uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Zhi Xu

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M. (Computer Science)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Passive Fault Detection for FSM and EFSM models

TITRE DE LA THÈSE / TITLE OF THESIS

Hasan Ural

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Jean-Pierre Corriveau

Robert Probert

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Passive Fault Detection for FSM and EFSM models

Zhi Xu

Thesis submitted to the
Faculty of Graduate and Post Doctoral Studies
In partial fulfillment of the requirements
For the Masters in Computer Science Degree*

School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa
© Zhi Xu, Ottawa, Canada, January, 2007

----

* The Masters program in Computer Science is a joint program with Carleton University, administered by the Ottawa-Carleton Institute for Computer Science

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-49300-7
Our file Notre référence
ISBN: 978-0-494-49300-7

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Abstract

Passive fault detection is a fundamental part of passive testing which determines whether a system under test (SUT) is faulty by observing the input/output behaviors of the SUT without interfering with its normal operations.

This thesis focuses on passive fault detection for SUTs whose specifications are given in finite state machine (FSM) and extended finite state machine (EFSM) models, and proposes a new approach to FSM and EFSM-based passive fault detection.

Compared with previous approaches for FSM-based passive fault detection based on the approach in [Lee97], this proposed approach for FSM-based passive fault detection has better performance with respect to computational complexity and provides more information about possible starting state and possible trace during the passive fault detection without the need for additional post-processing. We analyzed the worst case time complexity of the algorithms and gave results of experiments to estimate the average case time complexity of these algorithms.

This proposed approach for EFSM-based passive fault detection is derived from the proposed approach for FSM-based passive fault detection. It also provides information about possible starting state and possible trace at the end of passive fault detection; and utilizes a Hybrid method which combines the use of both Interval Refinement and Simplex methods for performance improvement during passive fault detection. Through experiments, we show that, compared with using only the Interval Refinement method or only the Simplex method, the Hybrid method guarantees the correctness of results with a reasonable time cost.

Acknowledgements

I would like to thank the many people who helped and supported me during my master's study at University of Ottawa.

First of all, I would like to thank my advisor, Prof. Hasan Ural, for introducing me to FSM and EFSM based testing. This thesis would not have happened without his support, guidance, patience, and penetrating insight.

I would also like to thank Jichao Zhang for helpful discussions and Dr. Fan Zhang for the help on the initial work.

I would like to acknowledge the generous financial support of Ontario Centres of Excellence, the Natural Sciences and Engineering Research Council of Canada and the University of Ottawa.

Finally, thanks to my parents and friends for their love and support through challenging times.

Table of Contents

1. Introduction 1

1.1 Passive Fault Detection 1

1.2 The Proposed Approach for FSM and EFSM-based Passive Fault Detection 3

1.3 Organization of This Thesis 4

2. Preliminaries 5

2.1 FSM-based Passive Fault Detection 5

2.2 EFSM-based Passive Fault Detection 7

3. Previous Work 16

3.1 The Approaches for FSM-based Passive Fault Detection 16

3.2 The Approaches for EFSM-based Passive Fault Detection 17

3.3 Other Related Research Directions 18

3.4 Summary 19

4. The Proposed Approach for FSM-based Passive Fault Detection ... 20

4.1 The Approach in [Lee97] 20

4.2 The Proposed Approach 22

4.3 Comparison of the Algorithms 30

4.4 Assertions 32

4.5 Experimental Evaluation 33

4.5.1 Experiment Setup 33

4.5.2 Results of the Experiment 34

4.6 Conclusions.. 37

5. The Proposed Approach for EFSM-based Passive Fault Detection. 39

5.1 The Proposed Approach.. 40

5.1.1 Algorithm Main.. 41

5.1.2 Algorithm Search_Trace_Tree.. 41

5.1.3 Algorithm Check_Trace.. 42

5.1.4 Implementation of the Function *action*.............................. 46

5.1.5 Optimization on Constraints... 49

5.2 Function *evaluate* with the Interval Refinement Method................. 51

5.2.1 Definitions.. 52

5.2.2 Algorithm Interval_Refinement.. 54

5.2.3 Experiments... 56

5.2.3.1 Experiment Setup... 56

5.2.3.2 Results of the Experiment.. 58

5.2.4 Dependency Problem... 59

5.3 Function *evaluate* with the Simplex Method............................. 60

5.3.1 Algorithm Simplex.. 62

5.3.2 Experiments and Comparisons... 63

5.4 The Hybrid Method... 65

5.4.1 New Algorithm Check_Trace... 66

5.4.2 Experiments and Comparisons... 68

5.5 Conclusions.. 71

6. Summary and Conclusions.. 72

List of Figures

Figure 2.1 An FSM *M* 6

Figure 2.2 An EFSM *M* 9

Figure 2.3 An SEFSM *M′* 9

Figure 2.4 An EEFSM *M″* 10

Figure 2.5 The structure of a transition *t* in an SEFSM 10

Figure 2.6 A configuration *c* 14

Figure 2.7 A Trace-Tree *Tree* and a *compatible trace* of *E* 15

Figure 5.1 The architecture of a Trace-Tree and its representation 44

Figure 5.2 The SEFSM *ATM* for an ATM system 57

Figure 5.3 The results of Case I by applying the Interval Refinement (IR) method 59

Figure 5.4 Solution of the CSP by the Simplex method 61

Figure 5.5 The results of Case I by applying the Simplex method and the Interval Refinement (IR) method 64

Figure 5.6 The results of Case I by applying the Hybrid method, the Interval Refinement (IR) method, and the Simplex method 69

Figure 5.7 Rates of time cost with three methods 70

List of Tables

Table 4.1 Average complexity analysis.. 29

Table 4.2 Computational complexities of three algorithms................. 31

Table 4.3 Experimental comparisons of four algorithms..................... 36

Table 5.1 Definitions of basic arithmetic operations [Han04]............... 52

Table 5.2 The results of Case I by the Interval Refinement method......... 58

Table 5.3 The results of Case II by the Interval Refinement method....... 58

Table 5.4 The results of Case I by the Simplex method....................... 64

Table 5.5 The results of Case II by the Simplex method..................... 64

Table 5.6 The results of Case I by the Hybrid method........................ 69

Table 5.7 The results of Case II by the Hybrid method...................... 69

Chapter 1

Introduction

1.1 Passive Fault Detection

Passive fault detection is a fundamental part of passive testing which determines whether a system under test (SUT) is faulty by observing the input/output (I/O) behaviors of the SUT without interfering with its normal operations [Lee02]. Compared with active fault detection, in which a tester has complete control over the inputs and devises a test sequence to reveal possible faults of the SUT, passive fault detection is more applicable under circumstances where such control is impractical or impossible, such as network fault management [Lee02].

This thesis focuses on passive fault detection for SUTs whose specifications are given in finite state machine (FSM) and extended finite state machine (EFSM) models, and proposes a new approach to FSM and EFSM-based passive fault detection where the specification of an SUT N is modeled as an FSM or EFSM M, N is treated as a blackbox, and the tester wishes to determine whether N is faulty with respect to M by observing the I/O behaviors of N represented as a sequence Q of I/O pairs in FSM or a sequence E of observed I/O events in EFSM.. Within this context, the *passive fault detection problem* can be defined as follows: given an M and the sequence of I/O behaviors of N where the starting state (when the sequence of I/O behaviors starts) of N is unknown, determine whether N is faulty with respect to M. Such a decision can be based on the number of states in M that are compatible with the given sequence of I/O behaviors of N. A state s of M is *compatible* with a sequence of I/O behaviors if the sequence of I/O behaviors is a

trace of M starting at s. If the number of states compatible with the sequence of I/O behaviors is zero, then that sequence of I/O behaviors is sufficient to determine that N is faulty with respect to M; otherwise, the sequence of I/O behaviors is declared to be insufficient to determine whether N is faulty. That is, there are one or more states that are compatible with the sequence of I/O behaviors and the sequence of I/O behaviors needs to be augmented by additional I/O behaviors of N to continue with the passive fault detection process.

In the literature, most approaches for FSM and EFSM-based passive fault detection are derived from the approach for FSM-based passive fault detection in [Lee97], such as [Tab99] and [Lee02]. The approach in [Lee97] can be summarized as follows: suppose that the starting state of N is any state of M, check the observed sequence Q of I/O pairs one-by-one from the beginning, and reduce the size of possible current states in M by eliminating impossible states until either no state is possible (N is faulty) or there is at least one state possible. An extension of this approach to EFSM-based passive fault detection was proposed in [Lee02] where the set of possible current states was replaced by the possible current Candidate Configuration Set (CCS). A review of previous work on passive fault detection is given in Chapter 3.

The approach for FSM-based passive fault detection in [Lee97] is comprehensive but not efficient enough. In this approach, every state of M needs to be checked. However, the number of states compatible with Q is usually comparatively small and checking every state of M would be unnecessary. Further, this approach only determines the set of possible current states when it terminates. Post-processing will be needed to retrieve the information about possible starting state and possible trace corresponding to Q (a

backward tracking approach for post-processing was introduced in [Alc04]). The approaches derived from [Lee97], such as the approach for EFSM-based passive fault detection in [Lee02], inherit these shortcomings.

1.2 The Proposed Approach for FSM and EFSM-based Passive Fault Detection

This thesis proposes a new approach for FSM and EFSM-based passive fault detection whose main features are as follows: assume that the subset S_0 ($S_0 \subseteq S$) of states of M contains all possible starting states of the sequence of I/O behaviors. Randomly pick a state s in S_0 and check whether the sequence of I/O behaviors is a trace of M at s; if s is compatible with the sequence of I/O behaviors, then stop and declare that the sequence of I/O behaviors is not sufficient to determine whether N is faulty. In this case, the sequence of I/O behaviors will be a trace of N starting from s and the current state of N can be determined readily. Otherwise, continue to check other states in S_0. After checking all states in S_0, if no state is found to be compatible with the sequence of I/O behaviors, then N will be declared faulty. S_0 is determined by the tester according to the specific application at hand (unless it is explicitly specified in M). In this thesis, S_0 is chosen equal to S when we conduct experiments for comparing the proposed approaches with existing approaches in an effort not to put the existing approaches at a disadvantage.

Compared with the previous approach in [Lee97], our proposed approach for FSM-based passive fault detection has better performance in computational complexity. That is, the proposed approach performs better in situations where there is only one state in S_0 that is compatible with Q and performs the same in situations where there is no state in S_0 that is compatible with Q. Moreover, the proposed approach provides more

information about possible starting state and possible trace corresponding to Q during passive fault detection.

The proposed approach for EFSM-based passive fault detection inherits the merits of the proposed approach for FSM-based passive fault detection. In addition, it addresses the fault detection with respect to the data portion and introduces a Hybrid method that combines the advantages of two existing methods, namely Interval Refinement and Simplex methods, in the implementation.

1.3 Organization of This Thesis

The rest of the thesis is organized as follows. Chapter 2 gives preliminaries needed for our discussions, including definitions and notations used in the proposed approach for FSM and EFSM-based passive fault detection. Chapter 3 reviews the existing approaches in the literature. Chapter 4 and Chapter 5 present the proposed approach for FSM and EFSM-based passive fault detection, respectively, in detail with both theoretical and experimental evaluations. Chapter 6 concludes this thesis with some final remarks, summary of contributions and directions for future research.

Chapter 2

Preliminaries

2.1 FSM-based Passive Fault Detection

The input to FSM-based passive fault detection consists of the specification of SUT N given as an FSM M and the sequence of I/O behaviors a tester actually observes during the execution of N given as a sequence Q of I/O pairs.

Definition 1:

A *Finite State Machine* (FSM) M is a quintuple

$(S, X, Y, \delta, \lambda)$

where $S = \{s_1, s_2, ..., s_n\}$ is a finite set of states with $n = |S|$ and $s_1 \in S$ as the *initial state*, X is a finite set of inputs, Y is a finite set of outputs, δ is a state transition function that maps $S \times X$ to S, and λ is an output function that maps $S \times X$ to Y. These two functions are extended to the input sequences $I \in X^*$ in the usual manner. M is *minimal* if, for any two different states s_i, $s_j \in S$, there is an input sequence $I \in X^*$ such that the respective resulting output sequences are different, i.e., $\lambda(s_i, I) \neq \lambda(s_j, I)$. M is *deterministic*, if for each input $x \in X$, there is at most one transition defined at each state of M. □

An FSM M can be represented by a (directed) graph $G = (Vertices, Edges)$ (Figure 2.1) where a set of vertices $Vertices = \{v_1, v_2, ..., v_n\}$ represents the set of states of M and a set of directed edges $Edges = \{(v_j, v_k; x/y): v_j, v_k \in Vertices\}$ represents all specified transitions of M. More specifically, each edge $edge = (v_j, v_k; x/y)$ represents a state

transition from state s_j to s_k with input $x \in X$ and output $y \in Y$, and the I/O pair x/y is the *label* of *edge*. The label of a path $edge_1 edge_2 \ldots edge_r$, $edge_i \in Edges$, $1 \leq i \leq r$, is the concatenation of the labels of $edge_i$ and is called an I/O sequence I/O. An I/O sequence $I/\lambda(s_i, I)$ is called a *trace of M at* state s_i.

In FSM-based passive fault detection, we assume that both M and its implementation N are deterministic FSMs, a sequence Q of I/O pairs = I/O is observed from N, and a set of possible starting states S_0 of M is given. We wish to determine whether there is no state s in S_0 such that Q is a trace of M at s. In other words, there is no start state in M from which Q is a valid I/O sequence. This would show that N is not a conforming implementation of M.

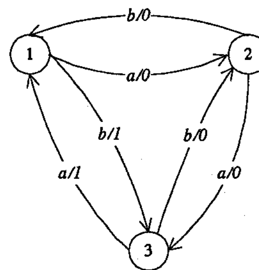

Figure 2.1 An FSM *M*

For example, suppose that M is as in Figure 2.1, Q_1 = "(a/0)(b/0)(a/0)(b/0)(a/0)(b/0)(b/1)" = "abababb/0000001" and $S_0 = \{s_1, s_2\}$. Since "0000001" = $\lambda(s_1$, abababb), Q_1 is a trace of M at s_1. Let Q_2 = "(a/0)(b/0)(a/0)(b/0)(a/0)(b/0)(a/1)" = "abababa/0000001". Q_2 is not a trace of M at s_1 or s_2. Thus N can be reported to be faulty as demonstrated by Q_2.

For convenience, in FSM-based passive fault detection, $Q = (x_1/y_1)(x_2/y_2)\ldots(x_k/y_k)$ always denotes the given I/O sequence, and $k = |Q|$ is the length of Q. For an integer $j \geq 1$,

Q_j^p denotes the prefix of Q of length j; Q_j^s denotes the remaining sequence of Q after deleting Q_j^p from Q.

2.2 EFSM-based Passive Fault Detection

The input to an EFSM-based passive fault detection exercise consists of the specification of SUT N given as a Simplified Extended Finite State Machine (SEFSM) (see Definition 3 below) and the sequence of I/O behaviors a tester has observed during the execution of N given as a sequence E of observed I/O events (see Definition 8). We propose the SEFSM model which is derived from the EFSM model [Lee96]:

Definition 2:

An *Extended Finite State Machine* (EFSM) M is a quintuple

$(I, O, S, \vec{x}, T)$

where I, O, S, T are finite sets of input symbols, output symbols, states, and transitions, respectively, and $\vec{x}$ is a vector of variables. Each transition t in the set T is a 6-tuple:

$(s_t, q_t, a_t, o_t, P_t, A_t)$

where s_t, q_t, a_t, o_t are the start (current) state, end (next) state, input, and output, respectively. $P_t(\vec{x})$ is a predicate on the current variable values and $A_t(\vec{x})$ gives an action on variable values. □

Definition 3:

A *Simplified Extended Finite State Machine* (SEFSM) M is a quadruple

$(S, E_m, \vec{x}, T)$

where

- $S = \{s_1, \ldots, s_n\}$ is a finite set of states;
- E_m is a finite set of I/O events. $e(\vec{y}) \in E_m$ is an input or output event (see Definition 5), and $\vec{y} = (y_1, y_2, \ldots, y_p)$ is a vector of parameters of the I/O event e, called *local variables*;
- $\vec{x} = (x_1, \ldots, x_r)$ is a vector of global parameter values, called *global variables*;
- T is a finite set of transitions (see Definition 4);

The difference between $\vec{y}$ and $\vec{x}$ is that $\vec{y}$ is observable from SUT N while $\vec{x}$ is unobservable. □

If the specification of an SUT N is given as an EFSM rather than as an SEFSM, the given EFSM can be transformed into an SEFSM. For example, an EFSM M given in Figure 2.2 can be transformed into an SEFSM M' in Figure 2.3. Two major differences between M' and M are: first, M' separates the input and output in M. For example, T4 in M is separated into T4a and T4b in M' for input event and output event respectively; second, the predicate is more specific in M' to facilitate passive fault detection (the predicate in an SEFSM is defined below in the Definition 6). An example of this is the splitting of T2 in M into T2a and T2b in M'.

Note that an *Event-driven Extended Finite State Machine* (EEFSM) model is used in [Lee02]. The most important difference between EEFSM and SEFSM models is that the SEFSM model simplifies the structure of predicates in the transitions of the EEFSM model to facilitate passive fault detection in the data portion of an EFSM. For example, Figure 2.4 presents the EEFSM M'' corresponding to M. Compared with M'', M' simplifies the structure of predicates in transitions by eliminating the "or" operator in M''.

Therefore, in M', a transition is executable if and only if all the constraints in the predicate are evaluated to be TRUE (see Definition 6). For example, in Figure 2.3, T2a and T2b represent the T2 in M''. Some other differences between EEFSM [Lee02] and SEFSM will be given in Section 5.1.4.

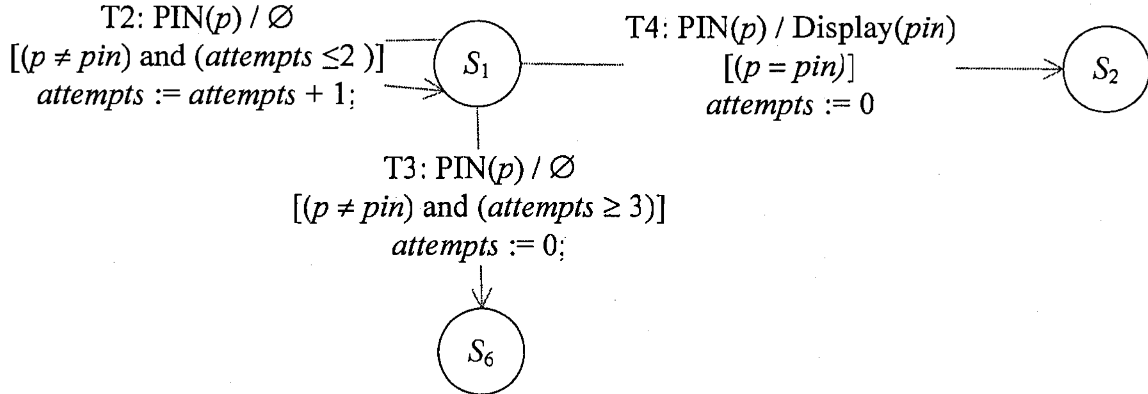

Figure 2.2 An EFSM *M*

(the notation for a transition in the diagram is as follows:
$t : a_t / o_t [P_t] A_t$)

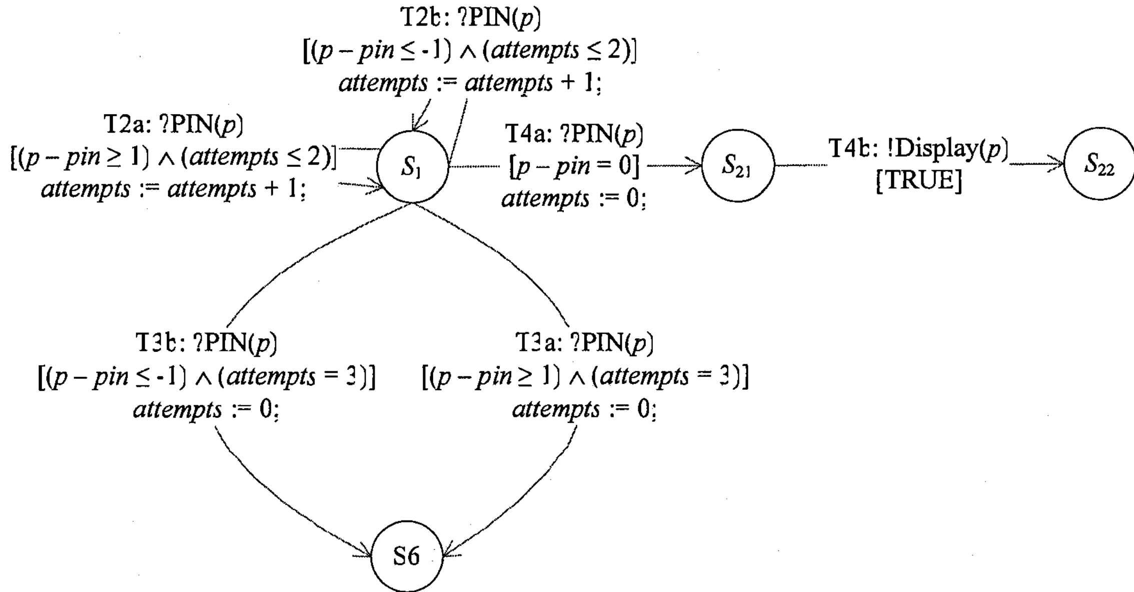

Figure 2.3 An SEFSM *M′*

(the notation for a transition in the diagram is as follows:
$t : e(\vec{y})[P(\vec{x}, \vec{y})]A(\vec{x}, \vec{y})$)

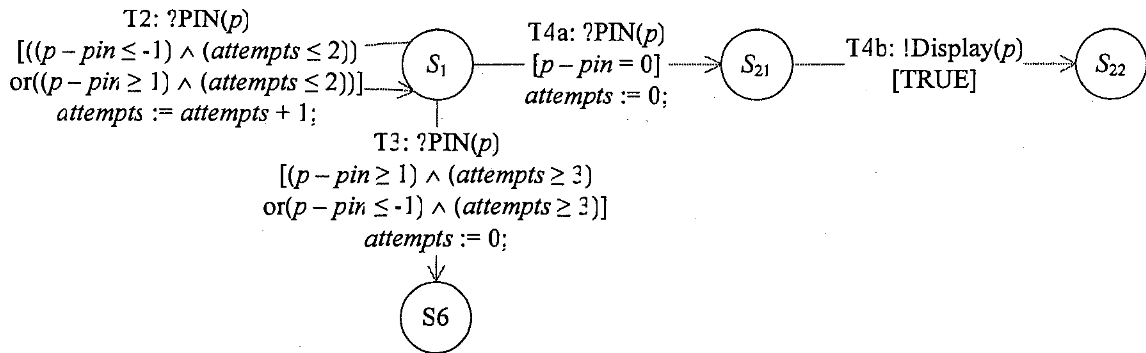

Figure 2.4 An EEFSM M''

(the notation for a transition in the diagram is as follows:
$t : e(\vec{y})[P(\vec{x},\vec{y})]A(\vec{x},\vec{y})$)

Definition 4:

A *transition* $t \in T$ in an SEFSM is a quintuple

$(s_i, s_j, e(\vec{y}), P(\vec{x},\vec{y}), A(\vec{x},\vec{y}))$

where

- s_i is the *starting state* of t;
- s_j is the *ending state* of t;
- $e(\vec{y}) \in E_m$ is an input or output event that can be observed once t is activated;
- $P(\vec{x},\vec{y})$ is a *predicate* expressing the conditions to be satisfied for the activation of t (see Definition 6);
- $A(\vec{x},\vec{y})$ is an *action* consisting of a sequence of assignment statements, updating $\vec{x}$ as functions of $\vec{x}$ and $\vec{y}$ (see Definition 7). (see Figure 2.5) □

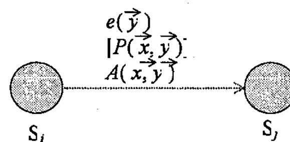

Figure 2.5 The structure of a transition t in an SEFSM

Definition 5:

An I/O event $e(\vec{y})$ is an input or output behavior associated with a transition in an SEFSM, which is categorized as an input event with the prefixed symbol "?" or an output event with prefixed symbol "!". □

For example, "!display(y)" means an I/O event "display" which outputs the value of variable y.

Because $\vec{y}$ is observable from N while $\vec{x}$ is unobservable, the I/O events with global variables as parameters must be modified. For example, assume x is a global variable, an input event "?read(x)" will be transformed to "?read(a) x:=a;" where a is a local variable and the action "x:=a;" assigns the value of a to x; similarly, an output event "!display(x)" will be transformed to "!display(a) [$a = x$]" where the predicate "[$a = x$]" guarantees the output value is equal to x.

Definition 6:

A predicate $P(\vec{x}, \vec{y})$ associated with a transition in an SEFSM consists of conjunctive terms, each of which is defined as a *constraint*, connected by "$\wedge$" (and) operators. A predicate P is in the form:

P ::= TRUE

| FALSE

| cs

| $P \wedge P$

cs ::= $Terms \geq$ INTEGER

| $Terms \leq$ INTEGER

| *Terms* = INTEGER

Terms ::= *Term*

| *Terms* + *Terms*

Term ::= INTEGER × INTEGER_VARIABLE

cs is a constraint, INTEGER is a integer constant, and INTEGER_VARIABLE stands for an integer variable, either a local variable or a global variable. □

For example, "$(3 \times x_1 + (-1) \times x_2 \geq 0) \wedge (1 \times x_1 + 4 \times y_2 \leq 4)$" is a predicate.

In this thesis, *cs* is represented by $\sum_{i=1}^{k} a_i x_i = I$ (a_i is a coefficient, x_i is a global variable, I is an interval) after replacing the local variables of $\vec{y}$ by the actual values of the parameters observed during the execution of N (see Section 5.1.3). For example, the constraint "$3 \times x_1 + (-1) \times x_2 \geq 0$" is represented by the expression "$3 \times x_1 + (-1) \times x_2 = [0, MAX]$". MAX is defined as 1×10^7 and MIN is defined as -1×10^7 in this thesis.

Definition 7:

An *action* $A(\vec{x}, \vec{y})$ associated with a transition in an SEFSM is a sequence of *assignments*. An *action* A is in the form of:

A ::= *assignment*;

| *assignment*; A ;

| NULL

An *assignment* is in the form of:

LHV := INTEGER

| *Terms* + INTEGER □

For example, "$x_3 := 3 \times x_1 + (-1) \times x_2 + (-5); x_1 := x_3;$" is an action.

Definition 8:

The sequence E of observed I/O events represents a sequence of I/O behaviors a tester observed during the actual execution of N, i.e. $e_1e_2 \ldots e_n$. Similar to the definition of the I/O event in SEFSM model, an observed I/O event e_i, $1 \leq i \leq n$, in E is also categorized as an observed input event with the prefixed symbol "?" or an observed output event with prefixed symbol "!". □

Different from the I/O event defined in the SEFSM model, an observed I/O event contains determined values instead of symbols for variables. For example, "?read(3)" is an observed I/O event in E while "?read(y)" is an I/O event in E_m.

Definition 9:

A *configuration* depicts a possible status of the SUT N during EFSM-based passive fault detection. A configuration c is a quadruple

$(\#, s, [\vec{x}], CS(\vec{x}))$

where

- \# is the number of observed I/O events that have been checked to reach the current configuration;
- s is the possible current state of N;
- $[\vec{x}]$ is a vector of *interval*s which represents the ranges of possible values which the variables in $\vec{x}$ can take;
- $CS(\vec{x})$ records the constraints on variables in $\vec{x}$. These constraints are obtained from both predicates and actions. As $CS(\vec{x})$ contains only global variables, we shall henceforth use CS as the abbreviation of $CS(\vec{x})$.□

For example, $c = (3, s_6, \{x_1 = [0, 5], x_2 = [1, 2]\}, \{x_1 + x_2 \geq 0; 3x_1 - x_2 \leq 9;\})$ is a configuration. (see Figure 2.6)

#:	3
s:	s_6
$[\vec{x}]$:	$x_1 = [0, 5], x_2 = [1, 2]$
$CS(\vec{x})$:	$x_1 + x_2 \geq 0; 3x_1 - x_2 \leq 9;$

Figure 2.6 A configuration *c*

According to the configuration c in Figure 2.6, 3 observed I/O events have been checked; the current possible state of N is s_6; the value of x_1 is greater or equal to 0 and less than or equal to 5, and the value of x_2 is greater or equal to 1 and less than or equal to 2; the values of x_1 and x_2 must satisfy two constraints "$x_1 + x_2 \geq 0$" and "$3x_1 - x_2 \leq 9$" at the same time.

Definition 10:

A *trace* represents the sequence of status of the SUT N during the EFSM-based passive fault detection. *Trace-Tree* records all the traces that have been checked during the EFSM-based passive fault detection.

1. A trace *trace* is a sequence of configurations, which are connected by transitions;
2. A Trace-Tree *Tree* for s consists of all the traces starting from a state $s \in S_0$. Each node in a Trace-Tree represents a configuration and each edge stands for a transition between two configurations (see Figure 2.7). Every trace $trace_i$ in *Tree* for s, starting from the root to a leaf with length k, is compatible with a prefix of E ($e_1 e_2 \ldots e_k$, $k \leq |E|$);
3. A *compatible trace* of E is defined as a trace in *Tree* for s with length equal to $|E|$. □

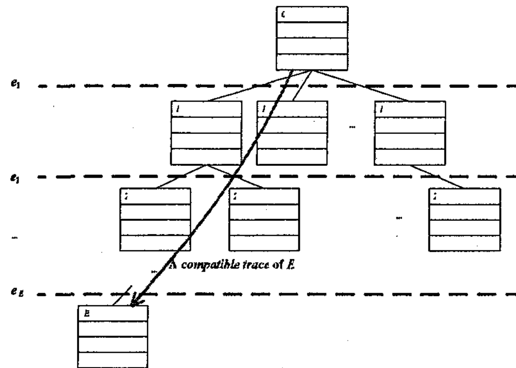

Figure 2.7 A Trace-Tree *Tree* and a *compatible trace* of *E*

Therefore, the process of searching for a trace corresponding to E can be considered to be a process exploring the Trace-Tree *Tree* for s for a compatible trace of E.

Chapter 3

Previous Work

Passive fault detection is useful when the SUT is "in-process". It has proved to be a powerful technique for network fault management [Lee02]. Since [Lee97], much effort has been made on FSM and EFSM-based passive fault detection. In this chapter, we give a review of existing approaches for passive fault detection and their applications to FSM and EFSM-based systems.

3.1 The Approaches for FSM-based Passive Fault Detection

Most of the existing approaches for passive fault detection are derived from the approach for FSM-based passive fault detection in [Lee97] which checks the observed sequence of I/O pairs one-by-one from the beginning, and reduces the size of possible current states by eliminating impossible states until either no state is possible or there is at least one state possible. The approach proposed in [Lee97] was applied to the Signaling System 7 (SS7) protocol mechanisms. It was also adopted by [Wu02] for routing protocol testing with the Protocol Integrated Test System (PITS) and by [Zha01] in the OnLine Test System (OLTS).

There are two major limitations of this approach. First, in this approach, every state of M needs to be checked. However, the number of states compatible with the observed sequence of I/O pairs is usually comparatively small. For example, if the SUT works correctly with respect to its specification, there may be only one state of M that is

compatible with an observed sequence of I/O pairs. In this case, checking all the states of M is unnecessary and inefficient.

Second, this approach only focuses on the set of possible current states of the SUT when passive fault detection terminates. Some useful information about possible starting state and possible trace corresponding to the sequence of I/O behaviors will be unknown after the passive fault detection. This information is useful for the fault location and fault identification processes which need to be performed after fault detection. To retrieve this information, a post-processing approach with backward checking is needed, as shown for example, in [Cav06]. Obviously, post-processing would be unnecessary and preventable if we can collect and record useful relevant information during passive fault detection.

3.2 The Approaches for EFSM-based Passive Fault Detection

[Lee97] briefly discussed the possibility of extending their approach for FSM-based passive fault detection to EFSM and communicating finite state machine (CFSM) models. Based on [Lee97], [Tab99] discussed three possible approaches for extensions of the approach proposed in [Lee97] to an EFSM model. The simplest approach was to generate the reachability graph, which is tantamount to unfolding the EFSM to its equivalent FSM. Unfortunately, this approach is impractical due to the state explosion problem [Tab99]. The second approach was to transform the EFSM into a nondeterministic finite state machine (NDFSM) by removing the enabling predicates of the transitions. But, ignoring the data portion fault detection aspect will certainly lessen the power of passive fault detection. The third approach recorded the value of a variable as "unknown" until an explicit hint was given in later passive fault detection which assigned a determined value

to this variable. This approach is also unsatisfactory because its accuracy depends on the occurrence of the corresponding explicit hint.

Based on [Lee97] and [Tab99], [Lee02] formally proposed an approach for EFSM-based passive fault detection with emphasis on data portion fault detection. This approach was claimed to be a practical approach and was applied to passive fault detection in Open Shortest Path First (OSPF) nearest neighbor state machine testing. However, this approach is based on the approach for FSM-based passive fault detection in [Lee97], and thus inherits the two major limitations discussed in the previous section. Later, [Alc04] and [Alc06] proposed backward checking approaches for exploring the information about possible starting state and possible trace by possible current Candidate Configuration Set (CCS) in EFSM-based passive fault detection. Further, [Lee02] uses the Interval Refinement method for data portion fault detection which may not be accurate because of the dependency problem (see Section 5.2.4). In [Che03a] and [Lee06], the Mixed Integer Linear Programming method was applied to replace the Interval Refinement method in the data portion fault detection of [Lee02]. The Mixed Integer Linear Programming method is accurate but much more time consuming than the Interval Refinement method (see Section 5.3).

3.3 Other Related Research Directions

Another research direction for performing passive fault detection is to utilize the characteristics of the specification, which were named as invariants in [Cav03]. In this approach, the specification is first analyzed before passive fault detection. Through the analysis, a set of invariants are generated to represent the requirements for the correct

behaviors of the SUT. Then, these invariants are applied to the sequence of I/O behaviors a tester observes to see if they match. Related research in this direction includes [Arn04] [Bay05] [Cav06]. This research direction focuses on distilling the characteristic invariants of a specification and is closely related to model checking techniques [Bay05].

Additionally, [Mil98] introduced a variant of communicating finite state machine (CFSM) for the specification of networks and discussed how passive fault detection procedures developed for FSM in [Lee97] could be adopted to apply. Besides passive fault detection, [Mil98] also discussed fault location and the limitations of fault detection in passive testing. Based on [Mil98], further research on fault identification and fault coverage in CFSM-based passive testing was discussed in [Mil01a] [Mil01b] and [Mil01c].

3.4 Summary

According to the previous work, most approaches for FSM and EFSM-based passive fault detection are derived from the approach for FSM-based passive fault detection in [Lee97]. These approaches, such as the approach for EFSM-based passive fault detection in [Lee02], inherit the two major limitations of the approach in [Lee97] given in Section 3.1.

In Chapter 4, we propose a new approach for FSM-based passive fault detection. Compared with the previous approach in [Lee97], the proposed approach improves the performance of FSM-based passive fault detection by reducing the number of possible starting states that need to be checked and providing more information about possible trace during passive fault detection so as to avoid additional subsequent post-processing.

Chapter 4

The Proposed Approach for FSM-based Passive Fault Detection

In this chapter, we first present an existing algorithm proposed by Lee et al [Lee97], then propose a new approach with three algorithms for FSM-based passive fault detection. The computational complexity of each algorithm is discussed after the algorithm is presented.

In order to conduct the analysis and make efficiency-based comparisons of the algorithms, we define the *number of comparisons* between the actual output y_j and the expected output $\lambda(s, x_j)$ to be the measure of computational complexity.

4.1 The Approach in [Lee97]

The idea of FSM-based passive fault detection in [Lee97] can be summarized as follows: suppose that the starting state N is any state of M (i.e., $S_0 = S$), check the sequence Q of I/O pairs one-by-one from the beginning, reduce the size of possible current states by eliminating impossible states until either no state is possible (N is faulty) or there is at least one state (no fault is detected by Q).

In order to facilitate comparisons, we formalized the idea of [Lee97] and rewrote the algorithm given in [Lee97] as Algorithm 0 without changing its computational complexity. Thus, the computational complexity of Algorithm 0 can be used in the comparison as the computational complexity of the approach in [Lee97].

```
Algorithm 0
Given: an FSM M,
        a sequence Q of I/O pairs = (x1/y1)(x2/y2)...(xk/yk), and
        S0 = {s1, s2, ..., sn}.
Return:  "N is faulty", or
         "No fault is detected by Q and the set of possible current states is S'".
Begin:
  j ← 1;                        // j is the counter for I/O pairs
  S' ← S0;
  while (j ≤ k)
    if (S' ≠ ∅)
      S'' ← ∅;
      for (every s ∈ S' ) // check each state in S'
        if (yj = λ(s, xj))
          S'' ← S'' ∪ δ(s, xj);     // remove redundant states in S''
      endfor
      S' ← S'';
      j ← j+1;
    else                        // S' = ∅
      return ("N is faulty");
  endwhile
  return ("No fault is detected by Q and the set of possible current states is S'");
End
```

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

Theorem 1 (Lee et al [Lee97]) Let S_j denote the set of possible current states right after the first j I/O pairs of Q, i.e., $S_j = \delta(S_0, x_1x_2 \ldots x_j)$. The complexity of Algorithm 0 is $C_1 = \sum_{j=1}^{k} | S_{j-1} |$.

Proof: In Algorithm 0, every state in the set of possible current states will be checked to compare its related I/O pair with the current I/O pair in Q, i.e., for a state s in S_j, there

will be one comparison between y_{j+1} and the expected output $\lambda(s, x_{j+1})$, and $| S_j |$ comparisons are needed to check the set S_j. Thus, the total number of comparisons is $\sum_{j=1}^{k} | S_{j-1} |$.

Example: Applying Algorithm 0 to M (Figure 2.1) and Q = "abababb/0000001", the number of comparisons is $\sum_{j=1}^{k} | S_{j-1} | = 3 + 2 + 2 + 2 + 2 + 2 + 2 = 15$.

4.2 The Proposed Approach

The following algorithm is based on our proposed approach which checks, for each state $s \in S_0$, whether or not Q is a trace of M at s. This algorithm terminates when Q is verified to be a trace of M at a state $s \in S_0 \subseteq S$ or when all states in S_0 are checked and no state is found compatible with Q. With this approach, if we know that certain state(s) are likely to be the starting state s, it may take less time than Algorithm 0 to find out when there is a state s of M such that Q is a trace of M at s.

Algorithm 1
Given: an FSM M,
  a sequence Q of I/O pairs = (x_1/y_1) (x_2/y_2)...(x_k/y_k), and
  $S_0 = \{s_1, ..., s_n\}$.
Return: "N is faulty", or
  "Q is a trace of M at state s_i and the possible current state is s".
Begin:
  $i \leftarrow 1$; // i is the state counter
  while $(i \leq n)$
    $j \leftarrow 1$; // j is the counter for I/O pairs
    $s \leftarrow s_i$; // s will represent $\delta(s_i, x_1 ... x_{j-1})$ when $j > 1$
    while ($j < k$ and $y_j = \lambda(s, x_j)$)
      $s \leftarrow \delta(s, x_j)$; // s is updated as the current state
      $j \leftarrow j+1$;
    endwhile
    if (($j = k$ and $y_j = \lambda(s, x_j)$)
      return ("Q is a trace of M at state s_i and the possible current state is s");

else

$i \leftarrow i+1$;

endwhile

return ("N is faulty");

End

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

Theorem 2 For the given state s_i of M and a sequence Q of I/O pairs $= x_1 \ldots x_k / y_1 \ldots y_k$, let $c_i(Q)$ denote the largest number j ($1 \leq j \leq$ k) such that $y_1 \ldots y_{j-1} = \lambda(s_i, x_1 \ldots x_{j-1})$ and $\lambda(\delta(s_i, x_1 \ldots x_{j-1}), x_j) \neq y_j$. Let $C_{2worst}(M, S_0, Q) = \sum_{i=1}^{n} c_i(Q)$. Let $C_2(M, S_0, Q)$ denote the complexity of Algorithm 1. If s_r is the first state of M such that Q is a trace of M at s_r, then $C_2(M, S_0, Q) = \sum_{i=1}^{r} c_i(Q)$; if N is faulty, then $C_2(M, S_0, Q) = C_{2worst}(M, S_0, Q) = \sum_{i=1}^{n} c_i(Q)$.

Proof: In Algorithm 1, each state in S_0 is checked whether it is compatible with Q. The checking procedure for a state s_i will not stop until it confronts a mismatch (then the next state s_{i+1} will be selected to check); or the entire sequence Q has been checked and no mismatch found (then s_i is reported to be compatible with Q). The whole checking procedure will terminate when a state compatible with Q is found or when all the states have been checked and no state is found to be compatible with Q. Assume $y_1 \ldots y_{j-1} = \lambda(s_i, x_1 \ldots x_{j-1})$ but $y_j \neq \lambda(\delta(s_i, x_1 \ldots x_{j-1}), x_j)$, it means j comparisons (denoted by $c_i(Q)$) are needed to decide whether or not Q is a trace of M at s_i. If $s_r \in S_0$ is the first state of M such that Q is the trace of M at s_r, Algorithm 1 will detect mismatch in checking $s_1 \ldots s_{r-1}$ and stop after checking s_r. Thus, the total number of comparisons needed is $\sum_{i=1}^{r} c_i(Q)$.

Example: Applying Algorithm 1 to M (Figure 2.1) and Q = "abababb/0000001", with r = 1, the number of comparisons is $C_2(M, S_0, Q) = \sum_{i=1}^{r} c_i(Q) = c_1(Q) = 7$.

Algorithm 1 is simple and straightforward, however, it encounters the *redundant checking problem* which is: two traces starting from different states converge to the same state after applying Q_j^p. In Algorithm 1, the common part Q_j^s will be rechecked redundantly. In contrast, Algorithm 0 avoids the redundant checking problem by removing redundant states in the set of possible current states.

The algorithm presented below, i.e. Algorithm 2, attempts to combine the merits of both Algorithm 0 and Algorithm 1. Let S_j denote the set of possible current states right after the first j I/O pairs of Q, i.e., $S_j = \delta(S_0, x_1x_2...x_j)$. In Algorithm 2, Algorithm 0 is used first to reduce the size of S_j and then Algorithm 1 is used on the current S_j with the remaining portion of the sequence Q.

Algorithm 2
Given: an FSM M,
  a sequence Q of I/O pairs = $(x_1/y_1)(x_2/y_2)...(x_k/y_k)$,
  an integer $q \le k$, and
  $S_0 = \{s_1, ..., s_n\}$.
Return: "N is faulty", or
  "No fault is detected by Q and the possible current state is s ".
Begin:
  $j \leftarrow 1$;   // j is the counter for I/O pairs
  $S' \leftarrow S_0$;
  while ($j \le q$)   // this while loop implements Algorithm 0 up to Q_q
    if ($S' \neq \varnothing$)
      $S'' \leftarrow \varnothing$;
      for (every $s \in S'$)
        if ($y_j = \lambda(s, x_j)$)
          $S'' \leftarrow S'' \cup \delta(s, x_j)$; // redundant states in S'' are removed
      endfor
      $S' \leftarrow S''$; $j \leftarrow j+1$;
    else   // $S' = \varnothing$

```
            return ("N is faulty");
        endwhile
        while (S' ≠ ∅)          // Algorithm 1 starting from S_q = S'
            randomly choose a state s from S';
            S' ← S' \ {s};
            j ← q + 1;           // j is the counter for I/O pairs
            while (j < k and y_j = λ(s, x_j))
                s ← δ(s, x_j);    // s represents the current state
                j ← j + 1;
            endwhile
            if (j = k and y_k = λ(s, x_k))
                return ("No fault is detected by Q and
                        the possible current state is s");
        endwhile
        return ("N is faulty");
    End
```

The computational complexity of Algorithm 2 is obtained by combining the results in Theorem 1 and Theorem 2 and it is affected by the selection of variable q. Suppose that, in Algorithm 0 part of Algorithm 2, j_1 is the minimum number of I/O pairs needed to eliminate the redundant checking problem and j_2 is the minimum number of I/O pairs needed to reduce the size of possible current states to be one or zero. Obviously, j_1 is smaller than j_2. Clearly,

if q is smaller than j_1, the redundant checking problem remains;

if q is larger than j_2, Algorithm 2 works the same as Algorithm 0;

if $j_1 \leq q < j_2$, Algorithm 2's performance is at least equal to the performance of Algorithm 0. However, since j_1 and j_2 are both determined by Q and M, it is difficult to determine a proper value for q. Moreover, the Algorithm 0 part makes Algorithm 2 unable to determine the possible starting state unless post-processing is performed.

Algorithm 0 is comprehensive but not efficient enough as it has to check all the states in S_0 and doesn't provide information about possible starting states. Algorithm 1 is straightforward but its efficiency is influenced by the redundant checking problem.

Algorithm 2 attempts to eliminate the redundant checking problem by combining the merits of both Algorithm 0 and Algorithm 1 but the effort is limited as it introduces another preset variable q for which it is difficult to find an appropriate value. Also, Algorithm 2 cannot determine the possible starting state.

To overcome the drawbacks of these three algorithms, an improved algorithm based on our approach is presented as the following:

Algorithm 3
Given: an FSM M,
    a sequence Q of I/O pairs = $(x_1/y_1)\ (x_2/y_2)\ldots(x_k/y_k)$, and
    $S_0 = \{s_1, \ldots, s_n\}$.
Return: "N is faulty", or
    "Q is a trace of M at state s_i and the possible current state is s".
Begin:
  $F_{1\ldots k-1} \leftarrow \varnothing$;
  $i \leftarrow 1$;
  while $(i \leq n)$
    $j \leftarrow 1$;   // j is the counter for I/O pairs
    $s \leftarrow s_i$;   // s will represent $\delta(s_i, x_1 \ldots x_{j-1})$ when $j > 1$
    while $(j < k$ and $y_j = \lambda(s, x_j))$
      $s \leftarrow \delta(s, x_j)$; // s is updated as current state
      if $(s \in F_j)$ // to eliminate *redundant checking problem*
        break;   // state s has already been checked,
              // thus end this trace
      else
        $j \leftarrow j + 1$;
    endwhile
    if $(j = k$ and $y_j = \lambda(s, x_j))$
      return ("Q is a trace of M at state s_i and the possible current state is s");
    else
      $i \leftarrow i + 1$;
      if $(j > 1)$
        add $\delta(s_i, x_1 \ldots x_l)$ to F_l, $l = 1, \ldots, j - 1$; // record the trace
  endwhile
  return ("N is faulty");
End

The data structure $F_{1\ldots k-1}$ is used to record the tracing history and therefore to avoid the redundant checking problem. If $\delta(s_i, x_1x_2\ldots x_j) \in F_j$, $1 \leq j \leq k$, then the $\lambda(\delta\ (s_i, x_1x_2\ldots x_j),$

$x_{j+1}\ldots x_k$) has already been checked and $y_{j+1}\ldots y_k \neq \lambda(\delta(s_i, x_1x_2\ldots x_j), x_{j+1}\ldots x_k)$. So, Q_j^s will not need to be checked and the checking started at s_i will stop. After checking a state s, if s is not an eligible starting state, Algorithm 3 adds the trace history starting from s into $F_{1..j-1}$ correspondingly; if s is compatible with Q, the possible starting state (s) and its corresponding trace are determined.

Theorem 3 For a given state s_i of M and a sequence Q of I/O pairs = $x_1\ldots x_k/y_1\ldots y_k$, let $c_i'(Q)$ denote the largest number j ($1 \leq j \leq$ k) such that (1) $y_1\ldots y_{j-1} = \lambda(s_i, x_1\ldots x_{j-1})$ and $\lambda(\delta(s_i, x_1\ldots x_{j-1}), x_j) \neq y_j$; (2) for every l ($1 \leq l \leq j$-1), $\delta(s_i, x_1\ldots x_l) \notin F_l$. If the rth state checked, s_r, is the first state of M such that Q is a trace of M at s_r, then the complexity of Algorithm 3:

$$C_3(M, S_0, Q) = \sum_{i=1}^{r} c_i'(Q);$$

if N is faulty, then $C_{3worst}(M, S_0, Q) = \sum_{i=1}^{n} c_i'(Q)$;

if $r = 1$, $C_{3best}(M, S_0, Q) = c_1'(Q)$.

Proof: Compared to Algorithm 1, the checking procedure of Algorithm 3 on state s_i will stop when it encounters a mismatch with Q, the whole Q has been checked compatible, or $\delta(s_i, x_1x_2\ldots x_j) \in F_j$. Similar to Theorem 2, let $c_i'(Q)$ denote the largest number j ($1 \leq j \leq k$) before checking on s_i terminates. If s_r is the first state of M such that Q is the trace of M at s_r, then the total number of comparisons needed is $\sum_{i=1}^{r} c_i'(Q)$.

Assuming the states compatible with Q are randomly dispersed in S_0, the process that sequentially checks the states within S_0 until a state compatible with Q is found can be modeled as the *Sampling without Replacement Model* [DeG02].

Theorem 4 According to the *Sampling without Replacement Model*, assume that there are m states in S_0 which are compatible with Q, and r ($1 \le r \le n\text{-}m+1$) states are randomly selected from S_0. Then, the probability that the rth state is the first state which is checked and found to be compatible with Q is given by

$$P_r(m) = \begin{cases} \dfrac{m}{n} & (r=1); \\ \dfrac{m(n-m)(n-m+1)...(n-m-r+2)}{n(n-1)...(n-r+1)} & (2 \le r \le n-m+1). \end{cases}$$

Proof: Each different arrangement of states selected from S_0 is called a *permutation.* Suppose that r states are selected one at a time and removed from S_0 ($1 \le r \le n\text{-}m+1$). Then each possible outcome of this selection will be a permutation of r states from S_0, and the total number of these permutations will be $P_{n,r} = n(n - 1)\ldots(n - r + 1)$ [DeG02]. This number $P_{n,r}$ is called the *number of permutations of n elements taken r at a time.* Thus, if $r = 1$, the number of permutations is n; if $2 \le r \le n\text{-}m+1$, the number of permutations, in which the rth state is the first state compatible with Q, is $mP_{n\text{-}m,r\text{-}1} = m(n - m)(n - m - 1)\ldots(n - m - r + 2)$. Thus, the probability of permutation that the rth state is the first state compatible with Q is:

if $r = 1$, $P_r(m) = \dfrac{m}{P_{n,r}} = \dfrac{m}{n}$;

if $2 \le r \le n\text{-}m+1$, $P_r(m) = \dfrac{mP_{n-m,r-1}}{P_{n,r}} = \dfrac{m(n-m)(n-m+1)...(n-m-r+2)}{n(n-1)...(n-r+1)}$.

Theorem 5 Suppose there are m ($0 \le m \le n$) states in S_0 which are compatible with Q. Let $P_r(m)$ denote the probability that the rth state is the first state which is compatible with Q. The average complexity of Algorithm 3 is

$$A_3 = \sum_{r=1}^{n-m+1} P_r(m) C_3(M, S_0, Q) = \sum_{r=1}^{n-m+1} (P_r(m) \sum_{i=1}^{r} c_i'(Q)) .$$

Proof: The average complexity of Algorithm 3 is the sum of the number of comparisons multiplied by its corresponding probability.

Example: Assume $n = 4$, $m = 1$, when $r = 1$, $C_3 = 4$; when $r = 2$, $C_3 = 5$; when $r = 3$, $C_3 = 7$; when $r = 4$, $C_3 = 9$ where C_3 stands for $C_3(M, S_0, Q)$; Then,

Table 4.1. Average complexity analysis

r	C_3	$P_r(m{=}1)$	$P_r(m{=}1)\ C_3$
1	4	1/4	1
2	5	1*3 / 4*3 = 1/4	5/4
3	7	1*3*2 / 4*3*2 = 1/4	7/4
4	9	1*3*2*1 / 4*3*2*1 = 1/4	9/4

Thus, $A_3 = \sum_{r=1}^{n-m+1} P_r(m) C_3(M, S_0, Q) = 1 + 5/4 + 7/4 + 9/4 = 25/4$.

In general, the number of states compatible with Q may be zero, or more. If this number is zero, it means that none of the states in S_0 is compatible with Q and N is faulty; if it is one or more than one, it means that the given Q is insufficient to determine whether N is faulty.

The general case can be simplified to the case in which the number of states which are compatible with Q is either one or zero. This stems from the fact that the essence of passive fault detection is to detect the existence of faults in N. If there are one or more states compatible with Q, it implies that the given Q is insufficient to come to a conclusion. An additional sequence ΔQ is needed to continue with the passive fault

detection. Thus, let $Q' = Q + \Delta Q$ denote the sequence concatenating Q to ΔQ. Thus, the new set (M, S_0, Q') contains at most one state that is compatible with Q'. Let $P_r(m = 1)$ denote the probability that there is one compatible state in S_0 and it appears at the rth ($1 \leq r \leq n$) selection. So, $P_r(m = 1) = \frac{1}{n}$.

Theorem 6: If there is only one state in S_0 that is compatible with Q, the average complexity of Algorithm 3 is $A_3 = \frac{1}{n}\sum_{r=1}^{n} C_3(M, S_0, Q) = \frac{1}{n}\sum_{r=1}^{n}(\sum_{i=1}^{r} c_i'(Q))$.

Proof: The average complexity of Algorithm 3 is the sum of the number of comparisons multiplied by its corresponding probability.

4.3 Comparison of the Algorithms

The computational complexities of the four algorithms are given in the theorems in the previous subsections. Table 4.2 below summarizes the results given in these theorems.

Table 4.2 Computational complexities of three algorithms

Type of algorithm	Computational complexity
Algorithm 0	$C_1 = \sum_{j=1}^{k} \mid S_{j-1} \mid$
Algorithm 1	$C_2(M, S_0, Q) = \sum_{i=1}^{r} c_i(Q)$
Algorithm 3	$C_3(M, S_0, Q) = \sum_{i=1}^{r} c_i'(Q)$

(k is the length of Q, $|S_j|$ is the number of states in the set of possible current states,
r is the number of states checked before a state compatible with Q is found,
$c_i(Q)$ denotes the largest number j $(1 \leq j \leq k)$
such that $y_1 \ldots y_{j-1} = \lambda(s_i, x_1 \ldots x_{j-1})$ and $\lambda(\delta(s_i, x_1 \ldots x_{j-1}), x_j) \neq y_j$
$c_i'(Q)$ denotes the largest number j $(1 \leq j \leq k)$
such that $y_1 \ldots y_{j-1} = \lambda(s_i, x_1 \ldots x_{j-1})$ and $\lambda(\delta(s_i, x_1 \ldots x_{j-1}), x_j) \neq y_j$ after eliminating the redundant checking problem)

Below, we compare the complexity of Algorithm 0 and Algorithm 3 when the number of states in S_0 which are compatible with Q is one or zero.

In Algorithm 0, once the set (M, S_0, Q) is fixed, the number of comparisons needed is determined and does not change during its application. On the other hand, the performance of Algorithm 3 is affected by the number of states in S_0 which are compatible with Q (see Theorem 4).

Algorithm 0 and Algorithm 3 represent different perspectives on tracing order in passive fault detection. Algorithm 0 checks all the states in the set of possible current states with one I/O pair in Q at a time whereas Algorithm 3 selects one starting state from S_0 and exhausts all the possible transitions starting from this state according to the sequence Q of I/O pairs.

If there is no state in S_0 which is compatible with Q, both Algorithm 0 and Algorithm 3 need to check the entire trace from every state in S_0 and thus these two algorithms perform equally. That is $\sum_{i=1}^{n} c_i'(Q) = \sum_{j=1}^{k} |S_{j-1}|$.

If there is only one state s in S_0 which is compatible with Q, then the total number of comparisons made by Algorithm 0 is $\sum_{j=1}^{k} |S_{j-1}|$ whereas the total number of comparisons made by Algorithm 3 is $\sum_{i=1}^{r} c_i'(Q)$ $(r \leq n)$. Clearly, $\sum_{i=1}^{r} c_i'(Q) \leq \sum_{i=1}^{n} c_i'(Q) = \sum_{j=1}^{k} |S_{j-1}|$, $(r \leq n)$ where the rth state checked is the state compatible with Q.

Thus, Algorithm 3 always performs at least as well as Algorithm 0. The equality in their computational complexity occurs when $r = n$.

4.4 Assertions

Based on the computational complexity of the algorithms presented in Section 4.3, several assertions can be made concerning their performance under different conditions.

Assertion 1: If N is not determined to be faulty and there is no redundant checking problem,

- The performance of Algorithm 1 will be the same as Algorithm 3 and be better or at least equal to that of Algorithm 0;
- The performance of Algorithm 2 will be between that of Algorithm 1 and that of Algorithm 0.

Assertion 2: If N is not determined to be faulty and there is a redundant checking

problem,

- The performance of Algorithm 3 will be better or at least equal to that of Algorithm 0 and Algorithm 1;
- It is not possible to compare the performances of Algorithm 0, Algorithm 1, and Algorithm 2 analytically because of the redundant checking problem.

Assertion 3: If N is determined to be faulty and there is no redundant checking problem,

- The performance of all the algorithms will be the same.

Assertion 4: If N is determined to be faulty and there is a redundant checking problem,

- The performance of Algorithm 3 will be equal to that of Algorithm 0;
- The performance of Algorithm 1 and Algorithm 2 will be worse or at most equal to that of Algorithm 0;
- It is not possible to compare the performances of Algorithm 0 and Algorithm 2 analytically.

4.5 Experimental Evaluation

This section describes an experimental evaluation that investigates the computational complexity of the four algorithms. There are two motivations for this experimental evaluation. First, the experiment will compare the average computational complexity of each algorithm; second, the experiment is expected to provide support to the assertions drawn above when $m = 0$ or 1.

4.5.1 Experiment Setup

In the experiment, we use a set of FSMs randomly generated by [TSG94]. This set

consists of FSMs with different numbers of states ($|S_0| = |S|$), set X of inputs and set Y of outputs. We select 5 configurations in the form of ($|S_0|$, $|X|$, $|Y|$), namely (5, 3, 3), (10, 4, 4), (15, 4, 4), (20, 5, 5), (30, 10, 10). For each configuration, we generate 5 FSMs respectively. For each FSM M, two cases are considered in the experiment.

In Case I, called *correct implementation*, there is exactly only one state in S_0 that is compatible with Q ($m = 1$). For every state s of M, we generate three random sequences of length $|S_0|*|X|*2$, $|S_0|*|X|*4$, $|S_0|*|X|*10$ respectively, starting from s. When generating each I/O sequence Q, we randomly select a transition of the current state of M and repeat this at the next state.

In Case II, called *faulty implementation*, there is no state in S_0 that is compatible with Q ($m = 0$) and "faulty" is expected to be reported. First, we create a faulty specification M' from M by altering either the output or next state of a (randomly) selected transition. For every state s of M', we generate three random sequences of length $|S_0|*|X|*2$, $|S_0|*|X|*4$, $|S_0|*|X|*10$ respectively, starting from s. When generating each I/O sequence Q, we randomly select a transition of the current state of M' and repeat this at the next state.

Then, we apply all four algorithms to the FSMs in these two cases and record the results.

4.5.2 Results of the Experiment

Table 4.3 shows the *number of comparisons* (between the actual output y_j and the expected output $\lambda(s, x_j)$), $1 \leq j \leq k$, $s \in S$, for each of the four algorithms applied to randomly generated FSM $M = (S, X, Y, \lambda, \delta)$, $S_0 \subseteq S$, and sequence Q of I/O pairs from its implementation FSM N which is M in Case I and M' in Case II. The results show the best

case, worst case, and average case.

From Table 4.3,

- Algorithm 1, in case I, has better performance than Algorithm 0 in average case and best case, but not in worst case. Also, in case II, Algorithm 1 cannot beat Algorithm 0;
- Algorithm 2 performs the same as the Algorithm 0 because the number of states in the set of possible current states shrinks to one or zero in the "Algorithm 0" part of Algorithm 2.
- Algorithm 3, in case I, needs fewer comparisons to find the compatible state and performs better than Algorithm 0; while in case II, these two algorithms perform the same.

Experimental results show that Algorithm 3 performs best overall among these four algorithms when there is one state in S_0 which is compatible with Q (Case I). The experimental results also confirm the assertions we present in Section 4.4.

Table 4.3 Experimental comparisons of four algorithms

			Case I : $m = 1$			Case II : $m = 0$		
Algorithm 0	$\|Q\|$	$\|S_0\|$	best	worst	average	best	worst	average
	60	5	64	74	65.5	6	58	17.8
	160	10	169	175	171.3	11	162	31.7
	240	15	254	262	258.0	16	252	42.8
	400	20	419	429	423.0	21	399	44.7
	1200	30	1229	1236	1231.9	31	1122	73.0
Algorithm 1	$\|Q\|$	$\|S_0\|$	best	worst	average	best	worst	average
	60	5	60	74	62.3	7	58	18.7
	160	10	160	172	165.5	11	202	36.9
	240	15	240	263	247.7	16	252	46.1
	400	20	400	432	409.3	21	744	52.9
	1200	30	1200	1235	1215.0	31	1531	83.7
Algorithm 2	$\|Q\|$	$\|S_0\|$	Best	worst	average	best	worst	average
$q = 5$ (q is	60	5	64	74	65.5	6	58	17.8
defined in	160	10	169	175	171.3	11	162	31.7
Algorithm 2)	240	15	254	262	258.0	16	252	42.8
	400	20	419	429	423.0	21	399	44.7
	1200	30	1229	1236	1231.9	31	1122	73.0
Algorithm 3	$\|Q\|$	$\|S_0\|$	Best	worst	average	best	worst	average
	60	5	60	65	61.9	6	58	17.8
	160	10	160	171	164.3	11	162	31.7
	240	15	240	262	247.9	16	252	42.8
	400	20	400	427	409.6	21	399	44.7
	1200	30	1200	1234	1215.3	31	1123	73.0

4.6 Conclusions

In this chapter, we proposed a new approach for FSM-based passive fault detection and present three algorithms based on this approach, i.e. Algorithm 1, Algorithm 2, and Algorithm 3.

Among three proposed algorithms, Algorithm 1 is straightforward but encounters the redundant checking problem; Algorithm 2 can partially eliminate the redundant checking problem by combining both Algorithm 0 and Algorithm 1, however it introduces another preset variable q for which it is difficult to find an appropriate value. Algorithm 3 uses the data structure $F_{1...k}$ to record the tracing history and therefore to avoid the redundant checking problem.

We also stated Algorithm 0, which represents the approach for FSM-based passive fault detection in [Lee97], as a comparison.

Through theoretical and experimental evaluations on computational complexities, we proved that (1) Algorithm 3 has the best performance among three proposed algorithms; (2) Algorithm 3 performs better than Algorithm 0 in situations where there is only one state in S_0 that is compatible with Q; (3) Algorithm 3 performs the same as Algorithm 0 in situations where there is no state in S_0 that is compatible with Q.

Further, Algorithm 3 provides more information about possible starting state and possible trace compatible with Q than Algorithm 0 during the passive fault detection.

According to the analysis in this chapter, we determine that the proposed approach for FSM-based passive fault detection improves the performance of FSM-based passive fault detection in [Lee97] by reducing the computational complexity and providing information about possible starting state and possible trace corresponding to Q during the

passive fault detection without the need for post-processing.

In the following chapter, we extend the proposed approach to EFSM-based passive fault detection.

Chapter 5

The Proposed Approach for EFSM-based Passive Fault Detection

This chapter proposes an approach for EFSM-based passive fault detection, which is derived from the proposed approach for FSM-based passive fault detection in Chapter 4.

Recall that, in EFSM-based passive fault detection, the specification of SUT N is modeled as an EFSM M, N is treated as a blackbox, and the tester wishes to determine whether N is faulty with respect to M by observing the I/O behaviors of N represented as a sequence E of observed I/O events. Therefore, the passive fault detection problem can be defined as follows: given an M and E of N where the starting state (when E starts) of N is unknown, determine whether N is faulty. Such a decision can be based on the number of states that are compatible with E. A state s of M is compatible with E if E is a trace of M starting at s. If the number of states compatible with E is zero, then E is sufficient to determine that N is faulty. Otherwise, E is declared to be insufficient to determine whether N is faulty. That is, there are one or more states that are compatible with E and E needs to be augmented by additional I/O behaviors of N to continue with passive fault detection.

An EFSM contains a control and data portion. The control portion can be considered as a FSM, which contains states and transitions. The data portion specifies functions that involve variables and input or output parameter values. Thus when determining whether a state is compatible with E, one has to consider both control portion and data portion of an EFSM. As the control portion fault detection has been discussed in FSM-based passive fault detection in Chapter 4, our emphasis of EFSM-based passive fault detection will be

the data portion fault detection.

The following sections are organized as follows. Section 5.1 introduces the proposed approach for EFSM-based passive fault detection. In the proposed approach, the data portion fault detection is performed by the function *evaluate*. Sections 5.2, 5.3, and 5.4 discuss implementing the function *evaluate* in the proposed approach with the Interval Refinement method, the Simplex method, and the Hybrid method, respectively. Finally, Section 5.5 gives our conclusions for EFSM-based passive fault detection.

5.1 The Proposed Approach

During EFSM-based passive fault detection, the input consists of the specification M of SUT N in the form of SEFSM and a sequence E of observed I/O events; the proposed approach attempts to find a compatible trace of E in M; and yields an output as either "N is faulty" or "a compatible trace of E in M is found".

The proposed approach can be briefly described as follows:

Given an SEFSM M, a sequence E of observed I/O events, and $S_0 \subseteq S$,

(1) Pick an unchecked state s from S_0;

(2) Try to build a Trace-Tree which is formed by all the possible traces starting from state $s \in S_0$;

(3) If a compatible trace *trace* of E is found, declare that "*trace* is a compatible trace of E in M"; if no compatible trace can be found, go to (1);

(4) If all states in S_0 have been checked and no compatible trace of E is found, declare that "N is faulty";

To implement these four steps, we propose the following algorithms.

5.1.1 Algorithm Main

In algorithm *Main*, we randomly select a state s from $S_0 \subseteq S$ of SEFSM M and try to find a compatible trace of E starting from s.

```
Algorithm Main
Given: an SEFSM M,
        a sequence E of observed I/O events, and
        S0 = {s1, s2, ..., sn}
Return:  "N is faulty", or
         "trace is a compatible trace of E in M"
Begin:
   S' ← S;     // S' represents the set of unchecked states
   while (S' ≠ ∅)
      randomly select a state s from S';
      S' ← S' \ {s};
      trace ← Search_Trace_Tree(M, s, E);    // search for a compatible trace of E
      if (trace ≠ NULL)
         return ("trace is a compatible trace of E in M");
   endwhile
   return ("N is faulty");              // no compatible trace of E is found
End
```

If a compatible trace *trace* is found, this algorithm will terminate and declare *trace* as a compatible trace of E; if all states in S_0 have been checked and no compatible trace is found, the algorithm will report "N is faulty".

5.1.2 Algorithm Search_Trace_Tree

The algorithm *Search_Trace_Tree* is applied to search for a compatible trace of E starting from the state s. In this algorithm, the data structures for configuration and Trace-Tree (defined in Chapter 2) are utilized.

Algorithm ***Search_Trace_Tree***
Given: an SEFSM M,
a state $s \in S_0$, and
a sequence E of observed I/O events
Return: a compatible trace *trace*, or
NULL
Begin:

```
Tree ← NULL;              // initialize the Trace-Tree Tree
trace ← NULL;             // initialize the trace trace
[x⃗]0 ← set the initial intervals of the global variables in M;
c0 ← (0, s, [x⃗]0, ∅);     // create the initial configuration c0 = (#, s, [x⃗], CS)
trace.add(c0);            // add c0 as the first configuration in this trace
Tree.add(trace);
while (Tree ≠ ∅)
    trace ← Tree.get(0);  //get the first trace in Tree
    succ ← Check_Trace(M, trace, E, Tree);   // check if this trace is
                                             // compatible with E
    if (succ = TRUE)      // if trace is compatible with E
        return (trace);
    else
        Tree.delete(trace); // delete the incompatible trace from Tree
endwhile
return (NULL);            // no trace corresponding to E has been found
```

End

5.1.3 Algorithm Check_Trace

As discussed above, a trace consists of a sequence of configurations which represents the sequence of changes in the status of N through E. Algorithm *Check_Trace* checks if a trace is a compatible trace of E.

Algorithm ***Check_Trace***
Given: an SEFSM M,
a trace *trace*,
a sequence E of observed I/O events, and
a Trace-Tree *Tree*,
Return: FALSE, or // *trace* is not a compatible trace of E
TRUE // *trace* is a compatible trace of E
Begin:

```
c_current ← trace.get(0); // get the first configuration
```

```
while (c_current ≠ NULL and c_current.# ≠ E.#)     // if there is an observed I/O event
                                                   // to be checked
    s ← c_current.s ;
    T_s ← all transitions in M starting at s ;
    e ← E.get(c_current.# + 1);   // get the observed I/O event e
                                  // for evaluating transitions
    C ← ∅;
    for each transition t in T_s  // evaluate transitions
        c ← c_current;
        if (control_portion_checking(c, t, e) = FALSE)
            end the for loop;     // the control portion is inconsistent
        else                      // the data portion fault detection commences
            newCS ← replace(t.P(x⃗, y⃗), e);  //eliminate local
                                              // variables in the predicate
            if (evaluate(c.[x⃗], c.CS, newCS) = FALSE)
                end the for loop;   // the data portion is inconsistent
            else
                C ← C ∪ {c}; // c has been modified and needs to be added to C
    endfor
    if (C = ∅)                    // if no executable transition is found
        return (FALSE);
    else
        for each configuration c in C
            c ← action(t, e, c) ;   // perform actions associated with transition t
            if (c = NULL)
                end the for loop;
            else
                if (c is the first configuration in C)
                    add c to trace;
                    c_current ← c;
                else
                    build a new trace branch_trace;
                    add c to branch_trace;     // create a new branch branch_trace
                    add branch_trace to tree;
        endfor
    endwhile
    return (TRUE);        // a compatible trace is found
End
```

Algorithm *Check_Trace* first determines the current possible state s and the observed I/O event e by the configuration $c_{current}$. Then, all transitions in M starting from s are checked by control portion and data portion. Those transitions passing both control portion and data portion fault detection will be considered as a possible transition corresponding to

the observed I/O event e. As there may be more than one possible transition, the algorithm *Check_Trace* picks the first one of them to continue checking and adds all other transitions as branches into the tree. The procedure of checking a Trace-Tree is described in Figure 5.1.

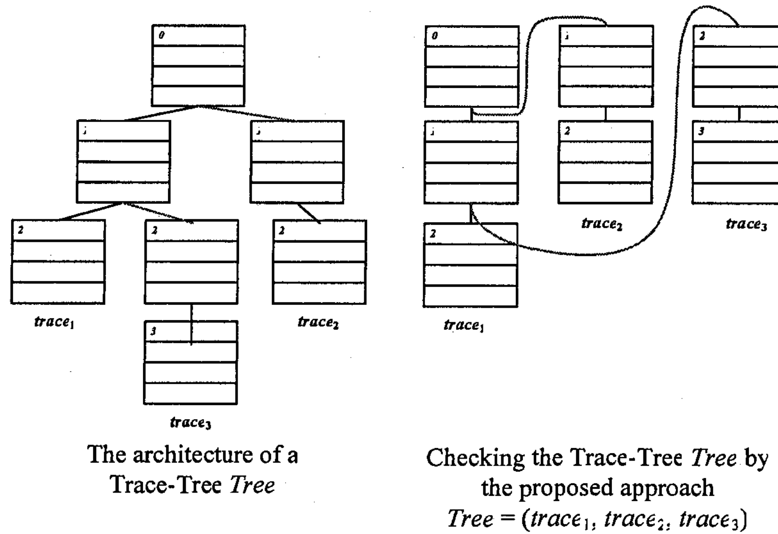

Figure 5.1 The architecture of a Trace-Tree and its representation

Within algorithm *Check_Trace*, two important steps are control portion fault detection and data portion fault detection, which correspond to the functions *control_portion_checking* and *evaluate*, respectively. When searching for possible transitions, these two steps are applied to each $t \in T_s$ which is the set of all transitions of M starting at a state s.

In the control portion fault detection, we match a transition's ending state and its I/O event by the same procedure as that of FSM-based passive fault detection in Chapter 4.

In the data portion fault detection, first, function *replace* is applied to replace the local variables of $\vec{y}$ in predicate $t.P(\vec{x}, \vec{y})$ by the actual values of the parameters of the observed I/O event e and to transform the predicate into a list of constraints stored in

newCS. Then, function *evaluate* is applied to detect possible inconsistency among constraints using the parameter values of the observed I/O event *e*. The checking procedure of function *evaluate* can be formalized as follows:

Given: (1) a configuration *c*, in which $c.[\vec{x}]$ contains a vector of intervals representing the ranges of possible values of global variables and *c.CS* stores existing constraints on $\vec{x}$; and (2) a set *newCS* of new constraints, which is generated from *t.predicate*($\vec{x}, \vec{y}$), function *evaluate* attempts to find a *solution* in $c.[\vec{x}]$ for both *c.CS* and *newCS* which is a combination of values in $c.[\vec{x}]$ that satisfies all the constraints.

More specifically, function *evaluate* can be abstracted as a solver for the following *Constraint Satisfaction Problem* (CSP):

Given: (1) a vector of intervals $[\vec{x}]$, which represent the ranges of possible values of global variables;

(2) a set of existing constraints *CS*;

(3) a set of new constraints *newCS*;

Determine: if there exist at least one combination of values in $[\vec{x}]$ that satisfies the existing constraints in *CS* and new constraints in *newCS* simultaneously.

If there exists a solution, the predicate *t.predicate*($\vec{x}, \vec{y}$) will be considered consistent with the configuration *c*; if no solution exists, the function *evaluate* will report FALSE, which means an inconsistency has been detected.

In Section 5.2, we implement function *evaluate* with the Interval Refinement method, which has been applied for data portion fault detection in [Lee02]. However, because of the dependency problem, the results of Interval Refinement method may not be accurate. Therefore, in Section 5.3, we discuss implementing function *evaluate* with the Simplex

method, which is accurate but slower than the Interval Refinement method. To combine the advantages of both Interval Refinement method and Simplex method, in Section 5.4, we propose the Hybrid method, which is accurate as the Simplex method and requires similar time cost as the Interval Refinement method.

If function *evaluate* returns TRUE, it means that t is executable and the configuration c has been modified within function *evaluate* by adding *newCS* to *c.CS* and updating the intervals in $c.[\vec{x}]$. The modified configuration c is added into the set C. After evaluating all the transitions in T_s, we continue to perform actions by function *action* on the configurations in C with their corresponding transitions in T_s.

After performing actions, we add the first configuration in C at the end of current trace *trace* and continue to check *trace* starting from this configuration. Other configurations in C will be considered as the initial configuration of new branches, which are represented as new traces in the Trace-Tree (see Figure 5.1).

5.1.4 Implementation of the Function *action*

When a transition has been evaluated to be executable, a new configuration will be constructed to record the status of SUT N after this transition. The construction of a new configuration depends on the *action* part, $A(\vec{x}, \vec{y})$, in the transition which consists of a sequence of assignments.

Given a configuration c standing for the current status of SUT N and a transition t which has been determined to be executable, function *action* performs the actions associated with t, and builds a new configuration c_{next} which stands for the status of SUT N after t. The details of algorithm *action* are presented as follows:

Algorithm *action*
Given: a transition t,
an observed I/O event e, and
the current configuration c,
Return: a new configuration c_{next}, or // the configuration after transition t
NULL // construction failed
Begin:
local_var ← set the values of the set of local variables according to e;
$c_{next} \leftarrow c$;
$assignments \leftarrow t.A(\vec{x}, \vec{y})$; // put the assignments in $t.A(\vec{x}, \vec{y})$ into a vector
while (*assignments* is not an empty sequence)
$a \leftarrow$ remove(a, *assignments*); // pick the first assignment from *assignments*;
replace the local variables in a using *local_var*; // the first step
if ($a.LHV$ is a local variable) // the second step
$q \leftarrow$ find the index of variable $a.LHV$ in *local_vars*;
$local_vars[q] \leftarrow a.RHE$; // replace by the value of RHE
else
$q \leftarrow$ find the index of variable $a.LHV$ in $c.[\vec{x}]$;
$[x_q] \leftarrow R(a.RHE)_{[\vec{x}]}$; // update the interval of $a.LHV$ in $[\vec{x}]$
if ($a.LHV$ appears in $a.RHE$)
for every constraint cs in $c_{next}.CS$ that contains $a.LHV$
replace the $a.LHV$ in cs by $(a.LHV - \sum_{i=1, i \neq q}^{k} a_i x_i) / a_q$;
endfor
else // if $a.LHV$ does not appear in $a.RHE$
for every constraint cs in $c_{next}.CS$ that contains $a.LHV$
replace the variable $a.LHV$ in cs with $[x_q]$;
change a to a new constraint cs';
$c_{next}.CS \leftarrow c_{next}.CS \cup cs'$; // add this new constraint
endfor
endwhile
return (c_{next});
End

The first step is to simplify the right hand expression (*RHE*) of assignments, which is the expression at the right hand of ":=". Considering the definition of *assignment* in Section 2.2, an assignment is in the form of:

LHV := INTEGER

| *Terms* + INTEGER

As *Terms* contains variables which can be either global variables or local variables, the

above *RHE* can be extended to be:

$$
\begin{aligned}
LHV := \quad & \text{INTEGER} && (1)\\
& \mid f(\vec{x}) + \text{INTEGER} && (2)\\
& \mid g(\vec{y}) + \text{INTEGER} && (3)\\
& \mid f(\vec{x}) + g(\vec{y}) + \text{INTEGER} && (4)
\end{aligned}
$$

The $f(\vec{x})$ stands for the *Terms* with global variables, and $g(\vec{y})$ stands for the *Terms* with local variables.

In the first step, we replace the local variables in (3) and (4) with their values. After the replacement, the expression of the assignment will be either (1) or (2) without local variables, which is $LHV := \sum_{i=1}^{k} a_i x_i$. Let *RHE* denotes $\sum_{i=1}^{k} a_i x_i$.

After the replacement, the *RHE* without local variables is used to update the value of *LHV* in the configuration *c*. If the *LHV* is a local variable, we use the value of *RHE* in the assignment to replace the existing value of this local variable. If the *LHV* is a global variable, we first replace the interval of *LHV* in $c.[\vec{x}]$ by the interval $R(RHE)_{[\vec{x}]}$, then update the constraints containing *LHV* in *c.CS*.

If the *LHV* appears in the *RHE*, for every constraint *cs* in *c.CS* that contains *LHV*, replace the *LHV* in *cs* by $(a.LHV - \sum_{i=1, i \neq q}^{k} a_i x_i) / a_q$; if the *LHV* does not appear in the *RHE*, for every constraint *cs* in *c.CS* that contains *LHV*, replace the appearances of the *LHV* with $[x_q]$ and add assignment to *c.CS* as a new constraint. For example, the assignment "$x_1 := x_2 + x_3 - 3$" can be changed to a constraint as "$x_2 + x_3 - x_1 = 3$".

The implementation of function *action* is similar to that of the actions part in EEFSM [Lee02] with the following differences:

First, [Lee02] only considered the situation where the *LHV* is a global variable, while we discuss the situations where the *LHV* could be either a global or a local variable. Secondly, in [Lee02], in the situation where the *LHV* does not appear in the *RHE*, all the constraints in $c_{current}.CS$ containing the *LHV* will be discarded. However, those discarded constraints may contain constraints on not only *LHV* but also other global variables. Considering this shortcoming, we keep those constraints and replace the *LHV* in them by the interval of the LHV in $[\vec{x}]$.

5.1.5 Optimization on Constraints

In the function *evaluate* and function *action*, evaluating and keeping the constraints are complex and time consuming. In order to reduce the complexity, we optimize the constraint related operations as follows:

First, throughout the passive fault detection, the values of global variables are represented by intervals. For a variable $x_i = [\underline{x_i}, \overline{x_i}]$, if its lower bound is equal to its higher bound (i.e. $\underline{x_i} = \overline{x_i}$), [Lee02] considers the value of variable x_i as a *determined value*. Whenever the value of a global variable is determined, [Lee02] replaces this variable in constraints with its determined value. For example, given the variable $x_1 = [1, 1]$ and a constraint cs: $x_1 + x_2 - x_3 = [-1, 5]$, x_1 in cs can be replaced by 1. Therefore, the new constraint after replacement would be cs: $x_2 - x_3 = [-2, 4]$. We adopt this replacement strategy in our approach.

Second, consider the situation in which a new constraint cs contains a single variable in the expression, for example $x_1 \leq 8$. It would be unnecessary to check cs with former

constraints in $c.CS$ and keep it in $c.CS$. Instead, we use cs to directly narrow the interval of this variable in $[\vec{x}]$. For example, given the existing interval of x_1 in $[\vec{x}]$ as $x_1 = [0, 20]$, and a new constraint cs as $x_1 \leq 8$, the narrowed interval is $x_1 = [0, 8]$. If the narrowed interval is not empty, we use the narrowed interval to replace the existing interval in $[\vec{x}]$. Otherwise, we report that an inconsistency is found.

Third, when searching for the compatible trace of E, a transition t in M may be encountered more than once, i.e. the observed I/O event e_i and e_k $(i \neq k)$ in E may correspond to the same transition t in M. In this case, we may have two constraints cs_1: $\sum_{i=1}^{k} a_i x_i = I_1$ and cs_2: $\sum_{i=1}^{k} b_i x_i = I_2$ such that $\forall i$, $1 \leq i \leq k$, $a_i = z \times b_i$ where z is a constant. We will call cs_1 and cs_2 *similar*. For example, $x_1 + x_2 = [1, 2]$ and $3x_1 + 3x_2 = [0, 9]$ are similar.

Then, given a new constraint cs, if there is a constraint within $c_{current}.CS$ that is similar to cs, we can reduce the number of constraints that needs to be checked by function *evaluate*. In order to determine whether there is a constraint cs' in current CS that is similar to cs, we apply the following algorithm *Similarity_Checking* before checking cs with the function *evaluate*.

Algorithm *Similarity_Checking*

Given: a new constraint cs ($\sum_{i=1}^{k} a_i x_i = I_1$), and

a set of existing constraints CS

Return: FALSE, or // inconsistency detected

TRUE // no inconsistency detected

Begin:

for each constraint cs' in CS // cs' : $\sum_{i=1}^{k} b_i x_i = I_2$

if (cs and cs' are similar)

$z \leftarrow a_i / b_i$;

```
            cs.I ← (cs'.I × z) ∩ cs.I;
            if (cs.I = ∅)                       // cs is inconsistent with cs'
                return (FALSE);
            else
                cs'.I ← cs.I;                   // cs is consistent with cs'
                return (TRUE);
    endfor
    return (TRUE);                              // no inconsistency is found by cs
End
```

If a constraint *cs'* similar to *cs* is found, we replace the interval of constraint *cs'.I* by the ($cs'.I \times z) \cap cs.I$. Thus, by applying algorithm *Similarity_Checking*, we can reduce the number of constraints that need to be checked by function *evaluate*.

5.2 Function *evaluate* with the Interval Refinement Method

To implement function *evaluate* in algorithm *Check_Trace*, first of all, we apply the Interval Refinement method, which has been used in [Lee02]. The interval arithmetic operations were introduced by R.E. Moore in [Moo66]. In the Interval Refinement method, the interval arithmetic operations are applied to narrow the intervals of variables in constraints. (The operations used in this thesis are introduced in Table 5.1) During the refinement, if the interval of a variable becomes empty, it means an inconsistency has been detected.

$$X+Y=[\underline{X}+\underline{Y},\overline{X}+\overline{Y}]$$

$$X-Y=[\underline{X}-\overline{Y},\overline{X}-\underline{Y}]$$

$$\mathrm{a}\times \mathrm{X}=[\mathrm{a}\times\underline{X},\mathrm{a}\times\overline{X}]$$

$$X*Y=\begin{cases}[\underline{X}\times\underline{Y},\overline{X}\times\overline{Y}] & \text{if } \underline{X}\geq 0 \text{ and } \underline{Y}\geq 0\\ [\overline{X}\times\underline{Y},\overline{X}\times\overline{Y}] & \text{if } \underline{X}\geq 0 \text{ and } \underline{Y}<0<\overline{Y}\\ [\overline{X}\times\underline{Y},\underline{X}\times\underline{Y}] & \text{if } \underline{X}\geq 0 \text{ and } \overline{Y}\leq 0\\ [\underline{X}\times\overline{Y},\overline{X}\times\underline{Y}] & \text{if } \underline{X}<0<\overline{X} \text{ and } \underline{Y}\geq 0\\ [\overline{X}\times\underline{Y},\underline{X}\times\underline{Y}] & \text{if } \underline{X}<0<\overline{X} \text{ and } \overline{Y}\leq 0\\ [\underline{X}\times\overline{Y},\overline{X}\times\underline{Y}] & \text{if } \overline{X}\leq 0 \text{ and } \underline{Y}\geq 0\\ [\underline{X}\times\overline{Y},\underline{X}\times\underline{Y}] & \text{if } \overline{X}\leq 0 \text{ and } \underline{Y}<0<\overline{Y}\\ [\overline{X}\times\overline{Y},\underline{X}\times\underline{Y}] & \text{if } \overline{X}\leq 0 \text{ and } \overline{Y}\leq 0\\ [\min(\overline{X}\times\underline{Y},\underline{X}\times\overline{Y}),\max(\underline{X}\times\underline{Y},\overline{X}\times\overline{Y})] & \text{if } \underline{X}<0<\overline{X} \text{ and } \underline{Y}<0<\overline{Y}\end{cases}$$

$$X\cap Y=\begin{cases}[\max(\underline{X},\underline{Y}),\min(\overline{X},\overline{Y})]\\ \varnothing, \quad \text{if } \max(\underline{X},\underline{Y})>\min(\overline{X},\overline{Y})\end{cases}$$

Table 5.1 Definitions of basic arithmetic operations [Han04]

(*X* and *Y* are interval numbers, a is a positive integer)

5.2.1 Definitions

Before discussing the Interval Refinement method, three definitions are needed.

Definition 11:

1. Let $\underline{x_i}$, $\overline{x_i}$, and [$\underline{x_i}$, $\overline{x_i}$] stand for the *lower bound*, the *upper bound*, and the *interval* of variable x_i, respectively, which means all the possible values x_i can take

are less than or equal to $\overline{x_i}$ and greater than or equal to $\underline{x_i}$;

2. Let $R(\sum_{i=1}^{k} a_i x_i)_{[\vec{x}]}$ denote the interval of expression $\sum_{i=1}^{k} a_i x_i$ on $[\vec{x}]$, which stands for all the possible values the expression $\sum_{i=1}^{k} a_i x_i$ can take on $[\vec{x}]$, i.e.

$$R(\sum_{i=1}^{k} a_i x_i)_{[\vec{x}]} = [\sum_{i=1}^{k} \min(a_i \underline{x_i}, a_i \overline{x_i}),\ \sum_{i=1}^{k} \max(a_i \underline{x_i}, a_i \overline{x_i})],\ x_i = [\underline{x_i},\ \overline{x_i}],\ 1 \le i \le k;$$

3. *Propagation* is defined as a set of refinements on all variables in $\vec{x}$. Given a constraint in the form of $\sum_{i=1}^{k} a_i x_i = I$, define the refinement on each variable x_j, $j = 1,\ldots, k$, as

$$[x_j]' = \begin{cases} [x_j], \text{ if } R(\sum_{i=1}^{k} a_i x_i)_{[\vec{x}]} \subseteq I & (1) \\ \varnothing, \text{ if } R(\sum_{i=1}^{k} a_i x_i)_{[\vec{x}]} \cap I = \varnothing & (2) \\ ((I - R(\sum_{i=1, i \ne j}^{k} a_i x_i)_{[\vec{x}]}) / a_j) \cap [x_j], \text{ otherwise} & (3) \end{cases}$$ □

The propagation describes how the constraint is applied to narrow the intervals of variables in $\vec{x}$. In case (1), the interval of variable x_j remains the same; in case (2), the interval of variable x_j will be empty, which means there is no solution for $\sum_{i=1}^{k} a_i x_i = I$ on $[\vec{x}]$; in case (3), the intersection will be taken as the new interval of variable x_j.

According to the definition, algorithm *Propagation* is presented as follows.

Algorithm *Propagation*

Given: a new constraint *cs*: $\sum_{i=1}^{k} a_i x_i = I$, and

a vector of intervals $[\vec{x}]$,

Return: FALSE, or // inconsistency detected

TRUE; // no inconsistency detected

Begin

$I' \leftarrow I \cap R(\sum_{i=1}^{k} a_i x_i)_{[\vec{x}]}$;

if ($I' = \varnothing$)

  return (FALSE); // inconsistency found

else

  $I \leftarrow I'$;

  $j \leftarrow 1$;

  while ($j \leq k$ and $a_j \neq 0$) // update the variables in $\vec{x}$

    $[x_j]' \leftarrow ((I - R(\sum_{i=1, i \neq j}^{k} a_i x_i)_{[\vec{x}]}) / a_i) \cap [x_j]$;

    if ($[x_j]' = \varnothing$)

      return (FALSE);

    else

      $[x_j] \leftarrow [x_j]'$;

  endwhile

  return (TRUE) // no inconsistency found

End

In case (2) of propagation, we consider the former vector of intervals $[\vec{x}]$ and the given constraint *cs* are inconsistent, thus this algorithm will return FALSE; while in case (1) and (3), the former intervals $[\vec{x}]$ and the given constraint are considered consistent, as a result, the algorithm will return TRUE.

5.2.2 Algorithm Interval_Refinement

Function *evaluate* is implemented using the Interval Refinement method by algorithm *Interval_Refinement* which is presented as follows.

Algorithm *Interval_Refinement*

Given: a vector of intervals $[\vec{x}]$,
a set of existing constraints CS, and
a set of new constraints $newCS$

Return: FALSE, or // inconsistency detected
TRUE // no inconsistency detected

```
Begin
    while (newCS ≠ ∅)
        CS' ← CS;
        [x]' ← [x];
        Select a constraint cs from newCS;  // cs is in the form of Σ_{i=1}^{k} a_i x_i = I
        newCS ← newCS \ {cs};
        I' ← I ∩ R(Σ_{i=1}^{k} a_i x_i)_[x];  // get the intersection
        if (I' = ∅)                   // case (2) in propagation
            return (FALSE);               // inconsistency detected
        if (I' = I)                   // case (1) in propagation
            CS ← CS ∪ cs;                 // add the new constraint cs to CS
        if (I' ≠ I)                   // case (3) in propagation
            I ← I';                       // update the new constraint cs
            if (Propagation(cs, [x]') = FALSE);
                return (FALSE);           // inconsistency detected
            else
                CS' ← CS' ∪ cs;           // add the new constraint cs to CS'
            while (any variable x_i in [x]' is changed);
                for every constraint cs' in CS' that contains x_i
                    if (Propagation(cs', [x]') = FALSE);
                        return (FALSE);
                endfor
            endwhile
            CS ← CS'; // update CS
            [x] ← [x]';                   // update [x]
    endwhile
    return (TRUE)    ;                // no inconsistency detected
End
```

In algorithm *Interval_Refinement*, the intersection between I and $R(\sum_{i=1}^{k} a_i x_i)_{[\vec{x}]}$ is first checked to decide if $[\vec{x}]$ is consistent with cs. If no inconsistency found, $[\vec{x}]$ will be updated based on the new constraint cs and cs will be added into the set CS. Moreover, if

any interval within $[\vec{x}]$ is changed in the updating process, the *Propagation* will be triggered to update all the existing constraints with changed variables in the set *CS*. The iteration of *Propagation* will stop when no interval within $[\vec{x}]$ is changed after previous *Propagation*. The result of algorithm Interval_Refinement could be either FALSE, which means inconsistency found, or TRUE, which means no inconsistency found and not only $[\vec{x}]$ but also *CS* have been updated.

When estimating the computational complexity of the algorithm *Interval_Refinement*, we count the number of calls to *Propagation*. *Propagation* reduces the length of intervals in $[\vec{x}]$ (i.e. the length of $[x_i]$ is $\overline{x_i} - \underline{x_i}$). Therefore, in the worst case, the number of calls to *Propagation* is proportional to the total length of all the intervals in $[\vec{x}]$. However, from practical experiments reported in [Lee06], the actual computational complexity is much lower than the worst case when it is applied in passive fault detection. For example, when the SUT being tested is a communications protocol, such as OSPF and TCP, the number of iterations for a new constraint normally is no more than 3 (see [Lee06]).

5.2.3 Experiments

Experiments have been carried out by applying the proposed approach for EFSM-based passive fault detection on an Automatic Teller Machine (ATM) system.

5.2.3.1 Experiment Setup

The specification of this ATM system is modeled as an EFSM then transferred to an SEFSM *ATM* (see Figure 5.2).

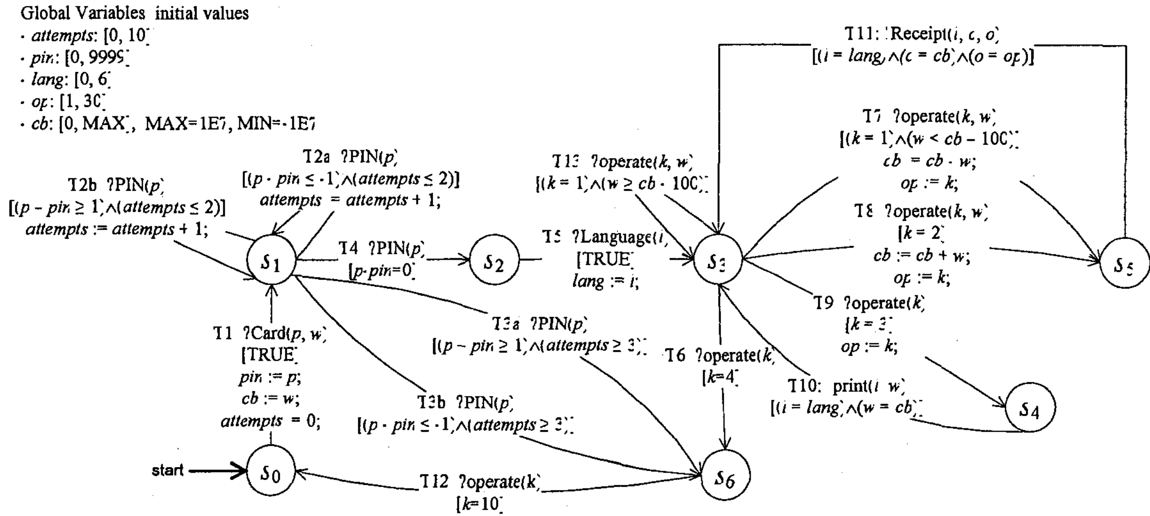

Figure 5.2 The SEFSM *ATM* for an ATM system

Within the SEFSM *ATM*, we define five global variables, i.e. $\vec{x} = (attempts, pin, lang, op, cb)$; seven states, i.e. $S_0 = \{s_0, s_1 \ldots s_6\}$; and fifteen transitions. Local variables are defined within transitions. Before the passive fault detection, each global variable is assigned an interval standing for its initial values. S_0 is determined by the tester according to the specific application at hand. In this experiment, S_0 is chosen to be equal to S.

In the experiment, we considered two cases. In Case I, called *correct implementation*, there is at least one trace in M that is compatible with E and this compatible trace is expected to be reported. In this case, we randomly generate a sequence E_s of observed I/O events ($| E_s | = 1000$) based on the SEFSM *ATM* and starting from state s_0. Within E_s, we randomly select five sequences with length of 20, 50, 100, 200, and 500.

In Case II, called *faulty implementation*, there is no trace in M that is compatible with E and "faulty" is expected to be reported. First, we create a faulty specification *ATM′* from *ATM* by altering the next state, expanding a constraint in the predicate, or narrowing a constraint in the predicate of a randomly selected transition. Then, we randomly

generate a sequence E_s of observed I/O events ($| E_s | = 1000$) based on the SEFSM *ATM′* and starting from state s_0. Within E_s, we randomly select ten sequences containing the altered transition with length of 30 observed I/O events.

Then, we apply the proposed approach for EFSM-based passive fault detection in these two cases and record the results.

5.2.3.2 Results of the Experiment

Table 5.2 and Table 5.3 show the time cost (in milliseconds) for the proposed approach applied to SEFSM *ATM*, subset S_0, and sequence E from its implementation SEFSM N which is *ATM* in Case I and *ATM′* in Case II.

Table 5.2 The results of Case I by the Interval Refinement method

$\|E\|$	Time Cost (ms)					Average Time Cost (ms)
20	63	47	63	47	78	59
50	94	94	110	125	79	100
100	125	109	109	110	125	115
200	172	125	125	125	140	137
500	187	187	203	188	204	193

Table 5.3 The results of Case II by the Interval Refinement method

Type	Time Cost (ms)									
Next State Fault	47	63	47	31	93	47	47	62	94	78
Expended Constraint Fault	78	78	63	94	109	62	78	63	47	63

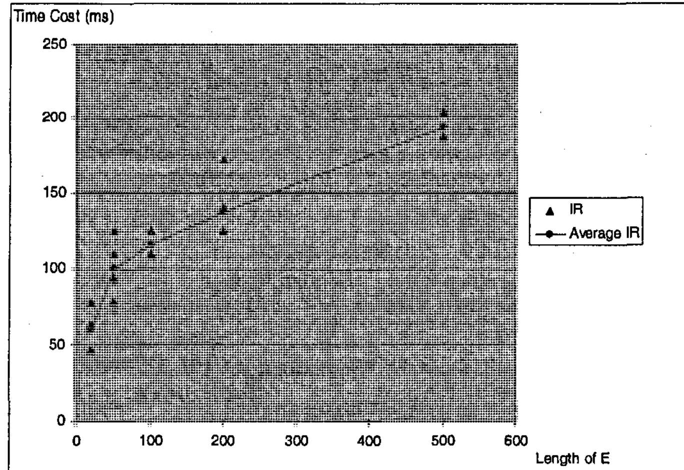

Figure 5.3 The results of Case I by applying the Interval Refinement (IR) method

In Case I, the proposed approach successfully finds the corresponding traces. The results in Table 5.2 also show that the time cost increases as the length of sequence E of observed I/O events increases.

In Case II with next state fault and expanded constraint fault, the proposed approach reports fault correctly, see Table 5.3.

The results of experiments with narrowed constraint fault are not listed because this type of fault cannot be detected by the proposed approach (by [Lee02] as well). Because, an observed I/O event generated by narrowed constraint will certainly satisfy the original constraint, thus it cannot be detected.

5.2.4 Dependency Problem

The Interval Refinement method works well in the experiments presented above. But, its accuracy may suffer from the *dependency problem.* When a given variable occurs more than once in an interval computation, it will be treated as a different variable in each occurrence. This causes widening of computed intervals and makes it more difficult to

obtain accurate results. This property of interval arithmetic is called the *dependency problem*. [Han04]

In our application, the dependency problem may mislead our judgment, especially when the values of variables are integers. For example: assume a configuration c with

$c.[\vec{x}]$: $x_1 = [1, 2]$, $x_2 = [1, 2]$, x_1 and x_2 are integers;

$c.CS$: $\{cs: x_1 - x_2 = 0;\}$, and

check two transitions

t_1 with a constraint cs_1 in its predicate as: $x_1 + x_2 = 3$;

t_2 with a constraint cs_2 in its predicate as: $x_1 + x_2 \leq 4$;

By applying the Interval Refinement method, both transition t_1 and t_2 will be judged as executable, even though t_1 is actually not executable, because x_1 and x_2 are integers and there is no solution for both cs and cs_1 at the same time. Therefore, if the Interval Refinement method is applied to our approach, the dependency problem may cause the transition t_1 to be falsely reported as executable.

To guarantee the correctness of results, we attempt implementing function *evaluate* with the Simplex method.

5.3 Function *evaluate* with the Simplex Method

The Simplex method is a CSP solver, which has been widely used in practice for optimization problems for linear real arithmetic constraints [Mar98]. Given an objective function, as a linear expression, and a set of linear constraints, the Simplex method searches the solution space of these constraints for a solution that can minimize or maximize the value of the objective function.

The main idea of the Simplex method is that, in a CSP, if a solution exists, it must be in a polygon confined by the constraints. And the extreme value for the objective function will always be obtained at the vertex of this polygon. Thus, the Simplex method checks all the vertices in order to find a suitable solution for the objective function. If no solution is found, it determines there is a conflict.

As an example, consider

a vector of intervals $[\vec{x}]$: $x_1 = [0, 10], x_2 = [0, 10]$;

a set of constraints,

$cs_1 : (x_1 + x_2 \leq 8)$;

$cs_2 : (x_1 + x_2 \geq 2)$;

$cs_3 : (x_1 - x_2 \leq 8)$;

$cs_4 : (x_1 - x_2 \geq 4)$;

$cs_5 : (x_1 \geq 6)$;

and an objective function: minimize$(x_1 + x_2)$; as a CSP.

By applying the Simplex method on this CSP, the solution found for the objective function is $(x_1 = 6, x_2 = 0)$, as presented in Figure 5.4.

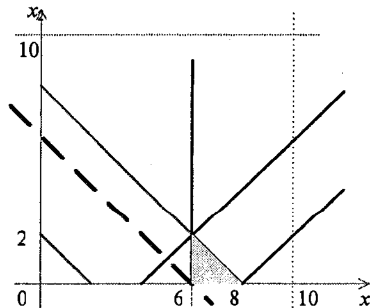

Figure 5.4 Solution of the CSP by the Simplex method

In Figure 5.4, the dashed line stands for the objective function and the real lines stand for

the constraints. All the nodes within the shadowed area satisfy the set of constraints. The solution found (x_1 =6, x_2 = 0) is at a vertex of this shadowed area.

5.3.1 Algorithm Simplex

For the implementation of function *evaluate*, the Simplex method can be used to overcome the shortcoming of the Interval Refinement method since the constraints in SEFSM model are all linear constraints which consist of linear equations or inequalities. To this end, we adopt an open source tool *lp_solve* as the implementation of the Simplex method. *lp_solve* is a free linear programming solver based on the revised Simplex method and the Branch-and-bound method [Lpsol]. The algorithm *Simplex* presented below implements *evaluate*(*c*.[$\vec{x}$], *c.CS*, *newCS*) using *lp_solve*.

Algorithm ***Simplex***
Given: a vector of intervals [$\vec{x}$],
        a set of existing constraints *CS*, and
        a set of new constraints *newCS*
Return: FALSE, or   // inconsistency detected
        TRUE   // no inconsistency detected
Begin
    $CS' \leftarrow CS \cup newCS$;
    if (*lp_solve.evaluate*([$\vec{x}$], *CS'*) = FALSE)
        return (FALSE); // inconsistency detected
    else
        $CS \leftarrow CS'$;   // no inconsistency detected
        return (TRUE);
End

First, the new constraints in the predicate are added to the set of existing constraints; then, the tool *lp_solve* is used to check if a solution exists for the combination of constraints. Unlike the Interval Refinement method, the Simplex method is a global optimization method, which means that both the new constraints and the existing constraints in *CS* will be considered at the same time.

lp_solve has also been used for EFSM-based passive fault detection in [Lee06] and [Che03a]. But they considered the problem of detecting possible inconsistency among constraints as a Mixed Integer Linear Programming problem and chose *lp_solve* as an implementation of Branch-and-Bound method for solving this problem. Branch-and-Bound is a general algorithmic method for finding optimal solutions of various optimization problems [Apt03]. However, in our context, the emphasis is to find out whether one solution exists, not to compare and find optimal solutions.

The mechanism of the Simplex method is more complex than the Interval Refinement method. Its worst case computational complexity is exponential. But [Lee06] shows that the average complexity of applying the Simplex method is polynomial. [Spi04] theoretically explains why it is usually polynomial.

5.3.2 Experiments and Comparisons

We conducted experiments to compare the time costs of the Simplex method and the Interval Refinement method under the same conditions. The experiment setup used for the application of the Interval Refinement method given in Section 5.2.3 is also used for the application of the Simplex method. The results of applying the Simplex method are stated in Table 5.4 and Table 5.5. Figure 5.5 depicts the relative costs of applying these two methods.

Table 5.4 The results of Case I by the Simplex method

$\|E\|$	Time Cost (ms)					Average Time Cost (ms)
20	203	219	250	188	250	222
50	468	422	453	500	390	446
100	625	657	625	719	718	668
200	1391	1281	1266	1344	1312	1318
500	2937	3625	3047	2906	3000	3103

Table 5.5 The results of Case II by the Simplex method

Type	Time Cost (ms)									
Next State Fault	141	140	125	125	312	172	265	156	250	187
Expended Constraint Fault	234	235	203	344	516	219	171	203	172	156

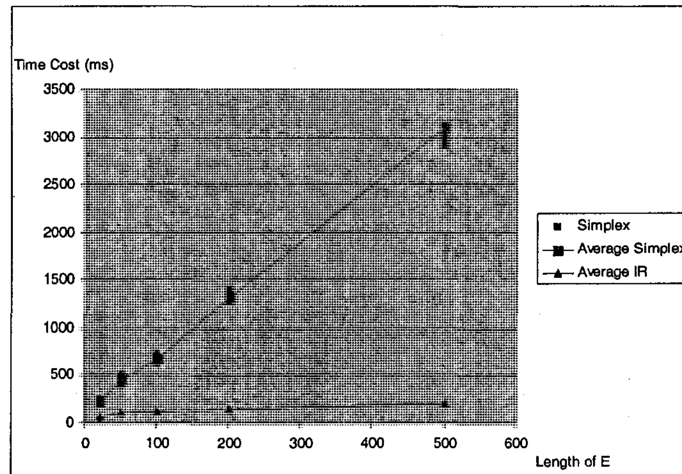

Figure 5.5 The results of Case I by applying the Simplex method and the Interval Refinement (IR) method

It can be seen in Figure 5.5 that the time cost of the Simplex method is much more than that of the Interval Refinement. As the length of *E* increases, the time cost of the Simplex

method increases faster than that of the Interval Refinement method.

The Simplex method guarantees the correctness of results, but is more time-consuming than the Interval Refinement method according to the results of experiments. The Hybrid method we propose in the next section combines the advantages of both the Simplex method and the Interval Refinement method.

5.4 The Hybrid Method

In the Hybrid method, we apply both the Interval Refinement method and the Simplex method for the data portion fault detection. The Simplex method is applied to confirm the results generated by the Interval Refinement method to eliminate the negative impact caused by the dependency problem.

The idea of the Hybrid method is: given a set T of candidate transitions, the current configuration $c_{current}$, and an observed I/O event e, the Interval Refinement method is first used to decide which transitions in T are executable. If only one transition is evaluated to be executable, the passive fault detection will continue with this transition; if none is evaluated to be executable, the current trace will be determined to be not compatible with E; if more than one transition is evaluated to be executable, the Simplex method will be applied to check the correctness of these executable transitions. At the end of one trace, before a trace is determined to be a compatible trace, we also use the Simplex method to confirm this result by checking if there exists inconsistency in the last configuration of this trace.

For example, if the Hybrid method is applied to the example in Section 5.2.4, in the first step, both transition t_1 and t_2 will be evaluated to be executable by the Interval

Refinement method; but in the second step, the inconsistency of t_1 will be revealed by the Simplex method.

If there is only one transition being evaluated to be executable, the Simplex method will not be applied after the application of the Interval Refinement method because this transition will be evaluated by the Simplex method implicitly by checking the last configuration of this trace. Consider a trace *trace* ($c_1c_2...c_k$, $k \leq |E|$) in the Trace-Tree *Tree*. If c_k is checked by the Simplex method and no inconsistency is found, *trace* is guaranteed to be compatible with a prefix of E ($e_1e_2...e_k$, $k \leq |E|$) because c_k contains all the constraints within the configurations from c_1 to c_{k-1}. Therefore, if no inconsistency found in the last configuration of a trace by the Simplex method, the transitions associated with *trace* should be all executable.

Additionally, in the Interval Refinement method, the function *Propagation* keeps narrowing the intervals; while in the Simplex method, the intervals will be untouched. So, in Hybrid method, the Interval Refinement method also helps to reduce the cost of applying the Simplex method by narrowing the intervals.

5.4.1 New Algorithm Check_Trace

To apply the Hybrid method in our problem, the algorithm *Check_Trace* needs to be modified. The new algorithm *Check_Trace* is presented as follows.

```
Algorithm Check_Trace
Given: an SEFSM M,
        a trace trace,
        a sequence E of observed I/O events, and
        a Trace-Tree Tree,
Return:  FALSE, or     // trace is not a compatible trace of E
         TRUE          // trace is a compatible trace of E
Begin:
  c_current ← trace.get(0); // get the first configuration
  while (c_current ≠ NULL and c_current.# ≠ E.#)   // if there is an observed I/O event
                                                   // to be checked
      s ← c_current.s_c ;
      T_s ← all transitions in M starting at s ;
      e ← E.get(c_current.# + 1);  // get the observed I/O event e
                                   // for evaluating transitions
      C ← ∅;
      for each transition t in T_s  // evaluate transitions
      c ← c_current;
          if (control_portion_checking(c, t, e) = FALSE)
              end the for loop;   // the control portion is inconsistent
          else                    // the data portion fault detection commences
              newCS ← replace(t.P(x⃗, y⃗), e);  //eliminate local
                                               // variables in the predicate
              if (Interval_Refinement(c.[x⃗], c.CS, newCS) = FALSE)
                  end the for loop;   // the data portion is inconsistent
              else
                  C ← C ∪ {c}; // c has been modified and needs to be added to C
      endfor
      if (C = ∅)                  // if no executable transition is found
          return (FALSE);
      else
          if (|C| > 1 or c_current.# + 1 = |E|) // checking by the Simplex method
              for each configuration c in C
                  if (Simplex(c.[x⃗], c.CS, ∅) = FALSE)
                      C ← C \ {c};
              endfor
          else
              continue;
      if (C = ∅)                  // if no configuration in C is consistent
          return (FALSE);
      else
          for each configuration c in C
              c ← action(t, e, c) ;   // perform actions associated with transitions
              if (c = NULL)
```

```
                end the for loop ;
            else
                if (c is the first configuration in C)
                    add c to trace;
                    c_current ← c;
                else
                    build a new trace branch_trace;
                    add c to branch_trace;     // create a new branch branch_trace
                    add branch_trace to tree;
        endfor
    endwhile
    return (TRUE);   // a compatible trace is found
End
```

The shadowed lines are the modified parts from the previous version of *Check_Trace*. As presented in this new algorithm, if there are more than one transition judged to be executable by the Interval Refinement method, or this is the end of one trace, the Simplex method will be applied to double-check the result of the Interval Refinement method.

5.4.2 Experiments and Comparisons

We conducted experiments to compare the time costs of the Hybrid method, the Interval Refinement method, and the Simplex method under the same conditions. The experiment setup used for the application of the Interval Refinement method given Section 5.2.3 is also used for the application of the Hybrid method. The results of applying the Hybrid method are stated in Table 5.6 and Table 5.7. Figure 5.6 depicts the relative costs of applying these three methods.

Table 5.6 The results of Case I by the Hybrid method

$|E|$	Time Cost (ms)					Average Time Cost (ms)
20	78	63	94	63	125	222
50	203	125	125	235	94	446.6
100	172	110	125	219	156	668.8
200	234	172	141	156	157	1318.8
500	281	234	312	218	265	3103

Table 5.7 The results of Case II by the Hybrid method

Type	Time Cost (ms)									
Next State Fault	109	63	47	47	62	63	78	62	110	46
Expended Constraint Fault	63	219	62	94	110	62	109	110	78	110

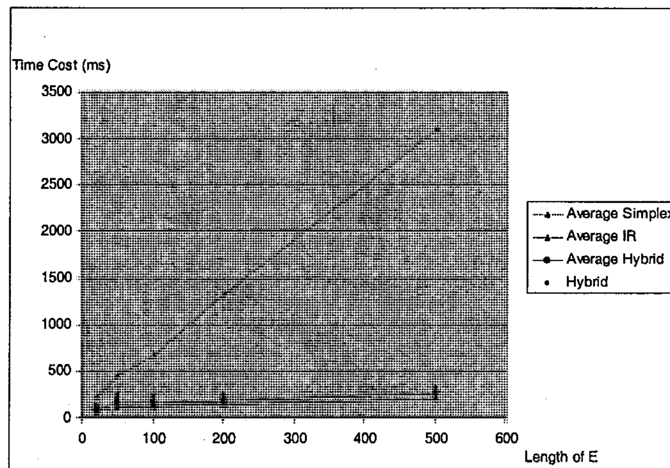

Figure 5.6 The results of Case I by applying the Hybrid method, the Interval Refinement method, and the Simplex method

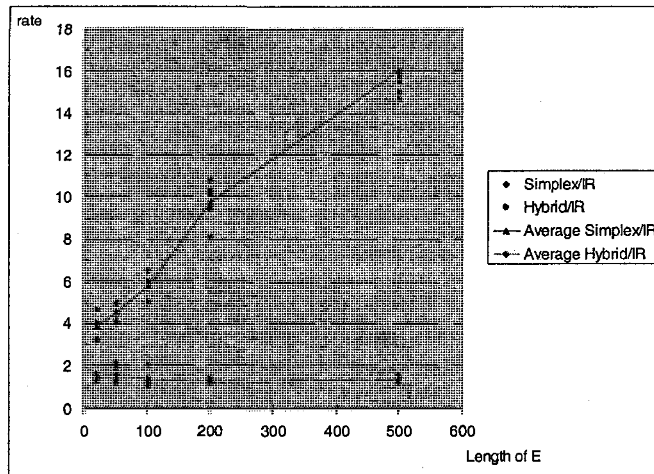

Figure 5.7 Rates of time cost with three methods

Figures 5.6 shows the comparison on the time cost of these three methods. According to the experiments, the Interval Refinement method requires the least amount of time; the Hybrid method requires a little bit more time than the Interval Refinement method; and the Simplex method is the most expensive in terms of time. As the length of sequence *E* of observed I/O events increases, the time consumed for these three methods all increases.

Moreover, to compare the rate of increase of time costs, along with the increase of $|E|$, we compute the rate of time costs between (1) Simplex method and Interval Refinement method (Simplex/IR in Figure 5.7); (2) Hybrid method and Interval Refinement method (Hybrid/IR in Figure 5.7). According to Figure 5.7, we see that (1) the time costs of the Interval Refinement method and Hybrid method are quite similar and, with the increase in the length of *E*, the difference between these two methods is not noticeable; (2) the time cost of Simplex method is much more than that of the Interval Refinement method, and as the length of *E* increases, the disparity between these two methods also increases.

Therefore, compared with the Interval Refinement method and the Simplex method, the Hybrid method guarantees the correctness of results with a reasonable time cost.

5.5 Conclusions

In this chapter, we proposed an approach for EFSM-based passive fault detection which is derived from the approach for FSM-based passive fault detection we have proposed earlier. This proposed approach provides information about possible starting state and possible trace at the end of the passive fault detection without post-processing.

In the implementation of this approach, we discussed three methods for the implementation of function *evaluate*. Among these three methods, the Interval Refinement method is fast but not accurate because of the dependency problem; the Simplex method is accurate but not efficient. The Hybrid method, which is proposed in this thesis, combines the use of both Interval Refinement and Simplex methods for performance improvement during the passive fault detection. Through experiments we show that, compared with using only the Interval Refinement method or only the Simplex method, the Hybrid method guarantees the correctness of results with a reasonable time cost.

Chapter 6

Summary and Conclusions

Passive fault detection is a fundamental part of passive testing which determines whether a system under test (SUT) is faulty by observing the I/O behaviors of the SUT without interfering with its normal operations.

In FSM and EFSM-based passive fault detection, the specification of an SUT N is modeled as an FSM or EFSM M, N is treated as a blackbox, and the tester wishes to determine whether N is faulty with respect to M by observing the I/O behaviors of N represented as a sequence Q of I/O pairs (FSM-based approach) or a sequence E of observed I/O events (EFSM-based approach). Within this context, the passive fault detection problem can be defined as follows: given an M and the sequence of I/O behaviors of N where the starting state (where the sequence of I/O behaviors starts) of N is unknown, determine whether N is faulty.

In the literature, most approaches for FSM and EFSM-based passive fault detection are derived from the approach for FSM-based passive fault detection in [Lee97], such as [Tab99] and [Lee02]. The approach in [Lee97] is comprehensive but has two limitations: i) inefficiency: in this approach, every state of M needs to be checked. However, the number of states compatible with Q is usually comparatively small and checking every state of M is usually unnecessary. ii) need for additional processing: [Lee97] only determines the set of possible current states when it terminates. Post-processing will still be needed to retrieve information about possible starting state and possible trace corresponding to Q.

In order to overcome the shortcomings of [Lee97], we propose an approach for FSM-based passive fault detection in Chapter 3. The main points of this approach are as follows: assume that the subset S_0 ($S_0 \subseteq S$) of states of M contains all possible starting states of the sequence of I/O behaviors. Randomly pick a state s in S_0 and determine whether the sequence of I/O behaviors is a trace of M at s; if s is compatible with the sequence of I/O behaviors, then stop and declare that the sequence of I/O behaviors is not sufficient to determine whether N is faulty. In this case, the sequence of I/O behaviors will be considered to be a trace of N starting from s and the current state of N can be determined readily. Otherwise, continue to check other states in S_0. After checking all states in S_0, if no state is found to be compatible with the sequence of I/O behaviors, then N will be declared faulty.

We prove that the proposed approach for FSM-based passive fault detection has better performance in computational complexity than the approach in [Lee97]. That is, the proposed approach performs better in situations where there is only one state in S_0 that is compatible with Q and performs the same in situations where there is no state in S_0 that is compatible with Q. Moreover, the proposed approach provides more information about possible starting state and possible trace corresponding to Q during passive fault detection and thus eliminates the need for post-processing.

In Chapter 5, we propose an approach for EFSM-based passive fault detection which is derived from the proposed approach for FSM-based passive fault detection in Chapter 4. This proposed approach provides information about possible starting state and possible trace at the end of the passive fault detection without post-processing. In the implementation of this approach, we discussed three methods for the implementation of

function *evaluate* and the Hybrid method we proposed is shown as the best choice through experiments and comparisons.

In future research, some model checking techniques can be adopted in the proposed approach in EFSM-based passive fault detection to help exploring the Trace-Tree. Also, it would be interesting to see how our proposed approach can help in solving the problems of fault location, fault identification, and fault diagnose.

Reference

[Alc04] B. Alcalde, A. Cavalli, D. Chen, D. Khuu and D. Lee, "Network Protocol System Passive Testing for Faulty Management - A Backward Checking Approach," *Proceedings of IFIP FORTE'04*, *LNCS*, vol.3235, pp.150-166, 2004.

[Alc06] B. Alcalde and A. Cavalli, "Parallel Passive Testing of System Protocols – Towards a Real-time Exhaustive Approach," *International Conference on Network, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies* (*ICN/ICONS/MCL'06*), pp.42-42, 2006.

[Apt03] K.R. Apt, *Principles of Constraint Programming*. Cambridge: Cambridge University Press, 2003.

[Arn04] J.A. Arnedo, A. Cavalli, and M. Nunez, "Fast Testing of Critical Properties through Passive Testing," *IFIP International Conference on Testing of Communicating Systems* (*TestCom'03*), *LNCS*, vol.2644, pp.295-310, 2004.

[Bay05] E. Bayse, A. Cavalli, M. Núñez, and F. Zaïdi, "A Passive Testing Approach based on Invariants: Application to the WAP," *Computer Networks and ISDN Systems*, vol.48, pp.247-266, 2005.

[Cav03] A. Cavalli, C. Gervy, S. Prokopenko, "New Approaches for Passive Testing Using an Extended Finite State Machine Specification," *Information and Software Technology*, vol.45, pp.837-852, 2003.

[Cav06] A. Cavalli and D. Vieira, "An Enhanced Passive Testing Approach for Network Protocols," *Proceedings of the International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies* (*ICN/ICONS/MCL'06*), pp. 169-169, 2006.

[Che03a] D. Chen, J. Wu, and T.L. Chu, "An Enhanced Passive Testing Tool for Network Protocols," *International Conference on Computer Networks and Mobile Computing* (*ICCNMC'03*), pp.513-516, 2003.

[DeG02] M.H. DeGroot and M.J. Schervish, *Probability and Statistics, 3rd Edition*. Boston: Addison-Wesley, 2002.

[Han04] E.Hansen and G.W. Walster, *Global Optimization Using Interval Analysis, 2nd Edition*. New York: Marcel Dekker Inc., 2004.

[Lee96] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines – a Survey," *Proceedings of the IEEE*, vol.84, pp.1090-1126, 1996.

[Lee97] D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla, and A. John, "Passive Testing and Applications to Network Management," *Proceedings of IEEE International Conference on Network Protocols* (*ICNP'97*), pp.113-122, 1997.

[Lee02] D. Lee, D. Chen, R. Hao, R.E. Miller, J. Wu, and X. Yin, "A Formal Approach for Passive Testing of Protocol Data Portions", *Proceedings of IEEE International Conference on Network Protocols* (*ICNP'02*), pp.122-131, 2002.

[Lee06] D. Lee, D. Chen, R. Hao, R.E. Miller, J. Wu and X. Yin, "Network Protocol System Monitoring – A Formal Approach with Passive Testing," *IEEE/ACM*

Transactions on Networking, vol.14, pp.424-437, 2006.

[Lpsol] Tool *lp_solve*, version 5.5.0.9, http://lpsolve.sourceforge.net/5.5/.

[Mar98] K. Marriott and P.J. Stuckey, *Programming with Constraints: An Introduction*. Cambridge, Mass.: MIT Press, 1998.

[Mil98] R.E. Miller, "Passive Testing of Networks Using a CFSM Specification," *IEEE International Performance, Computing and Communications Conference* (*IPCCC'98*), pp.111-116, 1998.

[Mil00] R.E. Miller and K.A. Arisha, "On Fault Location in Networks by Passive Testing," *IEEE International Performance, Computing and Communications Conference* (*IPCCC'00*), pp. 281-287, 2000.

[Mil01a] R.E. Miller and K.A. Arisha, "Fault Identification in Networks by Passive Testing," *Proceedings of 34th Annual Simulation Symposium*, pp.277-284, 2001.

[Mil01b] R.E. Miller and K.A. Arisha, "Fault Identification in Networks Using a CFSM Model by Passive Testing," *UMIACS-TR-2001-28*, technical report, 2001.

[Mil01c] R.E. Miller and K.A. Arisha, "Fault Coverage in Networks by Passive Testing," *International Conference on Internet Computing 2001*, pp. 413-419, 2001.

[Moo66] R.E. Moore, *Interval Analysis*. Englewood Cliffs, N.J.: Prentice-Hall Inc., 1966.

[Nea04] R. Neapolitan and K. Naimipour, *Foundations of Algorithms Using C++ Pseudocode, 3rd Edition*. Sudbury, Mass.: Jones & Bartlett Publishers, 2003.

[Spi04] D.A. Spielman and S.H. Teng, "Smoothed Analysis: Why the Simplex Algorithm Usually Takes Polynomial Time," *Journal of the ACM*, vol.51, pp.385-463, 2004.

[Tab99] M. Tabourier and A. Cavalli, "Passive Testing and Application to the GSM-MAP Protocol," *Information and Software Technology*, vol.41, pp.813-821, 1999.

[TSG94] T.M. Nguyen, *FSM-based Test Sequence Generator* (TSG), 1994. http://www.site.uottawa.ca/~ural/tsg/intro_fr.html

[Wu02] Y. Zhao, J. Wu, and X. Yin, "From Active to Passive: Progress in Testing of Internet Routing Protocols," *Journal of Computer Science and Technology*, vol.17, pp.264-283, 2002.

[Zha01] Y. Zhao, X. Yin, and J. Wu, "OnLine Test System, an Application of Passive Testing in Routing Protocols Test," *Proceedings of 9th IEEE International Conference on Networks*, pp.190-195, 2001.