A DIAGNOSTIC METHOD FOR NON-DETERMINISTIC FINITE STATE MACHINES

FRANK JIN YE LUO

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

SEPTEMBER 1996

ISBN 0-612-18413-7

Abstract

A Diagnostic Method for Non-Deterministic Finite State Machines

Frank Jin Ye Luo

In this research, we propose a diagnostic algorithm for the case where more than one fault (output and/or transfer) may be present in the transitions of an implementation modeled as a non-deterministic finite state machine (NFSM). If existing faulty transitions are identified by test cases in a test suite, this algorithm detects and localizes these faulty transitions. It generates a diagnosis, which is a set of faulty transitions with specific type of faults (output and transfer). The occurrence of all the faults allows the explanation of all observed outputs of the implementation. The algorithm guarantees the correct diagnosis of certain configurations of faults (output and/or transfer) in an implementation, which are characterized by a certain type of independence of the different faults. A simple example is used to demonstrate the different steps of the algorithm.

Acknowledgments

I would like to express my sincere gratitude to Dr. Ghedamsi, my thesis supervisor, for his financial support and valuable insight throughout this research.

I would also like to thank Dr. Atwood for his support in continuing supervision of the research.

I am indebted to my parents, grandmother, brother and sister for their patience, understanding and encouragement throughout this research.

I wish to acknowledge and thank my friend Mak Chung for reading and correcting the thesis.

Contents

List of Tables

List of Figures

List of Notations

Notation	Meaning
ε	null output
$s - a/b \rightarrow s'$	a transition
δ	fuzziness degree
γ	reset input
*	transfer fault
$^*(i/y)$	transfer fault candidate with fault information
Π	test suite
CC	set of correct candidates
F	upper bound of the number of transfer faults
FC	set of fault candidates
FH	fault hypothesis
FHG	fault hypothesis generation process
$i_{i,j}^k$	an input
I	set of inputs
K	maximum number of transitions with same input/output activities
Lc	maximum number of elements in a set of fault hypotheses for a given test case
Ls	number of test cases
^{o}	output fault candidate
$^o[y]$	output fault candidate with fault information
$o_{i,j}^k$	an output
O	set of expected output sequences
$\hat{O}$	set of observed output sequences
q	maximum number of tentative candidate sets
S	number of missing and additional output sequences
SFH	set of fault hypotheses
St	set of states
STC	set of tentative candidates
tc_i	a test case
$Trans$	transition
$T_{r,l}(S)$	an I/O tree rooted as a state s
$x.y$	concatenation of sets of input sequences
$T_{i,j}^k$	a transition
Y	set of outputs

Chapter 1

Introduction

Communication protocols are abstractly viewed as exhibiting *control flow* and *data flow*. Control flow deals with state changes in the system [Kaha 78]. Data flow deals with the input data and their representation/manipulation during state changes. The state of a protocol is defined as a stable condition in which the protocol rests until a stimulus, called an input, is applied. The protocol generates a response to the stimulus, called output (which may be null), when an input is applied, and moves into a new state (which may be the same as the previous state) where it stays until the next input.

Finite-state machine (FSM) models are very appealing for protocols since interactions can be modeled as inputs/outputs and protocol operations can be modeled by state changes. This is the reason that all protocol standards include a state table to describe the protocol behavior.

A protocol implementation has to be checked as to whether it conforms to its protocol specification or not. This activity is called *protocol conformance testing.* The data portion of a protocol is often tested using certain forms of static data flow analysis [Ural 87], [Vuon 89] and a functional testing approach [Sari 87]. To test the control portion of a protocol implementation, a set of *test cases*, called a *test suite*, is first generated according to a protocol specification, then the resulting test suite

is applied to the implementation under test (IUT) to see if the IUT conforms to its specification.

Typically, the formal conformance testing techniques generate a set of input sequences that will force the implementation to undergo all specified transitions. Very often, the test generation techniques assume the so-called *black box* approach where only the outputs generated by the implementation (upon receipt of inputs) are observable to the external tester. A considerable amount of work has been done relating to test generation methods for deterministic finite-state machine (DFSM) models. Some best known methods are Transition tour [Nait 81], W-method [Chow 78], Distinguishing Sequence method [Gone 70] and Unique-Input-Output (UIO) method [Sabn 88]. However, due to the nature of the non-determinism, test suites for nondeterministic finite-state machines (NFSMs) may become more complex, so that relatively few test selection methods for NFSMs have been reported. Some examples are [Kloo 93], [Trip 93], and [Luo 94a,b].

The protocol conformance testing mainly consists of three steps: Fault detection, Fault diagnosis and Fault correction. As indicated above, a large amount of work has been done in the first step, Fault detection, but little effort has been put into the second step. In the literature, we can only find a few papers in FSM diagnosis. Examples are [Ghed 92a,b] and [Ko 90]. However, none of these papers deals with diagnosing multiple faults of NFSM model. Therefore, we concentrate our efforts on solving the problem of localization of multiple faults in an IUT modeled by NFSMs. We propose an algorithm to localize identified faults in an IUT, through analyzing the test results. We apply test cases, generated by one of the existing test selection methods, to the IUT. If the observed output sequences, generated by the IUT, are different from the expected ones, then our algorithm is initialized. The algorithm allows the generation of different sets of possible transition faults. Each of them has the capacity of explaining all observed implementation behavior. In the case that more than one set is found, the algorithm will generate additional tests to distinguish these sets. After the tests are applied to the IUT and new output sequences are observed, the sets of transition faults which cannot explain new observations will be removed.

The thesis is organized as follows: The first part of chapter 2 introduces the definition of FSMs (DFSMs and NFSMs) and other related concepts. Then different test selection methods, for both DFSM and NFSM models, are presented. A review of diagnostic methods for FSMs can be found in chapter 3. Mainly we present three methods: two for diagnosing FSMs [Ghed 92b], [Ko 90] and one for diagnosing NFSMs [Ghed 92a]. We have three examples to illustrate different methods. In chapter 4, we describe our contribution: a method to diagnose multiple faults for NFSMs. A complete example is presented to illustrate different steps of the method. Chapter 5 introduces a diagnostic system, which is the implementation of the method described in chapter 4. It is able to diagnose multiple fault hypotheses for DFSM and NFSM. Chapter 6 contains a conclusion and discussion. In Appendix A, an algorithm to convert a normal NFSM to an Observable NFSM is presented. Finally, Appendix B contains object-oriented design notations used in Chapter 5.

Chapter 2

Test Selection Methods

Communication protocols are often modeled as an Extended Finite State Machine (EFSM) where the control portion is the Finite State Machine (FSM) [Kaha 78] and the data portion consists of the program segment. The state of a protocol is defined as a stable condition in which the protocol rests until a stimulus, called an **input**, is applied. The protocol generates a response to the stimulus, called **output** (which may be null), when an input is applied, and moves into a new state (which may be the same as the previous state) where it stays until the next input.

A protocol implementation has to be checked as to whether it conforms to its protocol specification or not. This activity is called **protocol conformance testing**. To test the control portion of a protocol implementation, a set of test cases, called **a test suite**, is first generated according to a protocol specification, then the resulting test suite is applied to the implementation under test (IUT) to see if the IUT conforms to its specification.

Typically, the formal conformance testing techniques generate a set of input sequences that will force the implementation to undergo all specified transitions. Very often, the test generation techniques assume the so-called *black box* approach where only the outputs generated by the implementation (upon receipt of inputs) are observable to the external tester. A considerable amount of work has been done relating to

test generation methods for deterministic finite-state machine (DFSM) models. Some well-known methods are Transition tour [Nait 81], W-method [Chow 78], Distinguishing Sequence method [Gone 70] and Unique-Input-Output (UIO) method [Sabn 88]. Due to the nature of the nondeterminism, test suites for nondeterministic finite-state machines (NFSMs) may become more complex, so that relatively few test selection methods for NFSMs have been reported. Some examples are [Kloo 93], [Trip 93], and [Luo 94a,b].

In this section, we will briefly introduce some test selection methods for DFSM and NFSM model. However before we continue, general concepts about finite state machine are described.

2.1 General concepts about finite state machines

Definition 2.1 *A* **finite state machine (FSM) M** *is defined as a quadruple (St, I, Y, Trans) where:*

St is the set of states of M. It includes an initial state s0,
I is the set of input symbols,
Y is the set of output symbols. It includes the null output (ε),
Trans is a relation between starting state and input on one hand, and ending state and output on the other hand.
$Trans \subseteq ((St \times I) \times (Y \times St))$

Couples $((s,a),(b,s')) \in Trans$ are called **transitions** of the machine M. The notation $s - a/b \rightarrow s'$ is also used to represent a transition. Two operations are applied on the transitions to get transition outputs and ending states. They are the O and the EndState operations, respectively.

In some FSM models, a **reset transition** is assumed to exist in each state of a machine. It is used to bring the machine back to its initial state. A reset transition takes the symbol γ as input and generates null output (the symbol ε) as output.

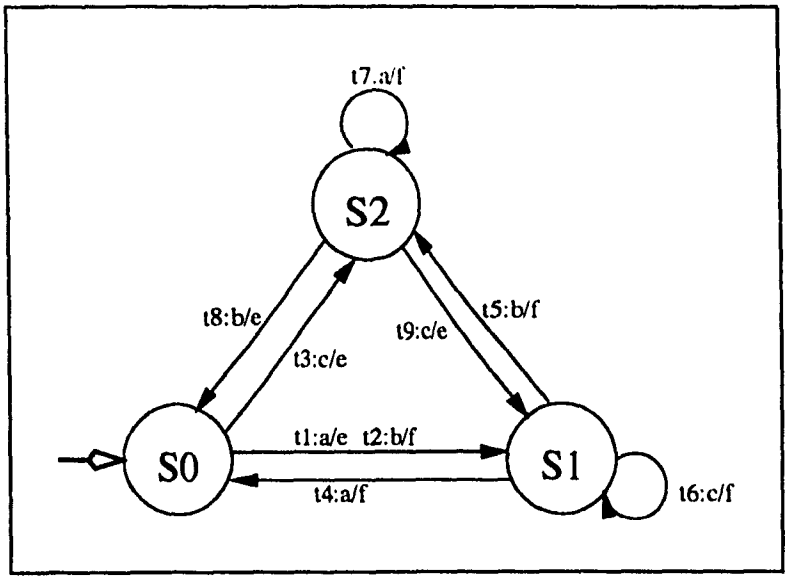


Figure 1: An example of a DFSM

In order to deal with null outputs (i.e, ε), we assume that the output ϵ is observed during a test by the application of an input and the non-observation of any output during a predetermined lapse of time. After deducing that a null output has occurred, the next input is allowed to be applied.

Definition 2.2 *A* **Deterministic FSM (DFSM)** *: If there is at most one transition with an input i ($i \in I$) defined for each state of an FSM M, then the machine M is called a Deterministic FSM.*

A graphic representation of a DFSM example, in the form of a **State transition diagram**, is given in Figure 1.

Definition 2.3 *A* **Nondeterministic FSM (NFSM)** *: If there may be more than one transition with an input i ($i \in I$) defined for any state of an FSM M, then the machine M is called a Nondeterministic FSM.* In the literature, DFSMs are usually referred as FSMs for simplicity, but in this thesis an FSM means a general finite state machine which could be either deterministic or nondeterministic.

An example of an NFSM is given in Figure 2.

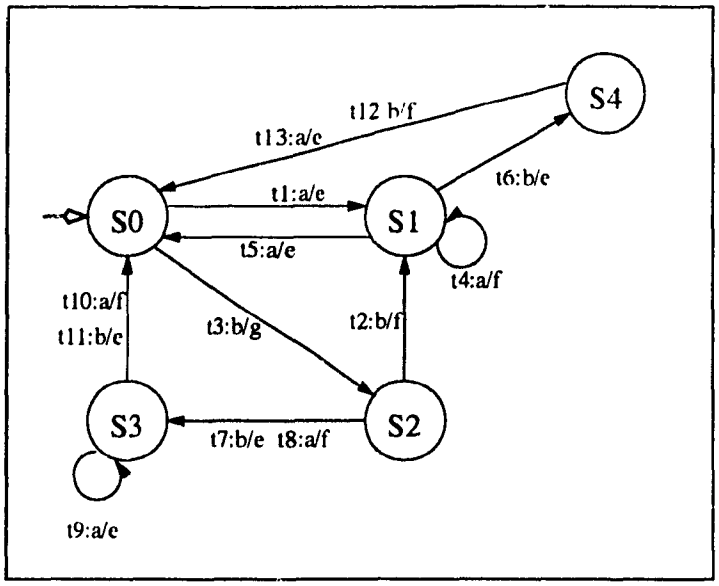


Figure 2: An example of an NFSM

Definition 2.4 *A FSM M is* **completely specified** *if for each input symbol i ($i \in I$), there exists one or more transitions defined for each state of M.* Usually, FSMs which are not completely specified are called **partially specified FSMs**.

Definition 2.5 *A FSM M is called* **strongly connected** *if M can transfer from any state to any other state.*

Definition 2.6 *A FSM M is called* **initially connected** *if M can transfer from the initial state to any other state.*

Definition 2.7 *In an FSM there may be some states which can never be distinguished from each other by their input/output behavior. States that can never be distinguished are called* **equivalent states**.

Definition 2.8 *Two DFSMs S and L are* **equivalent** *if their initial states S_0 and L_0 are equivalent.*

Definition 2.9 *An FSM is called* **reduced** *if and only if none of its states accepts the same set of input/output sequences.*

Definition 2.10 *Two states are* **distinguishable** *if and only if there is an input/output sequence x such that x can be accepted by only one of the two states but the input sequence that obtained by deleting all outputs in x can be accepted by both of them.*

Definition 2.11 *A FSM is* **minimal** *if and only if every two states are distinguishable.* A minimal FSM is reduced, but a reduced FSM is not necessarily minimal. Given a partially specified and reduced FSM M, if there are two states s_k and s_j, such that s_k only accepts input i_k but not i_j and state s_j accepts i_j but not i_k, then these two states are not distinguishable, hence machine M is not minimal.

Definition 2.12 Observable NFSMs (ONFSMs): *A NFSM M is said to be* **observable** *if, for every state s in M, there is no more than one transition that takes the same inputs, generates the same outputs and transfers to different states.* ONFSMs are a typical class of NFSMs where a state and an input/output pair uniquely determine the next state. However, an ONFSM may still be nondeterministic because, given a state and an input, one cannot determine a unique next state and a unique output. [Star 72] shows that each NFSM can be transformed into an equivalent ONFSM. The algorithm to convert from an NFSM to an ONFSM can be found in Appendix A.

2.2 Test selection methods for DFSM models

In this section, several test generation methods for DFSM are introduced very briefly. Most test generation methods can be viewed as consisting of two phases. The first phase derives some special input/output sequences, called state-identification sequences, for all states in the machine. Each of the states can be identified by its identification sequences. The second phase deals with the formation of the test sequences by concatenating test subsequences. Each test subsequence consists of an occurrence of a transition immediately followed by the state-identification sequences for the ending state of the transition. Test generation methods, such as DS [Gone 70], UIO [Sabn 88], W [Chow 78] and Wp [Fuji 91b] consist of these two phases, where

T-method [Nait 81]and CSP approach [Vuon90] do not. A comparison of various test selection methods can be found in [Fuji 91b].

2.2.1 The T-method

The T-method [Nait 81] generates test sequences known as a "transition tour". The tour exercises every transition of the implementation at least once and returns to its initial state. The drawback of the method is that it does not check the ending state of each transition, hence transfer faults may not be detected by the method. On the other hand, for certain protocols, which have a special message to determine the state of the protocol. the length of the tour can be minimized by the technique given in [Uyar 86], which is based on a graph theoretic concept called the Chinese Postman Problem [Kuan 62].

2.2.2 The DS-method

The DS-method [Gone 70] assumes that a DFSM is minimal, strongly connected, completely specified and possesses a **distinguishing sequence (DS)**. The distinguishing sequence is used as a state identification sequence. An input sequence is said to be distinguishing sequence for a DFSM, if the output sequence produced by the DFSM is different for each different state.

For each transition, $s_i - i/o \rightarrow s_j$, a test sequence is constructed by concatenating (1) the reset input γ, (2) the shortest path from initial state to state s_i, (3) the input symbol of the transition, i, and (4) the DS.

The DS-method is not applicable to all DFSMs since some of them may not have a DS.

2.2.3 The UIO-method

The UIO-method [Sabn 88] assumes a minimal, strongly connected and completely specified DFSM. This method involves deriving a **unique input/output (UIO)** sequence for each state of a DFSM. A UIO sequence for a state of a DFSM is an I/O behavior that is not exhibited by any other state of the DFSM.

For each transition, $s_i - i/o \rightarrow s_j$, a test sequence is constructed by concatenating (1) the reset input γ, (2) the shortest path from initial state to state s_i, (3) the input symbol of the transition, i, and (4) the UIO for state s_j. In general, test sequences generated by the UIO-method are shorter than those produced by the DS-method.

2.2.4 The W-method and Wp-method

The W-method [Chow 78] involves the derivation of a characterization set W of a DFSM. The W-set consists of input sequences that can distinguish between the behavior of every pair of states in the specification.

A set of test cases consists of the concatenation of a set P, and a set W. A set P is a transition cover, that is, for every transition $s_i - i/o \rightarrow s_j$, P contains an input sequence $x.i$ such that x and $x.i$ may lead the machine from the initial state to s_i and s_j respectively. The notation of $x.y$ is defined as the concatenation of the sequences x and y.

Provided that the number of states in the implementation remains within a certain bound, the W-method has the full power of detecting all output and transfer faults. However, test suites generated by the W-method are often longer than those produced by other test selection methods, such as the DS and UIO methods.

The Wp-method [Fuji 91b] is a modified version of the W-method. The only difference between the two methods is that instead of using the complete set W to check each reached state s_j, only a subset of the W-set, called the identification set, is used. Each state s_j in a DFSM has its own identification set, w_j. A set of test

cases are formed by concatenation of input sequences in the set P, say p, and the identification set w_j for the ending state s_j reached by p. The Wp-method has the same fault detection power as the W-method and the lengths of the test cases are shorter than the W-method.

2.2.5 The Constraints Satisfaction Problem Approach

Vuong [Vuon 90] proposed a novel method which does not make use of the conventional state and transition checking approach that was described previously. The proposed method applies a technique based on the constraints satisfaction problem (CSP) in AI. The basic idea of the approach is to generate a set of input/output sequences that can and only can be accepted by the given DFSM, and hence uniquely identifies the DFSM.

Generally speaking, a CSP involves a set of variables $X_1, \ldots, X_n$ having domains $D_1, \ldots, D_n$ where these variables take their values. A constraint $C_{i\ldots k}(X_i, \ldots, X_k)$ specifies the values $V_i, \ldots, V_k$, where $V_i \in D_i, \ldots, V_k \in D_k$, which the variables can take on. The CSP problem consists of finding all sequences of values $V_i, \ldots, V_k$ for a set of variables $X_i, \ldots, X_n$ that satisfy all the given constraints. In the test selection problem, each edge with label i/o forms a constraint on the two variables (sets of states) connected by this edge. The i/o constraints combined with the global constraints derived from the DFSM structure, restrict the values the states variables can take. In the CSP approach, the test sequence is generated incrementally from an initial test sub-sequence so as to satisfy the set of constraints which uniquely identify the given DFSM. Each additional sub-sequence is generated by considering the constraints imposed on the structure of the DFSM by the sub-sequences already generated previously. The test selection procedure can be summarized in the following three steps.

Step 1: Select an initial sequence and derive the corresponding constraints.

Step 2: Find the DFSMs which satisfy the constraints derived from the initial sequence.

Step 3: Generate additional sequences to distinguish the given DFSM from the remaining ones.

This method [Ko 90, Vuon 90] consists of the construction of all possible DFSMs which represent the faulty IUT, from the observed input/output sequences, hence it also allows fault detection and fault localization. The approach to detect and localize faults will be described in section 3.4.

2.3 Test selection methods for NFSM models

Generally, a nondeterministic FSM (NFSM) model may be used to represent different situations, such as the following [Boch 94]:

- a protocol entity with inherent nondeterminism;
- a set of deterministic protocol entities considered as options of a given protocol;
- a deterministic IUT embedded in a given system in such a way that a tester cannot directly observe [Petr 94b];
- nondeterminism due to concurrency. For example, a set of DFSMs, communicating with each other by input queues and channels, may have nondeterministic behavior and, in general, cannot be modeled as a global finite state machine.

By their nature, nondeterministic systems are more difficult to analyze than deterministic ones. Nondeterminism causes several problems such as the following [Boch 94]:

- an additional assumption about IUTs must be made, namely, the complete testing assumption, viz. the IUT should exhibit during testing all its nondeterministic choices;

- not just one, but several relations may be used as a conformance relation, such as Trace Equivalence [Luo 91a], Quasi-Equivalence [Luo 94b] and Reduction Relation [Petr 91a]. Therefore a test suite complete with respect to one relation might not be complete with respect to another (finer) relation;
- the notion of a fault must be defined in the context of a particular conformance relation. A simple mutation technique employed in the deterministic case may fail to explain faults in a nondeterministic implementation.

Test derivation and result analyses for NFSMs are relatively new research subjects. Only few test generation methods can be found in the literature. Examples are [Kloo 93], [Trip 93] and [Luo 94a,b]. The test methods are divided into the following two catalogs: (a) **Adaptive test generation methods.** A test is defined as a tree. By giving an input to an NFSM, different outputs may be seen, the next input that is given depends on the previous outputs. [Kloo 93] and [Trip 93] belong to this catalog; (b) **Preset test generation methods.** Like tests for the DFSM model, the entire input sequence is predefined independent of the outputs that are being produced. [Luo 91a,b] belong to this catalog.

2.3.1 Kloosterman's adaptive test cases generation method

According to Kloosterman's method [Kloo 93], each of the test cases is derived to check for one transition whether the output and the ending state of this transition conform the specification. A test case consists of four building blocks. They are as follows:

1. **Synchronizing Sequence (SS)**: With this sequence, the NFSM is transferred from any state to the initial state. It only consists of inputs.
2. **Transferring Sequence (TS)**: This sequence transfers the NFSM from the initial state to a starting state of the transition which is to be tested. It consists of inputs and outputs. This implies that the TS may become an adaptive experiment for non-deterministic specification.

3. **Input/ Output part (IO)**: This part consists of the input and the output of the transition to be tested. The input/output behavior of the transition is checked by this part.

4. **Unique Input Output sequence(UIOS)**: This is a sequence of inputs and outputs to check whether the tested transition transfers to the correct ending state. Like Transferring Sequence, UIOS is an adaptive experiment.

The following assumptions are made in [Kloo 93]:

- The NFSM is strongly connected,
- The NFSM is completely specified and
- The NFSM does not contain equivalent states.

Test Sequence Generation Algorithm:

a) Computing a synchronizing sequence (SS): A tree is used to compute the sequence. Each edge is labe' ı with an input and each node is labeled with a set of states, which denotes . states where the machine may reside. The initial node contains all states. The labels of the edges from a node with set of states **U** are inputs. The new node from a node with set U and an edge with input i contains all possible ending states of transitions whose starting states are in U and input is i. A path of the tree is terminated if:

- The set contains only the initial state. The machine is synchronized.
- The new set is already a node in the path to the root. It enters a cycle.

A synchronizing sequence is chosen from the paths from the root to a leaf node that contains only the initial state.

b) Computing transferring sequences (TS): A behavior tree is computed in this step. Each node of the tree is labeled by a set of states in which the machine can reside. The root node contains the initial state. The edges are labeled with

input-output pairs. A new node from a node with set U and edge with input i and output o is labeled by the ending states of transitions whose starting states are in U and which accept input/output i/o. The expansion of a node terminates if:

- The node contains only the state which is needed to transfer to.
- The new set is already a node in the path to the root. A cycle is entered.

A transferring sequence for the state s_i is chosen from the shortest path from the initial state to s_i. For each state in the machine, a transferring sequence is computed.

c) Computing unique input/output sequence (UIOS): In this step, a behavior tree is computed for each state of a machine. Each node of the tree is labeled with two sets of states. The edges are labeled with input-output pairs that alter the sets in the nodes. The first set of each node denotes the set of states in which the machine can reside when started in state(s) for which the UIOS has to be computed. The second set denotes set of states in which the machine can reside when started in any other states than the states for which the UIOS is computed and are not (yet) distinguished from the states in the first set.

Initially the first set contains all states for which the UIOS is computed. The second set contains all states except the states in the first set. The labels of the edges from a node with sets of states U_1 and U_2, $\langle U_1, U_2 \rangle$, are defined by the input/output pairs of different transitions whose starting states are in U_1. An input/output label of an edge is used to compute a new node. A new node $\langle U_{1'}, U_{2'} \rangle$ from a node labeled $\langle U_1, U_2 \rangle$ and edge with label i/o are computed as: first set of the new node, $U_{1'}$, contains all ending states of transitions whose starting states are in U_1 and accept input/output pair i/o, where second set of the new node, $U_{2'}$, contains all ending states of transitions whose starting states are in U_2 and accept input/output pair i/o. The expansion of a node terminates if:

- The second set is empty. A valid subsequence is found.
- There are states which are in both sets. A UIO sequence can never be found from the node because the second set cannot become empty.

- The new node already occurs in the path to the root. A circular path is entered.

After all nodes in the tree stop expanding, a behavior tree is specified. A UIO sequence is a subtree of the behavior tree with the same root node, and every leaf node of the subtree has a label in which the second set is empty.

d) Generating test cases: A test case for a transition t_i is formed by the concatenation of SS, TS, IO, UIOS, where TS is a transferring sequence for the start state of t_i; IO is the input/output which t_i accepts and UIOS is the UIO sequence of the ending state of t_i.

2.3.2 Tripathy's adaptive test case generation method

The method described in [Trip 93] is similar to [Kloo 93]. Both of them generate adaptive test cases and use UIO sequences as state identification sequences. The main differences between two methods are:

1. [Trip 93] assumes that reliable reset transitions are available for each state while [Kloo 93] explicitly introduces an algorithm to bring a machine from any state of an NFSM to its initial state (the synchronizing sequence).
2. [Trip 93] assumes that the NFSM is nondeterministic on only one input at any state, which means the method is only suitable for a specific type of NFSM.
3. The algorithms to derive UIO sequences are different.

The algorithm in [Trip 93] is described as follows:

An **Input/output(IO) tree** is defined as a rooted, unordered, and finitely branching tree generated from an NFSM M with branches labeled by the elements of the form a_k/o_l, where $a_k \in I$ and $o_i \in Y$.

The symbol $\mathbf{T_{r,l}(s)}$ represents an I/O tree rooted at a state s. The first index r represents the position from left to right of an event or a subtree in $T_{r,l}(s)$, while the

second index l represents the level of an event of a subtree in $T_{r,l}(s)$. The node of the tree is labeled by a state, and the branches of the tree are labeled with a/o_i or a_i/o_i depending upon whether a branch is nondeterministic or deterministic, respectively.

An IO tree is said to be an **adaptive input/output tree (AIO tree)** if the following conditions are satisfied:

i) each node in the IO tree has either one deterministic branch or all the nondeterministic branches from the node **and**

ii) each path from the root node to a leaf node is an UIO sequence for the root node in the NFSM M. □

An operator Θ is defined as follows:

- $\Theta(T_{r,l}(s)) = nil$ iff $T_{r,l}(s) = nil$
- $\Theta(a_b/o_b; T_{(b,1)}(s_b)) = a_b/o_b; \Theta(T_{(b,1)}(s_b))$
- $\Theta(T_{r,l}(s)) = \Big\{\sum_{i=1}^{q} \Theta(a/o_i; T_{i,l+1}(s_i)),$
  $\Theta(a_{q+1}/o_{q+1}; T_{q+1,l+1}(s_{q+1})),$
  $\Theta(a_{q+2}/o_{q+2}; T_{q+2,l+1}(s_{q+2})),$
  $\ldots\ldots$
  $\Theta(a_m/o_m; T_{m,l+1}(s_m))\Big\}$
- $\Theta(T_{r,l}(s)) = \Big\{\sum_{i=1}^{q} a/o_i; \Theta(T_{i,l+1}(s_i)),$
  $a_{q+1}/o_{q+1}; \Theta(T_{q+1,l+1}(s_{q+1})),$
  $a_{q+2}/o_{q+2}; \Theta(T_{q+2,l+1}(s_{q+2})),$
  $\ldots\ldots$
  $a_m/o_m; \Theta(T_{m,l+1}(s_m))\Big\}$

An algorithm to generate an AIO tree is given below.

Algorithm:	**Adaptive Input/Output Tree Generation**
Input:	NFSM M.
Output:	AIO Tree.

Method: Two sets $NBRANCH(s)$ and $DISTN(s)$ are used. $NBRANCH(s)$ is the set of all nondeterministic branches e with $Head(e) = s$, and $DISTN(s)$ is the set of all branches e with $Head(e) = s$.

step1: For each state $s \in S$, let p = $|NBRANCH(s)|$ and n = $|DISTN(s)|$. Compute

$$\Theta(T_{1,0}(s)) = \Big\{\sum_{i=1}^{p} a/o_i; \Theta((T_{i,1}(S_i)),$$
$$a_{p+1}/o_{p+1}; \Theta(T_{p+1,1}(S_{p+1})),$$
$$\ldots\ldots$$
$$a_n/o_n; \Theta(T_{n,1}(S_n))\Big\}$$

If there exists a tree rooted at state s, whose leaf nodes are *nil*, then that tree is the

AIO tree for the state s. Otherwise, set $l = 1$ and repeat the following step until an AIO is found.

step2: Let $T_{r,l}(S_r)$ be a subtree at level l, and $p = |NBRANCH(s_r)|$ and $n = |DISTN(s_r)|$, where $1 \leq r \leq n$. Compute

$$\Theta(T_{r,l}(s_r)) = \Big\{\sum_{i=1}^{p} a/o_i; \Theta(T_{i,l+1}(s_i)),$$
$$a_{p+1}/o_{p+1}; \Theta(T_{p+1,l+1}(s_{p+1})),$$
$$a_{p+2}/o_{p+2}; \Theta(T_{p+2,l+1}(s_{p+1})),$$
$$\ldots\ldots$$
$$a_n/o_n; \Theta(T_{n,l+1}(s_n))\Big\}$$

If there exists a tree rooted at state s, whose leaf nodes are *nil* , then that tree is the AIO tree for the state s . Otherwise, set $l = l + 1$ and go to step 2. □

A test case for a transition t_i is formed by the concatenation of the reset input γ, transferring sequence (described in the previous subsection), input/output of t_i, and the AIO tree for the ending state of t_i.

2.3.3 Luo's test case generation method based on partially-specified NFSM

The state machines derived from formal descriptions are often both partially-specified and nondeterministic. [Luo 94b] describes a test suite generation method for the software that is modeled by partially-specified nondeterministic FSMs (PNFSM). A conformance relation, called quasi-equivalence, is introduced to guide the test generation.

First, a PNFSM is transformed to an equivalent Observable PNFSM (OPNFSM), (see Definition 2.12) then test suites are generated from the resulting OPNFSM by a method called *Harmonized State Identification method (HSI-method).*

The method made the following assumptions:

- PNFSMs are initially connected. Otherwise, sub-machines which consist of all states and transitions that are reachable from the initial state are considered.
- The PNFSM specification is a reduced NFSM (Definition 2.9).
- The number of states in any implementation is bounded by a known integer. Without the assumption, no method can guarantee full fault coverage.
- A reliable reset input is available in any implementation of a PNFSM.

For conformance testing in the NFSM domain, different conformance relations may be applied. Because of the nature of PNFSMs, the Quasi-Equivalence relation is used to guide the test generation.

Definition 2.13 The Quasi-Equivalence *requires that, for every input sequence that can be accepted by a specification, the specification and its implementation produce the same set of output sequences.*

The paper also makes a so-called *complete-testing assumption*: it is possible, by applying a given input sequence to a given implementation a finite number of times, to exercise all possible execution paths of the implementation which are traversed by the input sequence [Fuji 91a, Luo 94a]. Without such an assumption, no test suite can guarantee full fault coverage.

The following two concepts are needed to describe the test generation algorithm.

Definition 2.14 Harmonized State Identification (HSI) sets $\{D_0, D_1, \ldots, D_{n-1}\}$: *for any two states that are distinguishable, there exists an input sequence in* $pref(D_i) \cup pref(D_j)$ *such that two different sets of output sequences are produced when this input sequence is applied to these states, respectively. The set* $pref(V)$ *is defined as a set that contains all sequences which are the prefixes of sequences in set* V. The size of HSI sets is generally smaller than the Characterization set (W-set). Both W-set and HSI-set generation algorithms can be found in [Luo 92].

Definition 2.15 Fuzziness degree δ *is the number of all different maximal sets of pairwise-distinguishable states in an OPNFSM. For a minimal OPNFSM,* $\delta = 1$.

Algorithm:	Test case generation for OPNFSMs
Input:	A specification S in the form of an OPNFSM, and the upper bound m on the number of states in the given NFSM implementation.
Output:	A test suite Π.

Step 1: Determine the fuzziness degree δ of S.

Step 2: Let the number of states in S be $n (n \leq \delta m)$. Find a set of harmonized state identification sets $\{D_0, D_1, \ldots, D_{n-1}\}$ for S.

Step 3: Construct a minimal **state cover** set Q. A set Q is a state cover if, for every state s_i in an FSM, Q contains an input sequence that may lead the machine from the initial state to s_i.

Step 4: Construct a test suite Π such that:

$$\Pi = \bigcup_{x_i \in Q.\{\epsilon\} \cup I \cup I^2 \cup \ldots \cup I^{\delta m - n + 1})} \{x_i\}.D_i$$

where x_i leads the machine from the initial state s_0 to s_i, D_i is the HSI set for state s_i, and I is the set of inputs of the machine. □

2.3.4 Luo's test case generation method based on communicating NFSM

The control portion of concurrent programs, especially in the area of communication software and communication protocols, can be modeled by a system of Communicating NFSMs (CNFSMs) where the NFSMs communicate with each other over queues and channels. A CNFSM system is defined as follows:

Definition 2.16 A system of CNFSMs, denoted by com($\mathbf{F_1}, \mathbf{F_2}, \ldots, \mathbf{F_n}$), *consists of a number of CNFSMs, $F_1, F_2, \ldots, F_n$, where*

1. *Each individual CNFSM is an NFSM plus an input FIFO queue; the NFSM only consumes the inputs in the queue.*
2. *Each pair of machines may have two FIFO channels for communication; each channel is designated for one direction of communication. A channel connects only two machines.*
3. *If a pair of machines, say F_1 and F_2, can communicate with one another through the channels between them, then signals from one machine pass through a FIFO channel and enter the input queue in the other machine.*

Channels can contain an unlimited number of signals. The signals can remain in a queue or in a channel, for an arbitrary period of time. An individual CNFSM has an input queue of an infinite length.

In [Luo 94a], a system of CNFSMs is first reduced into a single, trace-equivalence NFSM by reachability analysis; then the test sequences are generated from the resulting NFSM using the generalized Wp-method. The method is very similar to that reported in [Luo 94b]. The main difference is the construction of transition cover and state identification sets. Instead of a Harmonized State Identification (HSI) set in [Luo 94b], generalized Wp sets are used as the state identification sequences in [Luo94a].

Since we are mainly interested in test generation methods and result analysis, we will only discuss the test generation method for an NFSM, but skip the algorithm for the reduction from a CNFSM system to a single NFSM.

Definition 2.17 Trace equivalence *requires that a specification and its implementation produce the same set of possible output sequences for every input sequence.*

Definition 2.18 prefix set pref(V) for a given set of input sequences. *Given a set of input sequences* $V \subseteq I^*$, $\mathbf{pref(V)} = \{t_1 | T_2 \in I^* \ \& \ t_1.t_2 \in V \ \& \ t_1 \neq \varepsilon\}$ *where* $t_1.t_2$ *is concatenation of* t_1 *with* t_2.

Definition 2.19 A tuple of state identification sets $\{\mathbf{W_0}, \mathbf{W_1}, \ldots, \mathbf{W_{n-1}}\}$. *Given an ONFSM and a characterization set W,* $\{W_0, W_1, \ldots, W_{n-1}\}$ *is said to be a tuple of state identification sets if, for* $i = 0, 1, \ldots, n-1, W_i$ *is a set such that*

i) $W_i \subseteq pref(W)$;

ii) Given a state s_i, *for any other state* s_j, *there must exist an input sequences in* W_i *such that two different sets of possible output sequences are produced when this input sequence is applied to theses states respectively.*

Algorithm:	Test Case Generation for communicating NFSMs
Input:	A specification S in the form of a minimal ONFSM with n states, and the upper bound $m(n \leq m)$ on the number of states in the given NFSM implementation.
Output:	a test suite Π.

Step 1: Construct a characterization set W, and a tuple of state identification sets $\{W_0, W_1, \ldots, W_{n-1}\}$.

Step 2: Construct a minimal state cover set Q; that is, for every state s_i in an NFSM, Q contains an input sequence that may lead the machine from the initial state to s_i.

Step 3: Construct two sets P and R such that: $P = Q.(\{\varepsilon\} \cup I)$ and $R = P \backslash Q$.

Step 4: Construct a test suite Π in the following manner: $\Pi = \Pi_1 \cup \Pi_2$

where

$$\Pi_1 = Q.(\{\varepsilon\} \cup I \cup I^2 \cup \ldots \cup I^{m-n}).W,$$

and

$$\Pi_2 = \bigcup_{x_i \in R.I^{m-n}} \{x_i\}.D_i$$

where x_i leads the machine from the initial state s_0 to s_i, D_i is the HSI set for state s_i and I is the set of inputs of the machine. □

Chapter 3

Diagnostic Methods

3.1 Methodology for test result analysis and diagnostics

In the FSM diagnostic domain, the concept of "diagnosis based on a model" [Klee 87] is broadly used. The main idea is that, it is necessary to know how the system or the machine under test is supposed to work in order to be able to know why it is not working properly.

Observations of inputs and outputs show how the system to be diagnosed is behaving, while expectations, derived from its model, tell us how it is supposed to behave. The differences between expectations and observations, which are called **"symptoms"**, are hints of the existence of one or several differences between the model and its system. A diagnostic process often mainly consists of performing the following two tasks: the generation of candidates and the discrimination between candidates [Klee 87].

Generation of candidates: This process uses the identified symptoms and the model to deduce some diagnostic candidates. Each diagnostic candidate is defined to be the minimal difference, between the model and its system, capable

of explaining all symptoms. It indicates the failure of one or several components in the system.

Discrimination between candidates: Once the step of candidate generation terminates, we often end up with a number of diagnostic candidates. To reduce the number, some additional tests may be needed.

Both Ghedamsi's diagnostic methods [Ghed 92a,b] and our new proposed method consist of these two tasks, while Ko's method gives us a new point of view of diagnostic methods.

3.2 Ghedamsi's diagnostic tests for single transition faults in NFSMs

[Ghed 92a] proposed a diagnostic method for NFSM model. The method allows **implementations to have an output fault and/or a transfer fault, but only in one of its transitions**. The method localizes the faults (output and/or transfer) once the faults are detected by one or several test cases.

Algorithm:

Step 1: Construction of sets of expected output sequences

Because of non-determinism, a set of valid output sequences is expected for each test case in the given test suite. Given a test suite TS, $TS = \{tc_1, tc_2, \ldots, tc_n\}$, we construct a set of expected output sequences, O_i, for each test case $tc_i (1 \leq i \leq n)$. Each O_i contains all possible expected output sequences executed by tc_i, i.e., $O_i = \{o_i^1, \ldots, o_i^q\}$, where $o_i^k = \{o_{i,1}^k o_{i,2}^k \ldots o_{i,mi}^k\}$

Step2: Construction of sets of observed output sequences

In this step, the complete testing property [Luo 91a,b] is assumed to hold. By applying a test case, i.e., tc_i, to the implementation a sufficiently large number of times, all possible execution paths of the implementation are exercised and all possible output sequences are generated. For each test case tc_i, we form a set of observed output sequences, $\hat{O}_i$, by all generated output sequences corresponding to tc_i.

step3: Generation of symptoms

By comparing outputs in the sets $\hat{O}_i$ of observed output sequences with those in the corresponding sets O_i, we identify all differences and each difference represents a symptom.

Definition 3.1 *A transition $T^k_{i,j}$ of the specification where a symptom $(O_i \neq \hat{O}_i)$ (more precisely, $o^k_{i,j} \neq \hat{o}^k_{i,j}$) has been observed, is called a* **symptom transition**. *If we have the same symptom transition for all symptom, that transition is called* **the unique symptom transition (ust)**. *The observed output generated by the ust is called the* **unique symptom output (uso)**.

step 4: Generation of conflict sets

Definition 3.2 *A* **conflict set** *for a given symptom is defined to be the set of transitions which are supposed to participate (through their execution) in the generation of the symptom.*

For each symptom $(O_i \neq \hat{O}_i)$ identified in the step 3, a conflict set is constructed. We have the following three situations.

$O_i \subset \hat{O}_i$ The faulty implementation misses some expected output sequences. The conflict set will be formed by some transitions in the specification, which are

supposed to generate the missing expected output sequences.

$O_i \supset \hat{O}_i$ In this case, the implementation generates some extra output sequences. The conflict set will be formed by some transitions which participate in the generation of those extra observed sequences.

$O_i \not\subset \hat{O}_i$ and $O_i \not\supset \hat{O}_i$ In this case, the implementation generates some extra observed output sequences and has some expected output sequences missing. The conflict set is formed by some transitions that are supposed to generate the expected missing sequences and that generate the extra ones instead.

Also, a flag is introduced in this step and is initialized to be false. The flag is set to be true if in at least two sequences (an expected and its corresponding observed one), they differ with each other by at least two output symbols.

A formal description of the step follows:

If $(O_i \supset \hat{O}_i)$ Then $Plus_i = O_i - \hat{O}_i$
$Checkset_i = \hat{O}_i$
$ComputeConf(Conf_i, Plus_i, Checkset_i, O_i, flag)$
Else If $(O_i \subset \hat{O}_i)$ Then $Plus_i = \hat{O}_i - O_i$
$Checkset_i = O_i$
$ComputeConf(Conf_i, Plus_i, Checkset_i, O_i, flag)$
Else $Plus_i = \hat{O}_i - O_i$
$Checkset_i = O_i - \hat{O}_i$
$ComputeConf(Conf_i, Plus_i, Checkset_i, O_i, flag)$

Procedure ComputeConf(Conf, Plus, Checkset, O, flag)

/* Each call generates the minimal conflict set for the */

/* corresponding considered symptom. */

$Conf := \cap Conf_\sigma$, where $Conf_\sigma$ is computed as follows:

$\sigma \in Plus$

find $\sigma' \in Checkset$ such that it has the longest common prefix with σ, that is,

$\sigma = x_1x_2x_3 \ldots x_m x_{m+1} \ldots x_n$, $\sigma' = x_1x_2x_3 \ldots x_m x'_{m+1} \ldots x'_n$ and

$\sigma'' = x_1x_2x_3 \ldots x'_m x''_{m'+1} \ldots x''_n \Longrightarrow m' \leq m$

$Conf_\sigma = \{t_i | t_i$ is the specification transition determined by the input i_i,

the output x_i and their position in the corresponding sequences, $i \leq m\}$

$\bigcup \{t_{m+1} | t_{m+1}$ is determined by the input i_{m+1} , the output y_{m+1}

and their position in the corresponding sequences, where $y_{m+1} = x_{m+1}$

if $\sigma \in O$, otherwise, $y_{M+1} = x'_{m+1}\}$

The flag is set if $(x_{m+2} \ldots x_n \neq x'_{m+2} \ldots x'_n)$

Step 5: Generation of diagnostic candidates and their diagnoses

Definition 3.3 *A minimal set of faults, which has the capability of explaining all observed outputs, is called a* **diagnosis**. *The corresponding set of transitions, where these faults occur, is called a* **diagnostic candidate.**

In this step, an initial tentative candidate set "ITC" is formed first by the **intersection** of all conflict sets.

If there is a unique symptom transition "ust", it will be contained in the ITC. In this case, ITC will be split into two sets: **ustset**, which contains the ust, and **FTCtr** which contains the rest of the transitions in the ITC. These transitions are suspected to have at least one transfer fault. Otherwise, the full ITC set forms the FTCtr set and the ustset is kept empty.

According to the transitions in the sets (ustset and FTCtr) and the value of the flag, mutant machines are built and tested to help the generation of diagnoses. If a mutant machine passes the test, then the set of all faulty transitions, with specific outputs and/or ending states, are considered as a **diagnostic candidate**, or simply called a **diagnosis**.

Algorithm:

If the ustset is not empty, the following two cases are distinguished:

Case(1) the flag is false: The transition in ustset, ust, is suspected to have only an output fault. A mutant machine is built by changing in the specification machine the output of the transition ust to the unique symptom output (uso), then the given test cases are applied to the mutant. If the output sequences generated by the mutant are the same as those generated by the implementation, the mutant is considered to have passed the test. In this case, ust is considered as a diagnostic candidate and one diagnosis, stating that the ust might have the output fault uso, is created.

Case(2) the flag is true: The transition ust is suspected to have an output fault as well as a transfer fault. Again, we build a mutant machine by changing in the specification machine the output of the transition ust to the unique symptom output (uso). The end-state of the ust in the mutant needs to be changed also. We assign to the EndState of the ust all states in the machine, one state (i.e., s) at a time, except for the expected EndState of ust. After each assignment, the given test cases are applied to the resulting mutant. If the mutant generates the same output sequences as the observed output sequences generated by the implementation, the mutant passes the test. The ust is considered as a diagnostic candidate and one diagnosis, stating that ust might have the output fault uso and the ending state s, is created.

The elements (transitions) in set FTCtr are suspected to have transfer faults. For each transition, we consider all states in the machine, with the exception of the

expected ending state of T_k, one at a time. For each state s under consideration, a mutant machine is built by changing in the specification machine the ending state of T_k to the state s. If, by application of the given test cases, the mutant generates the same output sequences as the observed output sequences generated by the implementation, T_k is considered as a diagnostic candidate and one diagnosis, stating that ust might have the ending state s, is created.

Step 6: Additional diagnostic tests

Our goal is to localize the faults in the implementation. If only one diagnosis is found in Step 5, then the faults in the diagnosis are the real faults in the implementation. Otherwise additional test cases are needed to help reduce the number of diagnoses to only one.

The additional test cases selection approach consists of selecting a candidate transition which occurs in at least one of the diagnoses, and selecting addition diagnostic tests to determine which fault the transition contains, if any. This will eliminate all diagnoses that are not consistent with this new information. We repeat this approach until only one diagnosis is left.

An example:

Suppose the specification and the implementation are the machines represented in Figure 2 (page 7) and Figure 3, respectively. The test suite is given as follows.

$$TS = \{abbb, abab, aaab, bbabb, babab\}$$

Steps 1 and 2: The application of this TS to the specification and the implementation, yields the expected and observed output sequences, as shown in Table 1.

Step 3: Differences between the expected and observed output sequences are detected

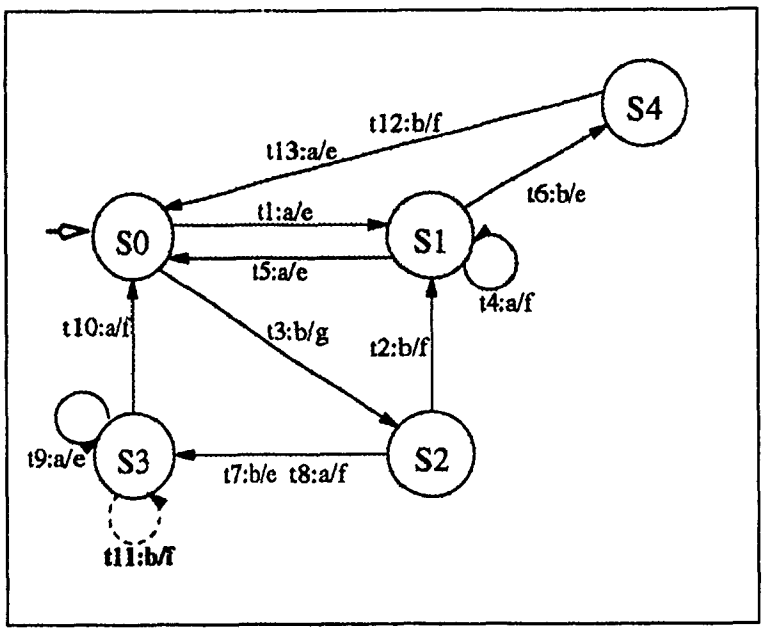


Figure 3: An implementation of the specification represented in Figure 2

Test Case	Expected Outputs	Observed Outputs
$tc_1 = a\ b\ b\ b$	$O_1 = \{e\ e\ f\ g\}$	$\hat{O}_1 = \{e\ e\ f\ g\}$
$tc_2 = a\ b\ a\ b$	$O_2 = \{e\ e\ e\ g\}$	$\hat{O}_2 = \{e\ e\ e\ g\}$
$tc_3 = a\ a\ a\ a$	$O_3 = \{e\ f\ e\ g,\ e\ e\ e\ e\}$	$\hat{O}_3 = \{e\ f\ e\ g,\ e\ e\ e\ e\}$
$tc_4 = b\ b\ a\ b\ b$	$O_4 = \{g\ f\ e\ g\ f, g\ f\ e\ g\ e,$ $g\ f\ f\ e\ f,$ **g e e e g,** $g\ e\ f\ g\ f, g\ e\ f\ g\ e\}$	$\hat{O}_4 = \{g\ f\ e\ g\ f, g\ f\ e\ g\ e,$ $g\ f\ f\ e\ f,$ **g e e f f,** $g\ e\ f\ g\ f, g\ e\ f\ g\ e\}$
$tc_5 = b\ a\ b\ a\ b$	$O_5 = \{$**g f e e e**$\}$	$\hat{O}_5 = \{$**g f f f g, g f f e f**$\}$

Table 1: Test cases and their output sequences (1)

for test cases tc_4 and tc_5. Those symptoms are

$$Sym_1 = (O_4 \neq \hat{O}_4)$$

$$Sym_2 = (O_5 \neq \hat{O}_5)$$

Step 4: Corresponding to the above symptoms, we have the following:

$$plus_1 = \hat{O}_4 - O_4 = \{g\ e\ e\ f\ f\}$$

$$checkset_1 = O_4 - \hat{O}_4 = \{g\ e\ e\ e\ g\}$$

$$plus_2 = \hat{O}_5 - O_5 = \{g\ f\ e\ e\ e\}$$

$$checkset_2 = O_5 - \hat{O}_5 = \{g\ f\ f\ f\ g,\ g\ f\ f\ e\ f\}$$

The expected output sequence "$g\ e\ e\ e\ g$" corresponds to path "$t_3t_7t_9t_{11}t_3$", therefore the conflict set is constructed as: $Conf_1 = \{t_3,\ t_7,\ t_9,\ t_{11}\}$ and $flag = true$. Similarly, we have the conflict set of Sym_2, $Conf_2 = \{t_3,\ t_8,\ t_{11}\}$ and $flag = true$.

Step 5: First, an initial tentative candidate set "ITC" is formed.

$$ITC = Conf_1 \cap Conf_2 = \{t_3,\ t_{11}\}$$

Then the ITC is split into two sets, *ustset* and $FTCtr$, as follows:

$$ustset = \{t_{11}\} \qquad FTCtr = \{t_3\}$$

Since the *flag* is set to true, the *ustset* has to be checked for simultaneous output and transfer faults. We build a mutant machine by changing, in the specification machine, the output of transition t_{11} to *uso*, which is "f", and the ending state of t_{11} to all states in the machine, one at a time, except for the correct ending state of the transition. After applying all test cases to the mutant, we will see that only when the ending state of t_{11} is changed to state S_3, does the mutant generate the same output sequences as the observations from the implementation. Hence a diagnosis that states that transition t_{11} has an output fault, which generates output symbol "f", and a transfer fault, which transfers to state S_3, is saved in a set called *PossFaults*.

We also need to check transitions in $FTCtr$ for transfer faults. In our example, only one transition t_3 is in $FTCtr$. We build a mutant machine by changing the ending state of transition t_3 to all states in the machine, one at a time, except for the correct ending state of the transition, and apply all test cases to the mutant. We will find that all resulting mutants fail to generate the same output sequences as observations. Hence no diagnosis is introduced by $FTCtr$.

Because there is only one diagnosis in $PossFaults$, we say that the diagnosis represents the real faults in the implementation and no additional test case is needed.

3.3 Ghedamsi's multiple fault diagnostics for DFSMs

In many cases, the assumption that implementations of an FSM have only single faults is not realistic, therefore a diagnostic method which allows multiple faults in the implementations of DFSMs is proposed in [Ghed 92b].

Definition 3.4 *A fault F of an implementation M in a transition t of S is said to be* **directly reached by a test case tc,** *if the execution of tc, as defined by the specification S, leads to the transition t, where there is no transfer fault in M on the path that leads from the initial state to t, and the subsequent path of the test case contains one or several symptoms.*

The algorithm makes the following assumption:
Assumption: For each fault in the implementation, there is a test case in the applied test suite which reaches that fault directly.

Algorithm:

Step 1: Generation of the expected and observed outputs.

Apply the test suite, TS, to the specification and the IUT. For each test case tc_i, the expected output sequence is written as $o_i = o_{i,1}o_{i,2}, \ldots, o_{i,m}$, while the observed output sequence is written as: $\hat{o}_i = \hat{o}_{i,1}\hat{o}_{i,2}, \ldots, \hat{o}_{i,m_i}$.

Step 2: Generation of symptoms.

Compare each observed output sequence with its corresponding expected output sequence and identify all differences for all test cases. Each difference is called a **symptom.**

Step 3: Construction of the set of tentative candidate sets.

Definition 3.5 *A transition name annotated by "or" represents a* **fault candidate** *which corresponds to an output fault candidate or a transfer fault candidate, respectively.*

Each **Tentative Candidate Set (STC)** is a set of fault candidates that are suspected to be faulty in the IUT.

Before the construction of the set of STC, we need to construct a set of **Fault Hypotheses (FH)** for each test case tc_i. Each FH is a pair **(FC, CC)**, where **FC** is a set of fault candidates and **CC** is a set of correct candidates, that is, fault candidates that are assumed not to be present.

A set of fault hypotheses SFH_i for test case tc_i is constructed as follows:

1. If tc_i has no symptom, SFH_i contains only one element of the form $\{\langle\{\}, \{t^o_{i,1}\}\rangle\}$, which indicates that the first transition, $t_{i,1}$, executed by tc_i, does not have an output fault.

2. If there are m symptoms in the comparison, we consider the different symptoms in order and include in FHset the fault hypotheses that correspond to the assumption that the symptom considered corresponds to a fault that is identified by the test case tc_i. The following situations may occur for the j-th symptom. Because of the assumption, the j-th symptom is considered only under the hypothesis that all earlier symptoms $j' < j$ correspond to output faults (an identified faulty transition must be a directly reached transition). We have the following two hypotheses:

   (i) The j-th symptom corresponds to an output fault (t^o_{i,k_j}), and there is no transfer fault on the execution path $(t_{i,k_{(j-1)}}, t_{i,k_{(j-1)}+1}, \ldots, t_{i,k_j-1})$ that leads to this symptom transition. In this case, the next symptom will also be considered (if it exists) since it corresponds to a directly reached fault. If there is no next symptom, then the fault hypothesis $\langle X, Y\rangle$ is added to FHset, where $X = \{t^o_{i,k1}, t^o_{i,k2}, \ldots, t^o_{i,k_j}\}$, $Y = \{t^o_{i,1}, t^*_{i,1}, t^o_{i,2}, t^*_{i,2}, \ldots, t^*_{i,k_j}\} - X$, t_{i,k_j} is the k_j-th transition where the j-th symptom has been observed.

   (ii) The j-th symptom corresponds to a transfer fault which is located in one of the transitions $(t_{i,k_{(j-1)}}, t_{i,k_{(j-1)}+1}, \ldots, t_{i,k_j-1})$ from the symptom transition (output fault in $t^o_{i,k_{(j-1)}}$) of the previous symptom (or from the initial state) to the symptom in question in the transition $t_{i,kj}$. In this case, the next symptom will not be considered, since it corresponds to a fault that is not identified by this test case, although it may be identified by another test case. For each transition $t_{i,n}$, $n = k_{(j-1)}, k_{(j-1)+1}, \ldots, k_j-1$, in the transition sub-sequence starting at $t_{i,k_{(j-1)}}$ and ending at the transition t_{i,k_j-1}, the fault hypotheses $\langle X, Y\rangle$ will be added to FHset, where $X = \{t^o_{i,k1}, t^o_{i,k2}, \ldots, t^o_{i,k_{(j-1)}}, t^*_{i,n}\}$, $Y = \{t^o_{i,1}, t^*_{i,1}, t^o_{i,2}, t^*_{i,2}, \ldots, t^o_{i,kj}\}$ - X.

The set STC is formed by all possible unions of fault alternatives taken from all test cases. Formally:

$$STC = \{TC | TC = \bigcup_{i=1,2,\ldots,Ls} FC_i \bigwedge TC \bigcap (\bigcup_{i=1,2,\ldots,Ls} CC_i) = \phi,$$

$$where \langle FC_i, CC_i \rangle \in SFH_i \ and \ Ls \ is \ the \ number \ of \ test \ cases\}.$$

Step 4: Generation of diagnostic candidates.

The set of tentative candidates, STC, derived in the previous step may contain a number of tentative candidates that do not explain the observations. Also, the ending states of transitions with transfer faults are unknown. Therefore, we build mutant machines, which correspond to the tentative candidate in question, to check whether or not the tentative candidate explains the observations, and, if yes, to find out the ending states of transitions with transfer faults.

Suppose that a tentative candidate, $Cand_i$, is under consideration. We build mutants by (1) assigning transitions, in the specification machine, that are suspected of having output faults as the corresponding symptom outputs, and (2) changing the ending states of all transitions that are suspected of having transfer faults. The possible ending states of a transition suspected of having transfer faults are states in the machine, except for the expected ending state of the transition. All other transitions remain unchanged. All test cases in TS are applied to the resulting machine (mutant). If the resulting outputs are the same as the observation, then the machine is considered as a possible implementation. The corresponding faults, with specific outputs and/or ending states, are saved in a set called **diagnostic candidate**, or simply **diagnosis**. The above process is repeated until all combinations of faults have been checked.

Step 5: Additional tests for reducing the number of diagnoses.

If more than one diagnosis is found in step 4, we need to reduce the number to one. We achieve this by selecting some additional test cases, applying the test cases to the IUT, and then removing diagnoses that fail to explain new observations.

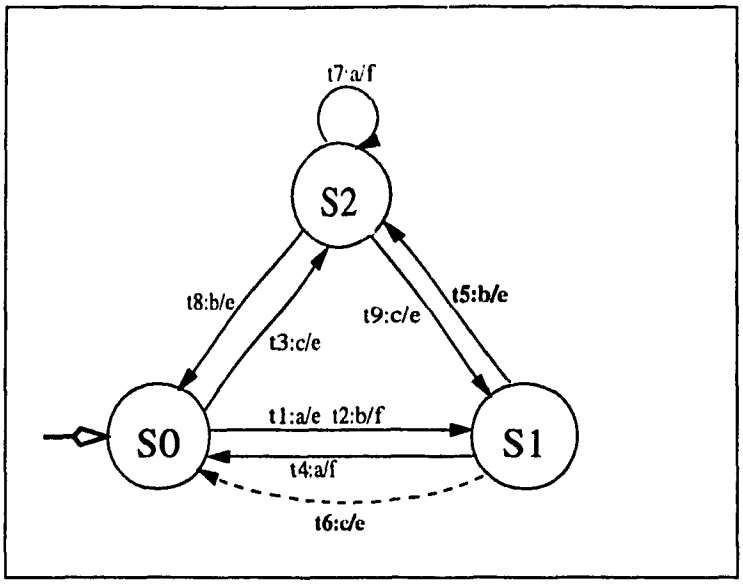


Figure 4: An implementation of the specification represented in Figure 1

Two approaches are presented in [Ghed 92b]. The first approach selects a fault in a diagnostic candidate and tries to select test case(s) to determine whether or not the fault exists in the implementation. After the application of the test case(s) to the implementation, all diagnoses that are not consistent with the new observation will be eliminated. The second approach compares the different diagnoses against one another. Inputs are applied to two DFSMs which correspond to two diagnoses until a difference between two outputs is observed. The drawback of the first approach is that the additional input sequences can not always be found and the drawback of the second is that it is time-consuming.

An application example

Suppose we have the specification machine shown in Figure 1 and the implementation machine in Figure 4. The following initial test suite, obtained by the Wp method, is given:

$$TS = \{aa;\ ab;\ bca;\ bcb\ ;\ baa;\ bbb;\ cab;\ cca;\ ccb;\ cba\}$$

Step 1: Apply the TS to the specification and the implementation, obtaining the expected and observed output sequences, as shown in Table 2.

tc #	Inputs	Specified transitions	Expected outputs	Observed outputs
1	a a	t1 t4	e f	e f
2	a b	t1 t5	**e f**	**e e**
3	b a a	t2 t4 t1	f f e	f f e
4	c c a	t3 t9 t4	e e f	e e f
5	c b a	t3 t8 t1	e e e	e e e
6	c a b	t3 t7 t8	e f e	e f e
7	b c a	t2 t6 t4	**f f f**	**f e e**
8	c c b	t3 t9 t5	**e e f**	**e e e**
9	b c b	t2 t6 t5	**f f f**	**f e f**
10	b b b	t2 t5 t8	**f f e**	**f e e**

Table 2: Test cases and their output sequences (2)

Step 2: Differences between the observed and expected outputs are found for test cases tc_2, tc_7, tc_8, tc_9, tc_{10}. Hence the following symptoms are generated:

$$Symp_{2,1} = (o_{2,2} \neq \hat{o}_{2,2}),$$
$$Symp_{7,1} = (o_{7,3} \neq \hat{o}_{7,3}). \qquad Symp_{7,2} = (o_{7,4} \neq \hat{o}_{7,4}).$$
$$Symp_{8,1} = (o_{8,3} \neq \hat{o}_{8,3}),$$
$$Symp_{9,1} = (o_{9,2} \neq \hat{o}_{9,2}),$$
$$Symp_{10,1} = (o_{10,3} \neq \hat{o}_{10,3}).$$

Step 3: Corresponding to the symptoms, the following sets of fault hypotheses are constructed:

$$
\begin{aligned}
SFH_1 &= \{(\{\}, \{t_1^o\})\} \\
SFH_2 &= \{(\{t_1^*\}, \{t_1^o\}), (\{t_5^o\}, \{t_1^o, t_1^*\})\} \\
SFH_3 &= \{(\{\}, \{t_2^o\})\} \\
SFH_4 &= \{(\{\}, \{t_3^o\})\} \\
SFH_5 &= \{(\{\}, \{t_5^o\})\} \\
SFH_6 &= \{(\{\}, \{t_3^o\})\} \\
SFH_7 &= \{(\{t_2^*\}, \{t_2^o\}), (\{t_6^*, t_6^o\}, \{t_2^o, t_2^*\}), (\{t_4^o, t_6^o\}, \{t_2^o, t_2^*, t_6^*\})\} \\
SFH_8 &= \{(\{t_3^*\}, \{t_3^o\}), (\{t_9^*\}, \{t_3^o, t_3^*, t_9^o\}), (\{t_5^o\}, \{t_3^o, t_3^*, t_9^o, t_9^*\})\} \\
SFH_9 &= \{(\{t_2^*\}, \{t_2^o\}), (\{t_6^o\}, \{t_2^o, t_2^*\})\} \\
SFH_{10} &= \{(\{t_2^*\}, \{t_2^o\}), (\{t_5^o\}, \{t_2^o, t_2^*\})\}
\end{aligned}
$$

According to the above sets of FHs, we construct the following set of tentative candidate (TC) sets:

$$
\begin{aligned}
\mathrm{STC} = \{ \quad & \{t_2^*, t_3^*, t_2^*, t_1^*\},\ \{t_2^*, t_3^*, t_2^*, t_5^o\},\ \{t_2^*, t_9^*, t_2^*, t_1^*\},\ \{t_2^*, t_9^*, t_2^*, t_5^o\}, \\
& \{t_2^*, t_5^o, t_2^*, t_1^*\},\ \{t_2^*, t_5^o, t_2^*\},\ \{t_6^o, t_3^*, t_5^o, t_6^*, t_1^*\},\ \{t_6^o, t_3^*, t_5^o, t_6^*\}, \\
& \{t_6^o, t_3^*, t_5^o, t_4^o, t_1^*\},\ \{t_6^o, t_3^*, t_5^o, t_4^o\},\ \{t_6^o, t_9^*, t_5^o, t_6^*, t_1^*\},\ \{t_6^o, t_9^*, t_5^o, t_6^*\}, \\
& \{t_6^o, t_9^*, t_5^o, t_4^o, t_1^*\},\ \{t_6^o, t_9^*, t_5^o, t_4^o\},\ \{t_6^o, t_5^o, t_6^*, t_1^*\},\ \{t_6^o, t_5^o, t_6^*\}, \\
& \{t_6^o, t_5^o, t_4^o, t_1^*\},\ \{t_6^o, t_5^o, t_4^o\}\}
\end{aligned}
$$

Step 4: For each element in STC, we build mutant machines to check if the tentative candidate explains the observation. If yes, we save the faults in the tentative candidate (with specific output and/or ending state) into a set called diagnosis.

The following diagnoses are constructed in this step:

$$
\begin{aligned}
Diag_1 &= \{t_9 \rightarrow S2, t_6 \rightarrow S0, t_1 \rightarrow S2, t_6 - e \rightarrow, t_5 - e \rightarrow\} \\
Diag_2 &= \{t_5 - e \rightarrow, t_6 - e \rightarrow, t_6 \rightarrow S0, t_9 \rightarrow S2\} \\
Diag_3 &= \{t_6 \rightarrow S0, t_1 \rightarrow S2, t_6 - e \rightarrow, t_5 - e \rightarrow\} \\
Diag_4 &= \{t_5 - e \rightarrow, t_6 - e \rightarrow, t_6 \rightarrow S0\}
\end{aligned}
$$

Step 5: Because more than one diagnosis is found in Step 4, additional test cases are needed to distinguish these diagnoses. In our case, the additional test cases can be "a a a" and "c c a b ". After applying the test cases to the implementation, we will find that only $Diag_4$ can successfully explain the new observations. Hence we can say

that $Diag_1$ contains the real faults in the implementation and other diagnoses can be removed from consideration.

3.4 Ko's approach

Section 2.2.5 describes a test generation method using CSP. The same approach can be used to detect and localize faults [Ko 90]. This approach is interesting since it, unlike other diagnostic approaches, does not take the specification as a reference to localize faults.

If the set of observed input/output sequences is different from the set of expected input/output sequences, which indicates that the IUT failed the test and that some of the transitions in it are faulty, then the CSP method is used to detect the faulty transitions.

First, the observed input/output sequences are used as the initial sub-sequence for the CSP method. After solving the CSP, each solution represents a possible DFSM model for the IUT. If more than one solution is found, new sub-sequences are selected to distinguish the real implementation from the other solutions.

On the other hand, if the CSP produces no solution, the basic assumptions of the method are violated and the IUT may have more states than the DFSM representing the specification, which means that the process of localizing faults becomes much harder. The observed inputs/outputs may be used as the initial sequences for DFSMs with more states.

Example: Consider the DFSM of Figure 5. The test suite is given as follows.

$$TS = \{aaab, bbbb, bbab\}$$

We assume that the application of the test sequence to the IUT produced the following:

Input:	a a a b	b b b b	b b a b
Expected Output:	e e e e	e f e e	e f e e
Observed Output:	e f e f	f f e f	f f e f

The observed output sequences are given to the CSP as initial test sequences. The number of states, in both the specification and the implementation, is assumed to be the same. The CSP has produced five DFSMs which are shown in Figure 3.4. In order to identify the real implementation, from all the other DFSMs, new sub-sequences are needed. The distinguishing sequences for the DFSMs can be the following:

Input sequences:	a b a a	a b b a b	a a b
Outputs (DFSM1):	e f e e	e f e e f	e f e
(DFSM2):	e f e f	e f e e f	e f e
(DFSM3):	e f e e	e f e f e	e f e
(DFSM4):	e f e e	e f e e f	e f f
(DFSM5):	e f e e	e f e f f	e f f

By applying these additional sub-sequences to the IUT and by observing the outputs, a definitive identification of the implementation DFSM can be reached.

It is to be noted that, one of the major disadvantages of the method is that the number of solutions could grow exponentially with the number of states in the given DFSM. The space required, therefore, could be huge.

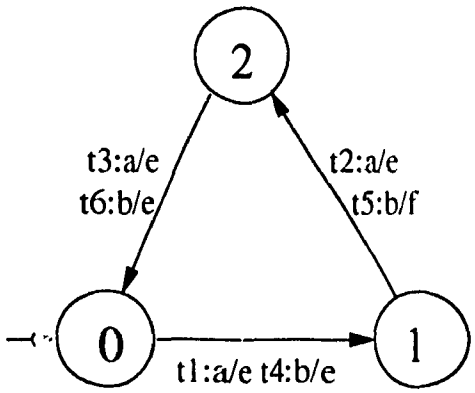


Figure 5: The DFSM for the given specification

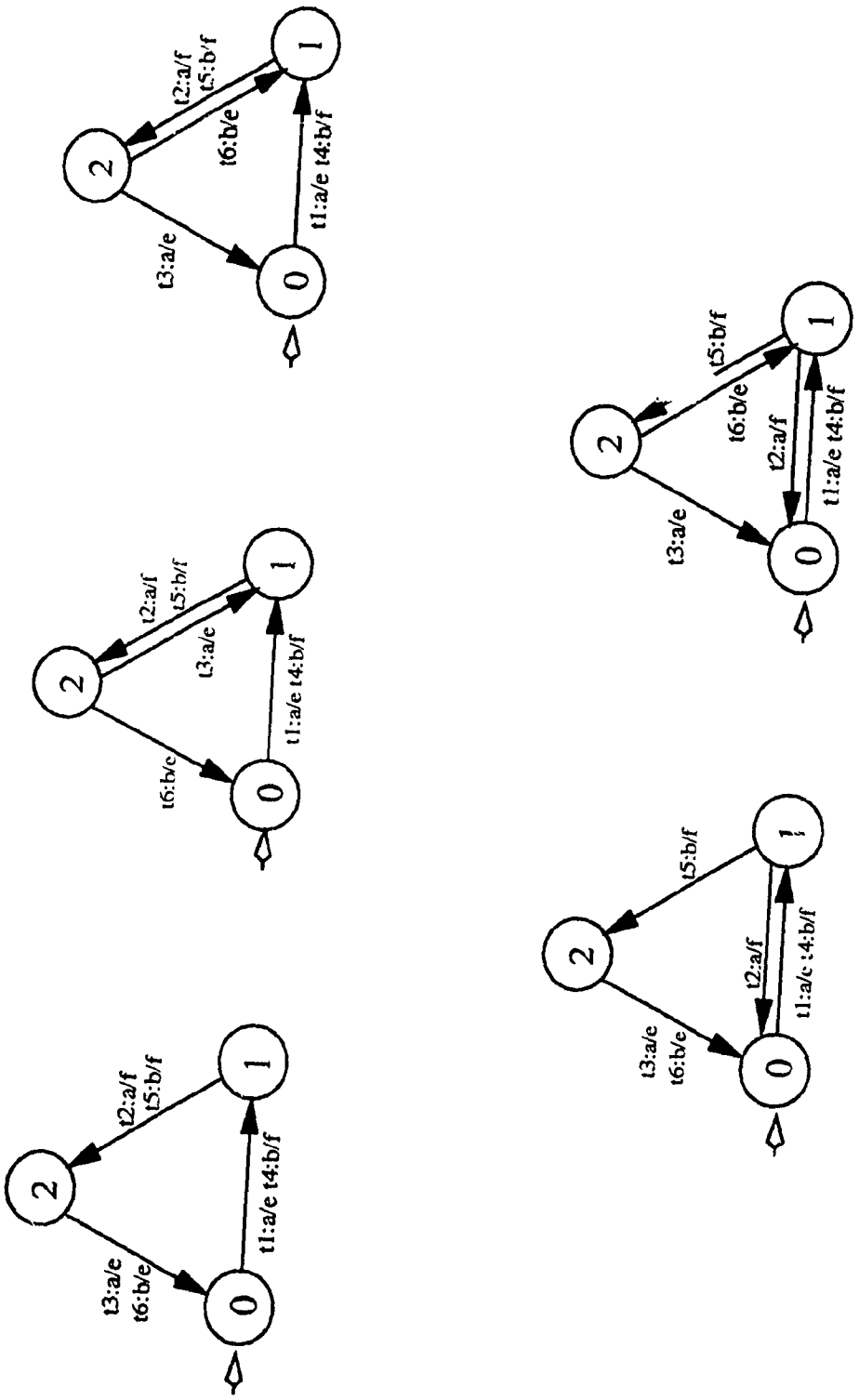


Figure 6: Possible DFSMs

Chapter 4

A Diagnostic Method for Non-Deterministic Finite State Machines

Nondeterminism and concurrency are two important features of formal specification languages for communication software, in particular, communication protocols. All the three major specification languages for communication software, LOTOS [Bolo87], ESTELLE [Budk 87], and SDL [Beli 89] support the description of nondeterminism and concurrency. So a machine abstracted from the formal description of a protocol might also be nondeterministic. Generally, a nondeterministic FSM (NFSM) model may be used to represent different situations.

Some work on test generation from NFSMs has been done in [Luo 94a], [Luo 94b], [Trip 93], [Kloo 93]. [Trip 93] and [Kloo 93] are based on the generalization of unique I/O sequences [Sabn 88], while [Luo 94a] uses a generalized Wp-Method and [Luo 94b] uses the HSI-method (Harmonized State Identification method).

The application of these test generation methods, however, provides only limited information about the locations of detected faults. In the communication protocol area, very little work has been done for the diagnostic and the fault localization

problems [Ghed 92a], [Ghed 92b], [Vuon 90]. [Ghed 92a] gives a diagnostic approach for single transition faults in NFSM models while [Ghed 92b] deals with multiple transition faults in FSM models. Based on these two approaches, we present in this chapter a diagnostic method for multiple transition faults in NFSM. The method allows the situation that more than one transition is faulty (output and/or transfer fault) in the implementation. It generates a minimal set of faulty transitions, with their specific faulty behaviors, if the faults in the implementation are detected and directly reached by test cases in a given test suite. The method is suitable for testing which conforms in accordance with Quasi-Equivalence [Luo 94a], Trace Equivalence [Luo 94b] or Reduction relations [Petr 94a]. If the method is applied to a DFSM, the time complexity is less than the one in [Ghed 92b].

The chapter is organized as follows: section 4.1 introduces a fault model of FSM, which will be used in our diagnostic method; section 4.2 includes definitions and concepts for diagnosing; in section 4.3, the proposed diagnostic method is described and a walk-through example is presented. Finally, in the last section, a concluding discussion is presented.

4.1 The FSM fault model

The FSM fault model [Boch 91] is based on faults made on labeled transitions. Some of these faults, which are essential in the diagnostic approach discussed in the following sections are defined as follows:

Definition 4.1 Output fault: *We say that a transition has an output fault if, for the corresponding state and received input, the implementation provides an output different from the one specified by the output function.*

Definition 4.2 Transfer fault: *We say that a transition has a transfer fault if, for the corresponding state and received input, the implementation enters a state other than that specified by the NextState function.*

Definition 4.3 Additional (missing) transition fault: *An implementation has an additional (missing) transition if, for a pair of present state and input, one more (one less) transition (with respect to the specification) is defined.*

For our diagnostic approach presented in the following sections, we assume the following fault model: **the implementation under test (IUT) may have one or several output faults and/or transfer faults in its transitions.**

In addition, certain cases of missing transition faults may also be explained by a combination of a transfer and/or an output fault as explained below. A missing transition might lead to an incompletely specified implementation (sometimes called a partially specified implementation). Different implementation assumptions may apply in this case, such as the following:

(a) Blocking: The input is blocked in the input queue, as defined by Estelle [Budk 87]. This case can not be modeled in general by a faulty transition.

(b) The input is dropped, as defined by SDL [Beli 89]. This case can be modeled by multiple faults, where the missing transition is seen as an existing one with a null output, which transfers back to its starting state.

(c) Some error indication. In this case, the fault is detected by the error indication. This case can be modeled by a multiple fault, where the faulty output has the error output and leads back to the same state.

4.2 Preliminary definitions

We assume in the following that an NFSM specification **S** is given, as well as an implementation under test (IUT) **M** with single or multiple output and/or transfer fault(s), as described in Section 4.1. In this chapter, the machine we refer to is an NFSM, unless indicated otherwise.

A **test suite**, **TS**, is defined as a set of test cases, where each test case is a

sequence of input symbols. We write $TS = \{tc_1, \ldots, tc_p\}$, where tc_i is a **test case.**

Because the specification S is a non-deterministic machine, when we apply a test case tc_i to S, different valid output sequences may be generated. **The set of expected output sequences of tc_i** covers all paths (sequences of transitions) which might be executed when the corresponding sequence of input symbols is applied. If a test case tc_i consists of m_i inputs $i_{i,1}i_{i,2}\ldots i_{i,mi}$, the corresponding set of expected output sequences is written as $\mathbf{O_i} = \{o_i^1, \ldots, o_i^q\}$, where $o_i^k = o_{i,1}^k o_{i,2}^k \ldots o_{i,mi}^k$ and output $o_{i,j}^k$ $(k = 1, \ldots, q)$ occurs after input $i_{i,j}$.

The implementation might also generate different output sequences for each test case tc_i. We write **the set of all possible observed output sequences of tc_i** as $\hat{\mathbf{O}}_\mathbf{i} = \{\hat{o}_i^1, \ldots, \hat{o}_i^r\}$, where $\hat{o}_i^k = \hat{o}_{i,1}^k \hat{o}_{i,2}^k \ldots, \hat{o}_{i,mi}^k$ (k = 1, ..., r), assuming that the **complete-testing property** [Luo 92] is satisfied. For a given test case tc_i, this property states that it is possible to execute all possible execution paths of the implementation, by applying the input sequence of the test case a finite number of times. The complete-testing property implies that the **"set of observed output sequences"** generated by the application of the test case tc_i is equal to $\hat{O}_i$.

In general, assuming that the nondeterministic behavior of an implementation can not be influenced by the environment, the complete-testing property does not hold. In many practical situations, however, the testing environment may influence the selection of alternative transitions through additional interaction parameters. In the case of random selection of alternatives, the probability that not all possible execution paths are executed at least once, may be reduced close to zero by applying the input sequence a sufficiently large number of times. Without the assumption, no conformance test checking the equivalence relation can be performed.

Definition 4.4 *A* **transition t** *of the specification S is said to be* **directly reached by a test case tc_i** *if the execution of tc_i, as defined by S, leads to the transition t, and there is no transfer fault in the implementation M on the path that leads from the initial state to t.*

For example, suppose the machine in Figure 7 represents the specification S and

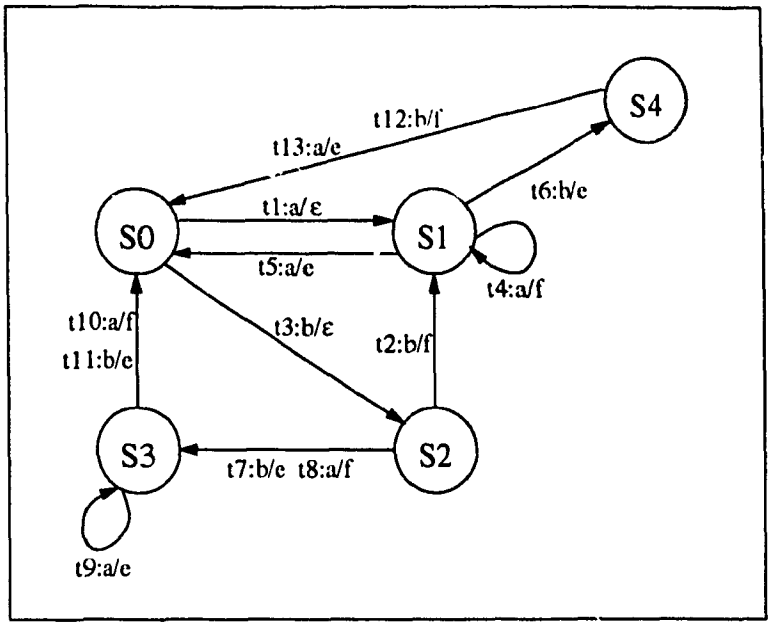


Figure 7: A state transition diagram of an NFSM

the one in Figure 8 represents the implementation M, we will find that test case "a a b b" exercises but does not directly reach transition t_{12} in M, because transition t_4 has a transfer fault and is exercised by the test case. However, t_{12} can be directly reached by another test case "a b b b". On this path no transfer fault occurs before t_{12} is reached.

Definition 4.5 *Suppose a test case* tc_i *exercises a path* p_i^k *in the specification S, the corresponding expected output sequence is* o_i^k. *If there exist faults in* p_i^k *in the implementation M, the observed output sequence produced by M becomes* $\hat{o}_i^k$ ($o_i^k \neq \hat{o}_i^k$). *We call* o_i^k *and* $\hat{o}_i^k$ *the* **missing and additional output sequence of the path** $\mathbf{p_i^k}$, *respectively.*

For example, suppose we have the specification S shown in Figure 7 and an implementation M' shown in Figure 9. When we apply the test cases "b b a b" and "b b b" to S and M' respectively, we will see some missing and additional output sequences of executed paths, as shown in table 3.

Definition 4.6 A missing output sequence of a test case $\mathbf{tc_i}$ *is the output*

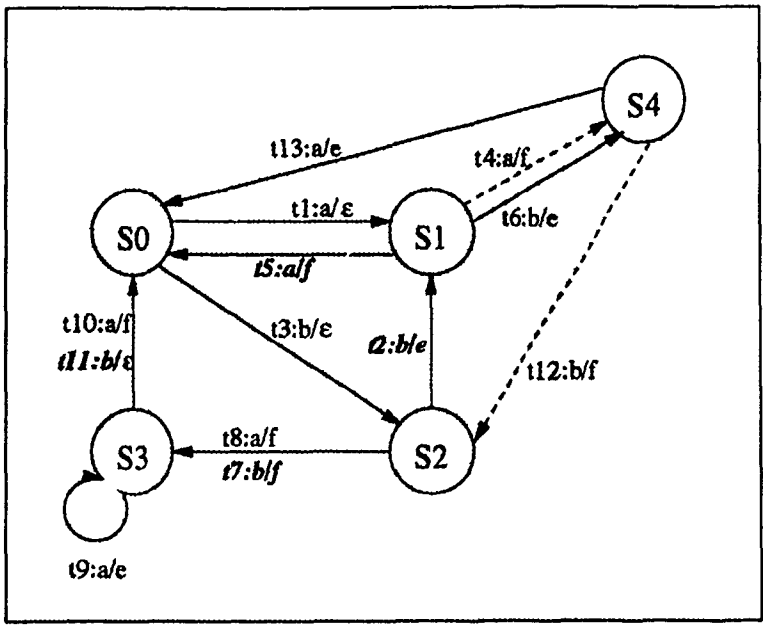


Figure 8: An implementation of the specification represented in Figure 7

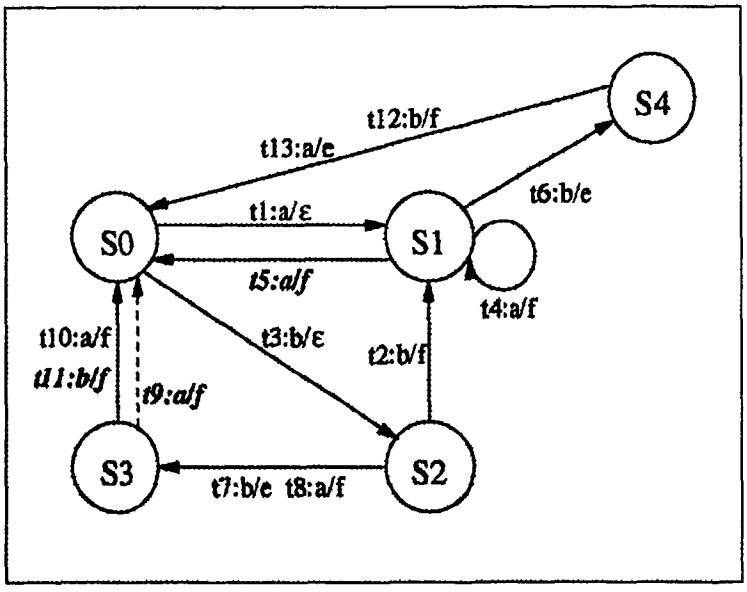


Figure 9: Another implementation of the specification represented in Figure 7

Test cases	Expected path in the specification	Expected output sequences		Observed output sequences	
b b a b	t_3, t_2, t_5, t_3	ε f e ε	Missing	ε f f ε	Additional
b b a b	t_3, t_2, t_4, t_6	ε f f e	Correct	ε f f e	Correct
b b a b	t_3, t_7, t_9, t_{11}	ε e e e	Missing	ε e f ε	Additional
b b a b	t_3, t_7, t_{10}, t_3	ε e f ε	Correct	ε e f ε	Correct
b b b	t_3, t_7, t_{11}	ε e e	Missing	ε e f	Additional
b b b	t_3, t_2, t_6	ε f e	Correct	ε f e	Correct

Table 3: Illustration of missing and additional output sequences of paths executed by "b b a b" and "b b b"

sequence found in the set of expected output sequences of tc_i, O_i, but not in the set of observed output sequences of tc_i, $\hat{O}_i$.

Definition 4.7 An additional output sequence of a test case tc_i *is the output sequence found in the set of observed output sequences of tc_i, $\hat{O}_i$, but not in the set of expected output sequences of tc_i, O_i.*

Table 4 lists missing and additional output sequences of test cases "b b a b" and "b b b". Comparing with table 3, we note that in most of the cases, the missing or additional output sequences for a path are the missing or additional output sequences for its corresponding test cases. However, this is not alway true, for example, the additional output sequence of path t_3, t_7, t_9, t_{11}, which is ε e f ε, happens to be the expected output sequence of path t_3, t_7, t_{10}, t_3, hence, it does not appear as an additional output sequence of test case "b b a b".

Definition 4.8 *A test case* **tc_i identifies a fault** *(output and/or transfer fault)* **in a transition t^k_{ij}** *if (1) $t^k_{i,j}$ is directly reached by tc_i, and (2) the faulty behavior can be observed from its missing or additional output sequence of tc_i.*

Taking the machine in Figure 7 as the specification S and the machine in Figure 8 as the implementation M, table 5 shows a few examples of identified and unidentified

Test case	Expected output sequences		Observed output sequences	
b b a b	ε f e ε	Missing	ε f f ε	Additional
	ε f f e	Not Faulty	ε f f e	Not Faulty
	ε e e e	Missing	ε e f ε	Not Faulty
	ε e f ε	Not Faulty		
b b b	ε e e	Missing	ε e f	Additional
	ε f e	Not Faulty	ε f e	Not Faulty

Table 4: Output sequences and types of faults of the test case "b b a b" and "b b b"

faults for the given test cases. Please note that some faults that are not identified by one test case may be identified by another. For example, the output fault in transition t_5 is not identified by test case "b b a b", but is identified by "a a a".

4.3 A Fault Hypothesis Generation (FHG) Process

In this section, we are going to present a process, called **Fault Hypothesis Generation (FHG) Process**, to identify possible faulty transitions (fault candidates) as well as correct transitions (correct candidates), and form **a set of fault hypotheses, FHset**, which contains such information. The process is a part of our proposed diagnostic algorithm (described in section 4.4). The FHset generated by the process will be used by Step 2 of the algorithm.

The original description of the process can be found in [Ghed 92b]. Readers who are interested can refer to that paper for more detail.

Definition 4.9 *A transition name annotated by* $^{\circ}$[y] *represents an* **output fault candidate**, *which indicates that the transition may generate "y" because of its output fault. A transition name annotated by* *(i/y) *represents a* **transfer fault candidate**, *which indicates that the transition may transfer to a state in the implementation*

Test case	Expected transition sequences	Expected output sequences	Observed output sequences	Fault(s) identified	Comments
a a a	t_1, t_4, t_4 t_1, t_4, t_5 t_1, t_5, t_1	ε f f ε f e ε e ε	ε f e ε f ε	output fault in t_5 and transfer fault in t_4	
a a b a	t_1, t_4, t_6, t_{13} t_1, t_5, t_3, t_8	ε f e e ε e ε f	ε f f f ε f ε f	output fault in t_5, transfer fault in t_4, but not transfer fault in t_{12}	t_{12} in implementation is not directly reached
b b a b	t_3, t_2, t_5, t_3 t_3, t_2, t_4, t_6 t_3, t_7, t_9, t_{11} t_3, t_7, t_{10}, t_3	ε f e ε ε f f e ε e e e ε e f ε	ε e f ε ε e f f ε f e ε ε f f ε	output fault in t_2, t_7 and t_{11}, transfer fault in t_4, but not output fault in t_5	output fault in t_5 is not detected

Table 5: Examples of identified and unidentified faults

because of its transfer fault, and that state has a transition which outputs "y" given an input "i". The notations [y] and (i/y) are called **fault information**. It can help us to reduce the number of tentative candidate sets, as well as the number of mutants, which will be described in section 4.4.

Definition 4.10 *A transition name annotated by $^\circ$ or * represents a* **correct candidate**, *i.e., the transition does not have output or transfer fault, respectively.*

Definition 4.11 *A* **Fault Hypothesis (FH)** *is a pair $\langle FC, CC \rangle$, where* **FC** *is a set of fault candidates and* **CC** *is a set of correct candidates.*

For example, the fault hypothesis $\langle \{t_2^*(a/f), t_2^o[g], t_1^o[g]\}, \{t_1^*\} \rangle$ represents the assumption that the transitions t_1 and t_2 have output faults that generate symbol g; t_2 has a transfer fault which transfers to a state s and s has a transition $t' : s - a/f \rightarrow s'$ (we do not know, at this stage, which state in the implementation M is s); t_1 does not have a transfer fault.

The process takes, as input, a quadruple of: a test case tc_i; a path expected to be

executed by tc_i, p_i^k; an expected output sequence o_i^m and an observed output sequence $\hat{o}_i^n$, and generates a set of fault hypotheses (FHset).

Definition 4.12 *The element FH in the FHset, which contains FC with the real implementation faults, and CC with no faulty transition in the implementation, is called the* **real FH or "RFH".**

The process ensures that it will generate an FHset that includes a **real FH (RFH), the FC of the RFH that contains all faults in path p_i^k that are identified by test case tc_i, and CC of the RFH that does not contain any fault in the implementation,** if the following relationship is confirmed:
the path p_i^k in the specification is implied to be executed by tc_i. The missing and additional output sequences of p_i^k are o_i^m and $\hat{o}_i^n$, respectively.

Let us say $tc_i = i_{i,1}i_{i,2}\ldots i_{i,l}$, $p_i^k = t_{i,1}^k t_{i,2}^k \ldots t_{i,l}^k$, $o_i^m = o_{i,1}^m o_{i,2}^m \ldots o_{i,l}^m$ and $\hat{o}_i^n = \hat{o}_{i,1}^n \hat{o}_{i,2}^n \ldots \hat{o}_{i,l}^n$. We know that, a transition $t_{i,j}^k$ may have an output fault if the expected and observed output symbols of $t_{i,j}^k$ are different, which means $o_{i,j}^m \neq \hat{o}_{i,j}^n$. $t_{i,j}^k$ may have a transfer fault if the expected and observed output sequences after the transition are different, which means $o_{i,j+1}^m o_{i,j+2}^m \ldots o_{i,l}^m \neq \hat{o}_{i,j+1}^n \hat{o}_{i,j+2}^n \ldots \hat{o}_{i,l}^n$. Therefore, we find a number **"h"** first, where the h-th symbol is the last one that is different in o_i^m and $\hat{o}_i^n$. We consider transitions in the path, one by one, identify all possible faults, until we meet the h-th transition $t_{i,h}^k$ in the path. Transitions after $t_{i,h}^k$ will not be considered even though they may be faulty, since the faults are not identified by the test case tc_i.

The following process illustrates how an FHset is constructed.

Process:	Fault Hypothesis Generation (FHG) Process
input:	1) test case tc_i
	2) path expected to be executed by tc_i, p_i^k
	3) missing output sequence of p_i^k, o_i^m and
	4) additional output sequence of p_i^k, $\hat{o}_i^n$
output:	a set of fault hypotheses (FHset)

$FHset = \phi$

For j = 1 to h $\quad \{*Total\ number\ of\ FHs\ will\ be\ h*\}$

$FC = \phi$

For j' = 1 to j

If $o^m_{i,j'} \neq \hat{o}^n_{i,j'}$ Then $FC = FC \cup t^o_{i,j'}[\hat{o}^n_{i,j'}]$

{* generate an output fault candidate *}

End $\quad$ {*For j' = 1 to j *}

If $o^m_{i,j+1}o^m_{i,j+2}\cdots o^m_{i,j+l} \neq \hat{o}^n_{i,j+1}\hat{o}^n_{i,j+2}\cdots\hat{o}^n_{i,j+l}$ Then $FC = FC \cup t^*_{i,j}(i_{i,j+1}/\hat{o}^n_{i,j+1})$

{* generate a transfer fault candidate *}

FC' is obtained from FC by eliminating fault information

$CC = \{t^o_{i,1}, t^*_{i,1}, t^o_{i,2}, t^*_{i,2}, \ldots, t^*_{i,j-1}, t^o_{i,j}\} - FC'$,

$FH = \langle FC, CC\rangle$ $\quad$ {* a pair of FC and CC forms an FH *}

$FHset = FHset \cup FH$ $\quad$ {* FHset is formed by union of all FHs *}

End $\quad$ {* For j = 1 to h *} □

The following example shows the construction of a set of fault hypotheses. Suppose we have the following:

Test Case:	$tc_i = i_{i,1}, i_{i,2}, i_{i,3}, i_{i,4}, i_{i,5}, i_{i,6}, i_{i,7}$
Transitions:	$p^k_i = t_{i,1}, t_{i,2}, t_{i,3}, t_{i,4}, t_{i,5}, t_{i,6}, t_{i,7}$
Expected outputs:	$o^m_i = o_{i,1}, o_{i,2}, \mathbf{o_{i,3}}, o_{i,4}, \mathbf{o_{i,5}}, \mathbf{o_{i,6}}, o_{i,7}$
Observed Outputs:	$\hat{o}^n_i = o_{i,1}, o_{i,2}, \mathbf{\hat{o}_{i,3}}, o_{i,4}, \mathbf{\hat{o}_{i,5}}, \mathbf{\hat{o}_{i,6}}, o_{i,7}$

where $o_{i,j} \neq \hat{o}_{i,j}$ when $j = 3, 5, 6$, then the following set of fault hypotheses is constructed:

FHset = {

$\langle\{t^*_{i,1}(i_{i,2}/o_{i,2})\}, \{t^o_{i,1}\}\rangle$, {* $t_{i,1}$ may have a transfer fault and its ending state may become a state that has a transition with input/output behavior $i_{i,2}/o_{i,2}$; $t_{i,1}$ does not have an output fault. *}

$\langle\{t^*_{i,2}(i_{i,3}/o_{i,3}), \{t^o_{i,1}, t^*_{i,1}, t^o_{i,2}\}\rangle$, {* $t_{i,2}$ may have a transfer fault, but no output fault: $t_{i,1}$ is not faulty *}

$\langle\{t^o_{i,3}[\hat{o}_{i,3}], t^*_{i,3}(i_{i,4}/o_{i,4})\}, \{t^o_{i,1}, t^*_{i,1}, t^o_{i,2}, t^*_{i,2}\}\rangle$, {* $t_{i,3}$ may have an output fault, instead of generating output $o_{i,3}$, $t_{i,3}$ generates $\hat{o}_{i,3}$; $st_{i,3}$ may have a transfer fault also; no faults in $t_{i,1}$ and $t_{i,2}$*}

$\langle\{t^o_{i,3}[\hat{o}_{i,3}], t^*_{i,4}(i_{i,5}/\hat{o}_{i,5})\}, \{t^o_{i,1}, t^*_{i,1}, t^o_{i,2}, t^*_{i,2}, t^*_{i,3}, t^o_{i,4}\}\rangle$,

$\langle\{t^o_{i,3}[\hat{o}_{i,3}], t^o_{i,5}[\hat{o}_{i,5}], t^*_{i,5}(i_{i,6}/\hat{o}_{i,6})\}, \{t^o_{i,1}, t^*_{i,1}, t^o_{i,2}, t^*_{i,2}, t^*_{i,3}, t^o_{i,4}, t^*_{i,4}\}\rangle$,

$\langle\{t^o_{i,3}[\hat{o}_{i,3}], t^o_{i,5}[\hat{o}_{i,5}], t^o_{i,6}[\hat{o}_{i,6}]\}, \{t^o_{i,1}, t^*_{i,1}, t^o_{i,2}, t^*_{i,2}, t^*_{i,3}, t^o_{i,4}, t^*_{i,4}, t^*_{i,5}\}\rangle\}$

Lemma 1: The process ensures that it will generate an FHset that includes a real FH (RFH). The FC set of RFH contains all faults in the path p^k_i that are identified by test case tc_i and the CC set of RFH does not contain any faulty transitions in the implementation.

Sketch of the proof: In the path p^k_i, if there is no transfer fault, or no transfer fault is identified by tc_i, then all identified output fault transitions will be contained in the FC of an FH, which is the RFH. If there is a transfer fault transition, say $t^k_{i,j}$, in the path and the fault is identified by tc_i, the transfer fault, as well as all output fault transitions before $t^k_{i,j}$, will be contained in the FC of an FH, and the corresponding CC will not contain any faulty transition in the implementation. The FH is RFH. Any faulty transitions after $t^k_{i,j}$ will not be contained in the FC of the RFH because they are not directly reached (hence not identified) by tc_i, although they may be identified by other test cases. □

4.4 The diagnostic algorithm

In this section, we will describe our proposed diagnostic algorithm. The diagnostic algorithm is based on a certain assumption about the faults contained in the implementation under test (IUT) and the test suite TS used to detect the presence of

faults. As explained below, the algorithm ensures correct and complete diagnosis if the following assumption is satisfied.

Assumption: There is at least one test case in the applied test suite that identifies each fault in the IUT.

4.4.1 Step 1: Generation of Expected Outputs and Observed Outputs

Apply the test suite, TS, to the specification and the IUT. For each test case tc_i, the corresponding set of expected output sequences is written as $O_i, O_i = \{o_i^1, \ldots, o_i^q\}$ where $o_i^k = o_{i,1}^k o_{i,2}^k, \ldots, o_{i,m_i}^k$ $(k = 1, \ldots q)$, and the corresponding set of observed output sequences is written as $\hat{O}_i = \{\hat{o}_i^1, \ldots, \hat{o}_i^r\}$, where $\hat{o}_i^k = \hat{o}_{i,1}^k \hat{o}_{i,2}^k, \ldots, \hat{o}_{i,m_i}^k$ $(k = 1, \ldots, r)$. The complete testing property (page 47) is assumed to be satisfied.

4.4.2 Step 2: Construction of a set of Fault Hypotheses (SFH) for each missing or additional output sequence

In this step, we construct a Set of Fault Hypotheses (SFH) for each missing or additional output sequence. Each SFH will contain a Real Fault Hypothesis (RFH). We make sure that, for each identified faulty transition in the implementation, there is an RFH whose FC contains the faulty transition (with corresponding fault).

By comparing O_i and $\hat{O}_i$ for each given test case tc_i, we identify all missing and additional output sequences of tc_i. A missing output sequence of tc_i is written as $o_i'^n$, and an additional output sequence of tc_i is written as $\hat{o}_i'^m$. For each missing or additional output sequence of tc_i, $o_i'^n$ ($\hat{o}_i'^m$), we construct a **Set of Fault Hypotheses** $SFH(o_i'^n)$ $(SFH(\hat{o}_i'^m))$ using the **Fault Hypothesis Generation process (FHG process)** described in the previous section. $SFH(o_i'^n)$ or $SFH(\hat{o}_i'^m)$ is constructed as follows:

(a) A missing output sequence $o_i'^n$ of tc_i indicates that the corresponding path p_i'', in the specification S, has one or more faulty transitions. If we knew which observed output sequence in $\hat{O}_i$ was the additional output sequence of $p_i''.\hat{o}_i''$, we could have used the FHG process to derive the $SFH(o_i'^n)$, given $tc_i, p_i^n, o_i'^n$ and $\hat{o}_i^n$ as the inputs. Unfortunately, because of the non-determinism, if there is more than one observed output sequence in $\hat{O}_i$, $(\hat{O}_i = \{\hat{o}_i^1, \hat{o}_i^2, \ldots, \hat{o}_i^m\})$, we are unable to tell which observed output sequence is $\hat{o}_i^k$. In order to ensure that $SFH(o_i'^n)$ contains the real fault hypothesis (RFH), we need to apply the FHG process m times for m different observed output sequences in O_i and obtain m sets of fault hypotheses (FHset). The $SFH(o_i'^n)$ is formed by the union of the FHsets. A formal description of the process is given below:

process: generation of SFH for a missing output sequence
$SFH(o_i'^n) = \phi$
for k = 1 to $|\hat{O}_i|$
    $FHset = FHG(tc_i, p_i^k, o_i'^n, \hat{o}_i^k)$ /* $(\hat{o}_i^k \in \hat{O}_i)$ */
    $SFH(o_i'^n) = SFH(o_i'^n) \cup FHset$
end /*for*/

(b) An additional output sequence of tc_i, $\hat{o}_i'^m$, indicates that, by applying test case tc_i to IUT, some faulty transition(s) in the IUT are executed and the IUT generates $\hat{o}_i'^m$. Similar to (a), if there is more than one path in the specification expected to be exercised by tc_i, we are unable to identify which path corresponds to the additional output sequence $\hat{o}_i'^m$. Therefore we give the FHG Process the quadruples as inputs: tc_i, p_i^k one $o_i'^k (o_i'^k \in O_i)$ each time, and $\hat{o}_i'^m$ and obtain an FHset for each input. SFH($\hat{o}_i'^m$) is formed by the union of the FHsets. A formal description of the process is given below:

process: generation of SFH for an additional output sequence
$SFH(\hat{o}_i'^m) = \phi$
for k = 1 to $|O_i|$
    FHset = FHG($tc_i, p_i^k, o_i^k, \hat{o}_i'^m$) /* $(o_i^k \in O_i)$ */
    $SFH(\hat{o}_i'^m) = SFH(\hat{o}_i'^m) \cup$ FHset
end /*for*/

Lemma 2: If Assumption 2 holds, each fault in the implementation is contained

in at least one RFH's FC set.

Sketch of the proof: According to Assumption 2, for each fault in a transition t, there is a test case, tc_i, that directly reaches t in the implementation M and identifies the fault. Then we must be able to find the corresponding missing or additional (or both) output sequence of tc_i (otherwise the fault is not identified). If the missing output sequence of tc_i, say o_i^k, is found, then $SFH(o_i^k)$ will be constructed. Since $SFH(o_i^k)$ is a superset of the *FHset* produced by $FHG(tc_i, p_i^k, o_i^k, \hat{o}_i^k)$, (where p_i^k is the path in the specification generating o_i^k, and $\hat{o}_i^k$ is the additional output sequence of p_i^k), and the *FHset* contains an RFH which includes the fault in t in its FC set (Lemma 1). In the case that the additional output sequence of tc_i is found, we can show the same conclusion. The proof is the same as the case of missing output sequences. □

4.4.3 Step 3: Construction of the set of Tentative Candidate Sets

Definition 4.13 A tentative candidate set (TCset) *is a set of fault candidates.*

Definition 4.14 A real TCset *is a TCset that contains all and only the faults in the implementation.*

In this step, we will construct **a set of TCset, (STC)** that contains the real TCset. Intuitively, due to **Lemma 2**, if an STC is formed by all possible unions of Fault Candidate sets (FC) of FHs derived in Step 2, the real TCset should be in the formed STC. However, if this is done, the size of the set of TCsets may become unnecessarily large. We therefore introduce a concept of **Conflict** to help us reduce the size.

Definition 4.15 Two FHs conflict with each other *if information contained in these two FHs is not consistent. Precisely,* **FH_i conflicts with FH_j** *if*

(1) there is a fault candidate $t_i^o[y]$ (or $t_i^*(i/y)$) in the set of fault candidate FC of FH_i and a correct candidate t_i^o (or t_i^*) in the set of correct candidate of FH_j. This represents the case that FH_i assumes that t_i is faulty while FH_j assumes that t_i is correct. *Or*

(2) there is a fault candidate $t_i^o[y]$ in the set of fault candidate FC of FH_i and another fault candidate $t_i^o[y']$ in the set of fault candidate FC of FH_j, where $y \neq y'$. This represents the case that FH_i assumes that t_i generates an output y while FH_j assumes that t_i generates y'.

For example, $FH_1\langle\{t_2^o[f], t_3^*(a/f)\}, \{t_1^o, t_1^*, t_2^*\}\rangle$ conflicts with FH_2 $\langle\{t_2^o[g], t_4^*(a/g)\}, \{t_1^o, t_1^*, t_2^o, t_2^*, t_3^o, t_3^*, t_4^o\}\rangle$ on two transitions: t_2 and t_3. FH_1 assumes that t_2 has an output fault which generates symbol f and t_3 has a transfer fault; FH_2 assumes that t_2 has an output fault that generates g (instead of f) and t_3 is not faulty.

We know that the real FHs in different sets of FHs will not conflict with each other, but other FHs (those are not real FHs) may have conflicts. Therefore, when we form a TCset, we will not take the FHs that are conflicting with each other.

We take <u>one and only one</u> FH from each SFH each time, and if there is no conflict between any two chosen FH, we form a Tentative Candidate set (TCset) by the union of the set of fault candidate FC of each FH. The set STC of the tentative candidate sets is formed by all resulting TCsets. Formally,

$$STC = \{TCset | TCset = \bigcup_{i=1,2,...,S} FC_i^k$$

where s is the number of missing and additional output sequences
& $\langle FC_i^k, CC_i^k\rangle = FH_i^k$, $FH_i^k \in SFH_i$
& FH_i^k does not conflict with $FH_{i'}^{k'}$ where $i \neq i'$ } □

Let us do an exercise to illustrate step 1 to 3. Again, assume the machine represented in Figure 7 is the specification and the one represented in Figure 9 the IUT. For the sake of simplicity, we only apply four test cases to these two machines: tc_1 :"b

b a b", tc_2 : "a b a b", tc_3 : "a b b b" and tc_4 : "b b b". We will find that only tc_1 and tc_4 generate missing and additional output sequences, which are already shown in table 4.

The set of fault hypothesis of the missing output sequence $o_{1,1}$ is formed as follows: apply the FHG Process three times, with inputs: $(tc_1, p_{1,1}, o_{1,1}, \hat{o}_{1,1})$, $(tc_1, p_{1,1}, o_{1,1}, \hat{o}_{1,2})$, and $(tc_1, p_{1,1}, o_{1,1}, \hat{o}_{1,3})$ (tc_1 = "$bbab$" and $p_{1,1} = t_3, t_2, t_5, t_3$), we will have three FH-sets, which are:

FHset#	Inputs	Fault hypothesis sets
$FHset_{1,1}$	$(tc_1, p_{1,1}, o_{1,1}, \hat{o}_{1,1})$	$\langle\{t_3 * (b/f)\}, \{t_3^o\}\rangle$, $\langle\{t_2 * (a/f)\}, \{t_2^o, t_3^o, t_3^*\}\rangle$, $\langle\{t_5^o[f]\}, \{t_2^o, t_2^*, t_3^o, t_3^*\}\rangle$
$FHset_{1,2}$	$(tc_1, p_{1,1}, o_{1,1}, \hat{o}_{1,2})$	$\langle\{t_3^*(b/f)\}, \{t_3^o\}\rangle$, $\langle\{t_2^*(a/f)\}, \{t_3^o, t_2^o, t_3^*\}\rangle$, $\langle\{t_5^o[f], t_5(b/e)\}, \{t_2^o, t_2^*, t_3^o, t_3^*\}\rangle$
$FHset_{1,3}$	$(tc_1, p_{1,1}, o_{1,1}, \hat{o}_{1,3})$	$\langle\{t_3^*(b/f)\}, \{t_3^o\}\rangle$, $\langle\{t_2^o[e], t_2^*(a/f)\}, \{t_3^o, t_3^*\}\rangle$, $\langle\{t_2^o[e], t_5^o[f]\}, \{t_2^*, t_3^o, t_3^*\}\rangle$

The set of fault hypothesis of the missing output sequence $o_{1,1}$, which is $SFH(o_{1,1})$, is formed by the union of $FHset_{1,1}$, $FHset_{1,2}$ and $FHset_{1,3}$.

Similarly, the resulting $SFH(o_{1,3})$, $SFH(\hat{o}_{1,1})$, $SFH(o_{2,1})$ and $SFH(\hat{o}_{2,1})$ are as follows.

$$
\begin{aligned}
SFH(o_{1,3}) = \{ \quad & \langle\{t_7^o[f], t_9^o[f], t_9^*(b/\epsilon)\}, \{t_3^o, t_3^*, t_7^*\}\rangle \\
& \langle\{t_7^*(a/f)\}, \{t_3^o, t_3^*, t_7^o\}\rangle \\
& \langle\{t_9^o[f], t_9^*(b/\epsilon)\}, \{t_3^o, t_3^*, t_7^o, t_7^*\}\rangle \\
& \langle\{t_3^*(b/f)\}, \{t_3^o\}\rangle \\
& \langle\{t_9^o[f], t_{11}^o[\epsilon]\}, \{t_3^o, t_3^*, t_7^o, t_7^*, t_9^*\}\rangle \\
& \langle\{t_7^o[f], t_7^*(a/f)\}, \{t_3^o, t_3^*\}\rangle \\
& \langle\{t_7^o[f], t_9^o[f]\}, \{t_3^o, t_3^*, t_7^*\}\rangle \\
& \langle\{t_3^*(b/e)\}, \{t_3^o\}\rangle \\
& \langle\{t_7^o[f], t_9^o[f], t_{11}^o[\epsilon]\}, \{t_3^o, t_3^*, t_7^*, t_9^*\}\rangle \ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{SFH}(\hat{o}_{1,1}) = \{\ & \langle \{t_3^{*}(b/f)\}, \{t_3^{o}\} \rangle \\
& \langle \{t_7^{o}[f], t_7^{*}(a/f)\}, \{t_3^{o}, t_3^{*}\} \rangle \\
& \langle \{t_7^{o}[f]\}, \{t_3^{o}, t_3^{*}\} \rangle \\
& \langle \{t_7^{o}[f], t_9^{o}[f], t_9^{*}(b/\epsilon)\}, \{t_3^{o}, t_3^{*}, t_7^{*}\} \rangle \\
& \langle \{t_2^{*}(a/f)\}, \{t_3^{o}, t_3^{*}, t_2^{o}\} \rangle \\
& \langle \{t_7^{o}[f], t_9^{o}[f], t_{11}^{o}[\epsilon]\}, \{t_3^{o}, t_3^{*}, t_7^{*}, t_9^{*}\} \rangle \\
& \langle \{t_4^{*}(b/\epsilon)\}, \{t_3^{o}, t_3^{*}, t_2^{o}, t_2^{*}, t_4^{o}\} \rangle \\
& \langle \{t_6^{o}[\epsilon]\}, \{t_3^{o}, t_3^{*}, t_2^{o}, t_2^{*}, t_4^{o}, t_4^{*}\} \rangle \\
& \langle \{t_5^{o}[f]\}, \{t_3^{o}, t_3^{*}, t_2^{o}, t_2^{*}\} \rangle \ \}
\end{aligned}
$$

$$
\begin{aligned}
SFH(o_{2,1}) = \mathrm{SFH}(\hat{o}_{2,1}) = \{\ & \langle \{t_3^{*}(b/e)\}, \{t_3^{o}\} \rangle \\
& \langle \{t_2^{o}[e], t_2^{*}(b/f)\}, \{t_3^{o}, t_3^{*}\} \rangle \\
& \langle \{t_2^{o}[e], t_6^{o}[f]\}, \{t_3^{o}, t_3^{*}, t_2^{*}\} \rangle \\
& \langle \{t_7^{*}(b/f)\}, \{t_3^{o}, t_3^{*}, t_7^{o}\} \rangle \\
& \langle \{t_{11}^{o}[f]\}, \{t_3^{o}, t_3^{*}, t_7^{o}, t_7^{*}\} \rangle \ \}
\end{aligned}
$$

Using the above sets of fault hypothesis, we construct the following set of tentative candidate sets:

$$
\begin{aligned}
\mathrm{STC} = \{\ & \{t_3^{*}(b/\epsilon), t_3^{*}(b/f)\}, \\
& \{t_7^{*}(a/f), t_2^{*}(a/f), t_7^{*}(b/f)\}, \\
& \{t_7^{*}(a/f), t_5^{o}[f], t_5^{*}(b/\epsilon), t_7^{*}(b/f), t_4^{*}(b/\varepsilon)\}, \\
& \{t_7^{*}(a/f), t_5^{o}[f], t_5^{*}(b/e), t_7^{*}(b/f), t_6^{o}[\varepsilon]\}, \\
& \{t_7^{*}(a/f), t_5^{o}[f], t_5^{*}(b/e), t_7^{*}(b/f)\}, \\
& \{t_7^{*}(a/f), t_5^{o}[f], t_7^{*}(b/f), t_4^{*}(b/\varepsilon)\}, \\
& \{t_7^{*}(a/f), t_5^{o}[f], t_7^{*}(b/f), t_6^{o}[\varepsilon]\}, \\
& \{t_7^{*}(a/f), t_5^{o}[f], t_7^{*}(b/f)\}, \\
& \{t_9^{o}[f], t_9^{*}(b/\varepsilon), t_2^{*}(a/f), t_{11}^{o}[f]\}, \\
& \{t_9^{o}[f], t_9^{*}(b/\varepsilon), t_5^{o}[f], t_5^{*}(b/e), t_{11}^{o}[f], t_4^{*}(b/\varepsilon)\}, \\
& \{t_9^{o}[f], t_9^{*}(b/\varepsilon), t_5^{o}[f], t_5^{*}(b/e), t_{11}^{o}[f], t_6^{o}[\varepsilon]\}, \\
& \{t_9^{o}[f], t_9^{*}(b/\varepsilon), t_5^{o}[f], t_5^{*}(b/e), t_{11}^{o}[f]\}, \\
& \{t_9^{o}[f], t_9^{*}(b/\varepsilon), t_5^{o}[f], t_{11}^{o}[f], t_4^{*}(b/\varepsilon)\}, \\
& \{t_9^{o}[f], t_9^{*}(b/\varepsilon), t_5^{o}[f], t_{11}^{o}[f], t_6^{o}[\epsilon]\}, \\
& \{t_9^{o}[f], t_9^{*}(b/\varepsilon), t_5^{o}[f], t_{11}^{o}[f]\}, \\
& \{t_7^{o}[f], t_7^{*}(a/f), t_2^{o}[e], t_2 :^{*} (a/f), t_2^{*}(b/f)\}, \\
& \{t_7^{o}[f], t_7^{*}(a/f), t_2^{o}[e], t_5^{o}[f], t_6^{o}[f]\}, \\
& \{t_7^{o}[f], t_9^{o}[f], t_2^{o}[\epsilon], t_2^{*}(a/f), t_9^{*}(b/\varepsilon), t_2 :^{*} (b/f)\}, \\
& \{t_7^{o}[f], t_9^{o}[f], t_2^{o}[e], t_2^{*}(a/f), t_{11}^{o}[\varepsilon], t_2 :^{*} (b/f)\}, \\
& \{t_7^{o}[f], t_9^{o}[f], t_2^{o}[e], t_2^{*}(a/f), t_2^{*}(b/f)\}, \\
& \{t_7^{o}[f], t_9^{o}[f], t_2^{o}[e], t_5^{o}[f], t_9^{*}(b/\varepsilon), t_6^{o}[f]\}, \\
& \{t_7^{o}[f], t_9^{o}[f], t_2^{o}[e], t_5^{o}[f], t_{11}^{o}[\varepsilon], t_6^{o}[f]\}, \\
& \{t_7^{o}[f], t_9^{o}[f], t_2^{o}[e], t_5^{o}[f], t_6^{o}[f]\}\ \}
\end{aligned}
$$

Complexity approximation of Step 3

Suppose we use **Ls** test cases; the maximum number of elements in a set of fault hypotheses for a given test case is **Lc**. Since the number of missing and additional output sequences is equal to the number of sets of fault hypotheses, if the number of missing and additional output sequences is **S**, then the maximum number of tentative candidate sets **q** , which is the number of all combinations of elements in the **S** sets of fault hypotheses, is bounded by $\mathbf{Lc^S}$. Please note that the actual number of tentative candidate sets **q** should be much smaller than $\mathbf{Lc^S}$ because a large number of elements

in a set of fault hypotheses will conflict with elements in another set, so they will not participate in forming a tentative candidate set (TCset).

4.4.4 Step 4: Generation of Diagnoses and Possible Implementations (PIs)

In Step 3, we derived a number of tentative candidate sets. In order to identify the real tentative candidate set, we need to find all **diagnostic candidates** first.

Definition 4.16 *A* **diagnostic candidate, or diagnosis for short,** *is a tentative candidate set and an assignment of faults (specific output and/or transfer to a specific state) to all its transitions which succeed in explaining all observations.* Note that a given tentative candidate set may lead to several diagnostic candidates.

A diagnostic candidate can be obtained by checking for each possible assignment of faults in the corresponding tentative candidate set (TCset), according to fault information, whether or not it explains all observations. We achieve this by applying all test cases in TS to the mutant machine that corresponds to the diagnostic candidate in question. If the outputs obtained from the mutant are identical to the outputs observed from the IUT, the diagnostic candidate is confirmed. The process to compute all possible faults for each tentative candidate, and hence all corresponding diagnostic candidates, is described as follows:

Suppose that a tentative candidate set, $\mathbf{TCset_i}$, in STC is under consideration and it has n transitions suspected of having transfer faults, and m transitions suspected of having output faults. We assign the transitions in the specification machine suspected of having these output faults in the $TCset_i$ the corresponding outputs according to the fault information. For example, when the tentative candidate set $\{t_9^o[f],\ t_9^*(b/\varepsilon),\ t_2^*(a/f),\ t_{11}^o[f]\}$ is under consideration, we change, in the specification S presented in figure 7, the output of transitions t_9 and t_{11} from 'e' to 'f'. We also change the ending states of the remaining transitions in the $TCset_i$ that are suspected of having transfer faults. The possible ending states of a transfer fault transition with

fault information $^*[i/y]$ are states s with a transition $s - i/y \rightarrow s'$ in the mutant machine. All remaining specification transitions are left unchanged. In our example, the possible ending states of $t_2^*(a/f)$ are S2 (because of $t_8 : S2 - a/f \rightarrow S3$) and S3 (because of t_9, the output of t_9 is changed from e to f), and the possible ending state of $t_9^*(b/\varepsilon)$ is S0. We apply test cases in TS to the resulting machine (mutant). If the resulting outputs are identical to those of the IUT, then the corresponding tentative candidate set, $\mathbf{TCset_i}$, is a possible real TCset. In this case, all assignments to the mutant form a **diagnostic candidate** and the mutant is called a **Possible Implementation (PI)**. If all combinations of faults for $TCset_i$'s elements fail to produce the same outputs as those obtained from the IUT, then $TCset_i$ is not the real tentative candidate set and needs no further consideration.

The above process is repeated until all combinations of faults for $TCset_i$'s elements are considered, which means that different possible ending states to $TCset_i$'s n elements suspected of having transfer faults are assigned, in addition to the new outputs to the remaining elements of $TCset_i$. For example, three mutants are built and tested corresponding to the TC set: $\{t_9^o[f], t_9^*(b/\varepsilon), t_2^*(a/f), t_1 1^o[f]\}$ The assignments are: outputs of t_9 and $t_1 1$: f, ending state of t_9: S0, and ending state of t_2: S2 and S3 for each mutant respectively.

Each resulting diagnosis has the ability to explain all observed outputs. It consists of the minimal set of faults (output and/or transfer), which might be present in the given implementation. In the example presented in the Step 3, all tentative candidate sets failed to explain the observations except for $\{t_9^o[f],\ t_9^*(b/\varepsilon),\ t_5^o[f],\ t_{11}^o[f]\}$ whose diagnostic candidate is $\{t_9 - f \rightarrow S0, t_5 - f \rightarrow, t_{11} - f \rightarrow\}$ ($t_i - \rightarrow s_j$ means that t_i might have transferred to s_j, while $t_k - x \rightarrow$ means that t_k might have the output fault of x).

Complexity approximation of Step 4:

Each element in the set of tentative candidates, produced in the previous step, contains several candidate transitions. Some of them are suspected of having transfer faults and the rest are suspected of having output faults. If **F** is the upper bound

of the number of transfer faults (this number can never be larger than the number of missing and additional output sequences **S**) in any of the $\mathbf{Lc^S}$ tentative candidates determined during Step 3, and the maximum number of transitions with same input/output activities is **K** (this number determines the maximum number of assignments of the ending state of a transition with transfer fault), the complexity of Step 3, in terms of possible number of diagnoses, will be $\mathbf{O((Lc^S).K^F)}$.

4.4.5 Step 5: Additional tests for reducing the number of diagnoses

Our main purpose is to localize implementation faults, in other words, to find one and only one NFSM that represents the IUT. Therefore, if the diagnostic process ends up with multiple diagnoses, (or multiple Possible Implementations (PIs)), additional tests are needed to help reduce the number to one.

We propose a process to select additional test cases. Given two PIs, the process generates test cases which distinguish the PIs. After the application of the additional test cases to the implementation, new output sequences can be observed. We remove those PIs, whose corresponding expected output sequences are different from the new observed ones, from further consideration. Please note that both PIs that are used to generate the additional test case may have different expected output sequences from the new observed ones. In this case, the two PIs will be removed. We repeat the process until only one PI is left. Then the PI is the NFSM representing the implementation, and the corresponding diagnostic candidate, which contains differences between the PI and the specification, is presenting the real faults in the implementation.

Process:	Additional Test Cases Selection
Input:	Two Possible Implementations, PI_j and PI_k
Output:	a test case that can distinguish PI_j and PI_k

Given two PIs, PI_j and PI_k, we generate a breadth first behavior tree. Each node of the tree is labeled with two sets of states, $(s_{j,m}, s_{k,m})$, of PI_j and PI_k, respectively.

Each edge of the tree is labeled with a pair of input and output.

Initially, the first set, $s_{j,0}$ and second set $s_{k,0}$ of the root node n_0 contain the initial states of PI_j and PI_k. The labels of the edges from a node with sets of states, $(s_{j,m}, s_{k,m})$, are computed by:
$edges(s_{j,m}, s_{k,m}) =$
$\{(i/o) | (((s,i)(o,s')) \in Trans_j \wedge s \in s_{j,m}) \vee (((s,i)(o,s')) \in Trans_k \wedge s \in s_{k,m})\}$
where $Trans_x$ is the set of transitions of PI_x.

The input and output in a label of an edge are used to compute a new node. The label of a new node, n'_m, from a node n_m, which is labeled by $(s_{j,m}, s_{k,m})$, and an edge with label (i/o) is computed by:

$$s'_{j,m} = \{s' | ((s,i),(o,s')) \in Trans_j \wedge s \in s_{j,m}\}$$

$$s'_{k,m} = \{s' | ((s,i),(o,s')) \in Trans_k \wedge s \in s_{k,m}\}$$

$$\text{and } n'_m = (s'_{j,m}, s'_{k,m})$$

The tree is now specified. We have to explain how the behavior tree becomes finite. We achieve this by terminating the expansion of the tree in some nodes. A path is terminated if:

- The label of a new node (the combination of the new sets of states) already occurs in the tree.
- One of the sets in a new node is empty. In this case, an additional test case is found, which is an input sequence associated with the path from the root to this node.

Example: Suppose that, we are given two Possible Implementations shown in Figure 10. We generate the tree of Figure 11. We find the additional test case "a, a, b". The sets of expected output sequences of PI_1 and PI_2 are { "f, f, g"; "f, f, f"} and {"f, f, g"}, respectively. Hence the test case can distinguish the PIs.

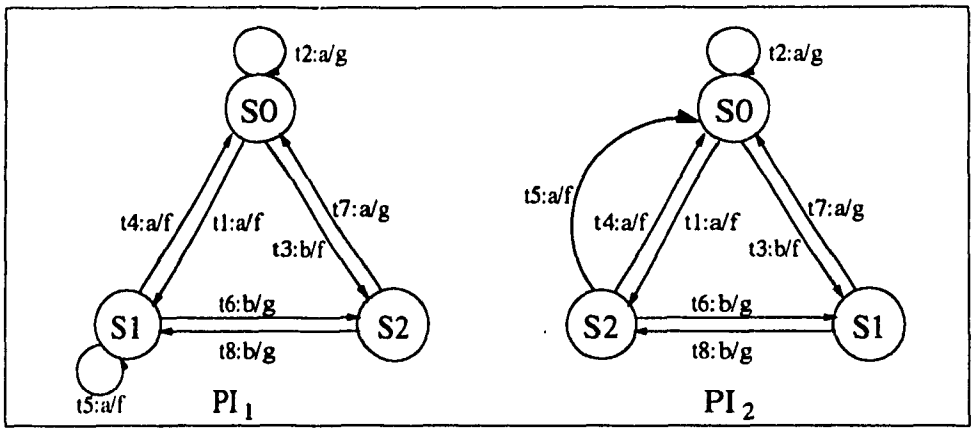


Figure 10: Two Possible Implementations

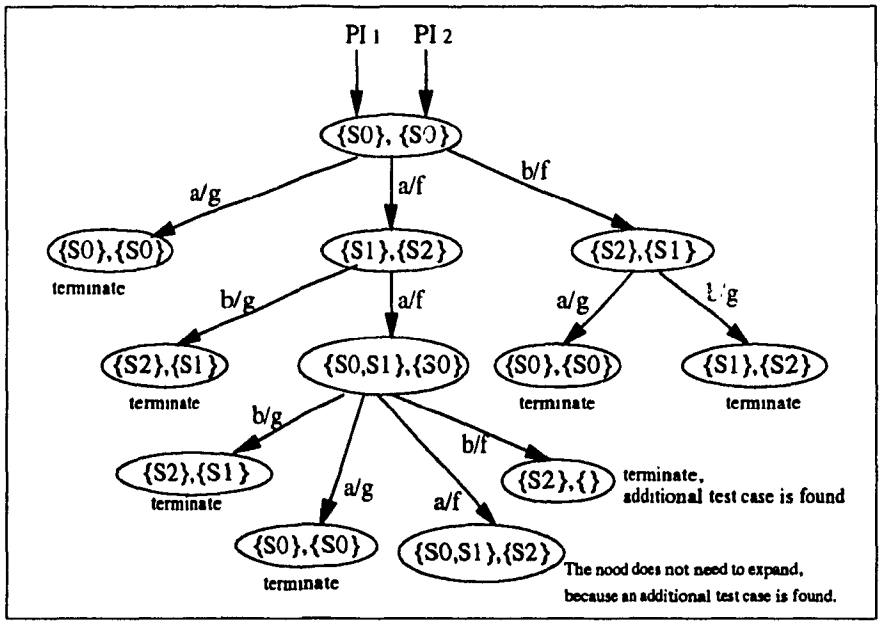


Figure 11: The breadth first behavior tree for the two Possible Implementations

The complexity approximation of Step 5

If the PIs generated in Step 4 are Observable NFSMs (ONFSMs), then the complexity of the Additional Test Case Section Process is small, because there is at most one state in each set of states in a node. The number of different nodes in the constructed tree is bounded by $\mathbf{O(n^2)}$, where $\mathbf{n}$ is the number of states in the specification machine. Since each node in the constructed tree is considered at most once by possible input/output pairs, the number of leaves in such a tree is bounded by $\mathbf{O(I \cdot O \cdot n^2)}$, where $\mathbf{I}$ and $\mathbf{O}$ are the numbers of different input and output symbols defined in the specification. Therefore, the overall complexity of the selection of the additional tests to distinguish between $\mathbf{N}$ PIs, is bounded by $\mathbf{O(N \cdot I \cdot O \cdot n^2)}$. From step 4, we know that $\mathbf{N}$, the number of PIs, is bounded by $\mathbf{O((Lc^S) \cdot K^F)}$, hence the complexity approximation of our algorithm is $\mathbf{O(Lc^S \cdot K^F \cdot I \cdot O \cdot n^2)}$. Nevertheless, we should notice that $\mathbf{N}$ is usually a very small number because, for a given test suite, there are only a small number of NFSMs, with the same number of states and transitions, that can produce the same outputs. Several experiments show that Step 3 and 4 take a large portion of the computation time, while Step 5 takes a lot less.

On the other hand, because we do not require the specification and implementation to be Non-Observable NFSMs, the PIs generated in Step 4 can be Non-Observable also, therefore we need to discuss the case when the PIs are Non-Observable. In general, the number of nodes in the tree grows exponentially as the numbers of states in Non-Observable NFSMs do. However, in practical applications, NFSM's are usually not very nondeterministic (i.e., there are only a few states each of which has more than one transition associated with the same input.); in this case, the number of nodes in the tree will not grow exponentially. In the given example, the PIs are Non-Observable NFSMs, but they are "Non-Observable" on only one state ($S1$ of PI_1 and $S2$ of PI_2). As we can see, such Non-Observable NFSMs will not introduce large complexity.

Chapter 5

An implementation of an NFSM diagnostic method

To verify the method described in Chapter 4, we have done some tests manually on small data sets. However, when a data set is too large, for example, an FSM with more than seven states, the number of TCsets and mutant machines becomes very large and manual verification becomes impossible. Therefore, we have written a program that is called NFSM-diags to verify the method.

Given three inputs: a specification, an implementation and a test suite, which are specified in files Specification.fsm, Implementation.fsm and Test_suite.tst, respectively, the program automatically applies test cases in Test_suite.tst to the specification and the implementation, and generates expected and observed output sequences. From the derived output sequences, it further generates a minimal set of faults existing in the implementation, if any. Besides the final result, which is a set of faults stored in the file faults.out, it also generates seven other files that contain intermediate values. The files are: specification.out; implementation.out; symptoms.out; fh.out; tc.out; diagnoses.out and additional_test.out.

We have applied NFSM-diags to some complicated NFSMs, as well as real life protocols, with different kinds of faults and various test cases obtained from test case

generation methods mentioned in Chapter 2. It shows that, as long as the assumptions are satisfied, NFSM-diags localizes all faults successfully.

We have applied the program to the alternating bit protocol and the NBS Transport Protocol (class 0) and (class 2) [Nati 83a]. We have changed outputs of all transitions and ending states of most transitions. We have found that, as long as the assumptions still hold, which means that all transfer fault transitions are directly reached by at least one test case, all faults are detected and localized. However, if there is one transition with fault that is fired by one test case but not directly reached by any test case, the program will generate no result, which indicates an assumption violation. We also have used NBS Transport Protocol (class 4) [Nati 83b] as an example of testing. It has 15 states, 27 inputs, 46 outputs and 62 specified transitions. We made the implementation with 6 transitions with output fault and 5 transitions with transfer fault; among the transitions, 3 of them have both output and transfer fault. We were using UIO test case to test the program. It has taken 18 hours for a dedicated UNIX machine to finish the job and all faults have been localized.

The difference between NFSM-diags and a real life diagnosing program is that NFSM-diags obtains observed output sequences by applying test cases to an FSM representing the IUT. If we modify this part to make NFSM-diags get the input of observed output sequences directly from other devices, for example, a file, then NFSM-diags can be used in real life to diagnose faults.

For people who are interested in using or further developing NFSM-diags, we have described the inputs, outputs and software design of it in the following sections.

5.1 Inputs and Outputs

A brief description of the input and output files is as follows.

- Specification.fsm and Implementation.fsm: These two input files specify two FSMs representing the specification and implementation to be diagnosed, respectively. The first line in the files specifies the initial state of an FSM, and each separate line following the first specifies a transition in the FSM. A transition is given as five strings: transition_id, starting_state, input_symbol, output_symbol and ending state. The strings are case sensitive, which means, for example, states "Closing" and "closing" are different. The machines must be completely specified, otherwise the program will not work.

- Test_suite.tst: This input file specifies test cases that will be applied to the specification and implementation to generate output sequences. Each line in the file, formed by a sequence of strings, represents a test case, where each string represents an input symbol.

- specification.out and implementation.out: These two output files contain two FSMs, which represent the specification and implementation respectively. After the program reads the input files Specification.fsm and Implementation.fsm, the output files will be generated. One can use these two files to check if there exists any typing mistake. Moreover, if there exist two identical transitions specified in the file Specification.fsm or Implementation.fsm, one of them will be eliminated. (Transitions with the same starting_state, input_symbol, output_symbol, ending state, and have the same or different transition_id will be treated as identical transitions.)

- symptoms.out: By comparing the expected and observed output sequences, the program will generate all symptoms. The symptoms and corresponding test cases will be stored in this file. If there is no symptom, then this file and the following files will not be generated and no-fault message will be given.

- fh.out: For each symptom, a set of Fault Hypotheses (SFH) will be generated and stored in this file.

- tc.out: A set of Tentative Candidates will be stored in the file.

- diagnoses.out: This file contains all diagnoses generated by the program.

- additional_test.out: If more than one Possible Implementation is found, additional test cases are needed to distinguish each each Possible Implementation. The additional test cases are stored in this file.

5.2 Object-Oriented Design of the Program

The method has been modeled and designed using **Object Modeling Technique (OMT)** [Rumb 91]. The design consists of three parts: **Object Modeling, Dynamic Modeling** and **Functional Modeling**. An **object model** captures the static structure of a system by showing the objects in the system, relationships between the objects, and the attributes and operations that characterize each class of objects. Those aspects of a system that are concerned with time and changes are the **dynamic model**. The **functional model** describes computations within a system. The following two sections present the object and functional model of our program. Because the dynamic aspects are well described in chapter 4, we will not present the dynamic model here. For those who are not familiar with OMT, the graphic notations can be found in Appendix B.

5.3 Object Model

Figure 12: Object Model: 1/7

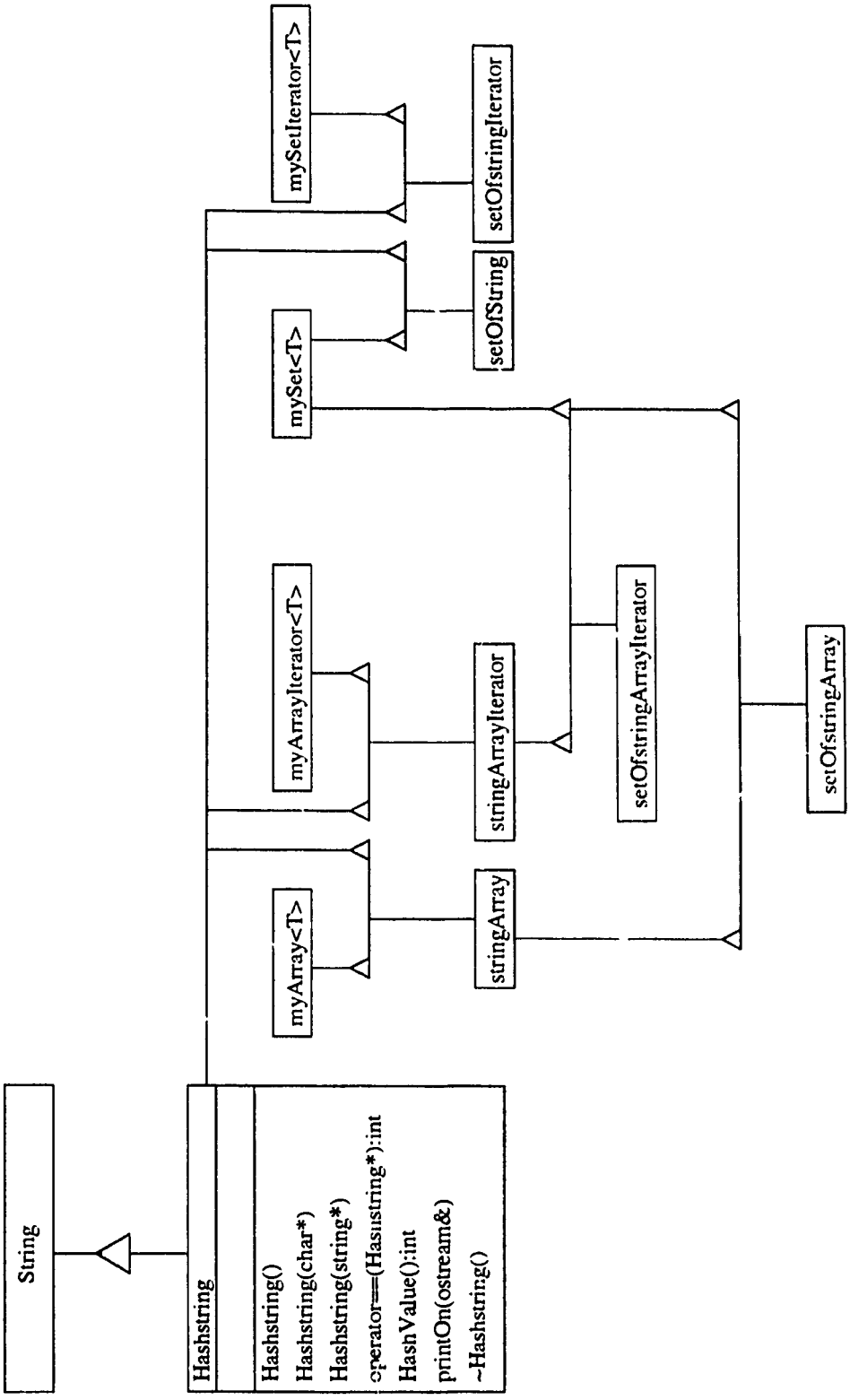


Figure 13: Object Model: 2/7

Figure 14: Object Model 3/7

Figure 15: Object Model: 4/7

Figure 16: Object Model: 5/7

Figure 17: Object Model: 6/7

Figure 18: Object Model: 7/7

5.4 Functional Model

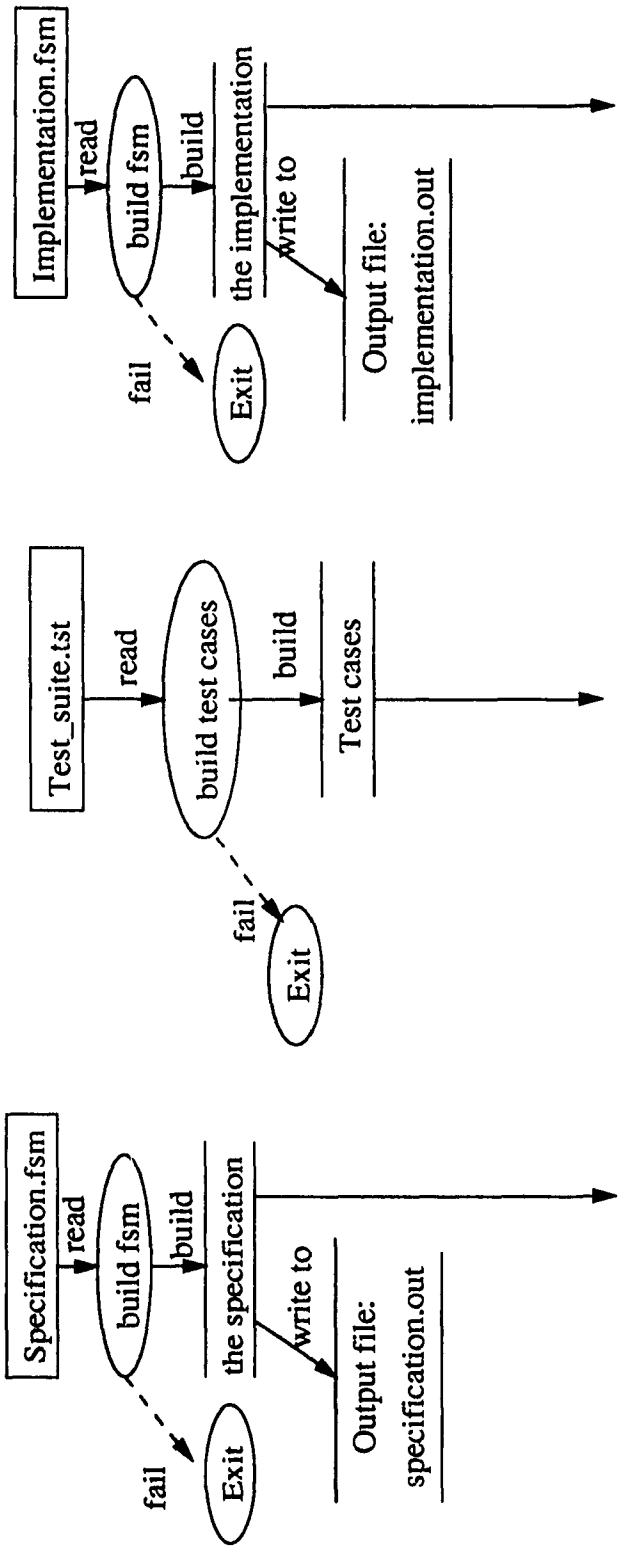


Figure 19: Functional Model: 1/4

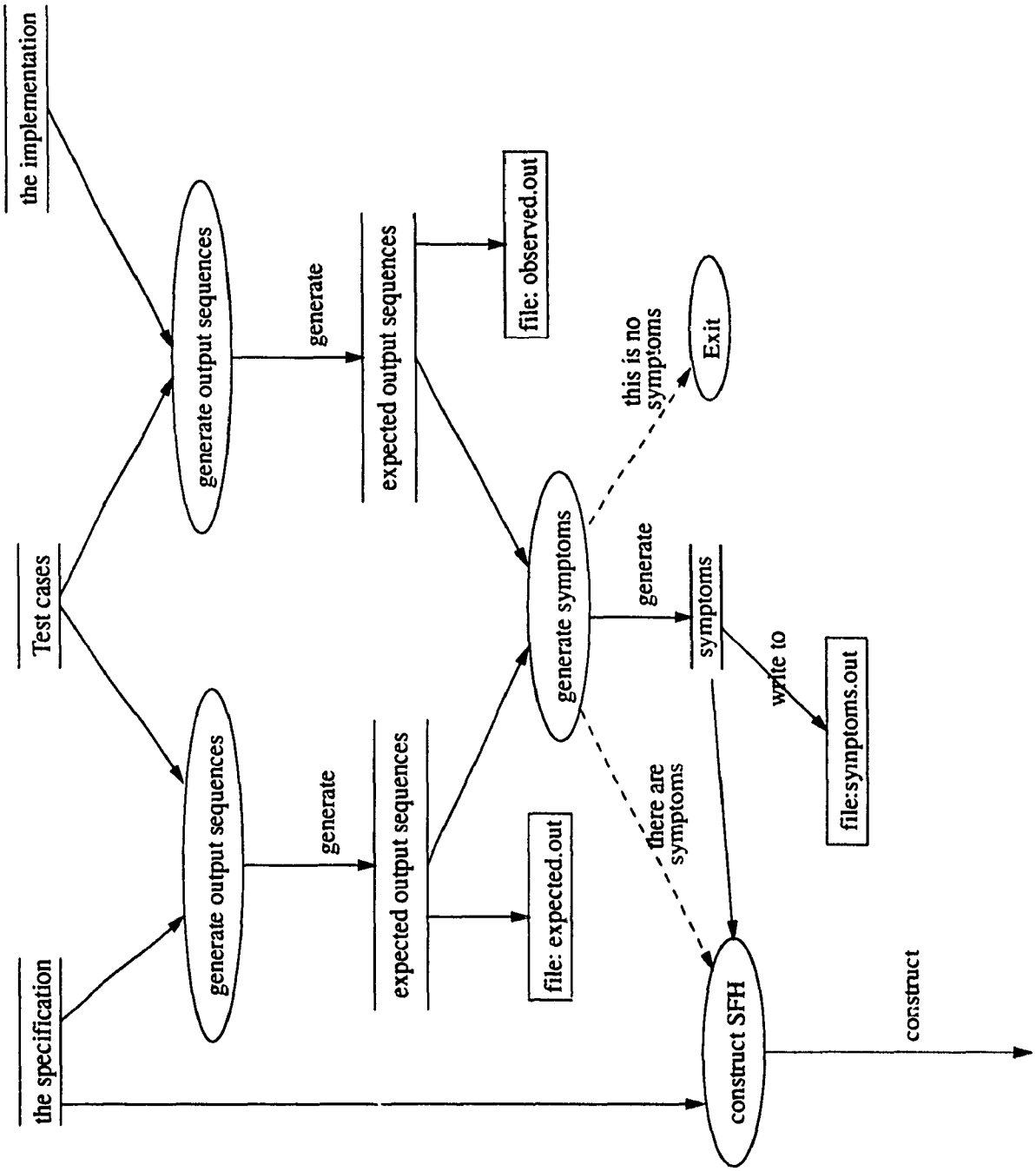


Figure 20: Functional Model: 2/4

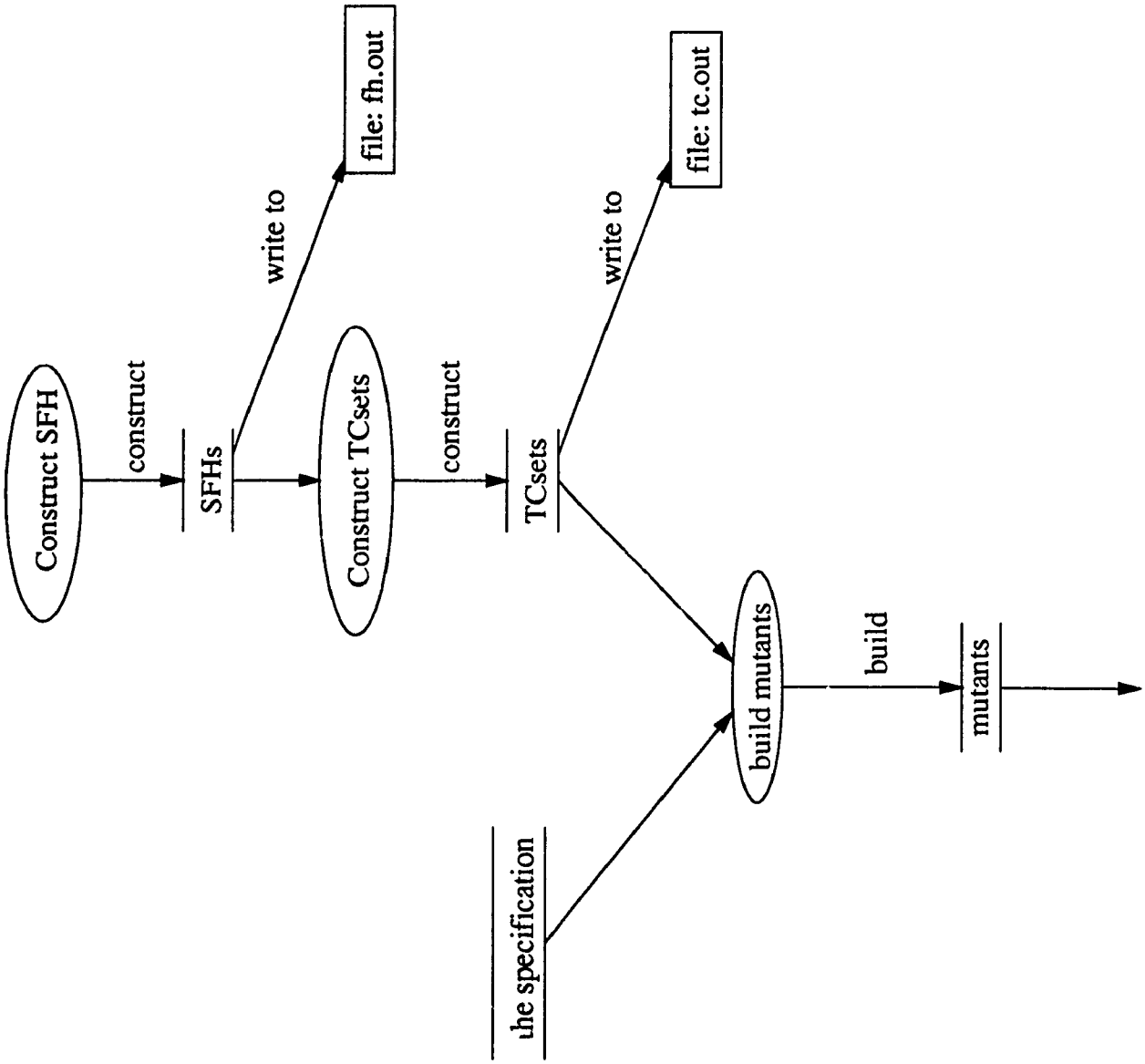


Figure 21: Functional Model: 3/4

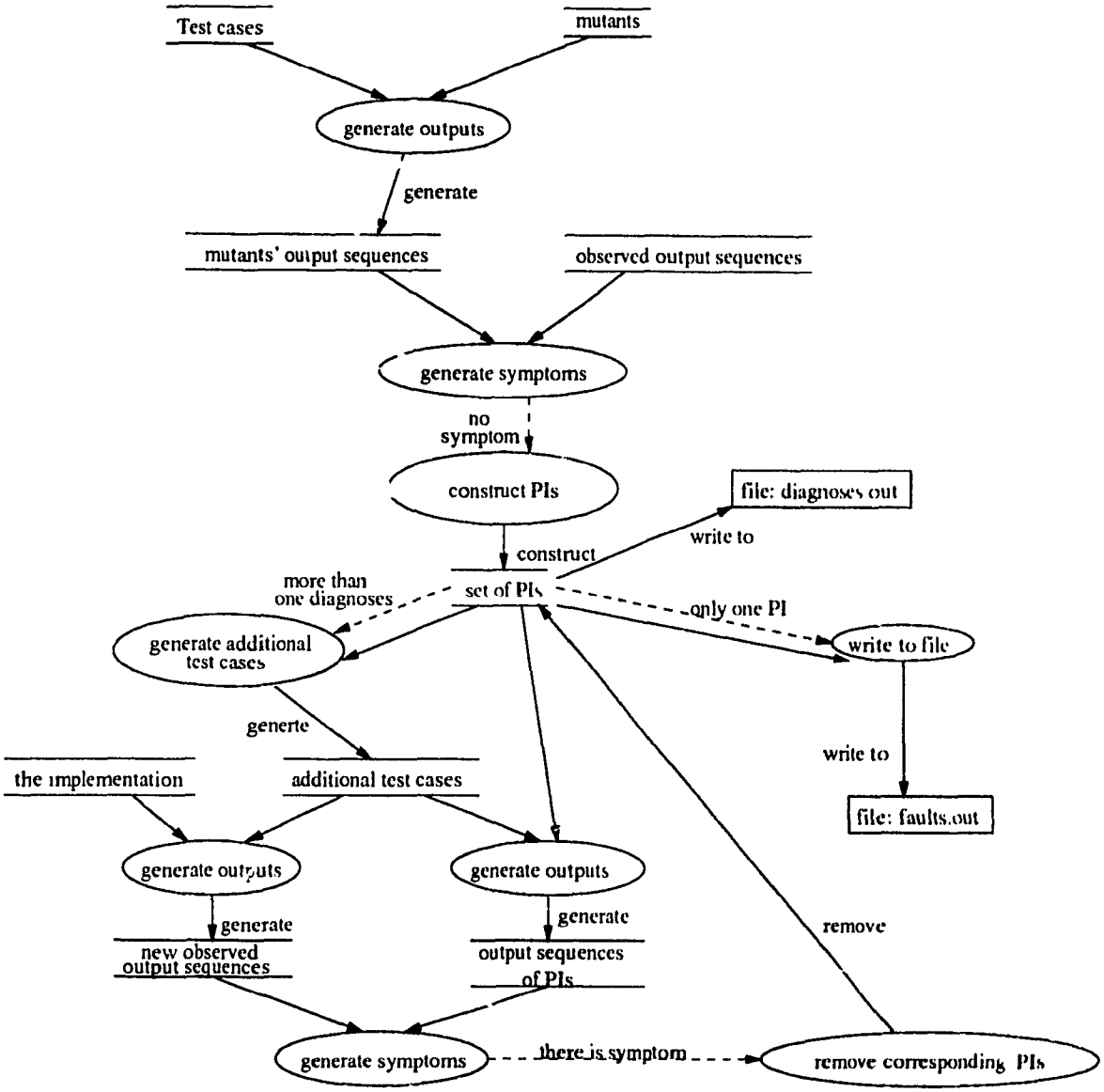


Figure 22: Functional Model: 4/4

5.5 Dictionary

NAME	TYPE	DESCRIPTION
myArray	Template	Implementation of an array
Add(T* a)	Member Function	Add a into the array
AddAt(T* a, int i)	Member Function	Add a into the array at position i
Detach(T* a)	Member Function	Remove a from the array
Detach(int i)	Member Function	Remove the member at position i
Find(T* a)	Member Function	Return the pointer of a member that is equal to a
Flush()	Member Function	Remove all members in the array
ForEach (void* Func, void* d)	Member Function	Apply Func to all members in the array, the function takes d as a list of parameters
GetItemIn Container()	Member Function	Return number of members in the array
HashValue()	Member Function	Return the hash value of the array
HasMember (T* a)	Member Function	Return 1 if a is found in the array, otherwise return 0
IsEmpty()	Member Function	Return 1 if the array is empty
myArray()	Member Function	Constructor
operator== (myArray &)	Member Function	Return 1 if equal, otherwise return 0
printOn (ostream & output)	Member Function	Print all members in the array to output
~myArray()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
myArrayIterator (T)	Template	Iterator for class myArray
Current()	Member Function	Return the member at current position
myArrayIterator()	Member Function	Constructor
operator++()	Member Function	Advance the iterator to the next position, and return the member at the next position
operator int()	Member Function	Return 0 if the end of the array is reached, otherwise return 1
Restart()	Member Function	Reset the iterator
~myArrayIterator()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
myDictionary <K, V \rangle	Template	Implementation of a Dictionary. A Dictionary contains pairs of *key* and *value*. While duplicates of values are allowed, duplicates of keys are not.
Add(K* k, V* v)	Member Function	A new key-value pair is added to the Dictionary
Detach(K* k)	Member Function	Remove a pair whose key is equal to k
Find(K* k)	Member Function	Return the pointer to a value whose key is equal to k
Flush()	Member Function	Remove all members in the Dictionary
GetItemIn Container()	Member Function	Return number of members in the Dictionary
HashValue()	Member Function	Return the hash value of the Dictionary
myDictionary()	Member Function	Constructor
printOn (ostream & output)	Member Function	Print all members in the Dictionary to output
~myDictionary()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
myDictionary Iterator \langle T \rangle	Template	Iterator for class myDictionary
Current()	Member Function	Return the member at current position
myDictionary Iterator()	Member Function	Constructor
operator++()	Member Function	Advance the iterator to the next position, and return the member at the next position
operator int()	Member Function	Return 0 if the end of the Dictionary is reached, otherwise return 1
Restart()	Member Function	Reset the iterator
~myDictionary Iterator()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
Hashstring	Class	A modified string class
Hashstring()	Member Function	Construct an empty Hashstring
Hashstring(char* str)	Member Function	Construct a Hashstring by copying str
Hashstring(string* str)	Member Function	Construct a Hashstring by copying str
HashValue	Member Function	Return hash value of the Hashstring
operator== (Hashstring* hstr)	Member Function	Return 1 if two Hashstrings are equal, otherwise return 0
printOn(ostream& output)	Member Function	Print the Hashstring on output
~Hashstring()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
mySet(T)	Template	Implementation of a Set
Add(T* a)	Member Function	Add a into the Set
Detach(T* a)	Member Function	Remove a from the Set
Find(T* a)	Member Function	Return the pointer of a member that is equal to a
Flush()	Member Function	Remove all members in the Set
ForEach (void* Func, void* d)	Member Function	Apply Func to all members in the Set, the function takes d as a list of parameters
GetItemIn Container()	Member Function	Return number of members in the Set
HashValue()	Member Function	Return the hash value of the Set
IsEmpty()	Member Function	Return 1 if the Set is empty, otherwise return 0
mySet()	Member Function	Constructor
operator== (mySet &)	Member Function	Return 1 if equal, otherwise return 0
printOn (ostream & output)	Member Function	Print all members in the Set to output
~mySet()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
mySetIterator \langle T \rangle	Template	Iterator for class mySet
Current()	Member Function	Return the member at current position
mySetIterator()	Member Function	Constructor
operator++()	Member Function	Advance the iterator to the next position, and return the member at the next position
operator int()	Member Function	Return 0 if the end of the Set is reached, otherwise return 1
Restart()	Member Function	Reset the iterator
~mySetIterator()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
stringArray	Class	Implementation of an Array of Hashstring
stringArrayIterator	Class	Iterator of stringArray
setOfString	Class	Implementation of a Set of Hashstring
setOfstringIterator	Class	Iterator of setOfString
setOfstringArray	Class	Implementation of a set of stringArray
setOfstringArrayIterator	Class	Iterator of setOfstringArray

NAME	TYPE	DESCRIPTION
transition	Class	Implementation of a transition
end_state()	Member Function	Return the ending state of the transition
end_state(Hash-string* state)	Member Function	Change the ending state to *state*
id()	Member Function	Return the id of the transition
input()	Member Function	Return the input of the transition
output()	Member Function	Return the output of the transition
HashValue()	Member Function	Return the hash value of the transition
operator==(transition& t)	Member Function	If everything (except transID) of two transitions is same, it returns 1. Otherwise it returns 0
printOn(ostream& ostr)	Member Function	Print the transition to ostr
start_state()	Member Function	Return the starting state of the transition
transition()	Member Function	Default constructor
transition(transition&)	Member Function	Copy constructor
transition(string* id, string* s, string* i, string* o, string* e)	Member Function	Construct a transition whose id=*id*, starting state = *s*, input = *i*, output = *o* and ending state = *e*
~transition()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
Outputs	Class	Record of an output sequence, and corresponding path and state sequences
HashValue()	Member Function	Return hash value
length()	Member Function	Return the length of the output sequence
operator==(Outputs& tester)	Member Function	If two Outputs are equal, return 1
Outputs()	Member Function	Default constructor
Outputs(Outputs & source)	Member Function	Copy constructor
printOn(ostream& ostr)	Member Function	Print the output sequence, corresponding path and state sequence to ostr
~Outputs()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
OutputList	Class	A list of Outputs (corresponding to a test case)
AddAtHead (Outputs* new)	Member Function	Add a new Outputs to the head of the list
Current()	Member Function	Return the Outputs at current position
DetachCurrent()	Member Function	Detach the Outputs at current position
gotoHead()	Member Function	Move the current position to the head
istail()	Member Function	Return 1 if the tail is reached
operator++(int)	Member Function	Advance the position to the next. It the current position is tail, then the position does not change
OutputList()	Member Function	Construct an empty list
printOn(ostream& ostr)	Member Function	Print the list on ostr
~OutputList()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
outputDict	Class	A Dictionary whose key is a test case and whose value is a list of Outputs (OutputList).

NAME	TYPE	DESCRIPTION
FSM	Class	A Finite State Machine class
buildUpIO(setOfstringArray& testSuite)	Member Function	Given a *testSuite*, the FSM generates output sequences. Test cases and corresponding sets of output sequences (implemented as *OutputLists*) are stored in an outputDict. The pointer of the outputDict is returned.
changeOutput(Hashstring* transID, Hashstring* output)	Member Function	Change the output of the transition with *transID* to *output*
changeNextState(Hashstring* transID, Hashstring* state)	Member Function	Change the NextState of the transition with *transID* to *state*
FSM()	Member Function	Construct an FSM without any state and transition
FSM(char * file)	Member Function	Read file *filename* and construct an FSM
FSM(FSM& source)	Member Function	Copy constructor
find(Hashstring* transID)	Member Function	Find a transition with *transID*
hasSuchState(Hashstring* input, Hashstring* output)	Member Function	Return a set of states containing all ending states of transitions with *input/output* behavior
printOn(ostream& ostr)	Member Function	Print information about the FSM to *ostr*
~FSM()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
symptom	Class	A class of symptoms. it carries information about the missing or additional output sequences
HashValue()	Member Function	Return hash value of the symptom
operator== (symptom&)	Member Function	Return 1 if two symptoms are equal, otherwise return 0
printOn(ostream& ostr)	Member Function	Print information about the symptom to *ostr*
symptom()	Member Function	Construct an empty symptom
symptom(stringArray* testCase, Outputs* output, char type)	Member Function	Construct a symptom, its output sequences are given by *output*, corresponding test case is given by *testCase* and type is given by *type* (type can be either "m" or "a")
symptomType()	Member Function	Return the type of the symptom
~symptom()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
symptomSet	Class	A Set of symptoms
symptomSetIterator	Class	The iterator of symptomSet

NAME	TYPE	DESCRIPTION
fault_candidate	Class	fault candidate
fault_candidate(string* transId, char type, string* output,...)	Member Function	Constructor
fault_candidate(fault_candidate&)	Member Function	Copy construct
HashValue()	Member Function	Return hash value of the fault candidate
operator==(fault_candidate& fc)	Member Function	If two fault candidates are same, return 1, otherwise return 0
printOn(ostream& ostr)	Member Function	Print information about the fault candidate to ostr
~fault_candidate()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
correct_candidate	Class	correct candidate
correct_candidate (string* transId, char type)	Member Function	Constructor
correct_candidate (correct_candidate&)	Member Function	Copy construct
HashValue()	Member Function	Return hash value of the correct candidate
operator== (correct_candidate& fc)	Member Function	If two correct candidates are same, return 1, otherwise return 0
printOn(ostream& ostr)	Member Function	Print information about the correct candidate to ostr
~correct_candidate()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
setOffault_candidate	Class	Set of fault candidates
setOffault_candidateIterator	Class	An iterator of setOffault_candidate
setCfTC	Class	Set of setOffault_candidate
setOfcorrect_candidate	Class	Set of correct candidate
setOfcorrect_candidateIterator	Class	An iterator of setOfcorrect_candidate

NAME	TYPE	DESCRIPTION
FH	Class	Fault hypothesis
add(correct_ _candidate* cc)	Member Function	Add a correct candidate to CC
add (fault_candidate* fc)	Member Function	Add a fault candidate to FC
conflict(FH*)	Member Function	Return 1 if two FCs are conflicting
FH()	Member Function	Default Constructor
FH(setOffault_ _candidate& fc, setOfcorrect_ _candidate& cc)	Member Function	Constructor
Flush()	Member Function	Remove all elements in the FC and CC
HashValue()	Member Function	Return hash value of the FC
myFCset()	Member Function	Return a pointer to FC set
operator== (FH&)	Member Function	Return 1 if two FCs are equal, otherwise return 0
printOn (ostream& ostr)	Member Function	Print FC
~FH()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
SFH	Class	Set of FHs
SFHIterator	Class	Iterator of SFH
symptomDictionary	Class	A Dictionary, whose key is symptom and value is SFH
symptomDictionaryIterator	Class	Iterator of symptomDictionary

NAME	TYPE	DESCRIPTION
symptom Generator	Class	A process to generate symptoms
no_symptom()	Member Function	Return 1 is no symptom is found, otherwise return 0
numOfSymptom()	Member Function	Return number of symptoms
SFHDict Generator()	Member Function	Generate an SFH for each symptom, and store it in SFHDict. The pointer of the SFHDict is returned
printOn (ostream& ostr)	Member Function	Print symptoms
symptomGenerator (OutputDict* S, OutputDict* M, setOfstringArray* TS)	Member Function	Given output sequences of the specification and implementation, as well as test suits (TS), a set of symptoms is generated.
~symptom() Generator	Member Function	Destructor

NAME	TYPE	DESCRIPTION
fault	Class	A fault with its specific output or ending state
fault(string* transID, char type, string* o_s)	Member Function	Construct a fault
fault(fault&)	Member Function	Copy constructor
print(ostream& ostr)	Member Function	Print the fault on ostr

NAME	TYPE	DESCRIPTION
TCset	Class	The class TCset is actually the step 3 of the method
TCset (symptomDictionary*)	Member Function	Given a symptomDictionary, a TCset is generated
pass()	Member Function	Return the pointer of the formed TCset
printOn(ostream& ostr)	Member Function	Print the TCset
~TCset()	Member Function	Destructor

NAME	TYPE	DESCRIPTION
step4	Class	The implementation of step 4 of the method
step4 (setOfTC& tcset, FSM& sp, Output-Dict& imp, setOf-stringArray& TS)	Member Function	Construct a set of diagnoses
numOfDiag()	Member Function	Return the number of diagnoses
Pass()	Member Function	Return the pointer of set of diagnoses
printOn(ostream& ostr)	Member Function	Print the diagnoses on ostr
~step4	Member Function	Destructor

NAME	TYPE	DESCRIPTION
step5	Class	The implementation of step 5 of the method
step5(FSM* imp, setOf_setOffault* DiagSet)	Member Function	Given the implementation and a set of diagnoses, the step will find one and only one diagnosis which contains real faults
writeOut (ostream*)	Member Function	Write out the additional test cases
~step5	Member Function	Destructor

Chapter 6

Conclusions and Discussion

In this research, we have proposed a diagnostic method for the case where system implementations, represented by NFSMs (including DFSMs), are allowed to have multiple faults. If existing faults are identified by test cases in a test suite, this algorithm detects and localizes these faulty transitions. It generates a diagnosis, which is a set of faulty transitions with specific outputs and/or ending states. One can correct faults in the implementation according to the information contained in the diagnosis.

The algorithm extends work by Ghedamsi, which treats the cases of single faults in an NFSM [Ghad92a], and multiple faults in a DFSM [Ghad92b]. Since it is not feasible to manually verify the correct operation of the algorithm for realistic examples, the algorithm has been implemented in a program that compares expected and observed output sequences and produces diagnoses. This program was applied to alternating bit protocol, and to NBS Transport Protocol class 0, 2 and 4.

As mentioned before, several conformance relations have been used in NFSM conformance testing. Our method can be applied directly to the implementations which conform to the Trace-Equivalence [Luo 94a] and Quasi-Equivalence [Luo 94b] relation. With slight modification, the method can also apply to the implementations that conform to the so called Reduction relations [Petr 94a]. The Reduction relation

requires that the implementation produces a (sub)set of output sequences that can be produced by the specification in response to every input sequence. In this case, we only require the disabling of the process of "generation of SFH for a missing output sequence" in Step 2 of the method.

In order to localize all faults in implementations, we assume that "there is at least one test case in the applied test suite that identifies each fault in the IUT". We define an identified fault as one that is detected and directly reached by a test case. Whether or not faults are detectable mainly depends on the power of test cases and it is not our main concern. Nevertheless, if transitions with faults in an implementation are not reached directly by any test case in a given test suite, full fault localization can not be ensured by our method. There are the following two cases:

Case 1: If the faulty transitions are not executed, or the faults are not detected by any test case, then other faults (those that are identified) can be localized. After the localized faults in the implementation are fixed and the test suite is again applied to the implementation, the remaining faults may be directly reached and detected, hence they may be localized.

Case 2: If the faulty transitions are executed and the faults are detected by one of the test cases, but not **reached directly by the test cases,** then the approach will result in no solution (no Possible Implementation can be found). In this case, other diagnostic methods may be needed.

Many questions are left for future work. An interesting research would be the extension of our work to the diagnostics of machines not respecting the assumption discussed above. Our diagnostic method has the other assumption that the IUT satisfies the complete-testing property. This assumption is usually difficult to hold in practice. Therefore, dropping such a condition will make the method more practical in diagnosing non-deterministic systems.

Appendix A

Transformation to Obtain ONFSMs

In [Luo 94b], an algorithm to construct a trace-equivalent ONFSM for an arbitrary NFSM is presented.

Algorithm:	Constructing a trace-equivalent ONFSM for a given NFSM.
Input:	an NFSM S.
Output:	an ONFSM S'.

Step 1: Build a graph G consisting initially a single unmarked node, labeled M_0. where $M_0 = \{S_0\}$.

Step 2: If there is no unmarked node in the graph G, stop; G is the ONFSM S' and the node M_0 represents the initial state of S'. Otherwise:

i find and mark an unmarked node M in G, where the label M is a set of states of S.

ii for every input/output pair i/o,

a) $M' = \{Q | P \in M$ & there is a transition: $P - i/o- > Q\}$,

b) if M' is not a node label in the resulting graph G, then create a node with label M',

c) create a directed edge from M to M' with label i/o.

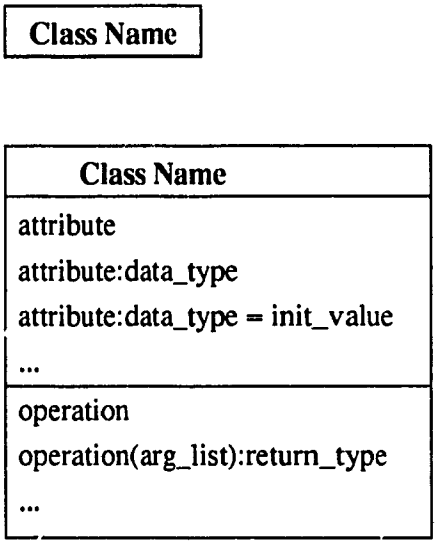
iii GOTO Step 2. □

Theoretically, the number of states in an ONFSM grows exponentially as the number of states in the original NFSM does. However, in practical applications, NFSMs are usually not very nondeterministic, hence the number of derived states will not grow exponentially.

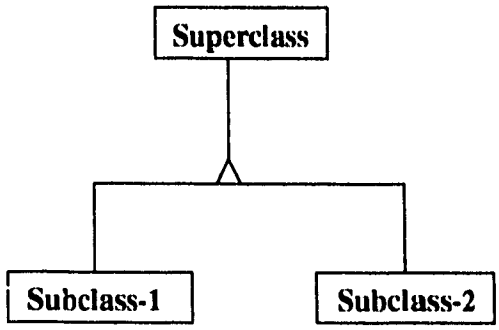
Appendix B

Object-Oriented Design Notation

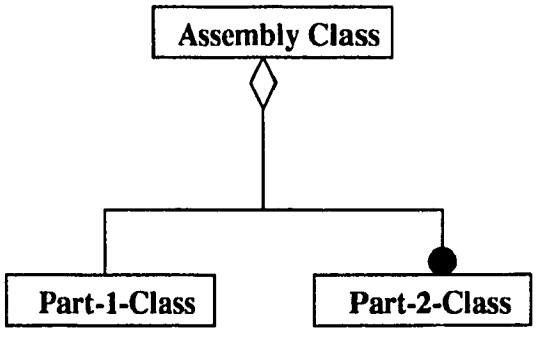
Class:



Generalization(Inheritance):



Aggregation:



Object Instances:

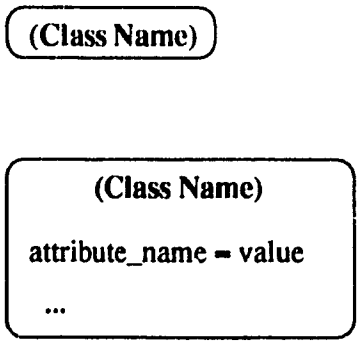
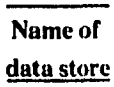


Figure 23: Object Model Notation

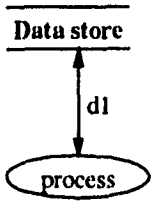
Process

Data Store of File Object

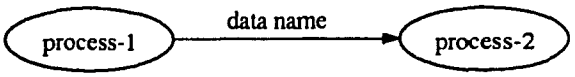


Actor Objects (as Source or Sink of Data):

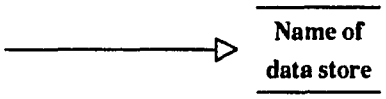
Access and Update of Data Store Value:



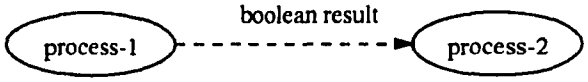
Data Flow between Processes:



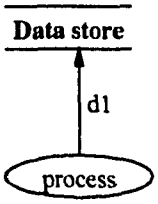
Data Flow that Results in a Data Store:



Control Flow:



Update of Data Store Value:



Access of Data Store Value:

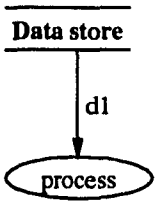


Figure 24: Functional Model Notation

Bibliography

[Beli 89] F. Belina et al., "The CCITT specification and description language SDL", Computer Networks and ISDN Systems, Vol. 16, pp. 311-341, 1989.

[Boch 91] G.v. Bochmann, R. Dssouli, A. Das, M. Dubuc, A. Ghedamsi and G. Luo, "Fault models in testing", Invited paper in 4-th International Workshop on Protocol Test Systems, Leidschendam, Holland, 15-17 Oct. 1991, (invited paper), pp. II.17-II.32.

[Boch 94] G. v. Bochmann and A. Petrenko "Protocol Testing: Review of Methods and Relevance for Software Testing" Dep. Publ. #923, University de Montreal, 1994.

[Bolo 87] T. Bolognesi, and E. Brinksma, "Introduction to the ISO Specification Language LOTOS", Computer Networks and ISDN Systems, 14, pp. 25-59, 1987.

[Budk 87] S. Budkowski et al.,"An introduction to Estelle: a specification language for distributed systems", Computer Networks and ISDN Systems, Vol. 14, No.1, pp. 3-23, 1987.

[Chow 78] T.s. Chow, "Testing software design modelled by finite state machines", IEEE Trans. S.E. 4,3, 1978.

[Fuji 91a] S. Fujiwara and G.v. Bochmann, "Testing Nondeterministic Finite State Machine with Fault Coverage", IFIP Transactions, Protocol Testing Systems IV, Ed. by Jan Kroon, Rudolf J. Heijink and Ed Brinksma, 1992, North-Holland, pp.267-280.

[Fuji 91b] S. Fujiwara, G.v. Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi, "Test Selection Based on Finite State Models" IEEE Transactions on Software Engineering, Vol. 17, No.6, June 1991.

[Ghed 92a] A. Ghedamsi, R. Dssouli, G.v. Bochmann "Diagnostic Tests for Single Transition Faults in Non-Deterministic Finite State Machines", IFIP Trans., Protocol Test Syst., V, Proc. 5th Int. Workshop Protocol Testing Syst., Montreal, Sept. 28-30, 1993, pp.105-116.

[Ghed 92b] A. Ghedamsi, G.V. Bochmann and R.Dssouli, "Multiple Fault Diagnostics for finite state machines", IEEE INFOCOM'93, San Francisco, USA, March, 1992.

[Ghed 92c] A. Ghedamsi, "Test selection and diagnostic methods", Ph.D Thesis, DIRO, Univ. of Montreal, 1993.

[Klee 87] J.de Kleer, and B.c. Williams, "Diagnosing multiple faults", Artificial Intelligence 32(1), 1987, pp. 97-130.

[Kloo 93] H. Kloosterman, "Test derivation from nondeterministic finite-state machines", IFIP Trans., Protocol Test Syst., V, Proc. 5th Int. Workshop Protocol Testing Syst., Montreal, Sept. 28-30, 1993, pp.297-308.

[Ko 90] K.C. Ko, "Protocol test sequence generation and analysis using AI techniques", Master thesis, Dept of Comp. Sci. UBC, JUI. 1990.

[Koha 78] Z. Kohavi, Switching and Finite Automata Theory. New York: MaCraw-Hill, 1978.

[Kuan 62] M. K. Kuan, Graph'c programming using odd or even points, Chinese Math, Vol.1, pp273-277, 1962.

[Luo 92] G. Luo, A. Petrenko and G. v. Bochmann, "Selecting Test Sequences for Partially-Specified Nondeterministic Finite State Machines", Publication #864 of D.I.R.O, University of Montreal.

[Luo 94a] G. Luo, G. v. Bochmann and A. Petrenko, "Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method" IEEE Trans. on S. E., Feb. 1994.

[Luo 91b] G. Luo. A. Petrenko and G. v. Bochmann. "Selecting test Sequences for Partially-Specified Nondeterministic Finite State Machines" International Workshop on Protocol Test Systems, VII, pp.91-106.

[Nait 81] S. Naito and M. Tsunoyama, "Fault Detection for Sequential Machines by Transition-Tours", Proc. of FTCS (Fault Tolerant Computing Systems), pp.238-243, 1981.

[Nati 83a] National Bureau Standards, "Specification of a transport protocol for computer communications, Volume 1: Overview and services", Washington, DC, Rep. ICST/HLNP-83-1, Jan. 1983.

[Nati 83b] National Bureau Standards, "Specification of a transport protocol for computer communications, Volume 3: Class 4 protocol", Washington, DC, Rep. ICST/HLNP-83-3, Jan. 1983.

[Petr 94] A. Petrenko, N. Yevtushenko and G.v. Bochmann, " Experiments on Nondeterministic Systems for the Reduction Relation".

[Rumb 91] J. Rumbaugh ... [et al.] Object-oriented modeling and design. New Jersey: Prentice-Hall, 1991.

[Sabn 88] K. Sabnani and A.T. Dahbura, "A protocol Test Generation Procedure", Computer Networks and ISDN Systems, Vol. 15, No. 4, pp. 285-297, 1988.

[Sari 87] B. Sarikaya, G.v. Bochmann and E. Cerny, "A Test Design Methodology for Protocol Testing", IEEE Trans. on SE, April 1987. pp.518-531.

[SDL 91] Newsletter, Dec. 1991.

[Star 72] P.H. Starke, Abstract Automata, North-Holland/American Elsevier, 1972, p.419.

[Trip 93] P. Tripathy and K. Naik, "Generation of adaptive test cases from nondeterminstic finite-state models", IFIP Trans., Protocol Test Syst. V, Proc. 5th Int, Workshop Protocol Testing Syst., G.v. Bochmann, R. Dssouli, and A. Das, eds. North-Holland, 1993, pp. 309-320.

[Ural 87] H. Ural, "A test Derivation Method for Protocol Conformance Testing", Proc. of the 7th IFIP Symposium on Protocol Specification, Testing and Verification, Zurich, May 5-8 1987.

[Uyar 86] M. U. Uyar and A. T. Dahbura, "Optimal test sequence generation for protocol: the Chinese postman algorithm applied to Q.931", Proc IEEE Global Telecommunication Conference, 1986.

[Vuon 89] S.T. Vuong, W. Chan and M.R. Ito, "The UIOv-Method for protocol test sequence generation", in the 2-nd International Workshop on Protocol Test Systems, Berlin, Germany, Oct. 3-6, 1989.

[Vuon 90] S.T. Vuong and K.C. Ko, "A novel approach to protocol test sequence generation", IEEE Global telecomm. conference and exhibition, San Diego, California, Dec. 2-5, vol. no. 3, 904.1.1-904.1.5, 1990.