# INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

# PROTOCOL RE-TESTING AND DIAGNOSTIC TESTING METHODS

by

Khaled Abdul-Ghani El-Fakih

A thesis submitted to the Faculty of Graduate and Post-Doctoral Studies in partial fulfilment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

Ottawa-Carleton Institute for Computer Science,
School of Information Technology and Engineering,
University of Ottawa,
Ottawa ON Canada

# Abstract

Many test selection methods have been developed for deriving tests when a system specification is represented in the form of a Finite State Machine (FSM). The purpose of these tests is to determine whether an implementation of the system conforms to (i.e. is correct with respect to) its specification. In realistic applications, maintaining a system modeled by a given specification machine involves modifying its specification as a result of changes in the user requirements. Testing the whole system implementation after each modification is considered expensive and time consuming. Therefore, it is important to generate tests that would only test the modified parts of the implementation that correspond to the modified parts of its specification. In the first part of this thesis, we present test generation methods that select tests for testing the modified parts of the system specification, in order to check that these modifications were correctly implemented in the system implementation. These methods are based on well-known test derivation methods called the W, Wp, HIS and distinguishing sequence methods.

As the purpose of conformance testing is to check whether an implementation conforms to its specification, an interesting complementary problem is to locate the differences between a specification and its implementation when the implementation is found to be nonconforming. In the second part of this thesis, we consider a system consisting of two communicating FSMs, called components. First, we show that it is not always possible to locate a fault within the given system, once a fault has been detected in its implementation (called System Under Test (SUT)). Accordingly, we present two new

two-level approaches for fault localization within the given system. The first method assumes that the SUT has a single fault in one of its components. Consequently, at the first diagnostic level the methods decide whether it is possible to identify the faulty component in the given system. If this is possible, the faulty component is identified, and if desired, at the second level, the methods determine whether it is possible to locate the fault within the faulty component. If that is possible, the methods provide additional test cases to locate the fault. The second method considers the case when the SUT may have multiple faults in at most one of its components. At the machine level diagnosis, the method decides whether it is possible to identify the faulty component machine in the given system, once faults have been detected in a system implementation. If this is possible, it provides tests for identifying the faulty component machine, and if desired, the method can be used to determine whether it is possible to locate the faults within the faulty component. If that is possible, he method provides additional test cases to locate the faults.

# Acknowledgments

A number of people have contributed to this work, either directly or indirectly, and I am pleased to thank them at this point. First, I would like to thank my supervisor Dr. Gregor v. Bochmann for his technical, moral, and financial support for my research work. I would like also to thank Dr. Nina Yevtushenko for her technical and moral help for this work. I have enjoyed and learned a lot from the inspiring discussions with Dr. Bochmann and Dr. Yevtushenko.

I would like to thank the members of my advisory committee, Dr. Hasan Ural and Dr. Danny Krizank, for their early suggestions on how to direct this work.

Many thanks also to Wissam Itani for implementing and experimenting with a fault diagnosis algorithm. Also, thanks for Hubert Garavel for his help is using the CADP tool used in implementing the algorithm.

I gratefully acknowledge the financial support by Communications and Information Technology Ontario (CITO), the Natural Sciences and Engineering Research Council (NSREC) Canada, and the University of Ottawa.

I am pleased to thank Dr. Nasha't Mansour, Khalil El-Khatib, and Gabriel Aghazarian for their continuous encouragement and moral support.

Last but not least, I wish to thank my parents, my wife May, and my sisters and their husbands for their love and for sharing the ups and downs of life.

# Table of Contents

# List of Definitions

# List of Figures

# List of Tables

# List of Symbols and Notations

"." : Denotes the *concatenation operation* (see Section 2.1)

$=$ : Equivalent machines or states (see Definition 2.6)( see Definition 2.8)

$\equiv_V$ : V-equivalent states (see Definition 2.6)

$\leq_{fsm}$ : Quasi-equivalent machines (see Definition 2.8)

$\Sigma^*$ : Denotes the set of all words constructed using an alphabet $\Sigma$ (see Section 2.1)

$|V|$ : Represents the cardinality of *set* $V$ (see Section 2.1)

"$\varepsilon$" : Denotes the empty word which consists of no elements at all (see Section 2.1)

$V^n$ : Denotes n-times concatenation of the set of words $V$ (see Section 2.1)

**DIS** : Defined input sequence (see Definition 2.2)

**DIS**($M_S|s_i$) : Set of all the defined input sequences at state $s_i$ of machine $M_S$

$TS$ : Test suite (see Definition 3.1)

m-complete : m-complete fault coverage (see Definition 3.4 )

$M_S$ : Specification FSM

$n$ : The number of states of the specification FSM $M_S$

$M_I$ : Implementation FSM

$m$ : The number of states of the implementation FSM $M_I$

$M_S'$ : Modified specification machine

$M_I'$ : Modified implementation machine

$Q$ : A state cover set of $M_S$ (see Definition 3.12)

$Q'$ : A state cover set of $M_S'$

$W$ : Characterization set $W$ (see Definition 3.8 )

$W_k$ : State identifier or separating set (see Definition 3.9)

$F$ : A family of state identifiers (or a separating family) (see Definition 3.9)

UIO : Simple or Unique Input/Output (UIO) Sequence (see Definition 3.10)

DS : Distinguishing Sequence (see Definition 3.11)

**Impl**($M_S$, $n$) : A set of all the completely defined deterministic machines with up to $n$ states and with the same I/O symbol set as $M_S$ (see Section 3.1.1)

$\overline{Q}$ : A state cover set of $M_S'$ for the case when $m > n$ (see Section 4.5.1)

$\overline{Q'}$ : A state cover set of $M_I'$ for the case when $m > n$ (see Section 4.5.1)

$Sub(A)$ : The set of all deterministic sub-machines of non-deterministic FSM $A$ (see Section 7.2)

$h^o(s,\alpha)$ : The output projection of the behavior function $h$ of an FSM (see Section 7.2)

$h^s(s,\alpha)$ : The state projection of the behavior function $h$ of an FSM (see Section 7.2)

$h^s{}_\gamma(s,\alpha)$ : The set of all states where the sequence $\alpha$ can take FSM $A$ from the initial state with the output response $\gamma$ (see Section 7.2)

*ComFSMs* : A system of two communicating FSMs

$RS = M_1 \Diamond M_2$ : Reference System of a system of two *ComFSMs* $M_1 \Diamond M_2$

$M_1$ : Embedded machine in a system of two *ComFSMs*

$M_2$ : Context machine in a system of two *ComFSMs*

$M'_1$ : Denote the implementation of $M_1$

$M'_2$ : Denotes the implementation of $M_2$

$\Diamond$ : Denotes the composition of two *ComFSMs*

$\times$ : Denotes the product of two *ComFSMs*

$X$ : The set of externally observable inputs of $M_1$

$Y$ : The set of externally observable outputs of $M_1$

$U$ : The set of internally unobservable (hidden) outputs of $M_1$ and the internally unobservable (hidden) input of $M_2$

$Z$ : The set of internally unobservable (hidden) inputs of $M_1$ and the internally unobservable (hidden) outputs of $M_2$

$tc_k$ : The $k^{th}$ test case of a given test suite

$o_k$ : The expected output sequence of test case $tc_k$

$\hat{o}_k$ : The obtained output sequence of test case $tc_k$

$T_k$ : A transition of machine $M_1$ or $M_2$

$TDC_{M_1 r}$ : The $r^{th}$ tentative diagnostic candidate of machine $M_1$

$TDC_{M_2 r}$ : The $r^{th}$ tentative diagnostic candidate of machine $M_2$

$PDC$ : A preliminary diagnostic candidate

$PDC^{(k)}_{M_1}$ : The $k^{th}$ preliminary diagnostic candidate of $M_1$

$PDC^{(l)}_{M_2}$ : The $l^{th}$ preliminary diagnostic candidate of $M_2$

$NumCandM_1$ : The number of preliminary diagnostic candidates of $M_1$

$NumCandM_2$ : The number of preliminary diagnostic candidates of $M_2$

$P_{M_1}$ : The set of preliminary diagnostic candidates of $M_1$

$P_{M_2}$ : The set of preliminary diagnostic candidates of $M_2$

$n_1$ : The number of states of $M_1$

$n_2$ : The number of states of $M_2$

$L_{TS}$ : The number test cases in a test suite

$L_{Tc}$ : The number of inputs in the longest test case of a given test suite

$Tr_{M_1}$ : The number of transitions of $M_1$

$Tr_{M_2}$ : The number of transitions of $M_2$

$FF$ : Fault Function (see Section 7.3.2)

*FF-Context* : Fault Function of the context machine (or $M_2$) in a system of two *ComFSMs* (see Section 7.3.2)

*FF-Embedded* : Fault Function of the embedded machine (or $M_1$) in a system of two *ComFSMs* (see Section 7.3.2)

$(M_1)_a^{out}$ : The FSM derived by adding new (faulty) transitions with all possible outputs to each transition of $M_1$ (see Section 7.3.2)

$(M_2)_a^{out}$ : The FSM derived by adding new (faulty) transitions with all possible outputs to each transition of $M_2$ (see Section 7.3.2)

$DisSet$ : Distinguishing set

# 1 Introduction

During the last 15 years, computer networks were subject to vast developments. Among others, one major aspect of these developments was the development of telecommunication networks. Roughly speaking, communication protocols are the rules that govern the communication between the different components within a distributed computer system [Boc94]. These components exchange information to provide certain communication services to the user. As with other software systems, the development life cycle of communication protocols encompasses different phases, such as functional specification, design, verification, implementation, testing, and maintenance. Formal Description Techniques (FDTs), such as LOTOS [ISO87a], ESTELLE [ISO87b] and SDL [SDL87], have been proposed and/or adopted for the specification of communication protocols. Formal verification and testing approaches have been also developed based on such formal description techniques [Boc94] [Bri90].

A large number of communication protocols have been standardized by different organizations such as ISO, ITU and IEEE. A protocol standard, in general, can lead to different implementations, which necessitates the need for conformance testing of an implementation to its standard [Boc94]. It is now widely accepted that protocol conformance testing is crucial to the achievement of the objective of Open Systems Interconnection (OSI) [Ray87]. The Finite State Machine (FSM) is the mathematical model that has been extensively used in the study of protocol conformance testing. This model has been widely used in the phases of development of communication protocols such as protocol specification and protocol conformance testing [Boc78]; [Yan95]; [Lee94]; [Ura92], [Sid89], etc.). Moreover, the FSM model has been used to describe a specification of a system in diverse domains such as switching and telephony systems. Extended FSMs are used as the semantic models for formal description techniques [ISO87b], and SDL [SDL87] and also in UML Statecharts.

## 1.1 Scope and Objectives of the First Part of the Thesis

As conformance testing is an important step in the protocol development cycle, many test selection methods have been developed for deriving tests when a system specification and implementation are represented by FSMs (for surveys see [Sid89] and [Boc94]). The purpose of these tests is to determine whether an implementation of the system conforms to (i.e. is correct with respect to) its specification. Usually a conforming implementation is required to have the same input/output behavior. As protocol implementations are tested for conformance against the protocol specification, in addition, these implementations are usually tested for properties not specified by the protocol. These properties may include implementation-specific choices, robustness to user errors and performance properties. This type of testing is called *assessment testing* [Boc94]. Moreover, there are other types of protocol testing, such as inter-operability testing between different implementations of the same protocol, and quality of service testing concerning specific performance questions [Boc94]. The research underlying the first part of this thesis concerns conformance testing.

For conformance testing, the best known test derivation methods are called, W [Vas73], [Cho78], partial W (Wp) [Fuj91], Unique-Input-Output (UIOv) [Vuo89], HIS [Pet93b] and the Distinguishing Sequence (DS) methods [Hen64] [Gon70] [Ura97].

In the area of FSM-based test generation, a fault model serves as a basis for fault coverage analysis [Cho78][Vuo89][Fuj91]. In FSM-based testing, the fault types usually considered are *output faults* (the output of a transition is wrong) or *transfer faults* (the next state of a transition is wrong) [Boc92]. A test suite derived by each of the above methods is considered complete since it detects all output and transfer faults of an implementation under test provided an upper bound on the number of states of this implementation is known.

In fact, the Wp, HIS, and UIOv methods are appropriate modifications of the W method which has two-phases. Tests derived at the first phase check that each state presented in the

specification also exists in the implementation, while tests derived at the second phase check all transitions of the implementation defined by the specification for correct output and final state. States and transitions of the implementation are tested by means of input sequences that distinguish states of the specification. The only difference between the above methods is how such distinguishing sequences are selected.

When using the W, Wp and HIS methods, we are required to apply several input sequences at certain states. Therefore, it is necessary to reach an appropriate state of the implementation several times during the testing procedure. For this purpose, all the methods assume that a reset operation, hereafter written as '*r*', has been correctly implemented which allows a safe return to the initial state of the implementation. This assumption is considered reasonable for a large class of implementations, such as telecommunications software.

For testing implementations where no reset feature is assumed in the implementation, the distinguishing sequence (DS) methods are used. Each of these methods generates a single test sequence (derived from a given specification FSM) that is supposed to be applied to the initial state of a given implementation to check its correctness. The applied test sequence and its expected output response sequence form a so-called *checking sequence*.

The DS methods use a *distinguishing sequence* (DS) as a facility for state identification. For a given FSM *M* with a distinguishing sequence, say *D*, all the states of *M* generate different output sequences in response to *D*. It is known that a distinguishing sequence may not exist for every FSM. Nevertheless, based on distinguishing sequences, a variety of methods for the construction of checking sequences have been proposed in the literature [Hen64][Gon70] [Ura97]. As stated previously, these DS methods generate a single test sequence. Therefore, in order to test two transitions with the same starting state, we need to ensure that this state is reached in the implementation before applying the inputs of these two transitions. Accordingly, transfer sequences should be designed in such a way that they can guarantee that the starting state is reached. Actually, the methods proposed by Hennie [Hen64] and Gonenc [Gon70] differ in the way that such transfer sequences are designed. Later in 1997, [Ura97] proposed a model that generates

short test sequences. Moreover, the proposed model is shown to include as special cases the other two methods.

In realistic applications, during the implementation phase of a system modeled by a given specification machine, designers usually implement incrementally the given system specification. Moreover, during the maintenance phase, designers usually change the specification of a given system due to changes in the users requirements or due to changes in the system environment. For instance, a communication protocol may have to be extended to incorporate new operations (behaviors) so that new services can be provided to the users. Two approaches are possible for the generation of test suites for the modified (incrementally developed, or changed) specification. The first one is to take the modified specification as a *new specification* and apply a test case generation method to this new specification. That is, to generate test cases that test the whole system implementation after each modification. The second approach, which we call *incremental testing approach*, is to generate tests that would only test the modified parts of the implementation that correspond to the modified parts of its specification. The second approach is more affective and less time consuming since it reuses the existing unmodified parts of the given system. Thus, it can be used to reduce the development and maintenance costs of such a system, which represents about two-thirds of the cost of software production [Sch92].

### 1.1.1 Contributions of the First Part of the Thesis

In the first part of this thesis, we present incremental test generation methods (called henceforth *re-testing methods*) that select tests (called *re-tests*) for testing the modified parts of the system specification, in order to check that these modifications were correctly implemented in the system implementation. Here we assume that the parts of the system implementation that correspond to the unmodified parts of the system specification are left intact. Moreover, we also reasonably assume that before modifying the system specification, its implementation was tested and found conforming to this specification. The re-testing methods are based on the W [Cho78], Wp [Fuj91], HIS [Pet93b] and UIOv [Vuo89] methods. Moreover, based on these methods, we present our DS based re-testing methods.

We note that although the re-testing methods presented in this thesis are developed for the FSM model, it is expected that they could be useful for the Labeled Transition System (LTS) model as well, since testing approaches developed for FSMs have been transformed for LTSs [Pet93a], [Tan95]. Here we note that the LTS model is based on the formalism Calculus for Communication Systems CCS [Mil80] and Communicating Sequential Processes CSP [Hoa85] and is the underlying model for the LOTOS language [ISO87a].

Another foreseeable application of the re-testing methods is in testing of object-oriented systems. In object-oriented systems, the components called objects are usually organized into object classes. An object class is a set of objects, which are called its instances. An object class definition specifies a set of allowable behaviors that each object instance in that class may exhibit. Furthermore, the inheritance mechanism allows one to define a new class (called *subclass*) from existing classes (called *superclasses*). A subclass can inherit the bahaviors defined for its superclasses. Therefore, if objects of the superclasses were already tested and found not faulty, the re-testing methods can be used to avoid unnecessarily testing the inherited parts of subclass objects.

## 1.2 Scope and Objectives of the Second Part of the Thesis

While the purpose of conformance testing is to check out if an implementation is different than its specification, an interesting complementary, yet more complex, step is to locate the differences between a protocol specification and its implementation [Lee93]. A solution to this so-called *diagnostic* problem has various applications. For example, it facilitates the job of correcting a protocol implementation so that it conforms to its specification [Lee93].

In general, diagnostics can be classified into two classes. The first class, called *experimental diagnostics*, is mainly used in Medicine [Sho76] and similar domains. Therefore, experimental diagnostics are not covered in this thesis. Our main interest is

related to the second class of diagnostics, called diagnostics based on a model [Dav88] [Kle87]. In this class of diagnostics is that it is necessary to know how the system or the machine under test is supposed to work in order to be able to know why it is not working correctly. Therefore, in model-based diagnostics, [Kle87] [Rei87], we assume the availability of the real system (i.e, implementation) which can be observed, and its model (i.e, specification) from which predictions can be made about its behavior.

Given a specification of a system and its implementation, observed input and output sequences demonstrate a behavior of the system at hand while expected input and output sequences, derived from the specification, tell us how the system at hand is supposed to behave. Any difference between expectations and observations is a *symptom* that the implementation at hand is faulty. In order to explain the observed symptoms, a diagnostic process should be initiated. It consists mainly of performing the following two tasks: the generation of fault candidates and the discrimination between candidates [Kle87].

In the software domain where a system may be represented by an FSM model, some work has already been done for the diagnostic problem [Ghe92b][Ghe92a][Ghe92d][Lee 93]. However, little work has been done for systems represented by two communicating finite state machines (ComFSMs) [Ghe93a].

The underlying research of the second part of this thesis concerns the diagnosis of systems represented by two communicating finite state machines. More specifically, we consider a system consisting of two communicating FSMs, called components. The first component (called the *context machine*) communicates with the environment and with the other component (called the *embedded machine*). The interaction between the context and the environment is assumed to be observable, while the interaction between the context and the embedded components is assumed to be hidden or unobservable. The previous work done on diagnosis for such a system [Ghe93a] assumes that one of the implementation machines is faulty (has "up to" one output or transfer fault), and tries to locate the faulty transition of the faulty machine. Unfortunately, it is not always possible (as will be shown later) to locate the faulty machine in such a system. Therefore, one first

would like to decide if it is possible to locate the faulty machine (we call this problem *machine level diagnosis*). If possible, further analysis can be done to locate the faulty transition within that machine (we call this *transition level diagnosis*).

### 1.2.1 Contributions of the Second Part of the Thesis

In the second part of this thesis we first show that it is not always possible to locate the faulty machine in a system of two ComFSMs. Moreover, we discuss the reasons behind this observation. Based on these reasons we then propose a machine level diagnostic method under the assumption of a single (output or transfer) fault in the implementation. This method would decide if it possible to locate the faulty machine in a system of two ComFSMs. If possible, it can go further and locate the faulty transition within the faulty machine. Finally, we propose another machine level diagnostic approach for the case where multiple output or transfer faults may occur in one of the system implementation machines. As in the previous diagnostic method, the method enables us to decide whether it is possible to identify the faulty machine in the system, once faults have been detected in a system implementation. If this is possible, it also provides tests for identifying the faulty component machine.

## 1.3 Organization of the Thesis

This thesis is organized as follows. Chapter 2 describes the FSM model and its related definitions. Based on this model, Chapter 3 describes the problem of FSM based testing and its related test generation methods, namely the W, Wp, HIS, UIOv, and the DS methods. Chapter 4 includes the FSM based re-testing problem and includes our proposed W based re-testing methods along with their related proofs and some application examples. These methods were presented for the case when the implementation is assumed to have the same number of states as the specification, and for the case when the implementation has more states than its specification. The methods presented for the former case were published in [Elf02] and were combined with the

methods of the latter case in order to be submitted for a journal publication. Furthermore, Chapter 4 includes an analysis on the length of the re-testing test sequences and a summary and a comparison with previous related research work. Furthermore, it includes the DS based re-testing methods along with some application examples. Chapter 5 introduces the diagnostic problem and describes the behaviour a system represented as two communicating FSMs. Moreover, it includes a summary of the previous research work done for the diagnostic of such a system. Chapter 6 includes a method to solve the diagnostic problem for the case where a single fault (output or transfer) may occur in the system implementation along with some application examples [Elf99b]. Moreover, it includes some related experiments and the complexity analysis of the proposed method. Chapter 7 includes a method to solve the diagnostic problem for the case where multiple output or transfer faults may occur in one of the system implementation machines [Elf01]. Moreover, it includes some application examples and a section on the complexity analysis of the proposed method. Chapter 8 concludes the thesis and includes our insights for further research.

# 2 The FSM Model

The specification of a system in diverse areas such as switching systems, telephony systems, communication protocols, lexical analysis, pattern matching, and machine learning can be modeled by a deterministic Finite State Machine (*FSM*) [Hen64] [Gon70] [Ura97]. In this chapter, we describe the FSM model and some related definitions to provide the necessary background to the detailed material that follows in the following chapters.

## 2.1 The FSM Model and its Related Definitions

A non-deterministic finite state machine is an initialized non-deterministic Mealy machine which can be formally defined as follows. A *non-deterministic finite state machine* (NDFSM) $M$ is a 6-tuple $M = (S, X, Y, h, D_M, s_1)$ (taken from [Pet93b]), where:

$S$ is a finite set of states, including a special state $s_1$ called the ***initial state***,

$X$ is a finite nonempty set of input symbols,

$Y$ is a finite set of output symbols, including a special ***null*** symbol '-',

$D_M$ is a specification domain: $D_M \subseteq S \times X$, and

$h$ is a behavior function: $h : D_M \rightarrow 2^{S\times Y}\backslash\varnothing$ where $2^{S\times Y}$ is the set of all subsets of the set $S\times Y$. The behavior function defines the possible transitions of the machine. Given present state $s_i$ and input symbol $x$, each pair $(s_j, y) \in h(s_i, x)$ represents a possible transition to the next state $s_j$ with the output $y$.

A finite state machine $M$ is said to be *deterministic* if for each pair $sx \in D_A$ it holds that $|h(s,x)| = 1$. In the deterministic FSM $M$ instead of behavior function $h$ we use two functions, next-state (or transition) function $\delta$: $S\times X \rightarrow S$ and output function $\lambda$: $S\times X \rightarrow Y$. i.e. a deterministic FSM is a 7-tuple $M = (S, X, Y, \delta, \lambda, D_M, s_1)$. An FSM $M$ is said to be *completely specified or simply a complete* finite state machine, if for each input $x \in X$, there is a transition defined at each state of $M$ i.e. if $D_M = S \times X$; otherwise, $M$ is a said to

be *partially specified* or simply a *partial* finite state machine. In the complete FSM, we omit the specification domain $D_M$, i.e. a complete deterministic FSM is a 6-tuple $M = (S, X, Y, \delta, \lambda, s_1)$ and a complete NDFSM is a 5-tuple $M = (S, X, Y, h, s_1)$.

We use as in [Fuj91] the notation "($s_i$ -$x$/$y$-> $s_j$)" to indicate that the FSM $M$ at state $s_i$ responds with an output $y$ and makes the transition to the state $s_j$ when the input $x$ is applied. This is also may be written as a transition of the form $s_i \xrightarrow{x/y} s_j$. State $s_i$ is said to be the *head* or *starting* state of the transition, while $s_j$ is said to be the *tail* or *ending* state of the transition. An input (or output) sequence $I$ (or $O$) is a suite of inputs $x$ (outputs $y$), which may be the empty sequence (ε).We use "$s_i$ - $I$ -> $s_j$" to indicate that the FSM $M$ is originally in state $s_i$ and goes to state $s_j$ when an input sequence $I$ is applied. In this notation, only the reached state $s_j$ is relevant, the output sequence is ignored.

The above given model is the initialized FSM model which captures the fact that, in practice, a system is often required to enter a particular state before doing anything meaningful. That particular state is called the *initial state* of the FSM and the FSM is said to be an *initialized machine*. By default, we assume throughout the thesis that the first state, i.e. the state with subscript 1, of a given FSM is the initial state. To guarantee that the system enters the initial state, a special reset function should be provided. In the FSM model, this reset function is represented by a number of reset transitions. For any state of the FSM, there is a transition leading from that state to the initial state. Such a transition is executed when the special reset input, denoted as "$r$", is applied. Since the purpose is to bring the FSM into its initial state, the output of a reset transition is accordingly not important and therefore is often assumed to be null (i.e. no output at all).

To simplify our discussions, we make several notational conventions here.

(1) $\Sigma^*$ denotes the set of all words constructed using an alphabet $\Sigma$;

(2) $|V|$ represents the cardinality of *set* $V$, i.e. the number of elements in $V$;

(3) The symbol "ε" denotes the empty word which consists of no elements at all; and

(4) The dot symbol "." denotes the *concatenation operation* of two words (or sequences), i.e. "a.b" is a word obtained by appending word "b" to the end of word "a". However, this symbol is often omitted when no ambiguity arises. This operation can be extended to the concatenation of a word $\alpha$ with a set of words $V$ as follows:

$$\alpha V = \{\alpha\beta \mid \beta \in V\}.$$

Furthermore, this operation can also be extended to the concatenation of two sets of words $W$ and $V$ as follows:

$$W.V = \{\alpha\beta \mid \alpha \in W \text{ and } \beta \in V\}.$$

Still further, if $m$ is a positive integer and $\alpha$ is a word, then $\alpha^m$ represents the absolute concatenation of $\alpha$ with itself for $m$ times, i.e.

$$\alpha^m = \underbrace{\alpha\alpha\cdots\alpha}_{m \text{ times}}$$

If $m = 0$, then $\alpha^m = \varepsilon$.

If $n$ is a positive integer, we let $V^n$ denote n-times concatenation of the set of words $V$ ($V^n = V.V^{n-1}$).

**Definition 2.1: Prefix-closed set**

Let $V$ be a set of words over alphabet $Z$. The prefix closure of $V$, written ***Pref***$(V)$, consists of all the prefixes of each word in $V$, i.e. ***Pref***$(V) = \{\ \alpha \mid \exists\ \gamma(\alpha\gamma \in V)\}$. The set $V$ is *prefix-closed* if ***Pref***$(V) = V$.

**Definition 2.2: Defined input sequence**

Given an input sequence $\alpha = x_1\ x_2\ ..\ x_k \in X^*$, $\alpha$ is called a *defined input sequence*(**DIS**) at state $s_i \in S$, if there exist $k$ states $s_{i1}, s_{i2}, ..., s_{ik} \in S$ such that there is a sequence of specified transitions $s_i$- $x_1$-> $s_{i1}$ --> ... --> $s_{i(k-1)}$- $x_k$-> $s_{ik}$ in the finite state machine $M_S$.

A defined input sequence [Yao93] is also called an ***admissible input sequence*** in [Pet91]. Hereafter, **DIS**($M_S|s_i$) will be used to denote the set of all the defined input sequences at state $s_i$ of machine $M_S$.

**Definition 2.3: State reachability**

For a given FSM $M$, state $s_j$ is said to be *reachable* from state $s_i$ if there exists $\alpha \in$ **DIS**($M|s_i$) such that $s_i$ -$\alpha$-> $s_j$.

**Definition 2.4: Initially connected and strongly connected FSM**

Let $M$ be a given FSM. $M$ is said to be *initially connected* if all its states are reachable from the initial state. $M$ is said to be *strongly connected* if, for any ordered pair of states $s_i$ and $s_j \in S$, there exists $\alpha \in$ **DIS**($M|s_i$) such that $s_i$ -$\alpha$-> $s_j$.

## 2.2 The Behavior of an FSM

An FSM specifies the behavior of a system being modeled. This is achieved by the two characterization functions, namely the transfer function $\delta$ and output function $\lambda$.

As we have seen in Definition 2.2, the application of an input sequence $\alpha = x_1x_2 .. x_k \in X^*$ to a state will produce an output sequence $\beta = y_1\, y_2 .. y_k \in Y^*$ (We will discuss shortly how to define the output sequence $\beta$ in the case that the input sequence $\alpha$ is not a defined input sequence for that state). Then, $\alpha/\beta = (x_1/y_1).(x_2/y_2)\cdots(x_k/y_k)$ is said to be an Input/Output sequence *I/O-sequence*.

It should be noted that if $(s_i, x_k) \in D_M$, then the output function $\lambda$ is uniquely defined for $(s_i, x_k)$, i.e. there is exactly one $y_l \in Y$ such that $y_l = \lambda(s_i, x_k)$. On the other hand,

if $(s_i, x_k) \notin D_M$, then the output function $\lambda$ is not defined for $(s_i, x_k)$. There are basically three approaches proposed in the literature to define the behavior of $M$ in this case.

***Approach* 1:** In this approach, an extra special state which is called the "*error state*" and denoted as $s_{err}$ and an extra special output symbol which is called "*error output*" and denoted as $y_{err}$ are introduced. Then, for any $(s_i, x_k) \notin D_M$, the transfer and output functions are defined as follows:

$\delta(s_i, x_k) = s_{err}$

$\lambda(s_i, x_k) = y_{err}$.

This approach was first proposed in [Sar82], but has not been in widespread use.

***Approach* 2:** In this approach, a special output symbol, called the **"null"** symbol and written as "-", is introduced to represent no output at all. If an input symbol is applied to a state for which the transfer and output functions are not defined, it is assumed that the FSM remains in its present state and produces no output. Formally, this can be stated as follows: if $(s_i, x_k) \notin D_M$, then

$\delta(s_i, x_k) = s_i$

$\lambda(s_i, x_k) = -$.

This approach, usually called the *completeness assumption*, has been widely used in the literature to convert partially specified FSMs to completely specified ones for test generation [Sab88] [Sid89]. The notion of strong conformance was proposed based on this approach [Sab88] [Sid89].

Although these two approaches seem to be quite simple, they are not proper approaches when *protocol conformance testing* is concerned. It is generally recognized [Boc86] [Boc90] that a protocol specification should define, but nothing more, those behavior aspects of a protocol entity which are required for the compatibility with other entities within the given protocol layer. This implies that those implementation dependent behavior aspects are left undefined in the specification. Conformance testing is then to ensure that any protocol implementation

satisfies those behavior aspects specified in the protocol specification. This guarantees that different protocol implementations built for a given protocol are compatible with each other and can therefore provide the required communication services to the user. Accordingly, conformance testing only concerns the behavior aspects defined in the specification. Those implementation dependent aspects should not be tested in the framework of conformance testing. Following this observation, the following third approach is proposed [Pet91] [Boc95].

***Approach 3:*** This approach interprets those $(s_i, x_k) \notin D_M$ as "don't care". In other words, for any $(s_i, x_k) \notin D_M$, $\delta(s_i, x_k)$ can take any state in $S$ and $\lambda(s_i, x_k)$ can take any output in $Y$. Approach 3 introduces non-determinism into the system behavior if $(s_i, x_k) \notin D_M$. However, an implementation which implements a given partially specified FSM is assumed to be completely specified and deterministic. This implies that we will adopt the method used in [Pet91] to explain such non-determinism, i.e., a given partially specified FSM actually represents a set of completely specified deterministic FSMs, each of which is obtained by choosing, for each $(s_i, x_k) \notin D_M$, state in $S$ for the transfer function $\delta$ and an output in $Y$ for the output function $\lambda$. A valid implementation of the system modeled by the given partially specified FSM can behave as one of these completely specified deterministic FSMs. However, a crucial point here is that these completely specified FSMs possess a common set of deterministic behavior aspects defined by the original partially specified FSM. For conformance testing, we will be concerned only with this common set of deterministic behavior aspects. Moreover, the conformance tests generated must avoid the unspecified parts of the specification.

Throughout this thesis, this third approach will be assumed for any $(s_i, x_k) \notin D_M$. For further discussions, we need to extend the definitions of the transfer function $\delta$ and output function $\lambda$ so that they apply not only to single inputs, but also to sequences of inputs.

**Definition 2.5: Extensions of the transfer and output functions**

Here, we extend functions $\delta$ and $\lambda$ to input sequences. Given a state $s_i \in S$ and the empty word $\varepsilon$, we assign $\delta(s_i, \varepsilon) = s_i$ while $\lambda(s_i, \varepsilon) = \varepsilon$. If an input sequence $\alpha x \in \mathbf{DIS}(M|s_i)$ and $\delta(s_i,\alpha)$ and $\lambda(s_i,\alpha)$ are already defined then $\delta(s_i,\alpha x) = \delta(\delta(s_i,\alpha),x)$ while $\lambda(s_i,\alpha x) = \lambda(s_i,\alpha).\lambda(\delta(s_i,\alpha),x)$.

We will only be interested in the behavior that has been specified by the given FSM. For an initialized deterministic machine, the (*specified*) *behavior* of the machine is the (*specified*) *behavior* at its initial state. Any behavior of a deterministic FSM is deterministic, and for comparing deterministic behaviors, we introduce the following relation and definitions.

Let $M_S = (S, X, Y, \delta_S, \lambda_S, D_S, s_1)$ and $M_I = (T, X, Y, \Delta_I, \Lambda_I, D_I, t_1)$ be two FSMs. Hereafter, $M_S$ usually represents a protocol specification while $M_I$ denotes an implementation, and thus, FSM $M_I$ is further assumed to be complete.

**Definition 2.6: Compatible, equivalent, and distinguishable states**

We say that states $s_i$ of $M_S$ and $t_j$ of $M_I$ are *compatible* if $\mathbf{DIS}(M_S|s_i) \cap \mathbf{DIS}(M_I|t_j) = \varnothing$ or if $\forall \alpha \in \mathbf{DIS}(M_S|s_i) \cap \mathbf{DIS}(M_I|t_j)$ it holds that $\lambda_S(s_i, \alpha) = \lambda_I(t_j, \alpha)$. Otherwise; we say that states $s_i$ and $s_j$ are *distinguishable*. Input sequence $\alpha \in \mathbf{DIS}(M_S|s_i) \cap \mathbf{DIS}(M_I|t_j)$ such that $\lambda_S(s_i, \alpha) \neq \lambda_I(t_j, \alpha)$ is said to distinguish the states $s_i$ and $t_j$. If the FSMs happen to be complete, then the definition of compatible states reduces to the definition of equivalent states [Gil62][Koh78]. Let $t_i$ be a state of $M_I$ and $s_j$ be a state of $M_S$. Consider set $V$ of input sequences such that $V \subseteq \mathbf{DIS}(M_S|s_j)$. State $t_i$ is said to be *equivalent* to $s_j$ with respect to the set $V$ (written as $t_i \equiv_V s_j$), if $\Lambda_I(t_i, \alpha) = \lambda_S(s_j, \alpha)$ holds for any $\alpha \in V$. In other words, for each input sequence of $V$, a behavior of $M_I$ at state $t_i$ coincides with that of $M_S$ at state $s_j$. Two states $t_i$ and $s_j$ are *equivalent* (written as $t_i = s_j$), if they are V-equivalent for any set $V$ [Cho78].

**Definition 2.7: Reduced FSM**

An FSM is said to be *reduced* if its states are pair-wise distinguishable. If the FSMs happen to be complete, then the definition of reduced FSM reduces to the definition of minimal FSM.

**Definition 2.8: Conforming, quasi-equivalent, and equivalent FSMs**

We say that $M_I$ *conforms to* $M_S$ (written as $M_I \leq_{\mathbf{fsm}} M_S$) if and only if $t_1 \equiv_{\mathbf{DIS}(M_S,s_1)} s_1$, where $t_1$ and $s_1$ are the initial states of $M_I$ and $M_S$, respectively. In other words, for each input sequence where a behavior of $M_S$ is defined, $M_I$ has the same behavior, i.e. the implementation is *quasi-equivalent* to the specification [Gil62]. This conformance relation corresponds to the notion of *weak conformance* [Sid89] [Sab88]. Furthermore, for completely specified machines, this relation corresponds to the notion of machine equivalence [Gil62] [Koh78]. Two complete FSMs $M_S$ and $M_I$ are equivalent if and only if for all possible input sequences, they produce the same output sequences. Formally, two FSMs $M_S$ and $M_I$ are *equivalent* (written as "=" ) if their initial states $s_1$ and $i_1$ are equivalent (i.e. $t_1 = s_1$) [Cho78].

# 3 FSM Based Testing Methods

Many test selection methods have been developed for the case that the specification of the system to be tested is given in the form of a finite state machine (FSM) [(for surveys see [Sid89] and [Boc94]]. The best known of these methods are called W [Cho78], Wp [Fuj91], Unique-Input-Output UIOv- [Vuo89], HIS [Pet93b] methods and the Distinguishing Sequence DS methods [Hen64] [Gon70] [Ura97].

In the following Section, the problem of FSM based testing will be described along with its related definitions and assumptions. This would facilitate the understanding of the description provided in Section 3.3 of the above mentioned test generation methods. Moreover, Section 3.2 describes the state identification facilities used by these methods.

## 3.1 The Problem of FSM Based Testing

Testing based on the FSM model can be formalized as the problem of *testing an FSM implementation* [Ura92]: given an FSM representation (specification) of a system (denoted henceforth as $M_S$) and an implementation of the system (denoted henceforth as $M_I$), we are required to determine if the implementation machine $M_I$ *conforms to* (i.e., is *correct* with respect to) the specification machine $M_S$ by testing $M_I$ as a black-box.

For the deterministic (with respect to the specification domain) and in general for the partially specified FSM model presented in Chapter 2, a conformance relation is the one given in Definition 2.8, where $M_I$ is said to conform to $M_S$ (written as $M_I \leq_{\mathbf{fsm}} M_S$) if $M_I$ is quasi-equivalent to $M_S$. In other words, for each input sequence where a behavior of $M_S$ is defined, $M_I$ has the same behavior. In other words, if and only if the initial states $t_1$ and $s_1$ of $M_I$ and $M_S$ are equivalent w.r.t. to the set of defined input sequences of $M_S$ at $s_1$ (i.e. $t_1 =_{\mathbf{DIS}(M_S,s_1)} s_1$). We also recall that this conformance relation corresponds to the notion of *weak conformance* [Sid89][Sab88]. Furthermore, for completely specified machines, this relation corresponds to the notion of machine equivalence [Gil62]

[Koh78]. We use this relation throughout this thesis as the criterion for judging the conformance of $M_I$ with respect to $M_S$.

According to Definition 2.8, $M_I \leq_{fsm} M_S$ implies that the specified behavior of $M_S$ is subset or equals to the behavior of $M_I$. Therefore, to judge the conformance of $M_I$ with respect to $M_S$ implies that the behavior of $M_I$ should be known. However, for FSM based testing, the implementation is normally treated as a *black-box*, i.e. the structure of $M_I$ and henceforth the behavior of $M_I$ is not known. Accordingly, in order to draw conclusions about the implementation, some *experiments* can be made by applying sequences of inputs (called a *Test Suite*(*TS*)) to the implementation and observing the resulting sequences of outputs (called *observed behavior* of $M_I$ with respect to *TS*). It is desired that whether $M_I$ conforms to $M_S$ can be decided based on the observed behavior of $M_I$ wrt *TS* .

As the structure of the implementation machine $M_I$ is not known, a test suite used in an experiment has to be generated from the specification machine $M_S$. In order to generate a *TS* from $M_S$ such that the observed behavior of $M_I$ w.r.t. *TS* provides sufficient information to decide the conformance of $M_I$ with respect to $M_S$, certain assumptions about the specification machine $M_S$ and the types of faults (fault model) that can be present in the implementation machine have to be made. Before discussing these assumptions in the next section, we introduce in the following the definitions of a test case and a test suite. As the structure of the implementation machine $M_I$ is not known, a test suite used in an experiment has to be generated from the specification machine $M_S$. Therefore we have the following definition.

**Definition 3.1: Test case and test suite**

Let $s_1$ be the initial state of $M_S$. Then a *test case* of $M_S$ is an input sequence $\alpha$ such that $\alpha \in \mathbf{DIS}(M_S | s_1)$. A *test suite* (*TS*) is a set of test cases.

It should be noted that, in practice, a test case is always of finite length and a test suite always consists of a finite number of test cases.

### 3.1.1 Assumptions for the FSM Based Testing

The test generation methods proposed in the literature and discussed in Section 3.3 were developed based on different combinations of assumptions made about the structural properties of the specification machine $M_S$ (which is deterministic by default). These properties are mainly about :

(1) If $M_S$ is completely or partially specified;

(2) If $M_S$ is strongly or initially connected;

(3) If $M_S$ is reduced or non-reduced;

Accordingly, each method can in general be applied only to the class of specification machines satisfying the required assumptions.

Other assumptions are made by the test selection methods about the types of faults (i.e., the fault model [Boc92] [Mor90] that can be present in an implementation. Without these assumptions, the number of implementation machines will be infinite. This makes the problem of test generation intractable. Therefore, these assumptions are introduced to limit the number of implementation machines to be considered [Gil62] [Koh78].

Moreover, we will see in the following section that each proposed test generation method has, explicitly or implicitly, made certain assumptions about the implementation. We give here a summary of these assumptions.

The first assumption is that the implementation is completely defined (with respect to the input symbol set of the specification machine $M_S$ including the special reset symbol when required) even if the given specification machine is only partially specified. This assumption implies that the implementation can execute when required all inputs from

the input symbol set of $M_S$ (including the special reset symbol) in all of its states. The second assumption is that the implementation is deterministic.

The above first two assumptions have been made by all the proposed test generation methods that are based on the deterministic FSM model. It should be noted that these two assumptions are always assumed to be satisfied by any implementation machine throughout the discussions of this thesis.

The third assumption is about the types of faults in the implementation machine that are guaranteed to be revealed or detected by these methods. For examples, all the proposed test generation methods have the capability of detecting the presence of output faults. In addition, most of these methods have the capability of detecting the presence of transfer faults. On the other hand, some of these methods allow the presence of extra state faults (i.e $M_I$ has more states than $M_S$), but within a relatively small number of additional states in order not to generate very long test sequences. In other words, these methods can allow the implementation to have more states (up to an upper bound) than the specification machine.

The fourth assumption is called the *reliable reset* assumption which has been made by a number of test generation methods. Under this assumption, the reset function is assumed to be implemented correctly, i.e. the implementation can be brought back to its initial state from any other state whenever the reset input is applied.

Any combination of these four assumptions actually specifies a set of FSMs representing possible valid (conform to) and invalid (does not conform to) implementations of the given specification machine $M_S$ that need to be considered. Let us denote, in general, such an implementation machine set as **Impl**($M_S$). Then our goal is to generate a test suite $TS$ from $M_S$ such that, for any given implementation machine $M_I \in$ **Impl**($M_S$), we can check whether $M_I \leq_{\mathbf{fsm}} M_S$ based on the observed behavior of $M_I$ wrt $TS$ resulting

from the experiments on $M_I$ with $TS$. However, whether this goal can be achieved depends on the so-called *fault coverage* of the test suite $TS$.

We note here that, during our later discussions, we will often use a particular implementation set denoted as **Impl**($M_S$, $m$) which consists of all the completely defined deterministic machines with up to $m$ states and with the same input symbol set and output symbol set as the specification $M_S$.

### 3.1.2 Test Generation

It is clear now, that FSM based testing is to decide if a given implementation machine $M_I$ conforms to the specification machine $M_S$, $M_I \leq_{fsm} M_S$, based on the observed behavior of $M_I$ w.r.t $TS$.

It should be noted that, with any test suite $TS$, the observed behavior of $M_I$ w.r.t $TS$ is a subset or equals to the behavior of $M_I$. Since $M_I$ is assumed to be deterministic and completely specified, the behavior of $M_I$ equals to its specified behavior. Therefore, the observed behavior of $M_I$ w.r.t. $TS$ is a subset or equals to the specified behavior of $M_I$. It should also be noted that if the test cases in $TS$ are applied to the specification machine $M_S$, we obtain the corresponding output sequences. The obtained set of I/O sequences, is said to be the *expected behavior* of $M_S$ with respect to the test suite $TS$. The crucial point here is if it is possible to generate a proper test suite $TS$ such that :

For all $M_I \in$ **Impl**($M_S$) , $M_I \leq_{fsm} M_S$ *iff* expected behavior of $M_S$ wrt $TS$ coincides with the behavior obtained by applying $TS$ to $M_I$.

**Definition 3.2: To pass a test suite**

$M_I$ is said to *pass* a given test suite *TS* written $M_I$ *pass TS*, iff the expected behavior of $M_S$ wrt *TS* equals to that obtained by applying *TS* to $M_I$ .Otherwise $M_I$ is said to *fail to pass* test suite *TS*.

**Corollary 3.1**

If $M_I$ fails to pass a test suite *TS*, then $M_I$ does not conform to $M_S$.

**Proof:** Follows directly from Definitions 2.8 and 3.2.

It should be noted here that *TS* is a finite test suite implies that *TS* has a finite number of test cases and each test case consists of a finite number of input symbols. We can now introduce the concept of complete fault coverage as follows.

**Definition 3.3: Complete fault coverage**

Let $M_S$ be a given specification machine and **Impl**($M_S$) its set of implementations. A test suite *TS* is said to provide *complete* or *full fault coverage* for **Impl**($M_S$) if, for any implementation $M_I \in$ **Impl**($M_S$),

$M_I \leq_{\mathbf{fsm}} M_S$ *iff* $M_I$ pass *TS*.

**Definition 3.4: m-complete fault coverage**

For the given specification machine $M_S$, a test suite *TS* is said to be m-complete if it provides complete fault coverage for **Impl**($M_S$, $m$).

As shown above, the notion of fault coverage provides a basis for assessing the completeness of a given *TS*, and thus it provides a basis for assessing the fault detection capabilities of a given test generation method. In the following section, we describe the

notion of FSM fault model that serves as a basis for fault coverage analysis [Cho78] [Vuo89][Fuj91] in the area of test generation.

### 3.1.3 The FSM Fault Model

As stated above, in the area of test generation, a fault model serves as a basis for fault coverage analysis [Cho78][Vuo89][Fuj91]. A general survey on a variety of fault models in testing was given in [Boc92].

The FSM fault model [Boc92] is based on faults made on labeled transitions. The fault models used by the test generation methods described in Section 3, are defined as follows:

**Definition 3.5: Output fault**

We say that a transition has an output fault if, for the corresponding state and received input, the implementation provides an output different from the one specified by the output function $\lambda$.

An implementation has a **single output fault** if one and only one of its transitions has an output fault.

**Definition 3.6: Transfer fault**

We say that a transition has a transfer fault if, for the corresponding state and received input, the implementation enters a different state than specified by the state transition function $\delta$.

An implementation has a **single transfer fault** if one and only one of its transitions has a transfer fault.

**Definition 3.7: Multiple faults**

An implementation has **multiple faults** if and only if some of its transitions have one or several faults defined above.

## 3.2 State Identification Facilities

The testing methods (presented in the next section) use state identification facilities in order to check that each state and transition defined by the specification also exists in the implementation. These facilities are certain input/output behaviors that can distinguish the states of an FSM. A number of state identification facilities have been proposed for test generation from FSM specifications. In the following, we will summarize the state identification facilities used by the test methods described in the following section.

For simplicity, let us assume that the specification machine $M_S = (S, X, Y, \delta_S, \lambda_S, D_S, s_1)$ is reduced and has $n$ states $s_1, s_2, ..., s_n$, input symbol set $X$, output symbol set $Y$. Its transfer and output functions are $\delta_S$ and $\lambda_S$. By default, $s_1$ is its initial state.

**Definition 3.8: Characterization Sequence Set (*W* set)**

A *characterization set* of the FSM $M_S$, often simply called a ***W set***, is a set of input sequences which satisfies the following conditions:

(1) For any $s_k \in S$, $W \subseteq \mathbf{DIS}(M_S|s_k)$,

(2) For any two states $s_i$, and $s_j$, $i \neq j$, there exists $\beta \in W$ such that $\lambda_S(s_i, \beta) \neq \lambda_S(s_j, \beta)$.

A $W$ set always exists for a reduced completely specified machine. However, the $W$ set does not always exist for a reduced partially specified machine.

**Definition 3.9: A family of harmonized state identifiers (or a separating family)**

Given a reduced FSM $M_S$ and a state $s_i \in S$, a set $W_i \subseteq \mathrm{DIS}(M_S|s_i)$ of defined input sequences at state $s_i$ is called a *state identifier* (or a *separating set*) of state $s_i$ if for any other state $s_j$ there exists $\alpha \in W_i \cap \mathrm{DIS}(M_S|s_j)$ such that $\lambda_S(s_i, \alpha) \neq \lambda_S(s_j, \alpha)$.

We now define a collection of state identifiers that has been first introduced in [Yev90] and then has been named a family of *harmonized identifiers* [Pet91] [Pet93b] or a separating family [Yan95]. A *separating family* is a collection of state identifiers $W_i$, $s_i \in S$, which satisfy the following condition:

For any two states $s_i$, and $s_j$, $i \neq j$, there exist $\beta \in W_i$ and $\gamma \in W_j$ which have common prefix $\alpha$ such that $\alpha \in \mathrm{DIS}(M_S|s_i) \cap \mathrm{DIS}(M_S|s_j)$, and $\lambda_S(s_i, \alpha) \neq \lambda_S(s_j, \alpha)$.

A separating family exists for any reduced (partial or complete) machine.

**Definition 3.10: Simple or Unique Input/Output (UIO) sequence**

Let $\{\tau_1, \tau_2, \ldots \tau_n\}$ be a set of input sequences which satisfies the following condition:

(1) For any $s_k \in S$, $\{\tau_1, \tau_2, \ldots \tau_n\} \subseteq \mathbf{DIS}(M_S|s_k)$, and

(2) For any two states $s_i$ and $s_j$, $i \neq j$, $\lambda_S(s_i, \tau_i) \neq \lambda_S(s_j, \tau_i)$.

Then $\tau_i/\beta_i$ is said to be a *Simple Input/Output Sequence* [Hsi70] or a *Unique Input/Output (UIO) sequence* [Sab88] [Yan90], for state $s_i$, where $\beta_i = \lambda_S(s_i, \tau_i)$. The set $\{\tau_1, \tau_2, \ldots \tau_n\}$ forms a $W$ set.

We note that a UIO sequence may not exist for some reduced completely specified FSMs.

**Definition 3.11: Distinguishing Sequence (DS)**

Let $\alpha$ be an input sequence which satisfies the following conditions:

(1) For any $s_k \in S$, $\alpha \subseteq \mathbf{DIS}(M_S|s_k)$, and

(2) For any two states $s_i \in S$, and $s_j \in S$, $i \neq j$, there exists a prefix $\gamma_{ij}$ of $\alpha$ such that

$\lambda_S(s_i, \gamma_{ij}) \neq \lambda_S(s_j, \gamma_{ij})$

Then $\alpha$ is said to be a *distinguishing sequence* or *diagnostic sequence* [Gil62], [Hen64], [Gon70], [Koh78] for the specification machine $M_S$.

For a distinguishing sequence $\alpha$, let $\beta_i = \lambda_S(s_i, \alpha)$, $i = 1, 2, ..., n$. Then $\alpha/\beta_1$, $\alpha/\beta_2$, ... $\alpha/\beta_n$ are UIO sequences for the $n$ states $s_1, s_2, ..., s_n$, respectively. On the other hand, if $\alpha_1/\beta_1$, $\alpha_2/\beta_2$, ... $\alpha_n/\beta_n$ are the respective UIO sequences for the $n$ states $s_1, s_2, ..., s_n$ and $\alpha_1 = \alpha_2 = ... = \alpha_n = \alpha$, then $\alpha$ is a distinguishing sequence.

We note that a distinguishing sequence may not exist for some reduced completely specified FSMs [Gil62].

The following definition is used by the test selection methods presented in the next section.

**Definition 3.12: State cover set**

A set $Q$ of input sequences is called a *state cover set* of FSM $M_S$ if for each state $s_i$ of $S$, there is exactly one input sequence $\alpha_i \in Q$ such that $s_1$-$\alpha_i$ -> $s_i$. As $s_1$-ε-> $s_1$, we normally include ε in a state cover set. We note that if $M_S$ is initially connected, i.e. each state is reachable from the initial state, then a state cover always exists. As mentioned in [Yan95], we consider only initially connected specification FSMs without a loss of generality, since any state of an FSM that is unreachable from the initial state, does not influence a behavior of the FSM.

## 3.3 Review of Testing Methods

Test generation methods can be classified as *single testing approaches* and *multiple testing approaches*, depending on whether reliable reset is assumed in the implementation under test (IUT). If reliable reset is available in the IUT, then we can generate a test suite which consists of a number of test cases. Each test case is preceded by the reset symbol "*r*". This guarantees that the sequence of inputs that follows *r* in the test case will be applied to the initial state of the IUT. Such test generation methods are therefore called *multiple testing approaches*. On the other hand, when reliable reset is not available in the IUT, a test suite consisting of single test case, which is supposed to be applied to the initial state of the IUT, has to be generated. Accordingly, such test generation methods are called *single testing approaches*. Both single and multiple testing approaches have been proposed for test generation from FSM specifications. The following two sections present an overview of these existing methods.

### 3.3.1 Review of Multiple Testing Methods

Multiple testing approaches assume the reliable reset function is available in the implementation under test. All of these methods follow the two testing phase principle described in the introduction.

#### The W, Wp, and HIS Methods

Given a reduced specification FSM $M_S = (S, X, Y, \delta_S, \lambda_S, D_S, s_1)$, $|S|=n$, and a complete implementation FSM $M_I = (T, X, Y, \Delta_I, \Lambda_I, t_1)$ such that $|T|=n$, let $W$ be a characterization set of $M_S$ (if exists) and $F = \{W_1, \ldots, W_n\}$ be a separating family of $M_S$.

All the methods have two phases in order to test the conformance of $M_I$ to $M_S$. Tests of the first, so-called *state identification* phase, check that each state specified by $M_S$ also

exists in $M_I$, or more formally they establish a one-to-one mapping $h_{S\text{-}I}$: $S \rightarrow T$ by the use of a characterization set $W$ or a separating family $F$. Given a prefix-closed state cover set $Q = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ of the specification FSM, for each state $s_j \in S$, the state identification phase comprises the sequences:

$r.\alpha_j.W_j$ (in the HIS method) or

$r.\alpha_j.W$ (in the W and Wp methods)

Each test sequence starts from the initial state, after the application of the reset operation $r$. In this case, in the W and Wp methods, to identify a reached state $s_j$ after a transition $\alpha_j$, all the sequences contained in $W$ are applied to $M_I$, separately. However, in the HIS it is enough to apply the sequences contained in the state identifier set $W_j$ of $s_j$.

We note that the original versions of the W and Wp methods [Vas73][Cho78][Fuj91] assume that the specification machine $M_S$ is completely specified, and accordingly $M_S$ is guaranteed to have a $W$ set. However, this assumption is only sufficient, but not necessary for the existence of a $W$. This implies that a partially specified machine may also have a $W$ set and accordingly the W and Wp method can be applied to such a partially specified machine [Yev90][Yao95]. However, if the specification FSM $M_S$ is partial, a characterization set $W$ may not exist; in this case, the W and Wp methods cannot be applied. In that case the HIS method can be applied.

If FSM $M_I$ passes the state identification test sequences, then there exists one-to-one mapping $h_{S\text{-}I}$: $S \rightarrow T$ such that:

$h_{S\text{-}I}(s_j) = t \Leftrightarrow s_j \equiv_{W_j} t$ in the HIS method, and

$h_{S\text{-}I}(s_j) = t \Leftrightarrow s_j \equiv_{W} t$ in the W and Wp methods **(X-1)**

The second so-called *transition testing* phase, assures that for each state $s \in S$, and input $x \in X$ that is defined at state $s$ the mapping $h_{S\text{-}I}$ satisfies the following property:

$$\lambda_S(s, x) = \Lambda_I(h_{S\text{-}I}(s), x) \text{ and } h_{S\text{-}I}(\delta_S(s, x)) = \Delta_I(h_{S\text{-}I}(s), x) \quad \textbf{(X-2)}$$

For this purpose, for each sequence $\alpha_j \in Q$ that takes the specification FSM to the appropriate state $s_j$, and each $x \in X$ that takes the $M_S$ from state $s_j$ to state $s_k$, the testing transition phase includes the following set of test sequences:

$r.\alpha_j.x.W_k$ in the HIS and Wp methods, where $W_k$ is a state identifier of the state $s_k$ in the specification FSM (for the Wp method, we have $W_k \subseteq W$ ) or

$r.\alpha_j.x.W$ in the W method

We note that the Wp method [Fuj91] is an improvement to the W method based on the observation that in the second testing phase, a subset $W_k \subseteq W$ may be sufficient to identify the ending state $s_k$ of a transition $s_i$ -x/y-> $s_k$.

In all the methods presented above, it is assumed that all states in $M_S$ and $M_I$ are reachable from the initial one. Moreover, is assumed that the number of states $m$ an implementation $M_I$ equals to $n$ the number of states of the specification FSM $M_S$. In Chapter 4, we will describe the W, Wp, and HIS methods for the case when the implementation can have more states than the specification (i.e. $m > n$).

If the FSM $M_I$ passes the test sequences of both testing phases, then it is quasi-equivalent to the specification FSM, i.e. is a conforming implementation. If the specification FSM is complete then the quasi-equivalence relation reduces to the equivalence relation, i.e. the specification FSM and its conforming implementation have the same Input/Output behavior.

As an application example of the W and Wp methods for the case $m = n$, let $M_S$ (taken from [Fuj91]) be the specification FSM $M_1$ shown in Fig.3.1, with the inputs $X = \{a, b, c\}$ and outputs $Y = \{e, f\}$.

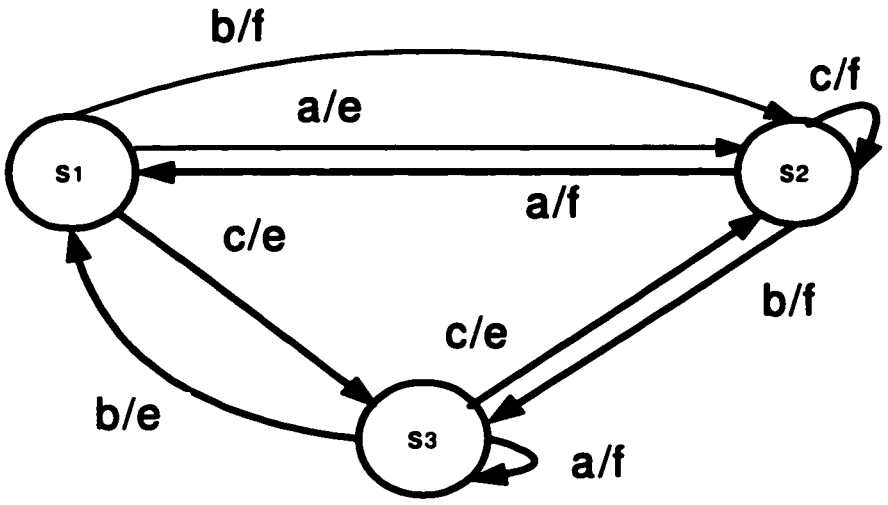


*Figure* 3.1. *Specification* $M_1$

We assume in addition a reset transition with no output (*r*/-) leading to the initial state $s_1$ from every state of $M_1$. $M_1$ has a $Q = \{\alpha_1=\varepsilon, \alpha_2= b, \alpha_3= c\}$as a state cover set, and a characterization set $W= \{a, b\}$. In fact, for each state $s_i$ of $M_1$ we have the following input/output sequences in response to *W*.

| | $s_1$ | $s_2$ | $s_3$ |
|---|---|---|---|
| *a* | *e* | *f* | *f* |
| *b* | *f* | *f* | *e* |

*Table* 3.1. *Responses of* $M_1$ *to W*

From the above, we get the following identification sets:

$w_1 = \{a\}$, distinguishes the state $s_1$ from all other states,

$w_2 = \{a, b\}$, all the sequences in *W* are needed to identify the state $s_2$, and

$w_3 = \{b\}$, distinguishes the state $s_3$ from all other states.

Based on these sets, the state identification phase of the W and Wp methods yield the test sequences: $r.\alpha_1.W + r.\alpha_2.W + r.\alpha_3.W = r.\varepsilon.\{a, b\} + r.b.\{a, b\} + r.c.\{a, b\} = r.a + r.b + r.b.a + r.b.b + r.c.a + r.c.b.$

The transition testing phase of the W method yields test sequences: $r.\alpha_1.X.W + r.\alpha_2.X.W + r.\alpha_3.X.W = r.\varepsilon.\{a, b, c\}.\{a, b\} + r.b.\{a, b, c\}.\{a, b\} + r.c.\{a, b, c\}.\{a, b\} = r.a.a + r.a.b + r.b.a + r.b.b + r.c.a + r.c.b + r.b.a.a + r.b.a.b + r.b.b.a + r.b.b.b + r.b.c.a + r.b.c.b + r.c.a.a + r.c.a.b + r.c.b.a + r.c.b.b + r.c.c.a + r.c.c.b.$

The transition testing phase of the Wp method yields test sequences: $r.\alpha_1.a.W_2 + r.\alpha_1.b.W_2 + r.\alpha_1.c.W_3 + r.\alpha_2.a.W_1 + r.\alpha_2.b.W_3 + r.\alpha_2.c.W_2 + r.\alpha_3.a.W_3 + r.\alpha_3.b.W_1 + r.\alpha_3.c.W_2 = r.\varepsilon.a.\{a, b\} + r.\varepsilon.b.\{a, b\} + r.\varepsilon.c.b + r.b.a.a + r.b.b.\{b\} + r.b.c.\{a, b\} + r.c.a.b + r.c.b.a + r.c.c.\{a, b\}$

We remove from the above sequences redundant sequences (i.e. prefixes of other sequences). For example, we remove the sequences *r.a* and *r.c.a* since they are prefixes of sequences *r.a.a* and *r.c.a.b*. The total length of the test sequences generated using the W method is 40 and the length of those generated using the Wp-methods is 28.

**The UIOv Method**

The UIOv method [Vuo89] is an improvement on the original UIO method [Sab88]. On the other hand, the UIOv method is a special case of the Wp method. This method makes the same assumptions about the specification machine $M_S$ and the implementation machine $M_I$ as the W method. In addition, it further assumes that the set $W_j$ of the Wp method can be chosen in such a way that they contain each only a single sequence, a so-called *Unique-Input-Output* (*UIO*) sequence.

Let $\alpha_1/\beta_1$, $\alpha_2/\beta_2$, ..., $\alpha_n/\beta_n$ be the respective UIO sequences for the $n$ states $s_1$, $s_2$, ..., $Ss_n$. Then the set $\{\alpha_1, \alpha_2, ..., \alpha_n\}$ constitutes a $W$ set of $M_S$ , and in the second testing phase, $\{\alpha_j\}$ is sufficient for the verification of the ending state of a transition $s_i$ -x/y-> $s_j$.

Therefore, we can define:

$W = < W_1, W_2, ..., W_n >$, where $W_i = \{\alpha_i\}$, for $i = 1, 2, ..., n$

Then, the test suite generated by this method can be given in exactly the same manner as the Wp method.

The assumption that the specification machine $M_S$ is completely specified is only sufficient, but not necessary for the existence of UIO sets. This implies that a partially specified machine may also have a UIO sets and accordingly the UIOv method can be applied to such a partially specified machine [Yev90][Yao95].

**The UIO Method**

The UIO method [Sab88] is based on the UIO sequences. It differs from the UIOv method in that it employs only the second testing phase.

The UIO method is proposed for the case $m = n$ only and it was originally claimed [Sab88] that such a test suite could provide complete fault coverage for all implementations of $M_S$ with up to $n$ states. However, a counter-example was later found [Vuo89] which proved that the UIO method could not guarantee to generate test suites with complete fault coverage for all implementations up to $n$ states as the first testing phase is not used in this approach. For this reason, hereafter, the UIO method will not be considered any more in this thesis.

### 3.3.2 Review of Single Testing Methods

A single testing approach generates a single test case which is supposed to be applied to the initial state of the IUT. Such an approach is particularly useful in case that the reset function is not available in the given implementation under test. An overview of the most known single testing approaches is given below.

**Distinguishing Sequence (DS) Methods**

Two distinguishing sequence methods are proposed by Hennie [Hen64] and Gonenc [Gon70]. These methods are shown later to be special cases of a method proposed by [Ura97]. In the following we will be describing these methods.

We have seen in the previous section that, for a state identifier consisting of several input sequences, transfer sequences have to be designed to ensure that the same state is reached in the implementation before applying each input sequence in the state identifier. On the other hand, for state identification, when a distinguishing sequence is used, only one transfer sequence is required since there is only one input sequence (i.e. the distinguishing sequence) that needs to be applied.

In the second phase of testing, however, in order to test two transitions $s_i$-$x_1$->$s_j$ and $s_i$-$x_2$->$s_k$ with the same starting state $s_i$, we need to ensure that the same starting state in the implementation is reached before the inputs $x_1$ and $x_2$ of these two transitions are applied. In a multiple testing approach, this is achieved by using the transition cover set based on the state cover set used in the first testing phase. In a single testing approach, however, these two transitions will have to be tested at different points of a single test case in the general format of $\alpha x_1 \beta x_2 \gamma$. Accordingly, transfer sequences such as $\alpha$ and $\beta$ should be designed in such a way that they can guarantee that the implementation is in the same state when inputs $x_1$ and $x_2$ of the two transitions under test are exercised. The

philosophy of two phase testing were used by both Hennie [Hen64] and Gonenc [Gon70] to develop their DS based single testing methods. Their methods differ in the way that such transfer sequences are designed.

Both Hennie and Gonenc's methods were developed by assuming that the specification machine $M_S$ is deterministic, reduced, strongly connected and has a distinguishing sequence. We will review these two methods in the following with the implementation set **Impl**($M_S$, $n$).

With Hennie's method, suppose that state $s_1$ is the initial state of $M_S$, then the test sequence for the first testing phase takes the following format:

$$d.\mathrm{T}(a_1, s_2).\ d.\mathrm{T}(a_2, s_3).\ d.\mathrm{T}(a_3, s_4)\ ..\ d.\mathrm{T}(a_{n-1}, s_n).\ d.\mathrm{T}(a_n, s_1).\ d$$

where:

(1) $d$ is a distinguishing sequence;

(2) $a_i$ is the state of $M_S$ reached when $d$ is applied to state $s_i$, for $i = 1, 2, ..., n$; and

(3) $\mathrm{T}(a_i, s_j)$ is the transfer sequence which brings $M_S$ from state $a_i$ to state $s_j$.

Suppose that $M_S$ is currently in state $a_k$. To test the transition $s_i$-$x$/$y$-> $s_j$, the following test subsequence is used:

$$\mathrm{T}(a_k, s_{i-1})\ .\ d.\mathrm{T}(a_{i-1}, s_i).x.\ d$$

By including $d.\mathrm{T}(a_{i-1}, s_i)$ in the transfer sequence $\mathrm{T}(a_k, s_{i-1}).d.\mathrm{T}(a_{i-1}, s_i)$, one can guarantee that the required state of the implementation corresponding to state $s_i$ (such a correspondence is established by the test sequence for the first phase of testing) is entered before the input $x$ of the transition under test is applied. Accordingly, the test sequence for the second phase of testing should be the concatenation of the test subsequences for all the specified transitions in $M_S$. The concatenation of the test sequences for the two testing phases gives us the final single test case which provides complete fault coverage for **Impl**($M_S$, $n$).

For the method proposed by Gonenc [Gon70], if $d$ is a distinguishing sequence, then in the first testing phase, $d.d$ is applied to each state of $M_S$. Therefore, the test sequence for the first testing phase takes the following general format:

$$d.d.T(a_1,s_2).\ d.d.T(a_2,s_3).\ d.d.T(a_3,s_4)\ldots\ d.d.T(a_{n-1},\ s_n).\ d.d.T(a_n,\ s_1).d$$

where

(1) $a_i$ is the state of $M_S$ reached when $d.d$ is applied to state $s_i$, for $i = 1, 2, ..., n$; and

(2) $T(a_i, s_j)$ is the transfer sequence which brings $M_S$ from state $a_i$ to state $s_j$.

By properly ordering the states, most of the transfer sequences can be eliminated and the subsequences $d.d$ can be overlapped to reduce the length of this test sequence. It should be noted that, by applying $d.d$ to state $s_i$, we have:

$$s_i - d-> b_i - d -> a_i.$$

The advantage of doing this is that both state $s_i$ and state $b_i$ (reached from $s_i$ when $d$ is applied) are verified.

In the second phase, the transition $s_i- x_1/y_1->s_j$ can be tested by using the test subsequence (called ***test cell*** in [Gon70]) $x_1.d$ which exercises the following sequence of transitions : $s_i -x_1-> s_j-d -> b_j$

Now, suppose there is another transition $b_j- x_2/ y_2 -> s_k$ that needs to verified. We can simply append the test subsequence $x_2.d$ to the end of the previous test subsequence $x_1.d$. This forces the following sequence of transitions to be executed:

$$s_i- x_1-> s_j-d-> b_j- x_2-> s_k -d-> b_k .$$

In other words, no transfer sequence is required between these two test subsequences. The reason is that state $b_j$ has already been verified in the first testing phase and can therefore be used as the starting state when the second transition is to be verified. If all

transitions leading from state $b_j$ have been checked, a transfer sequence is then needed to bring the machine to another state where transitions leading from that state remain to be checked. Apparently, by properly selecting the order in which all the specified transitions are to be verified, we can minimize the use of transfer sequences and therefore substantially reduce the length of the final single test case.

It should be noted that any single test case generated with Gonenc's method can provide complete fault coverage for **Impl**($M_S$, $n$).

[Ura97] proposed a general model for constructing minimum length test sequences. They also showed that Hennie's and Gonenc's methods are special cases of their proposed model. This model makes the same assumptions about the specification machine $M_S$ and the implementation machine $M_I$ as the methods presented by Hennie and Gonenc. That is it is assumed that the specification machine $M_S$ is deterministic, completely specified, reduced, strongly connected and has a distinguishing sequence.

# 4 FSM Based Re-testing Methods

## 4.1 Introduction

In realistic applications, maintaining a system modelled by a given specification FSM involves modifying its specification as a result of changes of the user's requirements. Moreover, developers usually implement incrementally the given specification. Testing the whole system implementation after each modification (or increment) is considered expensive and time consuming. Therefore, it is important to generate a test that would only test the modified parts of the implementation that correspond to the modified parts of its specification. As mentioned before, this would (a) reduce the maintenance cost of such a system, which was reported to account for as much as two-thirds of the cost of the software production [Sch92], and (b) allow designers to develop and test incrementally the implementation.

In this chapter, we present test generation methods (called henceforth re-testing methods) that select tests (called re-tests) for testing the modified parts of the system specification, in order to check that these modifications were correctly implemented in the system implementation. Here we assume that the parts of the system implementation that correspond to the unmodified parts of the system specification are left intact. Moreover, we also reasonably assume that before modifying the system specification, its implementation was tested and found conforming to this specification. These methods are based on the W [Cho78], Wp [Fuj91], HIS [Pet93b], and UIOv [Vuo89] methods and are also adapted to the DS methods.

This chapter is organized as follows. Section 4.2 introduces the re-testing problem. Section 4.3 includes a summary and a discussion on related research work. Section 4.4 includes the W, Wp, and HIS based re-testing methods used for solving the re-testing problem along with their related proofs and some application examples. Section 4.5 includes the W, Wp, and HIS based re-testing methods for the case when the

implementation is assumed to have more states than the specification. Moreover, it includes related proofs and some application examples. Section 4.6 includes the complexity analysis of the re-testing methods. In Section 4.7 the Wp re-testing methods are adapted to the UIOv method. Section 4.8 includes the DS based re-testing methods.

## 4.2 Problem Definition

The first problem studied in this thesis is defined as follows. Let the reduced FSM $M_S = (S, X, Y, \delta_S, \lambda_S, D_S, s_1)$ be the specification of a given system. We reasonably assume that the complete implementation FSM $M_I = (T, X, Y, \Delta_I, \Lambda_I, t_1)$ of $M_S$ with the same number of states has been tested and found conforming to $M_S$. Therefore, there exists a one-to-one mapping $h_{S\text{-}I}$: $S \rightarrow T$ such that for each state $s \in S$ and input $x \in X$ that is defined at state $s$, (X-1) holds.

Let the reduced $M_S' = (S, X, Y, \delta_S, \lambda_S, D_S', s_1)$ be the modified specification, and $M_I' = (T, X, Y, \Delta_I, \Lambda_I, t_1)$ be the modified implementation that should conform to $M_S'$. We assume that only transitions corresponding to the modified parts of $M_S'$ have been changed in $M_I'$ and we want to generate test sequences (hereafter called *re-testing sequences*) for the modified parts of the system specification, in order to check that these modifications were implemented correctly in the modified implementation $M_I'$. In other words, for each unmodified transition ($s_j - x/y \rightarrow s_k$) of the $M_S'$, we assume that transition ($h_{S\text{-}I}(s_j)$-$x/y$-> $h_{S\text{-}I}(s_k)$) has not been changed in the modified implementation $M_I'$. We note that in case where new states are added (or deleted) to (or from) $M_S$, we let $S'$ denote the set of states of $M_S'$ and $T'$ denote the set of states of $M_I'$, respectively.

In general, we have the following types of modifications that can be made in $M_S$ and implemented by a designer in $M_I$:

(1) outputs of some transitions are modified,

(2) tail states of some transitions are modified,

(3) outputs and tail states of some transitions are modified,

(4) new transitions are added,

(5) some transitions are deleted,

(6) new states are added, and

(7) some states are deleted.

## 4.3 A Summary and Comparison with Related Research Work

When the specification of a given system is a complete and reduced FSM, the problem of deriving re-testing sequences can be converted into the problem of test derivation from an FSM with a fault function [Pet92] or from its generalization [Kou99]. In both approaches, potential implementations of the given specification are represented as all complete and deterministic sub-machines of a given nondeterministic FSM that is called a Fault Function (FF) in the former case and a Mutation Machine (MM) in the latter case. When a test suite is derived from an FF or an MM, each modified transition of an implementation is checked for correct output and next state. Moreover, as shown in [Pet92] and as we show in this thesis, some unmodified transitions need to be checked for correct next state. Therefore, all the transitions must be examined thoroughly for this purpose. Moreover, since each implementation FSM is a submachine of the FF (or MM), the next state of an appropriate transition may belong to a proper subset of states; in this case, when testing the transition sequences are generated to distinguish the next state from the states in this subset rather from all states of the specification. In the paper [Pet92], the authors presented two procedures for test derivation from the FF under the assumption that an implementation does not have more states than its specification. Both procedures return a complete test suite that is a test suite that detects each implementation (i.e submachine of the FF) with a behavior different than that of the specification. In the first procedure, when checking the ending state of a transitions, the authors show how distinguishing sequences of a proper subset of states can be used instead of a complete state identifier. In the second, so-called *advanced procedure*, the authors proposed the notion of a *stable state identifier* that distinguishes a given state from each other state of a potential implementation. Given a state that has a stable state identifier, there is no need

to test any unmodified transition from the state. However, the question how to derive such stable identifiers was left open. In this thesis, we continue to study properties of the states and their distinguishing sequences and we derive appropriate stable identifiers for some of these states so that we do not need to test their outgoing unmodified transitions. Moreover, we derive appropriate identifiers to check each modified transition for correct output and ending state and we examine different cases that can be used to generate short re-testing sequences. As an example, we may use distinguishing sequences when checking certain transitions that pass through already tested transitions.

Compared to the FF approach [Pet92], the approach proposed in [Kou99] derives a complete test suite for an implementation with more states than its specification. The idea behind the approach is based on the derivation of the product of the specification and the mutation machines. The reachability analysis for the product then is performed to determine distinguishing sequences that test all transitions of a potential implementation. The authors do not discuss how to select distinguishing sequences in order to return a shorter test suite. However, in our case, the mutation machine from which re-testing sequences can be derived is special. Each unmodified specification transition is a deterministic transition of the mutation machine while each modified transition becomes chaotic; each pair (state, output) becomes possible as its tail state and output in a potential implementation. In other words, on one hand, the mutation machine has a number of deterministic transitions that can be used deriving a test suite, while on the other hand, a number of all possible paths that include modified transitions becomes exponential. By the latter reason, we propose a more appropriate technique for test derivation. First, we do not explicitly enumerate all possible implementation paths under an appropriate input sequence and secondly, we determine which unmodified transitions do not need re-testing and we derive appropriate transfer and distinguishing sequences that allow us not to test these transitions. For this purpose, we essentially use unmodified specification transitions that still remain deterministic in the mutation FSM. Moreover, unlike the work presented in [Pet2] and [Kou99], our work is generalized to deal with partial specifications.

## 4.4 Re-Testing Methods for the W, Wp, and HIS Methods

In this section, we present methods for generating re-testing sequences based on the W, Wp, and HIS methods for the case when $m=n$ (i.e the number of states of the specification machine is equal to the number of states in the implementation machine).

The re-testing methods adapted for the W, Wp, and HIS test derivation methods have also two phases. In the first phase, re-tests are selected in order to check (or re-identify) some states of the modified specification in the new implementation, and in the second re-testing phase, re-tests are selected to check each modified transition for a correct output and a correct tail state. Here, we examine different cases that can be used to generate short re-testing test sequences.

The first case, Case-1, occurs when the unmodified part of the modified specification is still reduced. In this case, there exists state identifiers that traverse only unmodified transitions and the mapping between the initial specification and its conforming implementation with the property (X-1) is still valid, and thus no state re-identification is needed. Moreover, only modified transitions need to be tested in order to check (X-2). This case always holds when new transitions are added when modifying the specification.

The second case, Case-2 occurs when the unmodified part of the modified specification is not reduced, but all the states of the modified specification are reachable through unmodified transitions. In this case, as in Case-1, the mapping between the initial specification and its conforming implementation is the only candidate that can satisfy (X-2) for the modified specification and its implementation. However, we have to check whether this mapping satisfies (X-1) for the states that have state identifiers pass through modified transitions. In order to check this, for each of these states, the corresponding re-testing test sequences only include the identification sequences that pass through modified transitions. Similar to Case-1, the re-testing phase includes only sequences that test modified transitions.

The third case, Case-3, occurs when the unmodified part of the modified specification is not reduced, and some states are only reachable through modified transitions. This case always holds if additional states are added when modifying the specification. In Case-3, for the states of the modified specification that are only reachable through modified transitions, the old conforming images might not be preserved in the new implementation. i.e. each of these states could correspond (be mapped) to a new state. This necessitates the re-identification of these states in the new implementation as well as checking not only modified but all outgoing transitions from these states for correct output and ending states. However, for some of these states, the old conforming image remains the only candidate that satisfies (X-1) and (X-2). For each such a state we do not need to re-test unmodified transitions from this state and in order to re-identify the state we derive a proper state identifier that kills implementations where the state has a new image (i.e. kills all other mappings of the state that correspond to wrong implementations).

For convenience, hereafter, we use the input symbols $a$ and $b$ for unmodified transitions, and $x$ and $z$ for modified ones.

### 4.4.1 Case-1: The Unmodified Part of the Modified Specification is Reduced

Here we assume that the unmodified part $UP\text{-}M_S'$ of the modified specification $M_S'$ is reduced. Then, there exist state identifiers $W_1,\ldots,W_n$, which satisfy the following property: each $W_i$ is a subset of the defined input sequences at state $s_i \in S$ in the $UP\text{-}M_S'$, We note that since the unmodified part of the specification can be partial, a characterization set W may not exist. However, we can always select state identifiers with the above property.

Due to the above property of the state identifiers, we observe that a sequence that distinguishes two states of the initial specification and traverses only unmodified transitions when applied at these states, also distinguishes the corresponding states of the

modified implementation $M_I'$. That is, given two states $s_j$ of $M_S'$ and $t$ of $M_I'$, it holds that:

$s_j \cong_{W_j} t \Leftrightarrow h_{S\text{-}I}(s_j) = t$ in the Wp and HIS methods, or

$s_j \cong_W t \Leftrightarrow h_{S\text{-}I}(s_j) = t$ in the W method.

Therefore, if the modified implementation is quasi-equivalent to the modified specification the mapping $h_{S\text{-}I}(s)$: $S \rightarrow T$ between the initial specification and its conforming implementation is the only candidate that can satisfy (X-1) and (X-2) for the modified specification and its implementation. Therefore, we do not need to re-identify states of the modified implementation or re-test unmodified transitions.

#### 4.4.1.1 General solution

For each modified edge ($s_j$ -x/y-> $s_k$), its corresponding re-testing test cases are formed as follows:

If we use the HIS method or if a characterization set $W$ exists and we use the Wp method:

$$r.\alpha_j.x.W_k, \qquad \textbf{(1-a)}$$

where $W_k$ is a state identifier of state $s_k$.

If a characterization set $W$ exists and we use the W method:

$$r.\alpha_j.x.W \qquad \textbf{(1-b)}$$

**Theorem 1.** Given a modified specification $M_S'$ and its implementation $M_I'$, let the unmodified part of $M_S'$ be reduced and have state identifiers $W_1$, ..., $W_n$. If implementation $M_I'$ passes the re-testing test suite which consists of the union of the test cases over all modified transitions as given in Formulae (1-a) or (1-b), then the implementation $M_I'$ is quasi-equivalent to $M_S'$.

We omit the proof of Theorem 1 since it is a particular case of Theorem 2.

We note that if the specification FSM is complete then the above case is a particular case of the advanced procedure in [Pet92] since each state identifier is a so-called stable state identifier, i.e. is a state identifier of the corresponding state in each potential implementation.

#### 4.4.1.2 **An optimized solution**

The test suite constructed using the formulae of the general solution may be shortened if we use, when checking certain transitions, shorter state identifiers that pass through already tested transitions rather than those generated only from the unmodified part of the modified specification. In other words, instead of using state identifiers derived in advance, we can generate shorter state identifiers, as modified transitions are tested. For this purpose, we assume that a linear order "<" over modified transitions of the specification is given. This order satisfies the following property: If $\alpha_r.z \in Q$ is a prefix of $\alpha_j \in Q$, then for any two modified transitions $(s_r$ -$z$-> $s_l)$ and $(s_j$ -$x$-> $s_k)$, transition $(s_r$ -$z$-> $s_l) < (s_j$ -$x$-> $s_k)$. In this way, when checking a modified transition, we use lower order transitions (or already checked transitions) to generate shorter re-testing sequences. In this section, we illustrate by an example the advantage of using such a linear order. However, we do not discuss how to derive an order that provides the shortest re-testing sequences.

For each modified edge $(s_j$ -$x$-> $s_k)$, its corresponding re-testing test cases are formed as in Formulae (1-a) or (1-b). However, as a state identifier of state $s_k$, we use state identifier $W_k$ (or characterization set $W$ in W-method) of the part of $M_S$ that comprises unmodified transitions or modified transitions $(s_r$ -$z$-> $s_l)$ where $(s_r$ -$z$-> $s_l) < (s_j$ -$x$-> $s_k)$. In other words, for testing transition $(s_j$ -$x$-> $s_k)$ the state identifier $W_k$ has sequences that if applied at state $s_k$ only traverse unmodified transitions or modified transitions $(s_r$ -$z$-> $s_l) < (s_j$ -$x$-> $s_k)$. This allows, when checking a modified transition, the use of already re-checked transitions (or lower order transitions) in order to generate shorter state identifiers.

**Theorem 2.** Given a modified specification $M_S'$ and its implementation $M_I'$, let the unmodified part of $M_S'$ be reduced, and for each modified transition $(s_j \text{ -}x\text{-> } s_k)$, let also $W_k$ be a state identifier of state $s_k$ in the part of $M_S$ that comprises unmodified transitions or modified transitions $(s_r \text{ -}z\text{-> } s_l) < (s_j \text{ -}z\text{-> } s_k)$. If implementation $M_I'$ passes the re-testing test suite which consists of the union of the test cases over all modified transitions as given in Formulae (1-a) or (1-b), then the implementation is quasi-equivalent to $M_S'$.

**Proof of Theorem 2:** Consider the one-to-one mapping $h_{S\text{-}I}(s)$: $S \rightarrow T$ between the initial specification and its conforming implementation. Given state $s_i$ of the modified specification, consider its state identifier $W_i$. Since the sequences of the set $W_i$ traverse only unmodified transitions when applied at state $s_i$, or already tested modified transitions (i.e. lower order transitions), and state $s_i$ is $W_i$-equivalent to state $h_{S\text{-}I}(s_i)$ and is not $W_i$-equivalent to any other state of the modified implementation. Therefore, we have

$$s_i \equiv_{W_i} t \Leftrightarrow t = h_{S\text{-}I}(s_i).$$

Therefore, it is enough to show that the old mapping $h_{S\text{-}I}$ possesses the property (X-2) only for modified transitions. For modified transitions, we use an induction over the ordered set $\{(s_r \text{ -}x\text{ -> } s_l)_i \mid i = 1,\ldots,m\}$ of $m$ modified transitions.

**Induction basis:** By definition of the order "<", given modified transition $(s_j \text{ -}x\text{ -> } s_k)_1$, the sequence $\alpha_j \in Q$ does not traverse a modified transition, i.e. the $\alpha_j$ takes the modified implementation from the initial state to the state $h_{S\text{-}I}(s_j)$. The re-testing test suite comprises test cases $r.\alpha_j.x.W_k$ where $W_k$ is a state identifier of state $s_k$ in the modified specification such that for each state $s_i$, $i \neq j$, there exist $\beta \in W_k$ with the following property. If sequence $\beta$ is applied at states $s_j$ and $s_i$ it does not traverse modified transitions and $\lambda_S(s_k, \beta) \neq \lambda_S(s_i, \beta) = \lambda_I(h_{S\text{-}I}(s_i), \beta))$. Therefore, if the modified

implementation passes the test cases $r.\alpha_j.x.W_k$ then $\lambda_S(s_j, x)=\Lambda_I(h_{S\text{-}I}(s_j), x)$ and the ending state of the transition $(h_{S\text{-}I}(s_j)$ -$x$-> $t_k)$ is $W_k$-equivalent to $s_k$, i.e. $t_k = h_{S\text{-}I}(s_k)$.. For each unmodified transition $(s_j$ -$a$-> $s_l)$ from state $s_j$ it holds that:

$\lambda_S(s_j, a)=\Lambda_I(h_{S\text{-}I}(s_j))$ and $h_{S\text{-}I}(s_l) = \Delta(h_{S\text{-}I}(s_j), a)$.

**Induction assumption**: For some $l<m$ we assume that for each transition $(s_j$ -$a$-> $s_k)_i$ with $i<l$ (X-1) holds.

**Induction step :** We now show that (X-1) holds for the $(l+1)$th transition $(s_j$ -$x$-> $s_k)$ of the ordered set of modified transitions. By definition of the order "<" and the induction assumption, sequence $\alpha_j \in Q$ takes the modified implementation from the initial state to the state $h_{S\text{-}I}(s_j)$. The re-testing test suite comprises test cases $r.\alpha_j.\ x.W_k$ where $W_k$ is a state identifier of state $s_k$ in the modified specification such that each sequence of the set $W_k$ applied at state $s_k$ traverses non-modified transitions or transitions $(s_j$ -$x$-> $s_k)_i$, $i<l$. Due to induction assumption, (X-2) holds for each such transition, i.e. $W_k$ is a state identifier of state $h_{S\text{-}I}(s_k)$ in the modified implementation. Therefore, if the modified implementation passes the test cases $r.\alpha_j.x.W_k$ then $\lambda_S(s_j, x)=\Lambda_I(h_{S\text{-}I}(s_j), x)$ and the ending state of the transition $(h_{S\text{-}I}(s_j)$ -$x$-> $t_k)$ is $W_k$-equivalent to $s_k$, i.e. $t_k = h_{S\text{-}I}(s_k)$.
For each non-modified transition $(s_j$ -$a$-> $s_l)$ from state $s_j$ it again holds that

$\lambda_S(s_j, a)=\Lambda_I(h_{S\text{-}I}(s_j))$ and $h_{S\text{-}I}(s_l) = \Delta(h_{S\text{-}I}(s_j), a)$.

Thus, (X-1) holds for each transition of the modified specification and implementation. In other words, if the modified implementation passes the above test sequences it is equivalent to the modified specification.□

#### 4.4.1.3 **An application example**

As an example for the general and optimized solution methods, we consider the modified specification FSM $M_1$ shown in Fig. 4.1. FSM $M_1$ has the input set $X=\{a, b, c\}$, output

set $Y = \{y1, y2, y3\}$. The labels of the modified transitions are shown in bold. The unmodified part of FSM $M1$ has a separating family $\{w1, w2, w3, w4\}$ of state identifiers, where $w1 = \{bb\}$, $w2 = \{bb\}$, $w3 = \{bb, c\}$ and $w4 = \{bb, c\}$. In fact, for each state $s_i$ of $M1$ we have the following input/output sequences in response to $W_i$.

| | $s1$ | $s2$ | $s3$ | $s4$ |
|---|---|---|---|---|
| $b\,b$ | $y1\,y2$ | $y2\,y2$ | $y2\,y1$ | $y2\,y1$ |
| $c$ | | | $y2$ | $y1$ |

*Table* 4.1. Responses of $M1$ to state identifiers (if defined)

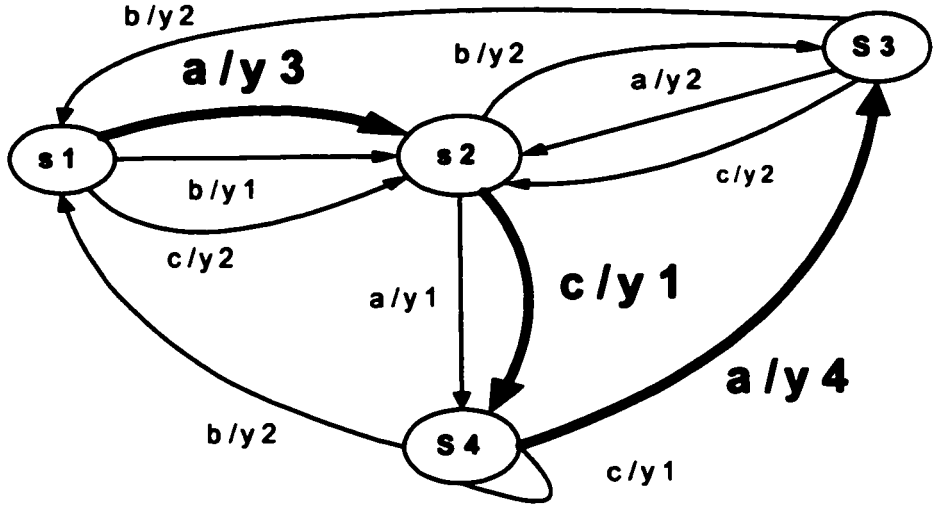


*Figure* 4.1. *Specification* $M1$

None of the above state identifiers passes through a modified transition if applied at an appropriate state, and thus, the unmodified part of $MS'$ is reduced. According to the general solution, in order to re-test the modified transitions, the following re-testing sequences of length 24 are generated using Formula (1-a): $\{r.\alpha1.a.w2 + r.\alpha2.c.w4 + r.\alpha4.a.w3\} = \{r.\varepsilon a.bb,\ r.b.c.bb,\ r.b.c.c,\ r.ba.a.bb,\ r.ba.a.c\}$, where $\alpha1 = \varepsilon$, $\alpha2 = b$ and $\alpha4 = ba$ are the corresponding sequences of the state cover set.

We now use the following linear order over modified transitions $(s1\ \text{-}a\text{->}\ s2) < (s4\text{-}a\text{>}s3) < (s2\ \text{-}c\text{->}\ s4)$. In order to re-test the modified transition $(s1\ \text{-}a\text{->}\ s2)$, the re-testing test

sequence $\{r.a.w2 = r.a.bb\}$ is used. If the implementation at hand passes this sequence, then we have the following responses to the input $a$ :

At state $hS\text{-}I(s1)$: $y3$, at state $hS\text{-}I(s2)$: $y1$, and at state $hS\text{-}I(s3)$: $y2$,

Consequently, in order to re-test transition ($s4$ -$a$ -> $s3$), the re-testing test sequence $r.\alpha4.a.a = r.b.a.a.a$ will be enough instead of $r.\alpha4.a.w3$, since $r.\alpha4$ reaches state $s4$ through unmodified transitions and if afterwards, the modified implementation produces the expected output $y4$ to the input symbol $a$, then $a$ becomes a distinguishing sequence for the part of the specification comprising unmodified transitions and transitions ($s1$–$a$-> $s2$) and ($s4$ -$a$ -> $s3$), where for the last transition only its output has been checked; therefore, $a$ is a distinguishing sequence for the corresponding part of the modified implementation.

Afterwards, in order to check the ending state $s4$ of the modified transition ($s2$ - $c$ -> $s4$), the distinguishing sequence $a$ can be used instead of the previous state identifier $w4$= $\{bb, c\}$. Hence, the re-testing test sequence of this transition is $r.\alpha2.c.a = r.b.c.a$. Therefore, according to the optimized solution method, in order to re-test the modified transitions, the following re-testing sequences are generated using the above linear order and Formulae (1-a): $\{r.a.bb + r.b.a.a.a + r.b.c.a\}$.

The total length of these sequences is 13, where the length of those generated using Formulae (1-a) of the general solution method is 24. The HIS method generates a test suite of length 51 if the whole specification of $MS'$ is considered for test derivation.

### 4.4.2 Case-2: Each State of the Modified Specification is Reachable through Unmodified Transitions and the Unmodified Part is not Reduced

In some cases, the unmodified part of the modified specification $M_S'$ is not reduced. However, if each state of $M_S'$ is reachable from the initial state through unmodified transitions then in the re-identification phase we only re-identify in the new implementation the states that have state identifiers passing through modified transitions, and similar to the previous case, only modified transitions need to be tested. Since each state of $M_S'$ can be reached through unmodified transitions, the only possible correct mapping between the states of $M_S'$ and $M_I'$ is the old mapping established between the states of $M_S$ and $M_I$. Therefore, in order to check that this mapping still holds for the states of the modified specification and implementation the only states that have state identifiers passing through modified transitions have to be re-identified in the new implementation. Moreover, in order to re-identify such a state, it is enough to apply only the sequences of the corresponding state identifier that pass through modified transitions.

Let $Q$ be a prefix-closed state cover set such that its sequences do not traverse modified transitions if applied at the initial state of $M_S'$. Let also $F = \{W_1, \ldots, W_n\}$ be a separating family of the modified specification, and $W$ be a characterization set (if exists).

**i) State re-identification phase**

For each state $s_r$ such that some sequences of $W_r$ traverse modified transitions if applied at $s_r$ the state re-identification sequences are formed as follows:

$$r.\alpha_r.W_r' \qquad \textbf{(2-1a)}$$

where $W_r' \subseteq W_r$ (or $W_r' \subseteq W$ for the W and Wp methods) comprises each sequence of the state identifier $W_r$ (of the characterization set $W$) that, if applied at state $s_r$ of the

modified specification, traverses a modified transition. We note that each state of $M_S'$ for which all sequences of the state identifier traverse only unmodified transitions, does not need to be re-identified.

**ii) Re-testing modified transitions phase**

For each modified edge ($s_j$ - $x$ -> $s_k$), its corresponding re-testing test sequences are formed as shown in Formulae (1-a) and (1-b).

**Theorem 3.** Given the modified specification $M_S'$ and its implementation $M_I'$, let $F = \{W_1, \ldots, W_n\}$ be a separating family of $M_S'$ and $W$ be a characterization set of $M_S'$ (if exists). Let also $Q$ be a prefix-closed state cover set of $M_S'$ such that each sequence of the set $Q$ does not traverse a modified transition if applied at the initial state. If the implementation $M_I'$ passes the re-testing test suite which is the union of the re-testing test sequences given in Formula (2-1a) and Formulae (1-a) or (1-b), then the implementation is quasi-equivalent to $M_S'$.

We omit the proof of Theorem 3 since it is a particular case of Theorem 4.

### 4.4.3 Case-3: Some States Are only Reachable through Modified Transitions

In some cases, the unmodified part of the modified specification $M_S'$ is not reduced and some states of $M_S'$ are only reachable through modified transitions. This case always holds when additional states are introduced when modifying the specification. Here, for the subset of states, say $S_{r\text{-}m}$ of the modified specification that are only reachable through modified transitions, the old conforming mapping might not be preserved between the new specification and its implementation, i.e. some $s_k \in S_{r\text{-}m}$ of the modified specification might be mapped to a new state of its implementation (say $t_l \in T_{r\text{-}m}$), different from $t_k$. Each such state must be re-identified in the new implementation and moreover, differently from former two cases, we have to check unmodified transitions from this state.

As an example, we modify the specification $M_S$ shown in the upper part of Fig. 4.2 and obtain the FSM $M_S'$ shown in the upper part of Fig. 4.3. The modified transitions are shown as bold lines. We note that the mapping between states of $M_S$ and its conforming implementation $M_I$, shown in the lower part of Fig. 4.2, is $h_{S\text{-}I}(s_k) = t_k$ for $k$= 1,...,4. Moreover, we let $M_I'$, shown in the lower part of Fig. 4.3, be the implementation of $M_S'$. $M_S'$ has $W = \{aa\}$ as a characterization set which is a state identifier of each state. In fact, we have the following output responses to $aa$.

For state $s_1$: $xx$,

For state $s_2$: $xy$,

For state $s_3$: $yy$, and

For state $s_4$: $yx$.

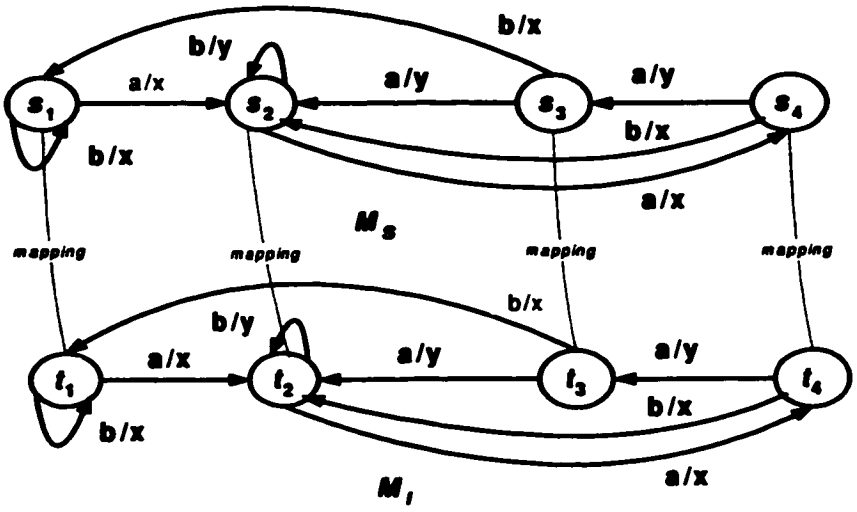


*Figure* 4.2. *Specification* $M_S$ *and implementation* $M_I$

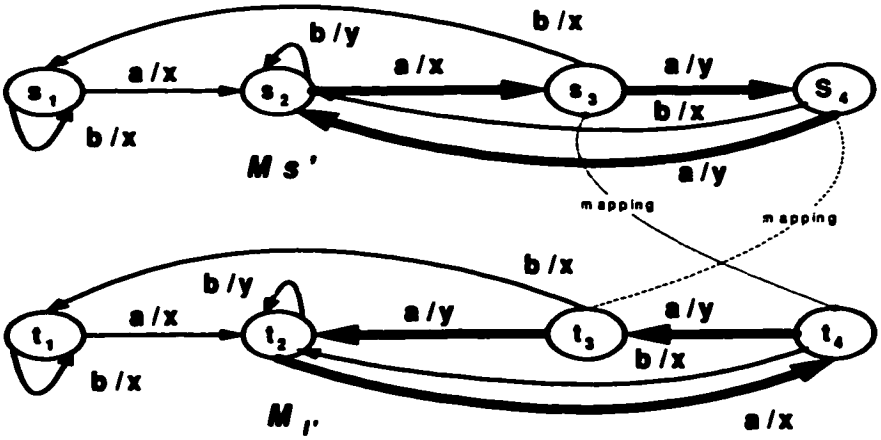


*Figure* 4.3. *Specification* $M_S'$ *and implementation* $M_I'$

We note that in the modified implementation state $t_4$ of $M_I'$ has the output response *yy* to *aa*, i.e. state $s_3$ of $M_S'$ is *W*-equivalent to $t_4$ of $M_I'$ while state $s_4$ of $M_I'$ is *W*-equivalent

to $t_3$ of $M_I'$. States $t_1$ and $t_2$ of $M_I'$ are $W$-equivalent to $s_1$ and $s_2$ of $M_I'$. The (X-2) holds for the modified transitions from states $t_3$ and $t_4$ under input $a$. However, it does not hold for the unmodified transitions at states $s_3$ of $M_S'$ and $s_4$ of $M_S'$ under input $b$, i.e. $M_I'$ is a wrong implementation of $M_S'$. Therefore, appropriate unmodified transitions also need to be checked in order to kill such a wrong mapping.

As in the previous section, we select a prefix-closed state cover set with the following property. If state $s_j \in S$ of $M_S'$ reachable through unmodified transitions, we select the sequence $\alpha_j \in Q$ that does not traverse any modified transition.

The re-testing method has two phases. In the first phase, some states of the modified specification are re-identified in the new implementation. It may occur that for some states of the modified specification that are only reachable through modified transitions their old image must still be preserved in the new implementation. In particular, those are the states that have a so-called stable identifier [Pet92] that distinguishes a state from any other state in each possible implementation. For each such state, we derive a state identifier (if it exists) that kills, through the re-identification phase, implementations where the state has a new image. We start from the set of states that are reachable through unmodified transitions. As in Case-2, the old mapping must still be valid for these states and only modified transitions from these states need to be checked. Moreover, if for each such state, the sequences of its state identifier do not traverse modified transitions then the state does not need to be re-identified. Otherwise, we select state re-identification sequences using Formula (2-1a) of Case-2, where in order to check the new mapping of a state, say $s_j$, we concatenate the sequence $r.\alpha_j$ with each sequence of the state identifier of the $s_j$ that passes through a modified transition. Then, we iteratively identify all other states for which the old mapping must be preserved; however, their state identifiers are derived in a proper way as described below. Afterwards, since each remaining state, say $s_j \in S_{r\text{-}m}$, of the modified specification could have a new image, i.e. $s_j$ could be mapped to say $t_k \in T_{r\text{-}m}$ instead of the old image $h_S$-

$f(s_j) = t_j$, re-tests are selected to re-identify the image of $s_j$ in the new implementation, i.e. check (or establish) that $s_j$ is W-equivalent to $t_k$, and to check that this mapping is conforming. In order to re-identify $s_j$, re-tests are selected by concatenating $r.\alpha_j$ with each sequence in the state identifier of state $s_j$ including sequences which traverse only unmodified transitions. Moreover, in order to check that this mapping is a valid one (i.e. kill wrong mappings), re-tests are selected to check each outgoing transition from $s_j$ for correct output and ending state in the new implementation.

In order to implement the above steps, we determine a subset $S_u$ of the set of states of the modified specification such that for the states in $S_u$ the old mapping between the states of the modified specification and its conforming modified implementation must still be preserved. The set $S_u$ enjoys a nice property. For each state in $S_u$, we do not need to check outgoing unmodified transitions from the state. In the following paragraph, we determine which states may be in the set $S_u$ and derive the set $S_u$ together with a separating family $F = \{W_1, \ldots, W_n\}$ (or characterization set $W$) so that if the implementation passes the re-identification test sequences, then there exists one-to-one mapping $h$: $S \rightarrow T$ such that the following property holds:

$$\text{For each state } s_i \in S_u \text{ we have: } s_i \equiv_{W_i} t \Leftrightarrow t = h_{S\text{-}f}(s_i). \qquad \text{(X-3)}$$

First, we add to the empty set $S_u$ each state $s_j$ that is reachable from the initial state through unmodified transitions. As in Case-2, the images of these states have to be still preserved in the new implementation. Then, for state $s_j \in S_u$ and each state $s_i \in S$, $s_i \neq s_j$, we include in the state identifiers $W_j$ and $W_i$ a sequence that distinguishes the states $s_j$ and $s_i$ in the modified specification. We note that, as discussed for Case-2, we recommend, while building the state identifier $W_j$, to select the sequences that do not pass through modified transitions if applied at state $s_j$, since we do not need to apply these sequences while re-identifying $s_j$.

Afterwards, we iteratively include in $S_u$ each state $s_j \in S \setminus S_u$, such that for each state $s_i \in S \setminus S_u$, $s_i \neq s_j$, there exists sequence $\beta_{ij}$ that does not traverse modified transitions if applied at states $s_i$ and $s_j$ and $\lambda_S(s_i, \beta_{ij}) \neq \lambda_S(s_j, \beta_{ij})$, or there exists input $x$ such that transitions $(s_j \text{ -}x\text{-> } s_k)$ and $(s_i \text{ -}x\text{-> } s_r)$ are unmodified, $s_k \neq s_r$ and $s_k, s_r \in S_u$. In the former case, we include sequence $\beta_{ij}$ in $W_i$ and $W_j$. Since $\beta_{ij}$ does not traverse modified transitions if applied at states $s_i$ and $s_j$ we have that $\Lambda_I(h_{S\text{-}I}(s_i), \beta_{ij}) = \lambda_S(s_i, \beta_{ij})$, and $\lambda_S(s_j, \beta_{ij}) \neq \Lambda_I(h_{S\text{-}I}(s_i), \beta_{ij})$. Thus, if $\beta_{ij}$ is included into $W_i$ and $W_j$ and the implementation passes the corresponding state re-identification sequences, then $s_j$ is not $W$-equivalent to $h_{S\text{-}I}(s_i)$ (i.e. $h_{S\text{-}I}(s_j) \neq h_{S\text{-}I}(s_i)$). In the latter case, we include into $W_i$ and $W_j$ the sequence $x\beta$ where $\beta$ is a common prefix of the appropriate sequences in $W_i$ and $W_j$ such that $\lambda_S(s_k, \beta) \neq \lambda_S(s_r, \beta)$.Thus, if $\Lambda_I(h_{S\text{-}I}(s_i), x.\beta) = \lambda_S(s_i, x.\beta)$, then $\lambda_S(s_j, x.\beta) \neq \Lambda_I(h_{S\text{-}I}(s_i), x.\beta)$. If $x.\beta$ is included in $W_i$ and $W_j$ and the implementation passes the corresponding state re-identification sequences, then $s_j$ is not $W$-equivalent to $h_{S\text{-}I}(s_i)$. Due to the definition of state identifiers for states in $S_u$, such a sequence exists. If any sequence of each state identifier is defined at each state then we derive the set $W$ as union of all state identifiers. We note that in order to kill for $s_j$ any mapping $h_I$ where $h_I(s_j) \neq h_{S\text{-}I}(s_j)$, the corresponding state re-identification sequences are derived by concatenating $r.\alpha_j$ with every sequence of the set $W_j$ (or $W$ for the Wp and $W$ methods).

Finally, we derive state identifiers for the remaining states in $S \setminus S_u$. For each pair of states $s_j$ and $s_i$ in $S \setminus S_u$ , $s_i \neq s_j$, we include a sequence $\beta_{ij}$ in $W_i$ and $W_j$ (if it does not already exist) such that $\lambda_S(s_i, \beta_{ij}) \neq \lambda_S(s_j, \beta_{ij})$. In order to re-identify $s_j$ in the new implementation and kill its possible wrong images, the corresponding re-testing sequences include all re-identification sequences and re-testing sequences for testing all outgoing transitions from state $s_j$. The characterization set $W$ (for the W, and Wp methods) can be obtained as the union of state identifiers $W_i$, $i$=1, ..., $n$ (if possible). We note that in order to reduce the number of transitions which need to be checked we use another technique than that based on stable state identifiers [Pet92]. The main idea behind

our approach is based on the observation that for each state reachable through unmodified transitions and some other states, the old image must be preserved in each conforming modified implementation and we select appropriate state identifiers, for which (X-3) holds. As we illustrate by the following example, not each such a state identifier is stable. Our technique can also be used to reduce a test suite derived from a mutation machine [Kou99] if the latter has many deterministic transitions.

**i) Phase of state re-identification**

For each state $s_j$ of the modified specification that needs to be re-identified in the new implementation, we derive its state re-identification test sequences as follows:

If $\alpha_j$ does not traverse a modified transition, the re-identification sequences are formed as in Formula (2-1a).

If $\alpha_j$ traverses a modified transition then there are test sequences

$r.\alpha_j.W_j$ (in the HIS method); **(3-1a)**

$r.\alpha_j.W$ (in the W and Wp methods). **(3-1b)**

Every sequence of the set $W_j$ (or $W$) must be applied after $\alpha_j$, whether the sequence applied at state $s_j$ traverses a modified transition or not.

**ii) Phase of re-testing modified transitions**

For each modified edge ($s_j$ - $x$ -> $s_k$), where $s_j \in S_u$, its corresponding test cases are formed as in Formulae (1-a) or (1-b). For each state $s_j \notin S_u$, Formula (1-a) or (1-b) are applied for each outgoing transition from state $s_j$ including those which are unmodified.

**Theorem 4.** Given the modified specification $M_S'$ and implementation $M_I'$, let $Q$ be a prefix-closed state cover set of $M_S'$ and $F = \{W_1, \ldots, W_n\}$ and $W$ be a separating family and a characterization set (if exists) of the modified specification $M_S'$ derived as described above. If the implementation $M_I'$ passes the re-testing test suite derived for Case-3, then the implementation is quasi-equivalent to $M_S'$.

**Proof of Theorem 4:** As we demonstrated by the example, in Case 3, a one-to-one mapping $h$: $S \rightarrow T$ such that state $s_i$ of the modified specification is $W_i$-equivalent to state $h(s_i)$ of the modified implementation can be different from $h_{S\text{-}I}$. We first need to check whether the one-to-one mapping $h$ exists at all.
We consider a relation $h \in S \times T$ such that

$$(s_j, t_j) \in h \Leftrightarrow s_j \equiv_{W_j} t_j.$$

If the implementation passes the re-identification test cases given by Formulae (2-1a), (3-1a) and (3-1b) then $h$ is a one-to-one mapping $h$: $S \rightarrow T$. We next show that $h(s_j) = h_{S\text{-}I}(s_j)$ holds for each $s_j \in S_u$.

Given state $s_j \in S$ of the modified specification such that the sequence $\alpha_j \in Q$ does not traverse modified transitions, if the implementation passes the re-identification test sequences given in Formulae (3-1a) then the state $h_{S\text{-}I}(s_j)$ to which $\alpha_j$ takes the implementation from the initial state, is $W_j$-equivalent to $s_j$, i.e. $h(s_j) = h_{S\text{-}I}(s_j)$. The initial state $s_1$ is in the set $S_u$, i.e. the base of induction holds.

Let us assume that $h(s) = h_{S\text{-}I}(s)$ holds for each state $s$ of a current set $S_u$ and that state $s_j$ is the next state we are going to include into $S_u$ using the procedure described above. Since $h$ is a one-to-one mapping, for each state $s \in S_u$ it holds that $h(s_j) \neq h_{S\text{-}I}(s)$. On the other hand, for each state $s_i \in S \backslash S_u$ , $i \neq j$, by construction of the state identifier, there exists a sequence $\beta_{ij} \in W_j \cap W_i$ such that $\beta_{ij}$ does not traverse modified transitions if

applied at states $s_i$ or $s_j$ and $\lambda_S(s_i, \beta_{ij}) \neq \lambda_S(s_j, \beta_{ij})$, or there exists a sequence $x\beta \in W_j \cap W_i$ such that the final states of unmodified transitions $(s\text{ -}x\text{->}\ s_k)$ and $(s_i\text{ -}x\text{->}\ s_r)$ are different and $\lambda_S(s_k, \beta) \neq \lambda_S(s_r, \beta)$.

In the former case, the modified implementation has different output responses to the sequence $\beta_{ij} \in W_j \cap W_i$ at the states $h(s_j)$ and $h_{S\text{-}I}(s_i)$, i.e. $h(s_j) \neq h_{S\text{-}I}(s_i)$. In the latter case, the modified implementation at states $h_{S\text{-}I}(s_k)$ and $h_{S\text{-}I}(s_r)$ has different output responses to the sequence $\beta \in W_k \cap W_r$, i.e. the modified implementation has different output responses to the sequence $x\beta \in W_k \cap W_r$ at the states $h(s_j)$ and $h_{S\text{-}I}(s_i)$, i.e. $h(s_j) \neq h_{S\text{-}I}(s_i)$. Therefore, by induction, $h(s_j) = h_{S\text{-}I}(s_j)$ for each state $s_j \in S_u$.

The proof of the fact that if the modified implementation passes re-testing test sequences then (X-2) holds for each transition of the modified specification coincides with the corresponding proof for Theorem 2.

Thus, if the implementation passes the test, then the mapping $h$ satisfies (X-2), i.e. the modified implementation is quasi-equivalent to the modified specification □

#### 4.4.3.1 An application example

As an application example for Case-3 with the HIS method, we add to the given specification a new state $s'4$ and its corresponding incoming and outgoing edges producing the modified specification $M_S'$ shown in Fig. 4.4 below.

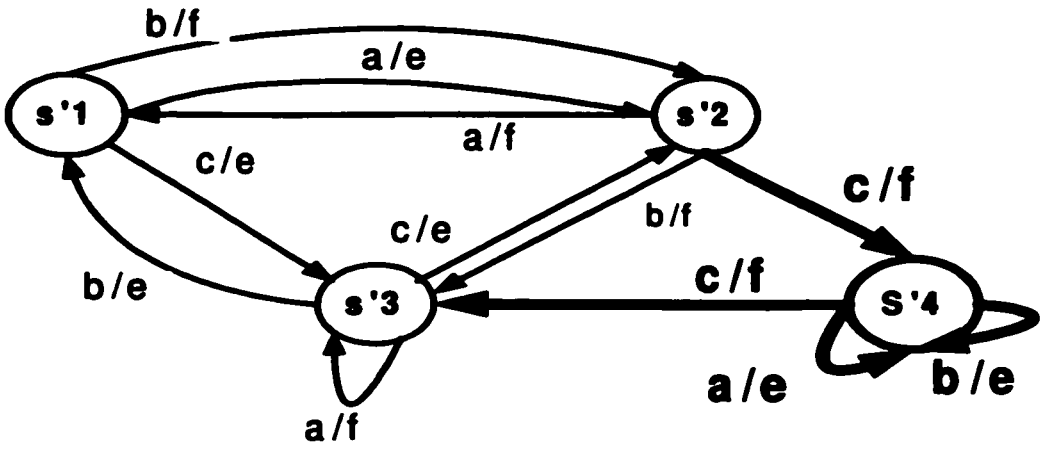


*Figure* 4.4. *Specification* $M_S'$

When new states are added to the modified specification we fall into Case-3. In fact, in this case, the unmodified part of the modified specification is not reduced, and the new states are only reachable through modified transitions. However, many states, namely those of the old specification, are distinguished with sequences which traverse unmodified transitions. Moreover, we notice that states $s'_1$, $s'_2$ and $s'_3$ have no stable identifiers, since the new state $s'_4$ can be faulty implemented being equivalent to any other state.

The state cover set of $M_S'$ is $Q'=\{\varepsilon, b, c, bc\}$. We consider each incoming and outgoing transition of the added state (here $s'_4$) as a modified transition. Therefore, the modified transitions of $M_S'$ are ($s'_2$-c/f->$s'_4$), ($s'_4$-b/f->$s'_4$), ($s'_4$-a/e->$s'_4$), and ($s'_4$-c/f->$s'_3$).

According to Case-3, we add to the set $S_u$ states $s'1$, $s'2$, and $s'3$ since these states are reachable through unmodified transitions and there exists a state identifier for each of these states that does not pass through modified transitions. In this example, the sequence *bb* is such an identifier. In fact, we have the following input/output sequences in response to *bb*:

For state $s'1$: *ff*,

For state $s'2$: *fe*,

For state $s'3$: *ef*, and

For state $s'4$: *ee*.

These states, i.e., $s'1$, $s'2$, and $s'3$ , do not need to be re-identified in the new implementation. In order to re-identify the added state $s'4$, the re-test sequence $r.\alpha 4.W4$ = *r.bc.bb* is selected using Formula (3-1a). If the modified implementation passes this sequence then there is a one-to-one mapping between states of the modified specification and implementation that are *bb*-equivalent.

Afterwards, in order to test the modified transition ($s'2$-c/f->$s'4$) whose head state $s'2$ is in $S_u$, the sequence $r.\alpha 2\ c.W4$ = *r.b.c.bb* is selected using Formula (1-a). Moreover, the following re-testing test sequences are selected using Formula (3-1c) in order to check the outgoing transitions from state $s'4 \in S \backslash S_u$ : $r.\alpha 4.a.W4 + r.\alpha 4.b.W4 + r.\alpha 4.c.W3$ = *r.bc.a.bb* + *r.bc.b.bb* + *r.bc.c.bb*

Consequently, the re-testing test suite has sequences of total length 18. The traditional HIS method derives a test suite of length 32 if the whole specification of $MS'$ is considered for test derivation.

## 4.5 Re-testing Methods when the Implementation Has more States than the Specification

In this section, we introduce the re-testing problem for the case when the implementation can have more states than the specification. We first briefly describe the W, Wp, and HIS test derivation methods for this case. Based on these methods we propose corresponding methods for deriving re-testing test cases.

Let the number of states $m$ of an implementation FSM be larger than the number $n$ of states of the specification FSM (i.e. $m > n$). We denote by $\overline{Q}$ the union $\alpha_r.(\mathbf{DIS}(M_S|s_r)\cap X^{m-n})$ over all $\alpha_r \in Q$. When an implementation can have more states than the specification the set $\overline{Q}$ is used instead of the state cover set $Q$, since now the set $Q$ is not enough in order to reach each state of the implementation in the state identification phase. For each sequence $\alpha_j \in \overline{Q}$ that takes the specification FSM to the corresponding state $s_j$, the state identification test sequences comprise the following sequences:

$r.\alpha_j.W$ in the W and Wp methods, or

$r.\alpha_j.W_j$ in the HIS method.

Here we note again that a $W$ set does not always exist for a reduced partially specified FSM. In this case, the W and Wp methods cannot be applied. However, we can apply the HIS method.

If the FSM $M_I$ passes these sequences, then $\overline{Q}$ is shown to be a state cover set of any implementation of $M_S$ up to $m$ states ([Cho78], for a complete specification FSM; [Yev90] for a partial specification FSM) and there exists a many-to-one mapping $h$: $T \rightarrow S$ such that:

$h(t) = s_j \Leftrightarrow t \equiv_{W_j} s_j$ in the HIS method, and

$h(t) = s_j \Leftrightarrow t \equiv_{W} s_j$ in the W and Wp methods (Y-1)

Moreover, the transition testing phase, assures that for each state $t \in T$ that is reachable from the initial state in the implementation at hand, and input $x \in X$ that is defined at state $s=h(s)$, the mapping $h$ satisfies the following property:

$$\Lambda_I(t, x) = \lambda_S(h(t), x) \text{ and } h(\Delta_I(t, x)) = \delta_S(h(t), x) \qquad \text{(Y-2)}$$

For this purpose, for each sequence $\alpha_j \in \overline{Q}$ that takes the specification FSM to the corresponding state $s_j$ and each $x \in X$ that takes the FSM $M_S$ from state $s_j$ to state $s_k$, the testing transition phase comprises:

$r.\alpha_j.x.W$ in the W method, or

$r.\alpha_j.x.W_k$ in the HIS and Wp methods, where $W_k$ is a state identifier of the state $s_k$ in the specification FSM and for the Wp method, we have $W_k \subseteq W$.

When deriving the re-testing test cases we assume that the complete implementation $M_I$ of the reduced FSM $M_S$ has been tested and found conforming to $M_S$. Therefore, there exists a many-to-one mapping $h_{I\text{-}S}$: $T \rightarrow S$ such that for each state $t \in T$ that is reachable from the initial state in the implementation at hand, and input $x \in X$ that is defined at state $s$, the mapping $h_{I\text{-}S}$ satisfies (Y-2). We denote by $h^{-1}_{I\text{-}S}$ the *inverse image* of $h_{I\text{-}S}$, i.e. given state $s_j$ of $M_S$, $h^{-1}_{I\text{-}S}(s_j) = \{t_j \mid h_{I\text{-}S}(t_j) = s_j\}$. Let $M_S' = (S, X, Y, \delta_S, \lambda_S, D_S, s_1)$ be the modified specification, and $M_I'=(T, X, Y, \Delta_I, \Lambda_I, t_1)$ be the modified implementation that must conform to $M_S'$. We assume that only transitions that correspond to the modified parts of $M_S'$ have been changed in $M_I'$ and we want to generate test sequences for the modified parts of the system specification, in order to check that these modifications were implemented correctly in the modified implementation $M_I'$. In other words, for each unmodified transition ($s_j$ - $a$ -> $s_k$) of $M_S'$, we assume that each transition ($t_j$-$a$->$t_k$) [where $t_j \in h^{-1}_{I\text{-}S}(s_j)$ and $t_k \in h^{-1}_{I\text{-}S}(s_k)$] has not been changed in the modified implementation $M_I'$. In this case, we regard the modified implementation as a grey-box

rather than a black-box since some parts of $M_I'$ are known (equal to the corresponding parts of the initial implementation $M_I$) while others are not, due to the modifications.
The re-testing methods have two phases similar to the methods presented for the case when the specification and the implementation have the same number of states. As in the previous methods, we also examine different cases that can be used to generate shorter re-testing test sequences. Moreover, we examine $M_S'$ and utilize some information from the specification $M_S$ and its conforming implementation $M_I$ to select (when possible) an appropriate state cover set of any implementation of $M_S'$ up to $m$ states that is shorter than that presented in ([Cho78]; [Yev90]).

Similar to the previous sections, we select a prefix-closed state cover set $Q$ with the following property. Given state $s_j \in S$ of $M_S'$ reachable through unmodified transitions, we select the sequence $\alpha_j \in Q$ that does not traverse any modified transition.

### 4.5.1 Case-4.1: The Unmodified Part of the Modified Specification is Reduced

If the modified specification is still reduced, as in Case-1, we do not need to re-identify states in the new implementation since the mapping between the states of $M_I$ and $M_S$ is still preserved between the states of $M_I'$ and $M_S'$ and it is the only candidate that can satisfy (Y-2).

Moreover, we know that since $M_I$ conforms to $M_S$, the union $\alpha_r.(\mathbf{DIS}(M_S|s_r) \cap X^{m-n})$ over all $\alpha_r \in Q$ (i.e $\overline{Q}$) is a state cover set of $M_I$. By utilizing this information, we build (when possible) a shorter state cover set for $M_I'$. Afterwards, when testing a modified transition $(s_j - x -> s_k)$ of $M_S'$, we select the sequences from this set that reach each state of each possible implementation of $M_S'$ that is the head state of a modified transition of the implementation that corresponds to $(s_j - x -> s_k)$, i.e. we reach each state $t_j$ such that $t_j \in h_{I-S}^{-1}(s_j)$.

In order to build a state cover set from $M_S'$ by utilizing accessible information from $M_I'$, for each state $s_j$ of $M_S'$ we proceed as follows:

- If state $t_j$ of $M_I'$ is the only inverse image of $s_j$ of $M_S'$ (i.e. $|h_{I-S}^{-1}(s_j)| = 1$), then we assign $Q_j' = \{\alpha_j\}$.
- If there is more than one inverse image for state $s_j$ of $M_S'$ (i.e. $|h_{I-S}^{-1}(s_j)| > 1$) and the sequence $\alpha_j$ does not traverse unmodified transitions if applied at the initial state of $M_S'$, then:

  For each inverse image $t$ of $s_j$ (other than $t_j$ that is reachable by $\alpha_j$ in $M_I$) that is reachable from the initial state of the implementation through unmodified transitions (if any), we let $\beta_t$ denote the sequence that reaches $t$. Then, we form the set of sequences:

  $Q_j' = \{\alpha_j\} \cup \{\beta_t \mid s_j = h_{I-S}(t)\}$
- If there is more than one inverse image for state $s_j$ of $M_S'$ and the sequence $\alpha_j$ traverses modified transitions if applied at the initial state of $M_S'$, then we assign $Q_j' = \{\alpha_j\}$.

Afterwards, we form the set of sequences $Q' = \bigcup_{Sj \in S} Q_j'$. If the sequences of the set $Q'$ take the modified implementation $M_I'$ to $|Q'|$ different states, then the union $\alpha_r.[\mathbf{DIS}(M_S|s_r) \cap X^{m-|Q'|}]$ over all $\alpha_r \in Q'$ (Hereafter denoted by $\overline{Q'}$) is a state cover set of any possible implementation with up to $m$ states.

For each modified transition $(s_j \text{ - } x \text{ -> } s_k)$, we let the set $R_{t_j}$ be the subset of the set $\overline{Q'}$ that covers the states in the set $h_{I-S}^{-1}(s_j)$ of $M_I'$. In other words, we include in $R_{t_j}$ each sequence in $\overline{Q'}$ that takes the modified specification from the initial state to state $s_j$. We derive the state identifiers $W_k$ and the set $W$ (if exists) which in this case do not traverse

modified transitions. Then, for each sequence $\gamma \in R_{t_j}$ we form its corresponding re-testing test sequences as follows:

If we use the HIS or Wp methods:

$$r.\gamma x.W_k \qquad \textbf{(4-a)}$$

If a characterization set $W$ exists and we use the W method:

$$r.\gamma x.W \qquad \textbf{(4-b)}$$

We note that we could use, as described in ([Cho78];[Yev90], the state cover set $\overline{Q} = \bigcup_{\alpha_r \in Q}[\alpha_r.\mathrm{DIS}(M_S \cap X^{m-n})]$ instead of $\overline{Q'} = \bigcup_{\alpha_r \in Q'}[\alpha_r.\mathrm{DIS}(M_S \mid s_r \cap X^{m-|Q'|})]$. However, this set is normally larger since it does not exploit some known information about $M_I'$ inherited from $M_I$.

**Theorem 5.** Given a modified specification $M_S'$ and its implementation $M_I'$, let the unmodified part of $M_S'$ be reduced and have state identifiers $W_1$, ..., $W_n$. If the implementation $M_I'$ passes the re-testing test suite which consists of the union of the test cases over all modified transitions as given in Formula (4-a) or Formula (4-b), then the implementation $M_I'$ is quasi-equivalent to $M_S'$.

In order to prove Theorem 5, we first prove the following lemma.

**Lemma 1.** Given state $s_j$ of $M_S'$ and the set $R_{t_j} \subseteq \overline{Q'}$. Let state $t_j$ in the inverse image of $s_j$ be reachable from the initial state of $M_I'$. If the implementation passes the re-testing test suite which is the union of the test cases given in Formula (4-a) or Formula (4-b) over all modified transitions, then there exists a sequence $\gamma \in R_{t_j}$ that takes $M_I'$ to state $t_j$. In other words, $R_{t_j}$ is a state cover set of all states of the inverse image of $s_j$.

**Proof of Lemma 1:** Consider the many-to-one mapping $h_{I\text{-}S}(t)$: $T \rightarrow S$ that holds between specification $M_S$ and implementation $M_I$. We first note that since the unmodified part $UP\text{-}M_S'$ of the modified specification $M_S'$ is reduced, there exist state identifiers $W_1,\ldots, W_n$, such that each $W_i$ is a subset of the defined input sequences at state $s_i \in S$ in the $UP\text{-}M_S'$, Therefore, for each state $t$ of the modified implementation it holds that :

$$t_j \equiv_{W_j} s \Leftrightarrow s = h_{I\text{-}S}(t_j). \qquad \text{(Y-3)}$$

Given state $t_j$ of the modified implementation $M_I'$, consider the sequence $\gamma \in \overline{Q'}$ that takes $M_I'$ from the initial state to state $t_j$. If $\gamma$ does not traverse modified transitions then $\gamma$ takes the modified specification to state $s_j$ such that $s_j = h_{I\text{-}S}(t_j)$, i.e.

$$\Delta_I(t_1, \gamma) \equiv_{W_j} \delta_S(s_1, \gamma). \qquad \text{(Y-4)}$$

Suppose now the sequence $\gamma$ traverses a modified transition. Without loss of generality we can assume that $\gamma = \beta z$ where $\beta$ does not traverse a modified transition if applied at the initial state of $M_S'$. In this case, if the implementation passes the test cases $\beta.z.W_j$ of the test suite given in Formula (4-a), then we are sure that after the sequence $\gamma = \beta z$ the modified implementation reaches state $t_j$ that is $W_j$-equivalent to $s_j$, i.e. the formula (Y-4) holds for any $\gamma \in \overline{Q'}$. Therefore, we can draw the following two conclusions:

a) Given state $t_j$ of the implementation $M_I'$ such that $h_{I\text{-}S}(t_j) = s_j$ and a sequence $\gamma \in \overline{Q'}$, if the sequence $\gamma$ takes the implementation from the initial state to state $t_j$, then $\gamma$ takes the specification $M_S'$ from the initial state to state $s_j$, and by definition of $R_{t_j}$, $\gamma \in R_{t_j}$.

b) The sequences in the state cover set $Q$ of $M_S'$ take the implementation from the initial state to $n$ different states. Moreover, for each sequence $\beta \in Q' \setminus Q$, by construction of $Q'$, the $\beta$ takes the implementation from the initial state to a state different than those that are reached by the sequences in $Q$. Therefore, by definition of $Q'$, sequences in $Q'$ take the implementation to $|Q'|$ different states, then in order to cover each state of $M_I'$, at

each state we need to apply all defined input sequences of length up to $m$-$|Q|$. Moreover, due to (a), $R_{t_j} \subseteq \overline{Q'}$ covers each state $t_j$. □

**Proof of Theorem 5:**

We now show the mapping $h_{I\text{-}S}$ satisfies (Y-2) for the modified specification $M_S'$ and implementation $M_I'$.

For each state $t_j \in T$, the following three cases are possible:

(i) State $t_j$ of $M_I'$ is the only inverse image of state $s_j$ of $M_S'$ (i.e. $s_j \equiv_{W_j} t_j$ and $|h^{-1}_{I\text{-}S}(s_j)| = 1$) and the sequence $\alpha_j$ does not traverse modified transitions if applied at the initial state of $M_S'$. In this case, the implementation $M_I'$ reaches state $t_j$ after $\alpha_j$.

Given a non-modified transition ($s_j$ - $a$ -> $s_k$), the modified implementation from state $t_j$ under input $a$ reaches state $t_k$ that is $W_k$-equivalent to state $s_k$. Thus, in this case, for each unmodified transition, it holds that

$$\Lambda_I(t_j, a) = \lambda_S(h_{I\text{-}S}(t_j), a) \text{ and } h_{I\text{-}S}(\Delta_I(t_j, a)) = s_k = \delta_S(h_{I\text{-}S}(t_j), a)$$

Given a modified transition ($s_j$ - $x$ -> $s_k$), if the implementation passes the test cases $r.\alpha_j.x.W_k$ given in Formula (4-a), then $\Lambda_I(t_j, x) = \lambda_S(h_{I\text{-}S}(t_j), x)$ and the ending state of the transition ($t_j$ - $x$ -> $t_k$) is $W_k$-equivalent to $s_k$, i.e. $s_k = h_{I\text{-}S}(t_k)$. Therefore, $h_{I\text{-}S}(\Delta_I(t_j, a)) = s_k = \delta_S(h_{I\text{-}S}(t_j), a)$.

(ii) State $t_j$ of $M_I'$ is the only inverse image of $s_j$ of $M_S'$ (i.e. $s_j \equiv_{W_j} t_j$ and $|h^{-1}_{I\text{-}S}(s_j)| = 1$); however, the sequence $\alpha_j$ traverses modified transitions if applied at the initial state of $M_S'$. Without loss of generality, we can assume that $\alpha_j = \beta z$ where $\beta$ does not traverse a modified transition if applied at the initial state of $M_S'$. In this case, if the implementation passes the test cases $\beta.z.W_j$ of the test suite given in Formula (4-a) we are sure that after

sequence $\alpha_j = \beta x$ the modified implementation reaches state $t_j$ that is $W_j$-equivalent to $s_j$, i.e. it reaches state $t_j$ such that $h_{I\text{-}S}(t_j) = s_j$.

Given a non-modified transition ($s_j$ - $a$ -> $s_k$), the modified implementation from state $t_j$ under input $a$ reaches state $t_k$ that is $W_k$-equivalent to state $s_k$ of the modified specification, i.e. $h_{I\text{-}S}(t_k) = s_k$. Thus, in this case, for each unmodified transition, it also holds that:

$$\Lambda_I(t_j, a) = \lambda_S(h_{I\text{-}S}(t_j), a) \text{ and } h_{I\text{-}S}(\Delta_I(t_j, a)) = s_k.$$

Thus, when the modified implementation passes the re-testing test cases $r.\alpha_j.x.W_k$ given in Formula (4-a), it holds that $\Lambda_I(t_j, a) = \lambda_S(h_{I\text{-}S}(t_j), a)$ and the ending states of the transitions ($s_j$ - $x$ -> $s_k$) are ($t_j$ - $x$ -> $t_k$) are $W_k$-equivalent, i.e. again $h_{I\text{-}S}(t_k) = s_k$.

(iii) The modified implementation $M_I'$ has at least two states state $t_j'$ and $t_j''$ that are equivalent to state $s_j$ of the unmodified specification, i.e. $h_{I\text{-}S}(t_j') = s_j$ and $h_{I\text{-}S}(t_j'') = s_j$. In this case, if states $t_j'$ and $t_j''$ of $M_I'$ are reachable from the initial state, then $M_I'$ reaches state $t_j'$ (or $t_j''$) through a sequence of the set $R_{t_j}$.

Given a non-modified transition ($s_j$ - $a$ -> $s_k$), the modified implementation from state $t_j'$ (or $t_j''$) under input $a$ reaches state $t_k'$ (or $t_k''$) that is $W_k$-equivalent to state $s_k$. Thus, in this case, for each unmodified transition, it holds that

$$\Lambda_I(t_j', a) = \lambda_S(h_{I\text{-}S}(t_j'), a) \text{ and } h_{I\text{-}S}(\Delta_I(t_j', a)) = s_k.$$

Given a modified transition ($s_j$ - $x$ -> $s_k$), if the implementation passes the test cases given in Formula (4-1a), then given state $t_j'$ (or $t_j''$), $\Lambda_I(t_j', x) = \lambda_S(h_{I\text{-}S}(t_j'), x)$ and the ending state of the transition ($t_j'$ - $x$ -> $t_k'$) is $W_k$-equivalent to $s_k$, i.e. $s_k = h_{I\text{-}S}(t_j')$. □

#### 4.5.1.1 **An application example**

As an example for Case 4.1 based on the W method, we modify the specification $M_S$ shown in the left part of Fig. 4.5 and obtain the FSM $M_S'$ shown in the left part of Fig. 4.6. The modified transitions are shown as bold lines. We let $M_I$, shown in the right part of Fig. 4.5, be the conforming implementation of $M_S$ where the mapping between the states of $M_I$ and $M_S$ is $h_{I\text{-}S}(t_1) = s_1$, $h_{I\text{-}S}(t_2) = s_2$, $h_{I\text{-}S}(t_3') = s_3$, and $h_{I\text{-}S}(t_3'') = s_3$. Moreover, we let $M_I'$, shown in right part of Fig. 4.6, be the implementation of $M_S'$ where the modified transitions of $M_I'$ that correspond to the modified transitions of $M_S'$ are shown in bold. The number $n$ of states of $M_S'$ is 3 while that of $M_I'$ is $m$=4. $M_S'$ has $b$ as its distinguishing sequence, and $Q = \{\varepsilon, a, b\}$ as its state cover set. Here, $Q'_1=\{\varepsilon\}$, $Q'_2=\{\alpha_2 = a\}$, $Q'_3=\{\alpha_3' = b, \beta_3' = c\}$. Therefore, $Q' =\{\varepsilon, a, b, c\}$ with $|Q|=4$, $X^{m-|Q|} = X^0 = \{\varepsilon\}$, and the union $\alpha_r.[\mathbf{DIS}(M_S|s_r)\cap X^{m-|Q|}]$ over all $\alpha_r \in Q'$ equals to $Q'$. Moreover, $R_{t_3} = \{b, c\}$, and $TS = r.\alpha_2.a.b + r.R_{t_3}.b = r.a.a.b + r.\{b, c\}.a.b$ has a total length of 12. The W method generates sequences of total length 123 if the whole specification $M_S'$ is used for test derivation.

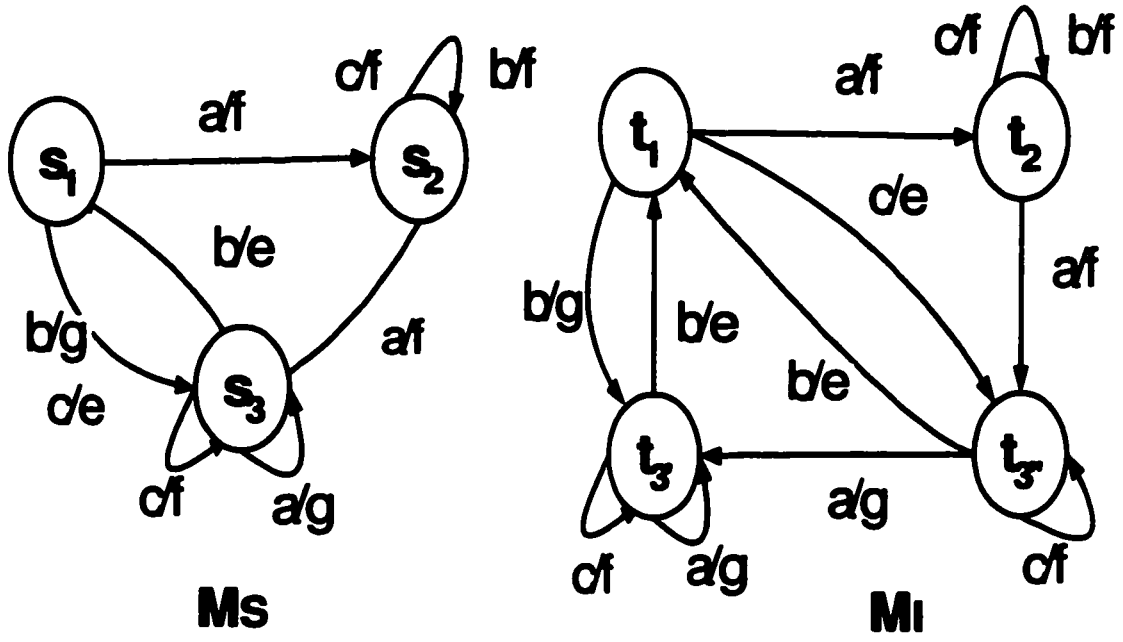


*Figure 4.5. Specification $M_S$ and its implementation $M_I$*

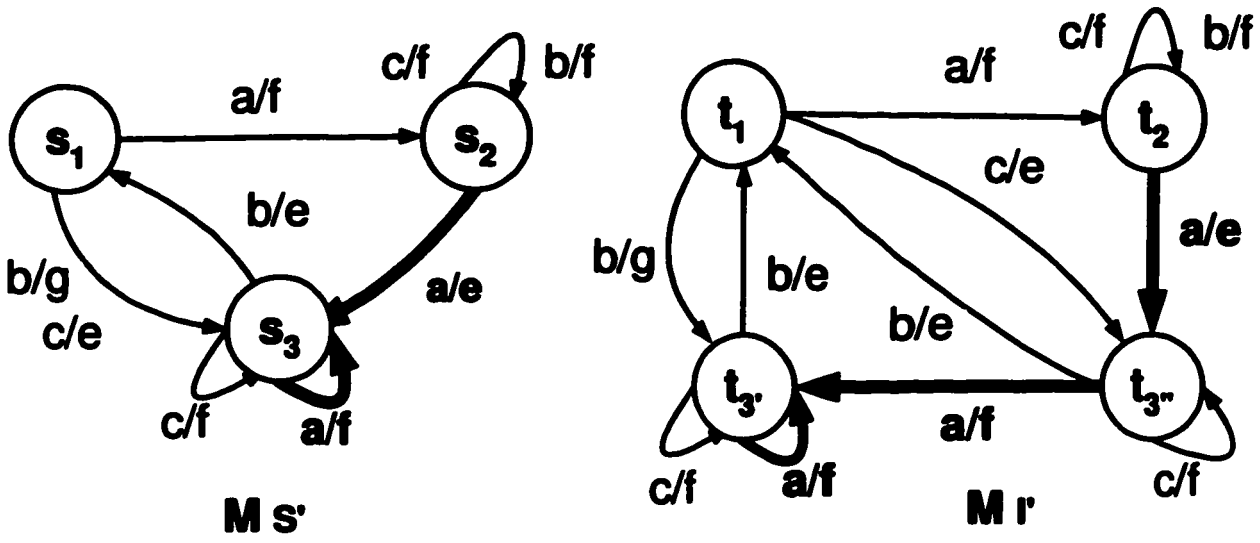


*Figure* 4.6. *Specification $M_S'$ and its implementation $M_I'$*

We note that if we do not utilize in $Q'_3$ the information $\beta_{3'} = c$, where $c$ is the input symbol that takes $M_I'$ to state $t_{3''}$ through the unmodified transition ($t_1$ - *c/e* -> $t_{3''}$), then $Q'_3$ would be equal to $\{\alpha_{3'} = b\}$. In this case, $Q' = \{\varepsilon, a, b\}$ with $|Q'| = 3$, $X^{m-|Q'|} = X$, and the union $\alpha_r.[\mathbf{DIS}(M_S|s_r) \cap X^{m-|Q'|}]$ over all $\alpha_r \in Q'$ equals to $= Q'.X = = \{a, b, c, aa, ab, ac, ba, bb, bc\}$. In this case, $R_{t_3} = \{b, c, aa, ba, bc\}$, and $TS = \{r.\alpha_2.a.b\} + \{r.R_{t_3}.a.b\} = \{r.a.a.b\} + \{r.b.a.b, r.c.a.b, r.aa.a.b, r.ba.ab, r.bc.a.b\}$ has total length 27.

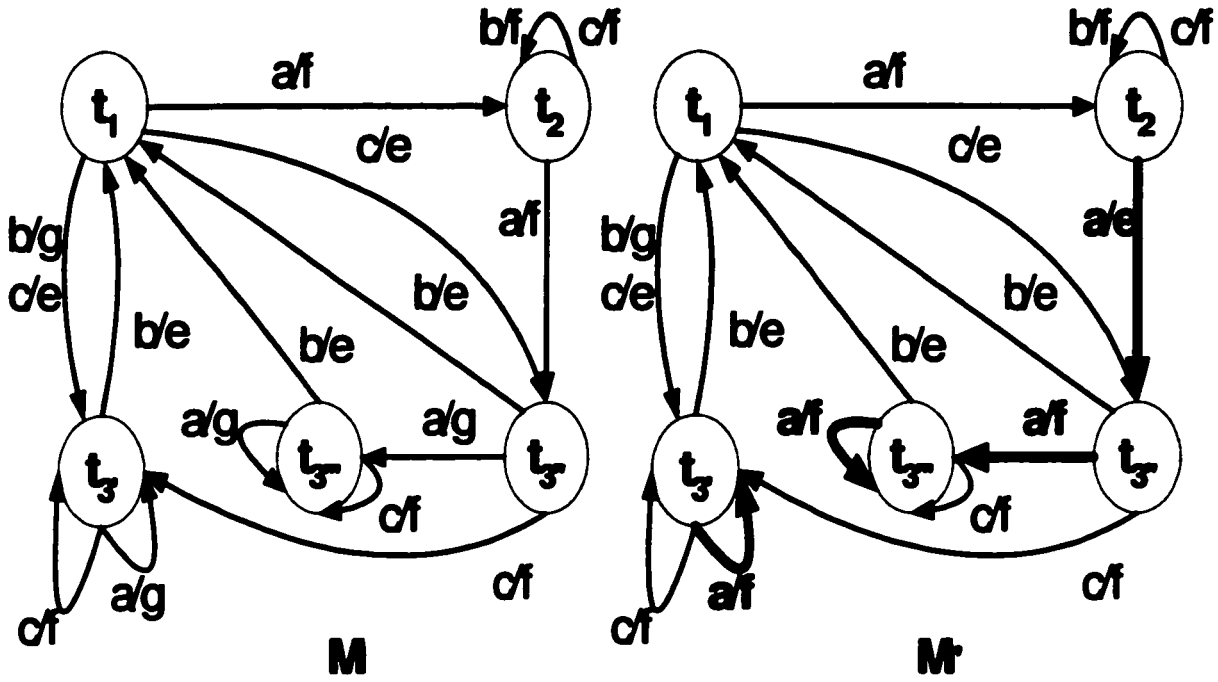


*Figure* 4.7. *Implementation $M_I'$ of specification $M_S'$ in Fig.* 4.6

As another example, we consider the specification $M_S$, shown in the left part of Fig. 4.5, and its conforming implementation $M_I$ shown in the left part of Fig. 4.7, and the modified specification $M_S'$ shown in the left part of Fig. 4.6. The mapping between the states of $M_I$ and $M_S$ is $h_{I\text{-}S}(t1) = s1$, $h_{I\text{-}S}(t2) = s2$, $h_{I\text{-}S}(t3') = s3$, $h_{I\text{-}S}(t3'') = s3$ and $h_{I\text{-}S}(t3''') = s3$. Moreover, we let $M_I'$, shown in the right part of Fig. 4.7, be the implementation of $M_S'$ where the modified transitions of $M_I'$ that correspond to the modified transitions of $M_S'$ are shown in bold. The number $n$ of states of $M_S'$ is 3 while that of $M_I'$ is $m$=5. $M_S'$ has $b$ as a distinguishing sequence, and $Q = \{\varepsilon, a, b\}$ as a state cover set. Here $Q'1=\{\varepsilon\}$, $Q'2=\{\alpha 2 = a\}$, $Q'3=\{\alpha 3'= b, \beta 3'= c\}$, $Q' =\{\varepsilon, a, b, c\}$ with $| Q' | = 4$. Therefore, $X^{m-|Q'|} = X$, and the union $\alpha_r.[\mathrm{DIS}(M_S|s_r) \cap X^{m-|Q'|}]$ over all $\alpha_r \in Q'$ equals to $Q'.X = \{a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc\}$. Here, $R_{t3} = \{\alpha 3'= b, \beta 3'= c, aa, ba, bc, ca, cc\}$. We note that the sequence $ab$ in $Q'.X$ is not included in $R_{t3}$ since this sequence does not traverse a modified transition if applied at the initial state $s1$ of $M_S'$ and the tail state of this sequence is not $s3$. By the same reasoning, we do not include in $R_{t3}$, the sequences

*ac*, *bb*, and *cb* in $Q'.X$. Then we form, $TS = r.\alpha_2.a.b + r.R_{t_3}.a.b = \{r.a.a.b\} + \{r.c.a.b, r.b.a.b, r.aa.a.b, r.ba.a.b, r.bc.a.b, r.ca.a.b, r.cc.a.b\}$ of total length 37. The W method generates sequences of length 452 if the whole specification $M_S'$ is used for test derivation.

Here we notice that similar to the optimized procedure of Case-1 we can use transitions that have been already tested in order to increase the cardinality of the set $Q'$. As shown above, this implies shorter re-testing sequences.

### 4.5.2 Case-4.2: Each State of the Modified Specification is Reachable through Unmodified Transitions and the Unmodified Part is not Reduced.

When the unmodified part of the modified specification $M_S'$ is not reduced and all states of the modified specification are reachable through unmodified transitions, all the states of the modified implementation are also reachable through unmodified transitions. Similar to Case-2, the mapping between the states of $M_I$ and $M_S$ is the only candidate that can satisfy (Y-2) for $M_I'$ and $M_S'$. However, we have to check whether this mapping satisfies (Y-1) for the states that have state identifiers that pass through modified transitions. In order to check this, for each of these states, the corresponding re-testing test sequences only include the identification sequences that pass through modified transitions. Moreover, the transition re-testing phase includes only sequences that test modified transitions.

Let $Q$ be a prefix-closed state cover set such that its sequences do not traverse modified transitions if applied at the initial state of $M_S'$. Let also $F = \{W_1, \ldots, W_n\}$ be a separating family of the modified specification, and $W$ be a characterization set (if exists). Moreover, we define and construct the sets $Q_j'$, $Q'$, $\overline{Q'}$, and $R_{t_j}$ as done in Case 4.1. We

note that in this case, $|Q'| = m$ since $Q'$ covers all the states of a modified implementation, and the sequences in the set $R_{t_j}$ do not traverse modified transitions.

**i) State re-identification phase**

For each state $s_r$ of $MS'$ such that some sequences of $W_r$ (or $W$ in the W or Wp methods) traverse modified transitions if applied at state $s_r$, we re-idenitfy each state $t_r \in h^{-1}_{I-S}(s_r)$ of the modified implementation. For each sequence $\gamma \in R_{t_r}$ that takes the modified specification to state $s_r$, the state re-identification re-testing sequences are formed as follows:

$$r.\gamma.W_r' \qquad \textbf{(4.2-a)}$$

where $W_r' \subseteq W_r$ (or $W_r' \subseteq W$ for the W and Wp methods) comprises each sequence of the state identifier $W_r$ (of the characterization set $W$) that, if applied at state $s_r$ of the modified specification, traverses a modified transition. We note that the inverse image(s) of each state of $MS'$ for which all sequences of the state identifier traverse only unmodified transitions, need not be re-identified.

**ii) Re-testing modified transitions phase**

In order to test a modified transition ($s_j$ - $x$ -> $s_k$), for each sequence $\gamma \in R_{t_j}$ that takes the modified specification to state $s_j$, the corresponding re-testing test sequences are formed as in Formulae (4-a) or (4-b).

**Theorem 7.** Given the modified specification $M_S'$ and its implementation $M_I'$, let $F = \{W_1, \ldots, W_n\}$ be a separating family of $M_S'$ and $W$ be a characterization set (if exists). Moreover, let $Q'$ be a prefix-closed state cover set of $M_S'$ such that each sequence of the set $Q'$ does not traverse a modified transition if applied at the initial state. If the implementation $M_I'$ passes the re-testing test suite which is the union of the re-testing test sequences given in Formulae (4.2-a) and Formulae (4-a) or (4-b), then the implementation is quasi-equivalent to $M_S'$.

We omit the proof of Theorem 7 since it is a particular case of Theorem 8.

### 4.5.3 Case-4.3: Some States Are only Reachable through Modified Transitions

In some cases, the unmodified part of the modified specification $M_S'$ is not reduced and some states of $M_S'$ are only reachable through modified transitions. Here, as in Case-3, we re-identify each inverse image of each state of $M_S'$ that is only reachable through modified transitions and we check all its outgoing transitions for correct output and ending state. Moreover, as in Case-4.2, we re-identify each inverse image of each state of $M_S'$ for which some sequences of the state identifier traverse modified transitions.

Let $F = \{W_1, \ldots, W_n\}$ be a separating family of the modified specification and $W$ be a characterization set (if exists), where sequences of the set $W_j$ (or $W$) may traverse modified transitions if applied at state $s_j$. Furthermore, we define and construct the sets $Q_j'$, $Q'$, $\overline{Q'}$, and $R_{t_j}$ as done in Case-4.1.

**i) State re-identification phase**

For each state $t_j$ that needs to be re-identified, we consider each sequence $\gamma \in R_{t_j}$ that takes the modified specification from the initial state to $s_j$, and we derive the state re-identification re-testing sequences:

If $\gamma$ does not traverse a modified transition, the re-identification sequences are formed as in Formula (4.2-a).

If $\gamma$ traverses a modified transition then there are test sequences:

$r.\gamma.W_j$ (in the HIS method) **(4.3-a )**

$r.\gamma.W$ (in the W and Wp methods) **(4.3-b)**

Every sequence of the set $W_j$ ($W$) must be applied after $\gamma$, whether the sequence applied at state $s_j$ traverses a modified transition or not.

**ii) Re-testing modified transitions phase**

For each modified transition ($s_j$ - $x$ -> $s_k$), we consider each sequence $\gamma \in R_{t_j}$ that takes the modified specification from the initial state to $s_j$.

If $\gamma$ does not traverse a modified transition, we derive the corresponding test sequences as in Formulae (4-a) or (4-b).

If $\gamma$ traverses a modified transition, we apply Formulae (4-a) or (4-b) to each outgoing transition from state $s_j$ including those which are unmodified.

**Theorem 8.** Given modified specification $M_S'$ and its implementation $M_I'$. let $F = \{W_1, \ldots, W_n\}$ be a separating family of $M_S'$ and $W$ be a characterization set (if exists). Moreover, let $Q'$, derived as described in Case-4.1, be prefix-closed state cover set of $M_S'$. If the implementation $M_I'$ passes the re-testing test suite which is the union of the re-testing test sequences given above, then the implementation $M_I'$ is equivalent to $M_S'$.

**Proof of Theorem 8:**

Consider a relation $h \subseteq T \times S$ such that :

$$(t_j,\ s_j) \in h \Leftrightarrow t_j \equiv_{W_j} s_j \quad (\text{or } (t_j, s_j) \in h \Leftrightarrow t_j \equiv_W s_j)$$

If the implementation passes the test cases given in Formula (4.2-a) and (4.3-a) or (4.3-b), then for every state $t_j$ there exists a state $s_j$ such that $t_j \equiv_{W_j} s_j$ ($t_j \equiv_W s_j$). Moreover, there does not exist another state $s_k$ different from $s_j$ such that $t_j \equiv_{W_j} s_k$ ($t_j \equiv_W s_k$), since $s_j$ is not $W_j$ -equivalent (is not $W$-equivalent) to $s_k$. Therefore, $h$ is a many-to-one mapping $T \rightarrow S$.

We note that if $t_j$ is reachable through unmodified transitions, then $h(t_j) = h_{I\text{-}S}(t_j)$. However, if $t_j$ is reachable through modified transitions, then the many-to-one mapping $h$: $T \rightarrow S$ such that $t_j$ of the modified implementation is $W_j$ -equivalent ($W$-equivalent) to state $s_j = h(t_j)$ of the modified specification, can be different from $h_{I\text{-}S}$.

Since the re-testing test suite has re-identification test sequences, similar to that in the proof of Lemma 1, if the implementation passes the test suite then we conclude as follows.

1) Sequences of $Q'$ take the implementation to $|Q'|$ different states, i.e. each state $t_j$ of the implementation reachable from the initial state can be reached through some sequence $\gamma \in \overline{Q'}$ such that $\gamma$takes the modified specification from the initial state to a state $W_j$-equivalent ($W$-equivalent) to state $t_j$, i.e. each state $s_j$ has an inverse image $t_j$ in the modified implementation.
2) Given state $s_j$ of the modified specification, for each sequence $\gamma \in \overline{Q'}$ that takes the specification to state $s_j$, it holds that $\Delta_I(t_1, \gamma) \equiv_{W_j} \delta_S(s_1, \gamma)$ [$\Delta_I(t_1, \gamma) \equiv_W \delta_S(s_1, \gamma)$].

For each state $t_j \in T$, the following three cases are possible:

If the modified implementation reaches $t_j'$ through a sequence $\gamma_j \in R_{t_j}$ that does not traverse a modified edge and $(s_j - a \rightarrow s_k)$ is an unmodified transition then $h(t_j') = h_{I\text{-}S}(t_j')$, and $h(\Delta_I(t_j', a)) = h_{I\text{-}S}(\delta_S(s_j, a)) = s_k$

If the modified implementation reaches $t_j'$ through a sequence $\gamma_j \in R_{t_j}$ that does not traverse a modified edge and $(s_j - x \rightarrow s_k)$ is a modified transition, and the implementation passes the test cases given in Formula (4-a) or (4-b) then :

$$\Delta_I(t_j', x) \equiv_{W_k} s_k \Rightarrow h(\Delta_I(t_j', x)) = s_k \text{ and } \delta_S(h(t_j'), x)) = s_k$$

If the modified implementation reaches state $t_j'$ through a sequence $\gamma_j \in R_{t_j}$ that traverses a modified edge and the implementation passes the test cases given in Formulae (4-a) or (4-b) over all outgoing transitions of $t_j'$, then for every transition $(s_j - x \rightarrow s_k)$ and $t_j'$ it holds that :

$$\Delta_I(t_j', z) \equiv_{W_k} s_k \Rightarrow h(\Delta_I(t_j', z)) = s_k \text{ and } \delta_S(h(t_j'), z)) = s_k \square$$

## 4.6 The Length of the Re-testing Sequences

For a given reduced FSM $M_S$ with $n$ states and $k$ input symbols, the test sequences generated by the W method can be constructed by first generating a set of defined input sequences say $\rho$ such that for every defined transition from state $s_i$ to state $s_j$ on input $x$, there are defined input sequences $\sigma$ and $\sigma_i x$ in $\rho$ so that $\sigma_i$ forces the machine into state $s_i$ from the initial state. Then, the sequences of $\rho$ are concatenated with the sequences of the $W$ set. One way to construct the set $\rho$ is to build a so-called "testing tree" of $M_S$. All partial paths in this tree constitute the set $\rho$. A "*partial path*" is a sequence of consecutive branches that starts at the root node and ends at either a terminal or non-terminal node of that tree. Each edge is labeled by an input symbol. The empty sequence is always an element of $\rho$. A procedure for constructing $\rho$ is given in [Cho78], where $\rho$ is obtained by constructing a testing tree then by enumerating the partial paths of the tree. Since each transition appears exactly once (or at most once for partially specified FSM) in the testing tree, the cardinality of the set $\rho$ is proportional to $n.k$, the upper bound on the number of (defined) transitions in the machine.

Every reduced completely specified FSM has a $W$ set consisting of no more than $n$-1 sequences, where each sequence has length of no more than $n$-1 [Yan95][Cho78]. However, for reduced partially specified FSMs, each sequence of the $W$ set (if exists) has length no more than $n^2$ [Yan95]. Therefore, the length of $W$ is of order $n^2$ for completely specified FSMs and it is of order $n^3$ for partially specified FSMs with a $W$ set. Consequently, the length of the test suite generated using the $W$ method has an upper bound of order $k.n^3$ for reduced completely specified FSMs and for reduced partially specified ones is of order $k.n^4$.

For the case when the implementation machine is assumed to have more states than the specification machine, i.e. $m > n$, the length of the test suite of the W method is multiplied by a factor $k^{m-n}$ since each sequence of the set $\rho$ that takes the specification to state $s_r$ is concatenated with the sequences ($\mathbf{DIS}(M_S|s_r) \cap X^{m-n}$). We note that, in the worst

case, the length of the test suite of the Wp is that of the W method. Moreover, the length of the test suite derived using the HIS method is of the same order as that of the W method since the order of the length of state identifiers is that of the $W$ set. This is due to the fact that the maximum number of sequences in a state identifier or a separating set is ($n$-1). Moreover, for fully specified FSMs, we can always find sequences so that the upper bound on the maximum length of separating sequence is no more than ($n$-1) [Yan95]. For partially specified FSMs, this bound is no more than $n^2$ [Yan95]. Therefore, for fully specified FSMs, the length of a separating set $W_i$ is roughly of order $n^2$ and for partially specified FSMs it is roughly of order $n^3$.

### 4.6.1 The Length of the Re-testing Sequences of Case-1:The Unmodified Part of the Modified Specification is Reduced

According to Case-1, no state re-identification is needed and re-testing sequences are generated to check only modified transitions of $M_S'$. For each modified transition, re-testing sequences are generated by concatenating one path of the $nk$+1 partial paths of a testing tree with the corresponding $W$ set ($W_k$ in the Wp and HIS methods). Therefore, for completely (partially) specified FSMs, the length of the re-testing test sequences is roughly of order $num.n^2$ ($num.n^3$), where $num$ is the number of modified transitions.

In order to calculate the upper bound of $num$, we note that according to Case-1, the unmodified part of the modified specification $M_S'$ is reduced, and for each state, say $s_i$, of $M_S'$ there exist a state identifier $W_i$ that is a subset of defined input sequences at $s_i$ in the unmodified part of $M_S'$. Consequently, according to Case-1, there are at least $n$ unmodified transitions in $M_S'$, and thus the upper bound of $num$ equals to $nk - n$. This happens when each state identifier $W_i$, for $i$=1..$n$, of $M_S'$ contains an input sequence of length equals to one. When $num$ is at its upper bound, the length of the re-testing sequences becomes roughly proportional to $n^3k$ for completely specified specifications and $n^4k$ for partially specified ones. However, we observe here that even when $num$ is at its upper bound, the maximum number of partial paths required for re-testing is $nk - n$

which is less than $nk + 1$ required for complete testing of $M_S'$. Moreover, as long as less than $n$ partials paths are needed for re-testing, the length of the re-testing sequences would stay roughly proportional to $n^2k$ for completely specified FSMs and $n^3k$ for partially specified ones

### 4.6.2 The Length of the Re-testing Sequences of Case-2: Each State of the Modified Specification is Reachable through Unmodified Transitions and the Unmodified Part is not Reduced

According to Case-2, each state say $s_k$ of the modified specification $M_S'$ that has sequences in the $W$ set ($W_k$ in the HIS method) traverse modified transitions, has to be re-identified in the new implementation. Moreover, re-tests are selected to check each modified transition for a correct output and a correct tail state. In the worst case, for *num* modifications, $n + num$ partial paths of the re-testing tree are needed since all of the $n$ states of the modified specification might have to be re-identified and *num* modified transitions have to be re-tested. Consequently, for completely specified FSMs the length of the re-testing test sequences is of $O(n + num)n^2$, and it is of order $(n + num)n^3$ for partially specified FSMs.

In order to calculate the upper bound of *num*, we note that according to Case-2 all (of the $n$-1) states of $M_S'$ are assumed to be reachable from the initial state through unmodified transitions. Therefore, there are at least $n$-1 unmodified transitions in $M_S'$, and the upper bound on *num* is the maximum number of defined transitions of the machine $nk$ minus $n$-1. This bound is of order $nk$. Consequently, for completely specified (partially specified) FSMs the length of the re-testing test sequences is of order $n^3k$ ($n^4k$). However, we note here that even when *num* is at its upper bound (i.e. $num = nk - (n\text{-}1)$), in the worst case, the number of partial paths required for re-testing will be $n + num = n + nk - (n\text{-}1) = nk + 1$, which is equal to that required for complete testing of $M_S'$. Moreover, as long as the number modifications plus the number of states to be re-identified is less than

$n$, i.e. when less than $n$ partials paths are needed for re-testing, the length of the re-testing sequences is of order $n^2k$ for completely specified FSMs and $n^3k$ for partially specified ones.

### 4.6.3 The Length of the Re-testing Sequences of Case-3: Some States Are only Reachable through Modified Transitions

According to Case-3, all the states that are only reachable through modified transitions have to be re-identified in the new implementation, and differently from the former cases, in the worst case, we have to check all unmodified transitions from these states. Moreover, as in Case-2, each state, say $s_k$, that has sequences in the $W$ set (or $W_k$ in the HIS method) traverse modified transitions, has to be re-identified. Therefore, for one modification (i.e. when $num = 1$), in the worst case, all the $n$ states of the modified specification might have to be re-identified and $nk - (k\text{-}num)$ outgoing transitions have to be retested. This happens when the initial state of $M_S'$ is the head state of the modified transition and all other states of $M_S'$ are reachable from the initial state only through the modified transition (that happens when all the outgoing unmodified transitions of the initial state (with an upper bound of $k\text{-}num$) loop back to the initial state itself). Consequently, $nk - (k\text{-}num)$ partial paths of the $nk+1$ partial paths of the testing tree are needed for re-testing, and thus the length of the re-testing sequences is roughly of order $n^3k$ for completely specified FSMs and $n^4k$ for partially specified ones. We note that as long as $num$ is still smaller than $k$, in the worst case, $nk - (k\text{-}num)$ transitions of $M_S'$ have to be re-tested. However, when $num$ is greater than or equals to $k$ and $num$ is smaller than or equals to its upper bound $nk$ (i.e. when all the transitions of $M_S'$ are modified), in the worst case, all the $nk$ transitions of $M_S'$ have to be re-tested. In this case, the re-testing test sequences generated by our re-testing methods coincide with the test sequences generated by the corresponding testing methods. However, as mentioned before, we note here that as long as the number of partial paths required for re-testing is still less than $n$, the length of the re-testing test sequences is roughly proportional to $n^2k$ for completely specified FSMs and $n^3k$ for partially specified ones.

### 4.6.4 The Length of the Re-testing Sequences when the Implementation Has more States than the Specification

The length of the re-testing sequences when the implementation has more states than the specification is equal to the one presented above for the cases when the implementation has the same number of states as the specification, but multiplied by a factor $k^{m-|Q'|}$. This is due to the fact that each sequence of the set $\rho$ that takes the specification to state $s_r$ is concatenated with the sequences ($\mathbf{DIS}(M_S|s_r) \cap X^{m-|Q'|}$).

For cases 4.1 and 4.3, in the worst case, $|Q'| = n$, the number of states of the specification machine. This happens when all the states of the modified implementation are reachable through modified transitions and the initial state, which is always reachable through unmodified transitions, has only one inverse image in the modified implementation. In this case, the factor $k^{m-|Q'|}$ equals to $k^{m-n}$ which is the lower bound factor reported in [Cho78]. However, for cases 4.1 and 4.3, in the best cases, $|Q'| = m$ and the lower bound on the factor $k^{m-|Q'|}$ is $k^0 = 1$. For Case-4.2, this happens when each state of the modified specification with more than one inverse image in the modified implementation, is reachable from the initial state through unmodified transitions. For Case-4.3 this happens when each state of the modified specification that is only reachable from the initial state through modified transitions, has only one inverse image in the modified implementation. Moreover, as long as there exist a state in the specification with more than one inverse image in the modified implementation and at least one of these images is reachable though unmodified transitions, the factor $|Q'|$ will be greater than $n$. Consequently, the factor $k^{m-|Q'|}$ will be less than $k^{m-n}$, the lower bound factor reported in [Cho78].

Finally, for Case-4.2, even in the worst case, all the states of the modified implementation are reachable from the initial state through unmodified transitions, and thus $|Q'| = m$. Consequently, the factor $k^{m-|Q'|}$ equals to one which is shorter than the lower bound factor $k^{m-n}$ reported in [Cho78].

## 4.7 Adapting the Wp Based Re-testing Methods to the UIOv Methods

The UIOv is a special case of the Wp method. This method makes the same assumptions about the specification machine $M_S$ and the implementation machine $M_I$ as the $W$ method. In addition, it further assumes that the set $W_j$ of the Wp method can be chosen in such a way that they contain each only a single sequence, a so-called Unique-Input-Output (*UIO*) sequence. A *UIO* sequence for a state of a complete $M_S$ is an I/O behavior that is not exhibited by any other state of $M_S$ (see Section 3.2 for formal definition).

Let $\tau_1/\beta_1$, $\tau_2/\beta_2$, ..., $\tau_n/\beta_n$ be the respective *UIO* sequences for the $n$ states $s_1$, $s_2$ ... $s_n$, then the set of input sequences $\{\tau_1, \tau_2, ..., \tau_n\}$ constitutes a $W$ set of $M_S$, and in the second testing phase $\{\tau_j\}$ is sufficient for the verification of the ending state of a transition $s_i$ -$x$/$y$-> $s_j$. Therefore, the test suites generated by the UIOv testing (or re-testing methods) can be given exactly in the same manner as the Wp testing (or re-testing methods) by having the set $W = \{W_1, W_2, ..., W_n\}$ where for any state identifier $W_k \in W$, $W_k = \tau_k$.

We note that UIO sequences may not exist for every reduced FSM. In this case the UIOv method cannot be applied. However, instead we can always apply the HIS method.

## 4.8 DS Based Re-testing Methods

In this section we first define the distinguishing sequence (DS) based re-testing problem, we present methods to solve the problem based on the re-testing methods presented in the previous sections. We recall that the DS methods generate a single test sequence (derived from a given specification FSM) that is supposed to be applied to the initial state of a given implementation to check its correctness. The applied test sequence and its expected output response sequence form a so-called *checking sequence.*

### 4.8.1 Problem Definition

The DS-based re-testing problem can be defined as follows: Let $M_S' = (S, X, Y, \delta_S, \lambda_S, s_1)$, $|S| = n$, be the modified version of the specification FSM $M_S = (S, X, Y, \delta, \lambda, s_1)$, $|S| = n$, where $M_S$ is completely specified, initialized, reduced, deterministic, strongly connected and has a distinguishing sequence $D$. We also let $D'$ be a distinguishing sequence of $M_S'$ and $M_I' = (T, X, Y, \Delta_I, \Lambda_I, t_1)$, $|T| = n$, be its implementation machine. We want to generate a test sequence (hereafter called *re-checking sequence*) in order to check that the modifications to $M_S$ were implemented correctly in the implementation $M_I'$. We assume that in the implementation $M_I'$, only transitions corresponding to the modified parts of the specification are changed. We also assume that before modifying $M_S$, its implementation $M_I = (T, X, Y, \Delta, \Lambda, t_1)$, $|T| = n$, was tested using a checking sequence and was found not faulty. Moreover, we assume that the modified specification machine $M_S'$ has the same assumptions as $M_S$. That is $M_S'$ is initialized, completely specified, reduced, strongly connected, deterministic, and has a distinguishing sequence.

In general, we have the following types of modifications that can be made in $M_S$ and implemented by a designer in $M_I$:

(1) outputs of some transitions are modified,
(2) tail states of some transitions are modified,
(3) outputs and tail states of some transitions are modified,
(4) new states are added, and
(5) some states are deleted.

In order to solve for the problem, as in the W based re-testing methods, we have to re-identify some states in the new implementation $M_I'$ and test some transitions for a correct output and a correct ending state. The selection of these states and transitions is done as

described for Cases 1 to 3 for the Wp re-testing methods. Accordingly, for each of these cases, we present below a corresponding method for generating a re-checking sequence. We note that now the $W$ set and the state identifiers of $M_S'$ each consists of the singleton $D'$.

For convenience we let the set $E_t = \{e_i \mid e_i$ is some transition $(s_j\text{-}x/y\text{->}s_k)$ of $M_S'$ that need to be tested, and $i=1,\ldots,l\}$ denote the set of $l$ transitions of $M_S'$ that need to be tested, where $l$ is less than or equals to the number of transitions of $M_S'$. Moreover, we let the set $S_{re\_id} = \{r_1, r_2, \ldots, r_r\} \subseteq S$, denote the set of $r$ states of $M_S'$ that need to be re-identified. Moreover, for a transition $e = (s_j\text{-}x/y\text{->}s_k)$, we let *label*($e$) denote the input/output pair $x/y$ of $e$. The *cost* (or *length*) of a transition is the number of *i/o* pairs in the label of that transition. The *cost* (or *length*) of a checking (or rechecking) sequence is the sum of the lengths of transitions included in the sequence.

### 4.8.2 A DS Method for Case-1: The Unmodified Part of the Modified Specification is Reduced

According to Case-1, when the unmodified part of $M_S'$ is still reduced and has a distinguishing sequence $D'$, we only check modified transitions for a correct transfer and a correct ending state.

For the set of $t$ transitions $E_t = \{e_1, e_2, e_t\}$ of $M_S'$ that need to be tested, we construct their re-checking sequence as follows:

$T(s_1,head(e_1))$. label($e_1$). $D'/\lambda_S(\text{tail}(e_1),D')$.$T(a_1,head(e_2)$.label($e_2$).$D'/\lambda_S(\text{tail}(e_2),D')$.
$T(a_2,head(e_3))$.label($e_3$).$D'/\lambda_S(tail(e_3),D')\ldots.T(a_{t-1},head(e_t))$.label($e_t$).$D'/\lambda_S(tail(e_t), D')$

where:

$T(s_1, head(e_1))$ = is an (*I/O*) transfer sequence from the initial state $s_1$ to head($e_1$) in $E_t$,

$a_i$ is the state of $M_S'$ reached when $D'$ is applied to state $tail(e_i)$; and

$T(a_i, head(e_{i+1}))$ = is an (*I/O*) transfer sequence from state $a_i$ to $head(e_{i+1})$ in $E_t$.

Note that the above transfer sequences may be empty sequences. Moreover, we always construct the re-checking sequence in such a way that the transfer sequences do not pass through modified transitions or they pass through already tested transitions. Furthermore, since $D'$ does not pass through modified transitions, the ending state after applying $D'$ at a known state of the implementation machine $M_I'$ is also known.

We note that we can always choose state $head(e_1)$ in $E_t$ in such a way that it is reachable from the initial state through unmodified transitions. Consequently, the ending state after the application of the first transfer subsequence $T(s_1, head(e_1))$ of the above re-checking sequence is known, since we can always construct this subsequence in such a way that it does not traverse any modified transition. Moreover, afterwards, subsequence $label(e_1).D'/\lambda_S(tail(e_1), D')$ checks the modified transition $e_1$ for a correct output and a correct ending state. The ending state after the above subsequence is also known, since $D'$ does not traverse modified transitions. Therefore, if the modified edge $e_1$ is implemented wrongly such that after the input $label(e_1)$ it transfers to a wrong state other than the specified one, the above application of $D'$ will detect this anomaly. Afterwards, we can always choose another transition in $E_t$ in such a way that the head state of such a transition is reachable from the current (known) state through either unmodified transitions or through already tested ones. Consequently, the transfer sequence from the current state to such a state is guaranteed to reach the intended state.

### 4.8.3 A DS method for Case-2: Each State of the Modified Specification is Reachable through Unmodified Transitions and the Unmodified Part is not Reduced.

Case-2 occurs when the unmodified part of the modified specification is not reduced, but all the states of the modified specification are reachable from the initial state through unmodified transitions. In this case, as shown previously, we re-identify each state for which $D'$ traverses a modified transition if applied at that state, and we test modified transitions.

First, for the $r$ states of the set $S_{re\_id} = \{r_1, r_2, \ldots, r_r\} \subseteq S$ of $M_S'$ that need to be re-identified, we construct their corresponding part (called *Part*-1) of the re-checking sequence as follows:

Part-1 = $T(s_1, r_1).D'/\lambda_S(r_1,D').T(a_1, r_2).D'/\lambda_S(r_2,D').T(a_2, r_3).D'/\lambda_S(r_3,D')....$

$.....T(a_{r-1}, r_r).D'/\lambda_S(r_r, D').T(a_r, r_1).D'/\lambda_S(r_1, D').T(a_1,r_2)$

where:

$T(s_1, r_1)$ = is an *I/O* transfer sequence from the initial state $s_1$ to $r_1$ in $S_{re\_id}$;

$a_i$ is the state of $M_S'$ reached when $D'$ is applied to state $r_i$ ; and

$T(a_i, r_{i+1})$ = is an *I/O* transfer sequence from state $a_i$ to state $r_{i+1}$ in $S_{re\_id}$.

The above state re-identification part of the re-checking sequence is constructed exactly in the same manner as done in [Hen64], but for the subset of states of $M_S'$ that need to be re-identified (i.e. for the states in $S_{re\_id}$).

Afterwards, in order to test modified transitions, we proceed as follows:

i) In order to test a transition $(s_i\text{-}x/y\text{->}s_j)$ whose head state $s_i$ is not in $S_{re\_id}$, we first transfer from the known current state $a_k$ of the machine to the initial state $s_1$ using a transfer sequence $T(a_k, s_1)$. Then we use $D'/\lambda_S(s_1, D')$ in order to check that we indeed reach $s_1$ after such a transfer sequence. Afterwards, we return back to the initial state from state $a_1$ reached by $D'/\lambda_S(s_1, D')$, using a transfer sequence $T(a_1, s_1)$. Finally, we transfer from $s_1$ to the intended state $s_i$ through unmodified transitions before the input $x$ of the transition under test is applied. All of the above is done in order to guarantee that the required state of the implementation corresponding to state $s_i$ (such a correspondence is established by the checking sequence that tested the conformance of $M_I$ to $M_S$) is entered before the input $x$ of the transition under test is applied. Formally, suppose that

$MS'$ is currently in state $a_k$. In order to test transition $s_i$-$x$/$y$->$s_j$ where $s_i \in S_{re\_id}$, the following test subsequence can be used:

$$T(a_k, s_1).D'/\lambda_S(s_1, D').T(a_1, s_1).T(s_1, s_i).x/y.D'/\lambda_S(s_j, D') \qquad \text{(I-b)}$$

However, before testing any of the above transitions using Formula (I-b), we first have to check that from the initial state $s_1$, the subsequence $D'/\lambda_S(s_1, D').T(a_1, s_1)$ returns the machine back to $s_1$ again. Consequently, in order to check the correctness of this subsequence, the state re-identification part of the re-checking sequence (i.e. Part-1) is concatenated with the following subsequence:

$$T(r_2, s_1).D'/\lambda_S(s_1, D').T(a_1, s_1)\ D'/\lambda_S(s_1, D').T(a_1, s_1) \qquad \text{(I-a)}$$

where,

$r_2$ is the state reached after applying Part-1, and

$a_1$ is the state of $MS'$ reached when $D'$ is applied to state $r_1$.

ii) In order to test a transition ($r_i$-$x$/$y$->$s_j$) whose head state $r_i$ is already re-identified in Part-1 (i.e. $r_i \in S_{re\_id}$ ), we use the same technique used in [Hen64]. Suppose that $MS'$ is currently in state $a_k$. To test the transition $r_i$-$x$/$y$->$s_j$ the following subsequence can be used:

$$T(a_k, r_{i-1}).D'/\lambda_S(r_{i-1}, D').T(a_{i-1}, r_i).x/y.\ D'/\lambda_S\ (s_j, D') \qquad \text{(II)}$$

By including $T(a_k, r_{i-1}).D'/\lambda_S(r_{i-1}, D').T(a_{i-1}, r_i)$ in the above sequence, one can guarantee that the required state of the implementation corresponding to state $r_i$ (such a correspondence is established by Part-1 of the re-checking sequence) is entered before the input $x$ of the transition under test is applied.

#### 4.8.3.1 An application example

As an application example for the DS re-testing method when Case-2 applies, consider the specification FSM $M_S$ given in Fig. 4.8 with inputs $X=\{a, b\}$ and outputs $Y=\{0, 1\}$. The cost of each transition of the FSM $M_S$ is 1. $M_S$ has a distinguishing sequence $D=baa$. In fact we have the following output sequences in response to $baa$ :

For state $s_1$: 110,

For state $s_2$: 101,

For state $s_3$ : 001,

For state $s_4$: 011, and

For state $s_5$: 010.

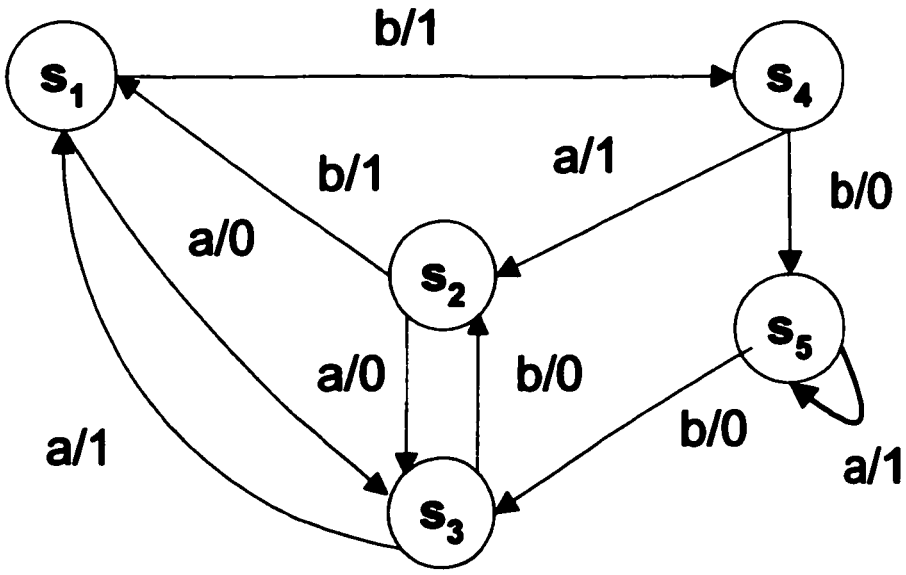


*Figure* 4.8. *Specification FSM* $M_S$

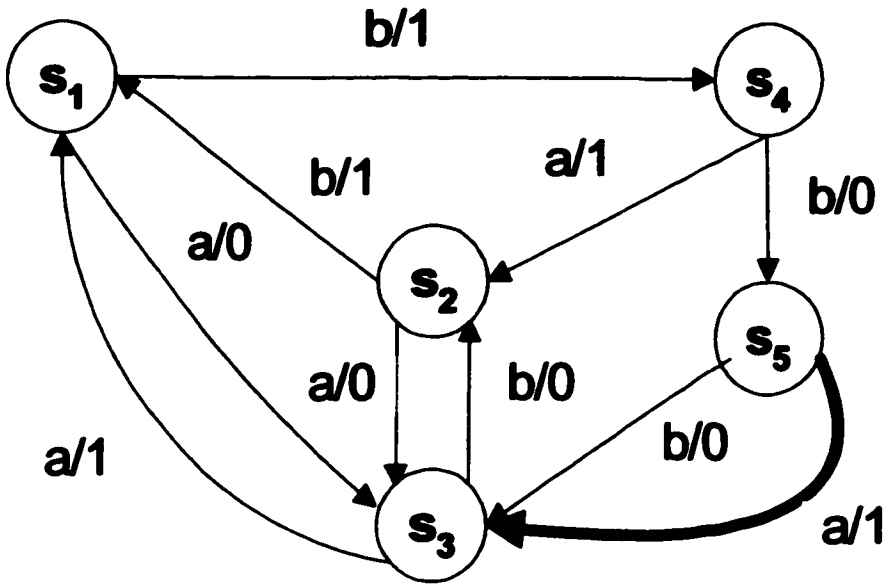


*Figure* 4.9. *Specification FSM* $M_S'$

We let the FSM $M_S'$ shown in Fig. 4.9 be the FSM $M_S$ with the modification of redirecting the ending state $s5$ of transition ($s5$-*a*/1->$s5$) to $s3$. $M_S'$ has still $D'$=*baa* as its distinguishing sequence. Moreover, the cost of each transition of the FSM $M_S'$ is 1.

State $s4$ of $M_S'$ is the only state for which $D'$ traverses the modified transition if applied to it. Consequently, the set $S_{re\_id}$ = {$s4$}. In order to re-identify $s4$ in the new implementation, from the definition of the first part of the re-checking sequence, we obtain:

Part-1 = $T(s1, s4).D'/\lambda_S(s4,D').T(s1, s4).D'/\lambda_S(s4,D').T(s1, s4)$

Part-1 = *b*/1. *baa*/011. *b*/1. *baa*/011.*b*/1

Afterwards, in order to test the modified transition ($s5$-*a*/1->$s3$) whose head state $s5$ is not in $S_{re\_id}$, we proceed as follows:

i) From the definition of (I-a), we concatenate Part-1, the state re-identification part of the checking sequence, with the following subsequence:

$T(s4, s1).D'/\lambda_S(s1, D').T(s3, s1)\ D'/\lambda_S(s1, D').T(s3, s1)$ =

*a*/1.*b*/1. *baa*/110. *a*/1. *baa*/110. *a*/1

ii) From the definition of (I-b), we concatenate the above subsequence with the following subsequence:

$T(s1, s1)D'/\lambda_S(s1, D').T(s3, s1).T(s1, s5).a/1.D'/\lambda_S(s3, D') =$

$\varepsilon/\varepsilon.\ baa/110.\ a/1.\ b/1b/1.\ a/1.\ baa/001$

The total length of the above constructed re-checking sequence is 29. The minimum length checking sequence obtained using the model proposed in [Ura97] has a length not less than 55, if the whole specification $M_S'$ is considered for checking sequence derivation.

### 4.8.4 A DS Method for Case-3: Some States Are only Reachable through Modified Transitions

According to Case-3, all the states that are reachable from the initial state only through modified transitions have to be re-identified in the new implementation. Moreover, we re-identify each state for which $D'$ traverses modified transitions if applied at that state. Furthermore, we test modified transitions and differently from the former cases, in the worst case, we have to test all unmodified transitions from the states that are reachable from the initial state only through modified transitions.

The re-checking sequence for this case can be constructed exactly in the same manner as done for Case-2. We note that according to Case-3, in order to test a transition whose head state is not in $S_{re\_id}$, we can always find a transfer sequence from the initial state to the head state of that transition that does not traverse any modified transition. This is due to the fact that if such a transfer sequence cannot be found, then such a state should be in the set $S_{re\_id}$.

#### 4.8.4.1 An application example

As an application example of the DS method for Case-3, we add to the specification machine $M_S$ in Fig. 4.8 a new state $s_6$ producing machine $M_S'$ shown in Fig. 4.10. The cost of each transition of the FSM $M_S'$ is 1.

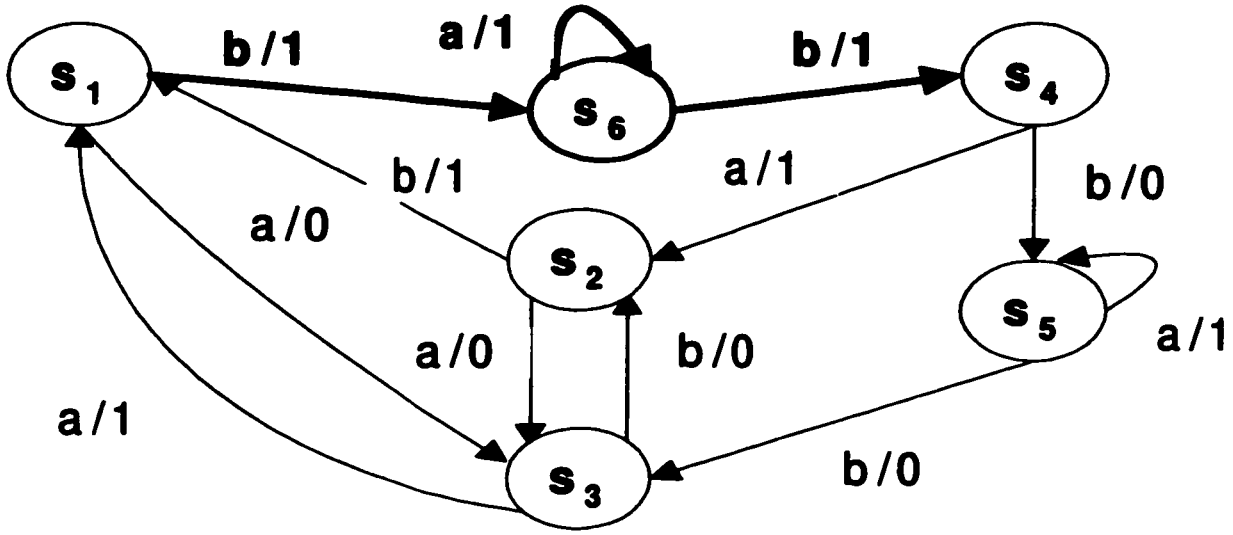


*Figure* 4.10. *Specification* $M_S'$

The specification $M_S'$ has still $D'$ =*baa* as its distinguishing sequence. In fact we have the following output sequences in response to *baa* :

For state $s_1$ : 111,

For state $s_2$: 101,

For state $s_3$ : 001,

For state $s_4$ : 011 ,

For state $s_5$: 010, and

For state $s_6$: 110.

According to Case-3, state $s_1$ needs to be re-identified since when $D'$ is applied at this state it traverses modified transitions. Moreover, we re-identify the added state $s_6$. Therefore, $S_{re\_id}$={$s_1$, $s_6$}, and from the definition of the first part of the re-checking sequence, we obtain:

Part-1 = $T(s_1, s_1).D'/\lambda_S(s_1, D').T(s_6, s_6).D'/\lambda_S(s_6, D').T(s_3, s_1).\ D'/\lambda_S(s_1, D').T(s_6, s_6)$

$= \varepsilon/\varepsilon.\ baa/111.\ \varepsilon/\varepsilon.\ baa/110.\ a/1.\ baa/111.\ \varepsilon/\varepsilon$

We consider each incoming/outgoing edge of $s_6$ as a modified transition. Therefore, the modified transitions of $M_S'$ are ($s_1$-$b$/1->$s_6$), ($s_6$-$a$/1->$s_6$) ($s_1$-$b$/1->$s_4$). Consequently, according to Case-3, we test these transitions and we also test each outgoing transition from state $s_6$ since $s_6$ is reachable from the initial state only through a modified transition. Therefore, the transitions that need to be tested are ($s_1$-$b$/1->$s_6$), ($s_6$-$a$/1->$s_6$) and ($s_1$-$b$/1->$s_4$).

Before testing these transitions, we notice that the head states of these transitions (i.e. $s_1$ and $s_2$) are already re-identified in Part-1. Therefore, in order to test these transitions, and from the definitions of (II), we obtain the following subsequences:

i) In order to test transition ($s_1$-$b$/1->$s_6$), we concatenate Part-1 with the following subsequence:

$T(s_6, s_6).D'/\lambda_S(s_6, D').\ T(s_3, s_1).b/1.D'/\lambda_S(s_6, D') =$

$\varepsilon/\varepsilon.\ baa/110.\ a/1.\ b/1.\ baa/110$

ii) Afterwards, in order to test transition ($s_6$-$a$/1->$s_6$), we concatenate the above subsequence with the following subsequence:

$T(s_3, s_1).D'/\lambda_S(s_1, D').T(s_6, s_6).a/1.D'/\lambda_S(s_6, D') =$

$a/1.\ baa/111.\ \varepsilon/\varepsilon.\ a/1.\ baa/110$

iii) Finally, in order to test transition ($s_6$-$b$/1->$s_4$), we concatenate the above subsequence with the following subsequence:

$T(s_3, s_1).D'/\lambda_S(s_1, D').T(s_6, s_6).b/1.D'/\lambda_S(s_4, D') =$

$a/1.\ baa/111.\ \varepsilon/\varepsilon.\ b/1.\ baa/011$

The length of the above re-checking sequence is 26. The model proposed in [Ura97] yields a checking sequence of length not less than 66, if the whole specification $M_S'$ is considered for checking sequence derivation.

# 5 Diagnostic Testing of Communicating Finite State Machines

## 5.1 Introduction

We recall that the purpose of conformance testing is to check whether an implementation conforms to its specification. An interesting complementary yet more complex problem is to locate the differences between a specification and its implementation when the implementation is found to be non-conforming [Lee93]. A solution to this problem has various applications. For example, it makes easy the job of correcting the implementation so that it conforms to its specification [Lee93].

In the software domain where a system may be represented as an FSM, some work has already been done for the *diagnostic* and *fault localization* problems [Ghe92a][Ghe92b] [Lee93]. However, little work has been done for distributed systems represented by communicating finite state machines (ComFSMs) [Ghe93a]. In [Ghe92a], [Lee93] and [Ghe93a] the differences between the systems specification and its implementation is located under the assumption of a single fault in the implementation. In [Ghed93b] the differences can be located for multiple faults under the assumption that each of the faults is reachable through non-faulty transitions.

In this second part of the thesis, we consider a system consisting of two communicating FSMs, called components, as shown as the "System Under Test (SUT) " in Figure 5.1. The system contains a machine, called *context machine* that communicates with the environment, and the other machine, called *embedded machine*. The interaction between these two components is assumed to be hidden or unobservable.

In [Ghe93a] a heuristic is proposed to locate a single fault within a system represented by two communicating FSMs, once a fault has been detected in its implementation. In this part of the thesis, we first show that it is not always possible to locate such a fault. This happens when a certain fault in an implementation of the context and another fault in the implementation of the embedded component may cause the same observable behavior of

the SUT. Accordingly, we present two new two-level approaches for the fault localization of the given system [Elf99b][Elf01]. At the first level (called *machine level diagnosis*) the methods decide if it is possible to identify the faulty component in the given system. If this is possible, the faulty component is identified, and if desired, at the second level (called *transition level diagnosis*) the methods determine if it is possible to locate the fault within the faulty component. If that is possible, the methods can provide additional test cases to locate the fault(s). We note that sometimes it is not possible to locate the fault(s) within the faulty component, since the same observable behavior of the SUT may be caused by different faults in the component implementation. The first method, called *single fault diagnostic method*, assumes as in [Ghe92a] [Lee93] [Ghe93a] that the SUT has a single fault in one of its transitions (i.e. in one of its component machines). However, the second method, called *multiple faults diagnostic method*, assumes that multiple output or transfer faults may occur in at most one component. Given a system decomposed into components, it is usually assumed for high-level abstractions that the structure of the system is preserved in all possible implementations, while a component may be either faulty or operating correctly [Kle87]. We note that the multiple faults method can be extended for identifying the faulty component FSM within a system of communicating FSMs when both component FSMs may be faulty.

The second part of the thesis is organized as follows. This chapter describes a behavior of a system of two communicating FSMs and introduces the diagnostic problem for such a system. Moreover, it includes an overview of previous research work done for solving this problem, namely the work presented [Ghe93a]. Furthermore, in the last section of the chapter, we show the problems of the method presented in that work. Chapter 6 includes the single fault diagnostic method along with two application examples that demonstrate its different steps. Moreover, it includes some experiments and the complexity analysis of the method. Chapter 7 includes the multiple faults diagnostic method along with some application examples that demonstrate its different steps. Moreover, it includes a section on the complexity analysis of the method.

## 5.2 A System of Two Communicating FSMs

Many complex systems are typically specified as a collection of communicating components [Pet96b]. Assuming that the behavior of each component of a system under test is known and can be described by an FSM, a system of communicating machines serves as a model of the given system.

We consider here a special case, where the system consists of two Communicating FSMs (*ComFSMs*), called *embedded machine* ($M_2$) and *context machine* ($M_1$), as shown in the upper part of Figure 5.1. The alphabet $X$ and $Y$ represent the externally observable input/output actions of the system, while the $U$ and $Z$ alphabets represent the internal (hidden) input/output interactions between the two components. As in [Pet96b], we assume that the sets $X$, $Y$, $U$, and $Z$ are pair-wise disjoint. The two (complete and deterministic) FSMs communicate asynchronously via bounded input queues where input/output messages are stored. An FSM produces an output in response to each input. We assume that the system at hand has at most one message in transit, i.e. the next external input is submitted to the system only after it produced an external output y to the previous input. Under these assumptions, the collective behavior of the two communicating FSMs can be described by product machine. A *product machine* describes the joint behavior of the component machines in terms of all actions within the system. The product machine $M_1 \times M_2$ is customary represented by a global graph, obtained by performing reachability computation [Wes78] [Boc80] [Bra83]. A global state consists of states of input queues and states of individual machines and can be represented in the form of a 2×2 matrix $\begin{bmatrix} a_1 & . & a_2 \\ s_1 & . & s_2 \end{bmatrix}$, where the first row represents the content of each input queue and the second row represents the current states of the two machines. According to the I/O ordering constraint, global states fall into the two categories, *stable* and *transient* states. A stable state has empty input queues, and thus it is ready to accept an external input action. Accepting such an action, the system changes its current state from a stable to a transient state where it cannot accept any external action. The system returns to a stable state after it has produced an external output action. Transitions between global states are labeled with an action causing a corresponding change of a global state. The product machine of a system of communicating machines can be deemed as a labeled transition system (LTS).

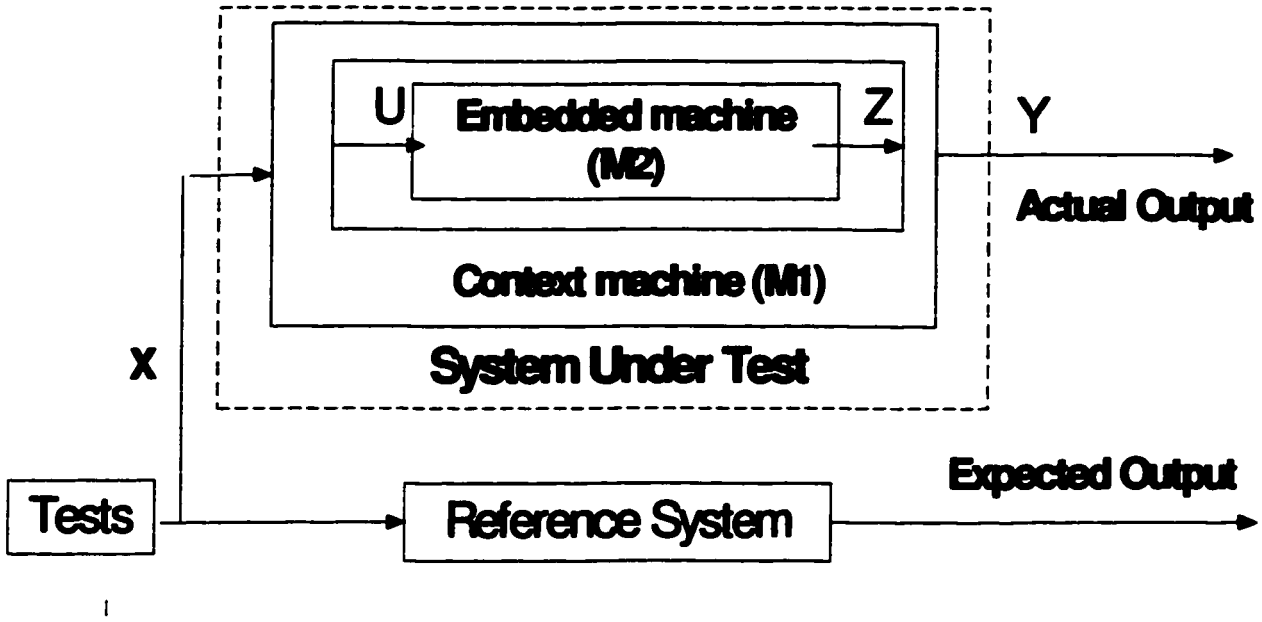


*Figure* 5.1. *A system of two ComFSMs and their tester*

If the product machine has a cycle labeled only with internal actions from the alphabet $U \cup Z$ then the system falls into live-lock when an input sequence leading to this cycle is applied; i.e. the system will not produce any external output. In this case, a behavior of the system cannot be described by an FSM, i.e. the composed machine does not exist. Otherwise, a *composed machine*, written as $RS = M_1 \Diamond M_2$ and called the "***Reference System***" in Fig 5.1, is obtained from the product machine by hiding all internal actions in the product machine, and pairing input with output actions [Pet96b].

A complete deterministic FSM $M_i$ ($i$ = 1, 2) in such a system of two *ComFSMs* is a 6-tuple $M_i = (S_i, I_i, O_i, D_i, \delta_i, \lambda_i, s_{i1})$ [Ghe93a], where :

$S_i$ is a finite set of states, including a special state $s_{i1}$ called the *initial state*,

$I_i$ is a finite set of input symbols, It includes the reset input ($r$),

$O_i$ is a finite set of output symbols. It includes the null output (-),

$D_i \subseteq S_i \times I_i$ is the specification domain,

$\delta_i$ is a state transition function : $D_i$ --> $S_i$, and

$\lambda_i$ is an output function : $D_i$ --> $O_i$.

We assume that both the specification and the implementation FSMs have a fault-free reset input "*r*". This input takes each FSM from its current state to its initial state with *NULL* output. This assumption allows the use a test suite consisting of several test cases instead of a unique checking (or diagnostic) sequence.

Consider for example the two machines $M_1$ and $M_2$ given in Fig. 5.2. The set of external inputs is $X=\{x_1, x_2\}$, the set of external outputs is $Y=\{y_1, y_2, y_3\}$, the set of internal inputs is $U=\{u_1, u_2\}$, and the set of internal outputs is $Z=\{z_1, z_2, z_3\}$. The corresponding product machine $M_1 \times M_2$ is shown in Fig. 5.3 and the $RS = M_1 \Diamond M_2$ extracted from it is shown in Figure 5.4, where the stable states are represented by big labeled circles, and the transient states are represented by small unlabeled circles.

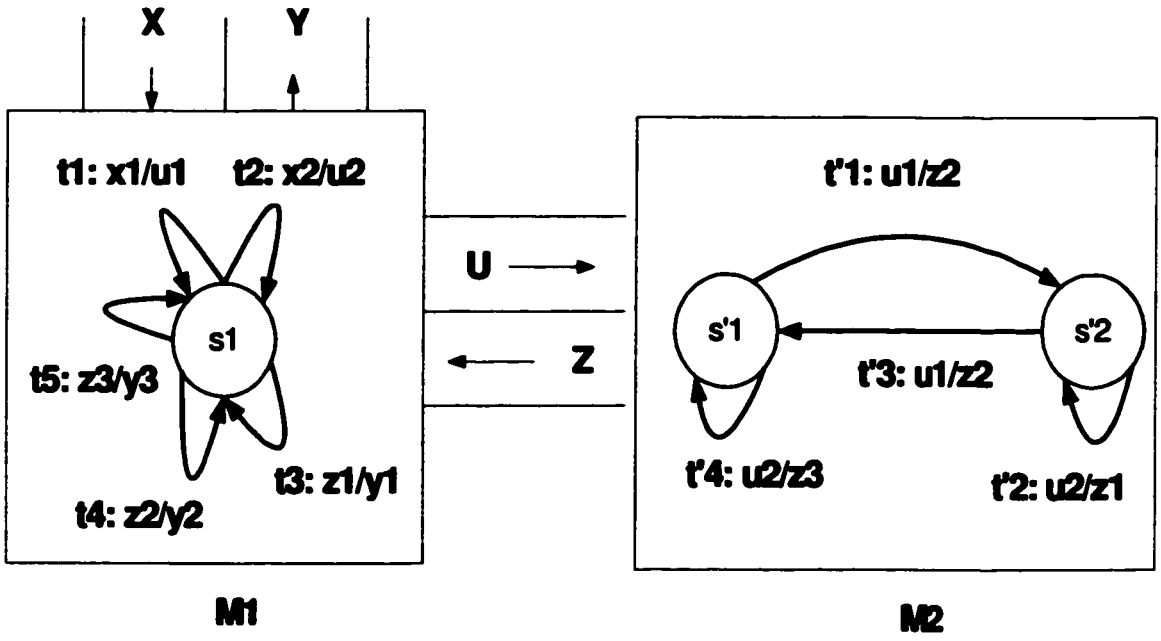


*Figure 5.2. A system of two ComFSMs $M_1$ and $M_2$*

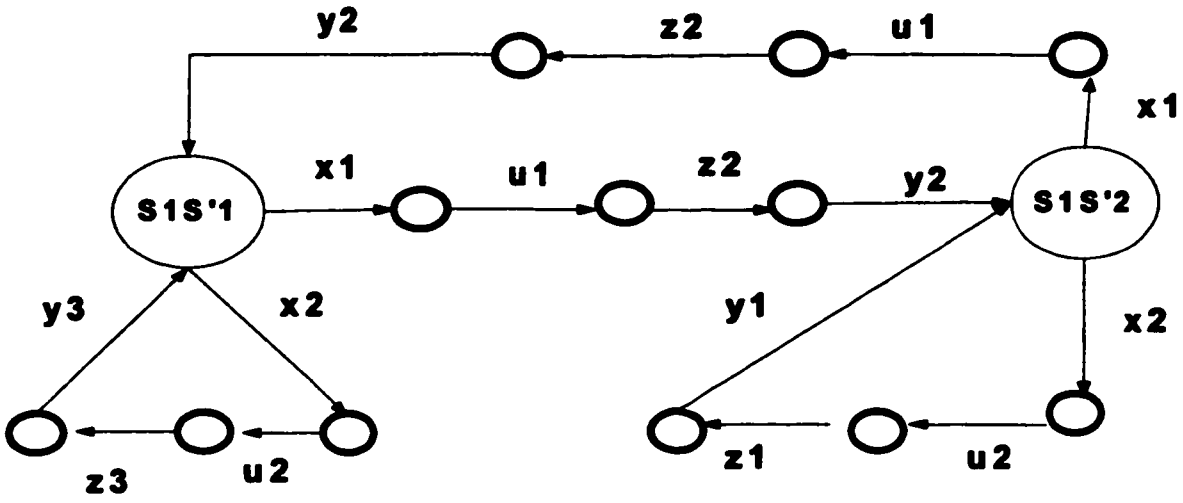


*Figure* 5.3. *Product System* ($M_1 \times M_2$) *of the* $M_1$ *and* $M_2$ *of Fig.* 5.2

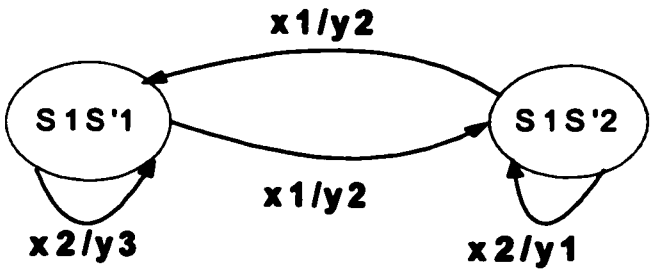


*Figure* 5.4. *Reference System* ($M_1 \Diamond M_2$) *of the* $M_1$ *and* $M_2$ *of Fig.* 5.2

The tester, lower part of Figure 5.1, implements a given test by executing external input sequences (test cases) simultaneously against both the SUT consisting of the implementation of $M_1$ and $M_2$, and the reference system in order to generate the observed and expected outputs. If for each test case, the sequences of the observed and expected outputs coincide then the system is said to *pass* the test suite.

## 5.3 The Diagnostic Problem

Diagnostics is a well documented subject in other areas such as Artificial Intelligence (AI), complex mechanical systems and medicine [Sho76]. In general, diagnostics can be classified into two classes. The first class, called *experimental diagnostic*, is mainly used in Medicine [Sho76] and similar domains. Therefore, experimental diagnostic is not covered in this thesis. Our main interest is related to the second class of diagnostic, called diagnostic based on

model [Dav88] [Kle87]. The main idea of this type of diagnostic is that it is necessary to know how the system or the machine under test is supposed to work correctly in order to be able to know why it is not working correctly. Therefore, in model-based diagnostics, [Kle87] [Rei87], we assume the availability of the real system (i.e, implementation) which can be observed, and its model (i.e, specification) from which predictions can be made about its behavior.

Given a specification of a system and its implementation, observed input and output sequences demonstrate a behavior of the system at hand while expected input and output sequences, derived from the specification, tell us how the system at hand is supposed to behave. Any difference between expectations and observations is a *symptom* that the implementation at hand is faulty. In order to explain the observed symptoms, a diagnostic process should be initiated. It consists mainly of performing the following two tasks: the generation of candidates and the discrimination between candidates [Kle87]. However, in order to derive the set of all possible candidates that can explain the observed symptoms proper assumptions about faults in the implementation must be made, i.e. we need a fault model [see the following subsection].

**Task 1: Generation of candidates**: This process uses the identified symptoms and the specification to determine diagnostic candidates. Each *diagnostic candidate* is a system with the given structure capable to explain all symptoms where each component is an element of an appropriate fault domain. It indicates the failure of one or several components in the system.

**Task 2: Discrimination between candidates:** Usually the number of diagnostic candidates generated by the previous step is huge. To reduce this number one may derive additional tests cases called *distinguishing tests* [Gen84] or introduce additional observation points in the implementation under experiment and execute the initial tests again.

### 5.3.1 A Fault Model for a System of Two Communicating FSMs

Different fault models are being used in diagnostics. Given a system decomposed into components, it is usually assumed for high-level abstractions that the structure of the system is preserved in all possible implementations, while a component may be either faulty or operating correctly [Kle87]. For lower level abstractions, such as gates and transitions, the "stuck at 0/1" fault model [Kle89] [Str89] is considered in different papers devoted to the diagnostics of digital circuits. In the software area, and more precisely for FSMs, we consider a fault model based on output and transfer faults of a deterministic FSM [Ghe92a]. We recall from Chapter 2 that, given a complete deterministic specification and an implementation FSMs $\langle S, I, O, \delta, \lambda, s_1 \rangle$ and $\langle S, I, O, \Delta, \Lambda, s_1 \rangle$, we say a transition $(s,i)$ has an *output fault* if $\delta(s,i)=\Delta(s,i)$ while $\lambda(s,i)\neq\Lambda(s,i)$. An implementation has a *single output fault* if one and only one of its transitions has an output fault. We say that a transition has a *transfer fault* if $\delta(s,i)\neq\Delta(s,i)$, i.e. given state $s$ and input $i$, the implementation enters a different state than that specified by the next-state function of the specification. An implementation has a *single transfer fault* if there is no output fault in the implementation and one and only one of its transitions has a transfer fault. We say that the implementation under test of the given system may have *multiple faults* if several transitions have output or transfer faults.

In the context of fault diagnosis and for the rest of this thesis, we also mention the fault model based on output and transfer faults that is often used for diagnosis of a system decomposed into components, where only one component may be faulty [Kle87] [Ghe92a] [Ghe93a]. In our context, the specification is a decomposed system; i.e. its implementation is a system of two *ComFSMs*, where at most one of these machines is faulty. In the single fault diagnostic method [Elf99] presented in Chapter 6, we assume, as in [Lee93], [Ghe92a] and [Ghe93a], that the implementation under test may have a single output or transfer fault in one of its transitions. However, in the multiple faults diagnostic method [Elf01] presented in Chapter 7, we assume that multiple output or transfer faults may occur in at most one component.

## 5.4 Previous Work on the Fault Diagnoses of Systems Modeled as Two Communicating FSMs

Up to date, little work has been reported for generating diagnostic tests for systems modeled as two ComFSMs. The only method, we know about, was the one presented in [Ghe93a]. In this section we provide a brief description of this method.

In [Ghe92a], a diagnostic algorithm for systems represented by a single FSM is proposed. Such an algorithm localizes the faulty transition in the system implementation once the fault has been detected. It generates, if necessary, additional diagnostic tests that depend on the observed symptoms in order to locate the detected fault. The algorithm guarantees the correct diagnosis of any single (output or transfer) fault in the system.

In [Ghe93a], the diagnostic approach presented in [Ghe92a] is generalized to the case where the (distributed) system specifications and implementations are represented by two ComFSMs. One of the two implementations is assumed to have a single fault, either transfer or output. The algorithm is **assumed** to localize the faulty transition in the system implementations once the fault has been detected.

The method starts by the generation of expected outputs for a given test suite *TS*. Afterwards, the observed outputs are generated by the application of *TS* to the system under test. A comparison of expected and observed outputs identifies the symptoms.

For each test case $tc_i$ of *TS* with symptoms and for each machine $M_i$ ($i$=1, 2) of the system, its corresponding conflict set is determined. A so-called *conflict set* for a given test case is defined to be the set of transitions which are supposed to participate (through their execution) in the generation of the symptom outputs. The conflict set for a machine $M_i$ ($i$=1, 2) is formed by all transitions executed in the $M_i$ specification when the corresponding test case is applied. No transitions, executed after the observation of the first symptom in $tc_i$, are included in the different conflict sets of $M_i$.

Afterwards, for each machine $M_i$ ($i$=1, 2) of the system, the intersection of all conflict sets is formed. Each transition say $T_k$ (or so-called *tentative candidate transition*) in this intersection, represents a tentative faulty transition (with an output or transfer fault) that may explain all symptoms.

For each $T_k$ all possible faulty outputs are considered, one at a time. Each possible faulty output is then eliminated if it does not explain the observations (i.e, the expected and the observed outputs are not equal), else a diagnostic candidate (diagnosis) with an output fault stating that $T_k$ might have an output fault is generated.

Moreover, for each transition $T_k$, a set of all faulty transfer states called "EndState$_k$", to which $T_k$ might transfer is computed. Here, all states of the machine are considered with the exception of the expected next state of $T_k$, one at a time. For each state $s$ under consideration, $s$ in included in EndState$_k$, if under the assumption that $s$ is the next state of $T_k$, the expected and the observed outputs are equal for all succeeding transitions for all test cases of $TS$. Afterwards, all correct transitions $T_k$ are removed (i.e transitions with empty EndState$_k$). For each remaining transition $T_k$, a corresponding diagnostic candidate stating that $T_k$ might have a transfer fault is generated.

Afterwards, additional tests are generated and applied to the SUT in order to discriminate between the generated diagnoses. For each diagnosis candidate with a possible transfer fault at transition $T_k$, additional tests are generated and executed in order to be able to know exactly to which state does $T_k$ transfers. These tests should have the ability of distinguishing between the different states contained in the corresponding ending state set "EndStates$_k$ " of $T_k$ and possibly its correct ending state. Therefore, a set called a "*limited characterization set*" $W_k$ is computed for the states in EndStates$_k$ and the correct state. It is different from the characterization set defined in [Cho78], since it concerns only a subset of states rather than the whole set of states in the machine. This set is formed by sequences of inputs such that if applied to the machine in one of the states in EndStates$_k$, the produced outputs will be

different from the outputs obtained if the same input sequences were applied to the machine in any other state of EndStates$_k$ or the correct state. Each additional test case is a concatenation of an input sequence, called *transfer sequence*, required to take the machine from its initial state to the starting state of $T_k$, the input for $T_k$ and a sequence of inputs from the $W_k$.

Moreover, in order to distinguish between diagnoses with internal output faults (i.e output faults that correspond to the transitions that communicate their output to the other machine), for each diagnosis with a possible internal output fault at transition $T_k$, additional tests are generated and executed in order to be able to know exactly what output $T_k$ produces. Each additional test case is a concatenation of an input sequence, called transfer sequence, required to take the machine $M_i$ from its initial state to the starting state of $T_k$, the input for $T_k$ and a sequence of inputs from what is called "*the distinguishing set*" $U_k$. As stated in [Ghe93a], the characteristic of the sequences in $U_k$ is that once incorporated in the additional test cases, they will have the ability of distinguishing between the different possible outputs which might be generated by $T_k$ and communicated to the machine $M_j$. In other words, if $M_j$ in a state s receives an input symbol $x$ (i.e. the output of $T_k$ ) from $M_i$, it will execute a precise corresponding transition $t$ and will reach a state $s'$, then, a sequence from $U_k$ will be applied to $M_j$ in state $s'$. If a faulty input symbol $x'$ (instead of $x$) is received by $M_j$ in state $s$, a different transition $t'$ will be executed and **possibly** a different output will be generated and a different state will be reached. Therefore, as stated in [Ghe93a], the different sequences of $U_k$ will identify such an anomaly. Consequently, if the application of these additional tests generates the expected outputs, the transition $T_k$ is declared correct.

The method stops when one diagnosis is left, and identifies the faulty transition accordingly.

## 5.5 Limitations of Ghedamsi's Method

Unfortunately, the method proposed in [Ghe93a] and described above does not work in all cases although it is based on a simplifying assumption. In some cases, it is impossible to externally distinguish between a fault in machine $M_i$ and a fault in machine $M_j$, for $i \neq j$, and $i$, $j$ = 1,2. The simplifying assumption considered in [Ghe93a] states that if a component machine say $M_1$ in a state s receives an internal input symbol say z from the other component $M_2$, it will execute a precise corresponding transition $t$ and will reach a state $s'$. If a faulty input symbol $z'$ (instead of $z$) is received by $M_1$ in state $s$, a different transition $t'$ will be executed and a different output will be generated and a different state will be reached [Ghe93a]. Thus, if we relax the above assumption, for a selected diagnosis with a possible internal output fault at transition $T_k$, it might not be possible to construct the distinguishing set $U_k$ (described above) for $T_k$.

As an example for the case when we cannot distinguish between a fault in $M_1$ and a fault in $M_2$, let us consider the system of two ComFSMs $M_1$ and $M_2$ shown in Fig. 5.5. The reference system of $M_1$ and $M_2$, $RS = M_1 \Diamond M_2$ is shown in Fig. 5.6.

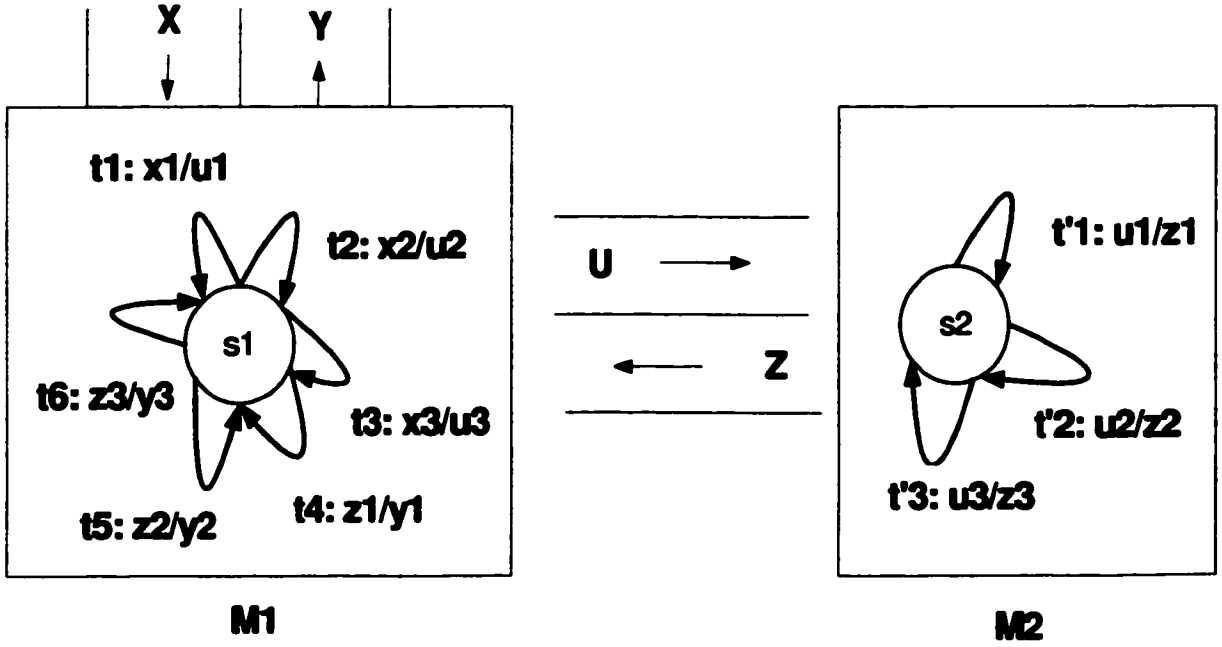


*Figure 5.5. A system of two CFSMs, $M_1$ and $M_2$*

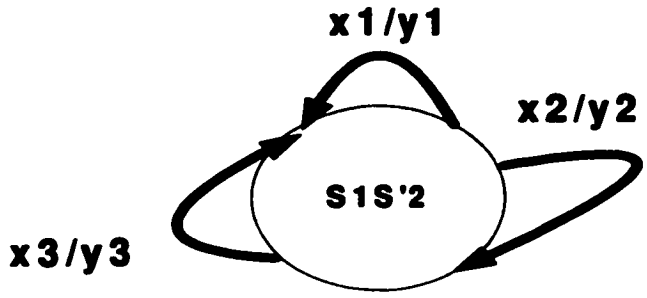


*Figure 5.6. Reference System of the $M_1$ and $M_2$ of Fig. 5.5*

Suppose that the test suite $TS$ = {$r.x_1$, $r.x_2$, $r.x_3$} is given for the two *ComFSMs* specification shown in Figure 5.5. The application of $TS$ to the specification of Figure 5.5 and its corresponding implementation of $M_1$ and $M_2$ yields the expected and observed output sequences depicted in Table 5.1. Here, for this example, we assume that the implementation of $M_1$ has an output fault, where $t_1$ produces $u_2$ instead $u_1$of in response to the input $x_1$.

Based on the observed outputs of Table 5.1, one of the following tentative candidate transitions, $t_1$ or $t_4$ of $M_1$, and $t'_1$ of $M_2$ could have an output fault. We let $DC_{M1,1}$ and $DC_{M1,2}$, be the diagnostic candidates of machine $M_1$, and $DC_{M2,1}$ be the diagnostic candidate of machine $M_2$, where:

$DC_{M1,1}$ = $M_1$ where $t_1$ has produced $u_2$ instead of $u_1$ in response to the input $x_1$.

$DC_{M1,2}$ = $M_1$ where $t_4$ has produced $y_2$ instead of $y_1$ in response to the input $z_1$.

$DC_{M2,1}$ = $M_2$ where $t'_1$ has produced $z_2$ instead of $z_1$ in response to the input $u_1$.

| *tc#* | *tc1* | *tc2* | *tc3* |
|---|---|---|---|
| Inputs | $r.x_1$ | $r.x_2$ | $r.x_3$ |
| Specified transitions | $t_1$, $t'_1$, $t_4$ | $t_2$, $t'_2$, $t_5$ | $t_3$, $t'_3$, $t_6$ |
| Expected Outputs | $y_1$ | $y_2$ | $y_3$ |
| Observed Outputs | $y_2$ | $y_2$ | $y_3$ |

*Table 5.1. Test cases and their outputs*

Unfortunately, we cannot distinguish (externally) between the given faulty transition $t_1$ of the implementation of $M_1$, where $t_1$ produces $u_2$ instead of $u_1$ in response to the input $x_1$ at state $s_1$, and another possible faulty transition $t'_1$ of the implementation of $M_2$, where $t'_1$ produces $z_2$ instead of $z_1$ in response to the input $u_1$ at state $s'_2$. The reason behind this observation is that the reference system of the implementations assumed at hand (i.e $DC_{M1,1} \Diamond M_2$), is equivalent to the reference system of other possible implementations, namely, $M_1 \Diamond DC_{M2,1}$ and $DC_{M1,1} \Diamond M_2$, due to the fact that :

$$DC_{M1,1} \Diamond M_2 = M_1 \Diamond DC_{M2,1} = DC_{M1,2} \Diamond M_2$$

This is the reason why we can not distinguish externally between the given faulty implementation of $M_1$ with an output fault at $t_1$ where $t_1$ produces $u_2$ instead of $u_1$ in response to the input $x_1$ at state $s_1$, and a possible faulty implementation of $M_2$ with an output fault at $t'_1$ where $t'_1$ produces $z_2$ instead $z_1$ of in response to the input $u_1$ at state $s'_2$.

Based on the above observation, in the following chapter, we present a new single fault diagnostic method.

# 6 Diagnosing Single Faults in Communicating Finite State Machines

## 6.1 Introduction

In this chapter, we propose a two-level diagnostic method [Elf99b] for the case that the system specification and implementation are given in the form of two communicating finite state machines and at most a single component machine may have up to one fault. At the first diagnostic level, the method enables us to decide if it is possible to identify the faulty machine in the system, once a fault has been detected based on the external interactions of the system. If this is possible, at the second level, it determines whether it is possible to locate the fault within the faulty component. If this is possible, the method provides additional test cases to locate the fault. Two examples are used to demonstrate the different steps of the method and some experimental results are provided. Moreover, the chapter includes the complexity analysis of the method.

## 6.2 The Diagnostic Approach

We recall from the previous chapter that in the context of fault diagnosis, often the fault model based on output and transfer faults is used for diagnosis of a system decomposed into components, where only one component may be faulty [Kle87] [Ghe93a]. In our context, the specification and its implementation can be represented as a system of two *ComFSMs*, where at most one of these machines is faulty. Here we assume as in [Lee93], [Ghe92a] and [Ghe93a], that the implementation under test may have a single output or transfer fault in one of its component machines.

### 6.2.1 An Overview of the Diagnostic Approach

If the implementation of the given system, i.e. the implementation of the context and the embedded component, does not pass a given test suite, the method starts by generating the *symptoms*, that is the difference between how the system should behave (expectations), and how it is actually behaving (observations). In order to explain these symptoms, for each component machine in the system, the transitions that are suspected to be faulty are

determined. One of these transitions must be faulty if the given component machine contains the fault. For each of these transitions, its corresponding *tentative diagnostic candidates* (machines containing a fault) are generated. Each of these candidates describes the behavior of the component with a possible fault in a suspected faulty transition. Afterwards, each tentative diagnostic candidate is combined with the specification of the other component to obtain what we call a *behavior diagnostic candidate*. This candidate describes the behavior of the overall system for the fault predicted by its corresponding tentative diagnostic candidate. Then, each behavior diagnostic candidate that does not succeed to explain the observed behavior of the SUT is eliminated. A particular behavior diagnostic candidate fails to explain the observed behavior, if its expected outputs are not equal to the observed outputs of the SUT for some test case in the *TS*. All remaining behavior candidates are considered as *preliminary diagnostic candidates* (PDC). If there are no preliminary candidates left for one of the two components, the other component is declared faulty. Otherwise, additional tests are generated and applied to the SUT to distinguish between these candidates and the SUT. Each of these tests reduces, at least by one, the number of these candidates. If after such a step, there are no remaining preliminary candidates for one of the two component machines, then the other component is declared faulty. In this case, and if the second level of diagnostic is desired, the method continues by generating and applying to the SUT, additional tests to reduce the number of the remaining candidates of the faulty component to a single candidate in order to locate the fault. Otherwise, if there are two equivalent preliminary candidates, one for each component, which are equivalent to the SUT, then the faulty machine cannot be identified. Similarly, in certain cases, the second level of diagnostic may not be possible if different single faults in the faulty component lead to a behavior equivalent to that of the SUT.

In order to draw one of the above conclusions, we should eliminate the preliminary diagnostic candidates that do not correspond to the SUT. This can be done by generating for each pair of candidates, say $PDC^{(1)}$ and $PDC^{(2)}$, a test sequence that allows to distinguish between them. This sequence can be generated using the method presented in [Gil62].

Applying this sequence to the SUT will lead to one of the following situations:

(i) The observed output is equal to the one expected for $PDC^{(1)}$.

(ii) The observed output is equal to the one expected for $PDC^{(2)}$.

(iii) The observed output is different from both of the outputs expected for $PDC^{(1)}$ and $PDC^{(2)}$.

In cases (i) or (ii), we know that $PDC^{(2)}$ or $PDC^{(1)}$, respectively, is a wrong diagnosis. In case (iii), we know that both, $PDC^{(1)}$ and $PDC^{(2)}$ are wrong diagnoses. This reduces the number of preliminary candidates at least by one.

### 6.2.2 The Algorithm

**Step-1: Generation of expected outputs, observed outputs, and symptoms**

A given Test Suite (*TS*), which could be derived using the methods presented in [Cho78] or [Fuj91]. Each test case $tc_k = x_{k,1},x_{k,2},...,x_{k,mk}$ of *TS* is applied to both the reference system ($RS = M_1 \Diamond M_2$) and the SUT. Moreover, for each test case $tc_k$, its expected and observed output sequences are generated. We let $o_k = o_{k,1},o_{k,2},...,o_{k,mk}$ denote the expected output sequence of test case $tc_k$, and $\hat{o}_k = \hat{o}_{k,1},\hat{o}_{k,2},...,\hat{o}_{k,mk}$ denote the observed one, where $o_{k,l}$ ($\hat{o}_{k,l}$) is expected (observed) after input $i_{k,l}$. The observed outputs are compared with the corresponding expected outputs and their differences are identified. Each difference ($o_{k,l} \neq \hat{o}_{k,l}$) represents a symptom.

**Step-2: Generation of tentative, behavior and preliminary diagnostic candidates**

For each symptom ($o_{k,l} \neq \hat{o}_{k,l}$) and machine $M_i$ ($i$=1,2), we determine the sequence of transitions (called a *fault-containing path*) that is supposed to be executed by the machine, according to the specification of the two components, for the generation of the symptom output sequence. One of these transitions must be faulty if the given machine contains the fault. For example, a fault-containing path for machine $M_1$ is formed by all transitions

executed by $M_1$ when the corresponding test case is applied. No transition executed after the observation of the symptom in a test case will be included in the fault-containing path.

Afterwards, for each machine in the system, we determine the transitions that are suspected to be faulty by forming the intersection of the transitions of the fault-containing paths for all symptoms. For each of these transitions, say $T_k$, of machine $M_i$, we form the corresponding tentative diagnostic candidates (*TDC*). Each of these candidates is formed by computing and assigning to $T_k$ a specific fault (specific output or specific transfer) and by leaving all remaining transitions of $M_i$ unchanged.

For convenience of presentation, we let $n$ (or $m$) denote the number of tentative diagnostic candidates of machine $M_1$ (or $M_2$), and $TDC_{M_1,r}$ (where $r = 1...n$) be the tentative candidates of $M_1$, and $TDC_{M_2,s}$ (where $s = 1..m$) be the tentative candidates of $M_2$. For each of these candidates we generate its corresponding behavior diagnostic candidate by forming the composition $TDC_{M_1,r} \Diamond M_2$ and $M_1 \Diamond TDC_{M_2,s}$. In our context, each behavior diagnostic candidate corresponds to a particular (faulty) implementation of $M_1$ (or $M_2$) obtained by the fault predicted by $TDC_{M_1,r}$ (or $TDC_{M_2,s}$) and the assumed non-faulty implementation of the other machine $M_2$ (or $M_1$). Finally, we eliminate from these behavior candidates those that do not succeed to explain the observed behavior of the SUT. A particular behavior diagnostic candidate fails to explain the observed behavior, if its expected outputs are not equal to the observed outputs of the SUT for some test case of *TS*. All remaining behavior candidates are considered as *preliminary diagnostic candidates* (*PDC*).

For convenience of presentation, we let $NumCandM_1$ be the number of preliminary diagnostic candidates of $M_1$ and the mutli-set $P_{M_1} = \{PDC^{(k)}_{M_1} \mid (k = 1... NumCandM_1)\}$ be the preliminary diagnostic candidates of $M_1$. Also, we let $NumCandM_2$ be the number of preliminary diagnostic candidates of $M_2$ and the multi-set $P_{M_2} = \{PDC^{(l)}_{M_2} \mid (l = 1... NumCandM_2)\}$ be the preliminary diagnostic candidates of $M_2$. A *multi-set* is a set that may have equal elements (i.e. equivalent preliminary candidates obtained through different faulty transitions).

**Step-3A: Diagnostic level-1.** If one of the sets $P_{M_1}$ (or $P_{M_2}$) is empty while the other $P_{M_2}$ (or $P_{M_1}$) is not then we conclude that $M_2$ (or $M_1$) is faulty. Go to Step-3B. If both sets are empty then the SUT has a fault that does not match our fault model. If both sets are not empty and all their preliminary diagnostic candidates are equivalent to one another, then the faulty component machine $M_1$ or $M_2$ cannot be identified. This is due to the fact that there is a possible fault in $M_1$ and a possible fault in $M_2$ such that the behavior of the SUT is equal to both of these faulty implementations (End of diagnostic process). If the set $P_{M_1}$ contains a machine $PDC^{(k)}_{M_1}$ such that $PDC^{(k)}_{M_1} \neq PDC^{(l)}_{M_2}$ is true for some machine $PDC^{(l)}_{M_2}$ in $P_{M_2}$, then we add to the test suite a new test sequence $\alpha$ that distinguishes these machines, and we apply this sequence to the SUT. If the output response of the SUT to $\alpha$ does not coincide with that of $PDC^{(k)}_{M_1}$, we delete $PDC^{(k)}_{M_1}$ from the set $P_{M_1}$, and if this response does not coincide with that of $PDC^{(l)}_{M_2}$, we delete $PDC^{(l)}_{M_2}$ from the set $P_{M_2}$. Go back to Step-3A.

**Step-3B: Diagnostic level-2.** If FSM $M_i$ is faulty, then we proceed as follows: If the set $P_{M_i}$ has a single preliminary candidate, then the faulty transition is the one predicted by that candidate (End of diagnostic process). If the set $P_{M_i}$ has only equivalent machines, then the faulty transition cannot be located (End of diagnostic process). If the set $P_{M_i}$ has machines $PDC^{(j1)}_{M_i} \neq PDC^{(j2)}_{M_i}$ for some $j1 \neq j2$ in $k$ if $M_i = M_1$ or for some $j1 \neq j2$ in $l$ if $M_i = M_2$, then we derive a new test sequence $\alpha$ that distinguishes these machines, and we apply this sequence to the SUT. If the output response of the SUT to $\alpha$ does not coincide with that of machine $PDC^{(j1)}_{M_i}$, we delete that machine from the set $P_{M_i}$, also if this response does not coincide with that of machine $PDC^{(j2)}_{M_i}$, we delete that machine from the set $P_{M_i}$. If the set $P_{M_i}$ is empty, then the SUT has a fault that does not match our fault model (End of diagnostic process). If the set $P_{M_i}$ is not empty, then we go back to Step-3B.

## 6.3 Two Application Examples

In the following two subsections, two examples are given to demonstrate the different steps of the diagnostic method described in the previous section. As mentioned before, the diagnostic method can be used for the case where a single transfer or output fault may be present in one of the implementation component FSMs. However, for simplicity of presentation we assume, for the following examples, that only a single output fault may occur in one of the two component implementations.

Consider the two machines $M_1$ and $M_2$ shown in Figure 6.1. The set of external inputs is $X=\{x_1, x_2\}$, the set of external outputs is $Y=\{y_1, y_2, y_3\}$, the set of internal inputs is $U=\{u_1, u_2\}$, and the set of internal outputs is $Z = \{z_1, z_2\}$. The corresponding composed machine $M_1 \Diamond M_2$ is shown in Figure 6.2.

Suppose the test suite $TS = \{r.x_1.x_1.x_1,\ r.x_1.x_2.x_2,\ r.x_2.x_1.x_1.x_2,\ r.x_2.x_2.x_1\}$ is given for the two *ComFSMs* specification shown in Figure 6.1. *TS* was derived from the reference system $RS=M_1 \Diamond M_2$ of Fig. 6.2, using the Wp-method [Fuj91].

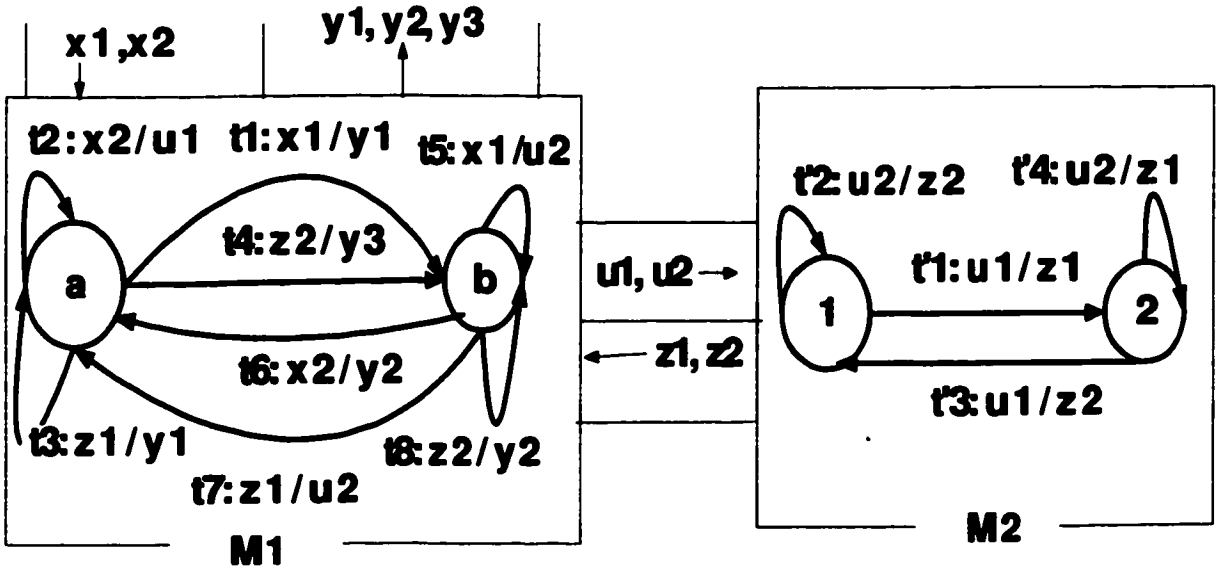


Figure 6.1. *A system of two ComFSMs $M_1$ and $M_2$*

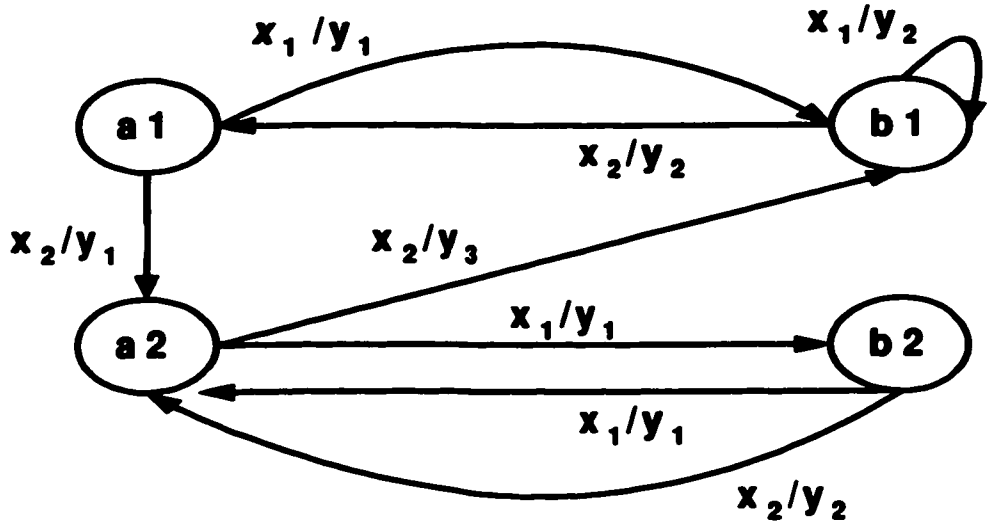


Figure 6.2. *Composed Machine ($M_1 \Diamond M_2$) obtained from the $M_1$ and $M_2$ of Fig.* 5.1

### 6.3.1 Example-1

The application of *TS* to the specification of Figure 6.1 and the corresponding implementation of $M_1$ and $M_2$ (which are equal to the specification except that $t_5$ of $M_1$ has the faulty output $y_2$) yields the expected and observed output sequences depicted in Table 6.1.

| *Tc#* | *tc*1 | *tc*2 | *tc*3 | *tc*4 |
|---|---|---|---|---|
| Inputs | *r*. *x*1. *x*1. *x*1 | *r*. *x*1. *x*2. *x*2 | *r*. *x*2. *x*1. *x*1. *x*2 | *r*. *x*2. *x*2. *x*1 |
| Specified Transitions | *r*,*t*1,*t*5, *t*'2, *t*8, *t*5, *t*'2, *t*8 | *r*,*t*1, *t*6, *t*2, *t*'1, *t*3 | *r*,*t*2, *t*'1, *t*3, *t*1,*t*5, *t*'4, *t*7, *t*'4, *t*3,*t*2, *t*'3, *t*4 | *r*,*t*2, *t*'1, *t*3, *t*2, *t*'3, *t*4, *t*5, *t*'2, *t*8 |
| Expected Outputs | *y*1 *y*2 *y*2 | *y*1 *y*2 *y*1 | *y*1 *y*1 *y*1 *y*3 | *y*1 *y*3 *y*2 |
| Observed Outputs | *y*1 *y*2 *y*2 | *y*1 *y*2 *y*1 | *y*1 *y*1 **y2** *y*2 | *y*1 *y*3 *y*2 |

*Table* 6.1. *Test cases and their outputs*

A difference between observed and expected outputs is detected for test cases *tc*3 (shown in bold). Therefore, the symptom is: $Symp_1 = (o_{tc3,3} \# \hat{o}_{tc3,3})$

Corresponding to the above symptom, we determine the following fault-containing paths for both machines $M_1$ and $M_2$, which are equal to the transitions that are suspected to be faulty (intersection of the transitions of fault-containing paths of each machine) for this particular example:

$Conf^{M1} = \{ t_1, t_2, t_3, t_5, t_7 \}$ ; $Conf^{M2} = \{ t'_1, t'_4 \}$.

Corresponding to these transitions, we compute the following tentative diagnostic candidates for $M_1$ and $M_2$:

$TDC_{M_{1,1}}$ , $TDC_{M_{1,2}}$, $TDC_{M_{1,3}} = M_1$ except that $t_1$ has been changed from $x_1/y_1$ to $x_1/y_2$, $x_1/u_1$ and $x_1/u_2$, respectively.

$TDC_{M_{1,4}}$ , $TDC_{M_{1,5}}$, $TDC_{M_{1,6}} = M_1$ except that $t_2$ has been changed from $x_2/u_1$ to $x_2/u_2$, $x_2/y_1$ and $x_2/y_2$, respectively.

$TDC_{M_{1,7}}$ , $TDC_{M_{1,8}}$, $TDC_{M_{1,9}} = M_1$ except that $t_3$ has been changed from $z_1/y_1$ to $z_1/u_2$, $z_1/u_1$ and $z_1/y_2$, respectively.

$TDC_{M_{1,10}}$ , $TDC_{M_{1,11}}$, $TDC_{M_{1,12}} = M_1$ except that $t_5$ has been changed from $x_1/u_2$ to $x_1/y_2$, $x_1/y_1$ and $x_1/u_1$, respectively.

$TDC_{M_{1,13}}$ , $TDC_{M_{1,14}}$, $TDC_{M_{1,15}} = M_1$ except that $t_7$ has been changed from $z_1/u_2$ to $z_1/y_1$, $z_1/y_2$ and $z_1/u_1$, respectively..

$TDC_{M_{2,1}} = M_2$ except that $t'_1$ has been changed from $u_1/z_1$ to $u_1/z_2$.

$TDC_{M_{2,2}} = M_2$ except that $t'_4$ has been changed from $u_2/z_1$ to $u_2/z_2$.

We notice that all the behavior diagnostic candidates that correspond to the above tentative diagnostic candidates, except $TDC_{M_{1,10}}$ and $TDC_{M_{2,2}}$, do not explain all the observed outputs of the SUT, and thus are not considered as preliminary diagnostic candidates. For example, if $TDC_{M_{1,1}}$ ($t_1$: $x_1/y_2$) is the faulty implementation, then the SUT = $TDC_{M_{1,1}} \Diamond M_2$ must produce the output sequence $y_2y_2y_2$ for $tc_1$. However, it produces the external output $y_1y_2y_2$ as shown in Table 6.1. The remaining behavior diagnostic candidates, $PDC^{(1)}{}_{M_1} = TDC_{M_{1,10}} \Diamond M_2$ and $PDC^{(1)}{}_{M_2} = M_1 \Diamond TDC_{M_{2,2}}$, are preliminary diagnostic candidates and are

found to be equivalent. Therefore, the faulty machine of the SUT cannot be identified. However, we know that the faulty transition is either that predicted by $TDC_{M1,10}$ of $M1$ or $TDC_{M2,2}$ of $M2$.

### 6.3.2 Example-2

As another example, we assume that the application of *TS* to the specification of Figure 6.1 and its corresponding implementation of $M1$ and $M2$ yields the expected and observed output sequences depicted in Table 6.2.

| *Tc#* | *tc1* | *tc2* | *tc3* | *tc4* |
|---|---|---|---|---|
| Inputs | $r\ x1\ x1\ x1$ | $r\ x1\ x2\ x2$ | $r\ x2\ x1\ x1\ x2$ | $r\ x2\ x2\ x1$ |
| Specified Transitions | $r, t1, t5,$ $t'2, t8,$ $t5, t'2, t8$ | $r, t1, t6,$ $t2, t'1, t3$ | $r, t2, t'1, t3,$ $t1, t5, t'4, t7, t'4,$ $t3, t2, t'3, t4$ | $r, t2, t'1, t3,$ $t2, t'3, t4,$ $t5, t'2, t8$ |
| Expected Outputs | $y1\ y2\ y2$ | $y1\ y2\ y1$ | $y1\ y1\ y1\ y3$ | $y1\ y3\ y2$ |
| Observed Outputs | $y1\ \mathbf{y1}\ y1$ | $y1\ y2\ y1$ | $y1\ y1\ y1\ y3$ | $y1\ y3\ \mathbf{y1}$ |

*Table 6.2. Test cases and their outputs*

By applying the diagnostic method, we find that the transitions that are suspected to be faulty are $t5$, and $t8$ of $M1$ and $t'2$ of $M2$. Corresponding to these transitions, we compute seven behavior diagnostic candidates, four of which are eliminated since they do not correspond to the observed behavior of the SUT. The remaining three candidates (shown in Figure 6.3) $PDC^{(1)}{}_{M1} = TDC_{M1,4} \Diamond M2$, $PDC^{(1)}{}_{M2} = M1 \Diamond TDC_{M2,1}$, and $PDC^{(2)}{}_{M1} = TDC_{M1,5} \Diamond M2$ are preliminary candidates, where $TDC_{M1,4}$, $TDC_{M1,5} = M1$ except that $t8$ has been changed from $z2/y2$ to $z2/y1$ and $z2/u1$, respectively. $TDC_{M2,1} = M2$ except that $t'2$ has been changed from $u2/z2$ to $u2/z1$.

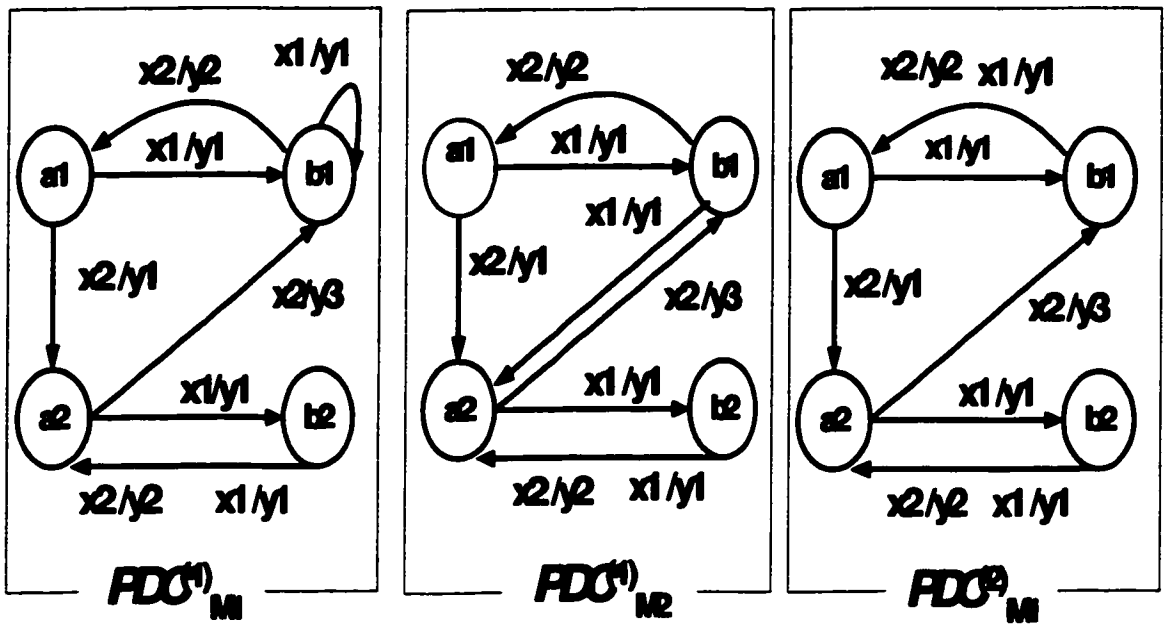


*Figure* 6.3. *Preliminary diagnostic candidates* $PDC^{(1)}{}_{M1}$ , $PDC^{(1)}{}_{M2}$ *and* $PDC^{(2)}{}_{M1}$

Using Gill's method [Gil62], the test sequence $x_1x_1x_2$ is found to distinguish between $PDC^{(1)}{}_{M1}$ and $PDC^{(1)}{}_{M2}$. If the SUT produces the output response $y_1y_1y_2$ to the test case $x_1x_1x_2$ then $TDC_{M_{1,4}}$ is the faulty implementation of $M_1$. If it produces the output sequence $y_1y_1y_3$ then $TDC_{M_{2,1}}$ is the faulty implementation of $M_2$. Moreover, if the SUT produces the output response $y_1y_1y_1$ to $x_1x_1x_2$ then both candidates $PDC^{(1)}{}_{M1}$ and $PDC^{(1)}{}_{M2}$ are eliminated, and $TDC_{M_{1,5}}$ is the faulty implementation of $M_1$.

## 6.4 Experimental Results

We have implemented and experimented with our diagnostic method in order to determine how often a faulty machine can be identified for various implementations of specifications of the given system. Most of the specifications are generated randomly using a uniform distribution. For each specification, corresponding faulty implementations (experiments) are created by altering the specified output of a randomly selected transition of one of the two component machines. The diagnostic method is implemented as tcl scripts that interface with INRIAs CADP tool. The tool is used to generate the behavior diagnotic candidates of the component machines. In Table 6.3, the results of the experiments are presented. Column

I includes the number of states of the component machines $M_1$ and $M_2$ i.e. $|n_1|;|n_2|$. Column II includes the number of external input and output symbols of $M_1$, i.e $|X|;|Y|$, and column III includes the number of input and output symbols of $M_2$, i.e $|U|;|Z|$. Column IV contains the number of generated implementations (experiments) for component FSMs $M_1$ or $M_2$. Column V indicates for how many of these implementations (experiments) the faulty component machine could not be identified. We note that the first row of Table 6.3 includes the experiments done on the specifications given in Figure 6.1 used and discussed in the previous sections, and row 7 includes the experiments done on the specifications of a telephone services system taken from (Cavali et al., 2000). Moreover, the experiments shown in all rows except those of rows 5 and 8 were exhaustive in the sense that they consider all possible faulty implementations of machine $M_1$(or $M_2$) and for each of these implementations, it is determined if there exists a possible faulty implementation of $M_2$ ($M_1$) such that the behavior of the system for both these faulty implementations is the same (i.e. could not locate the faulty machine).

It is clear that for 13 of the below 3816 experiments, the faulty component in the system could not be identified. Moreover, it seems that as the system gets larger, the ability to locate the faulty component increases.

| I<br>$\|n_1\|;\|n_2\|$ | II<br>$\|X\|;\|Y\|$ | III<br>$\|U\|;\|Z\|$ | IV<br>(Experiments) | V<br>(faulty component can not be identified) |
|---|---|---|---|---|
| 2; 2 | 2; 3 | 2; 2 | 32;0 | 1 |
| 4; 2 | 2; 3 | 2; 3 | 64;0 | 0 |
| 6; 3 | 6; 6 | 6; 6 | 792;0 | 3 |
| 6; 6 | 6; 6 | 6; 6 | 792;0 | 0 |
| 6; 6 | 6; 6 | 6; 6 | 0;180 | 0 |
| 8; 4 | 8;8 | 4; 4 | 1056;0 | 0 |
| 8; 2 | 6; 7 | 4; 3 | 720;0 | 9 |
| 12; 6 | 6; 6 | 6; 6 | 0;180 | 0 |
| | | **Total** | **3816** | **13** |

*Table* 6.3. *Experimental results based on the first diagnostic level*

Furthermore, we have also experimented with our diagnostic method in order to determine how often a faulty transition of a faulty component machine can be identified for various implementations of specifications of component machines. The specifications considered are the same as those of Table 6.3 above. For each specification of a particular component machine, corresponding faulty implementations (experiments) are created by altering the specified output of a randomly selected transition of the machine. Afterwards, the second diagnostic level (i.e. Step-3B of the algorithm) is used to determine if there is another possible faulty transition of the component machine such that the behavior of the system for both these transitions is the same. Table 6.4, include the results of these experiments, where Column V indicates for how many experiments the faulty transition could not be located.

| I<br>$\|n_1\|;\|n_2\|$ | II<br>$\|X\|;\|Y\|$ | III<br>$\|U\|;\|Z\|$ | IV<br>(Experiments) | V<br>Could not locate the faulty transition |
|---|---|---|---|---|
| 2; 2 | 2; 3 | 2; 2 | 32;0 | 4 |
| 4; 2 | 2; 3 | 2; 3 | 64;0 | 32 |
| 6; 3 | 6; 6 | 6; 6 | 792;0 | 28 |
| 6; 6 | 6; 6 | 6; 6 | 792;0 | 6 |
| 6; 6 | 6; 6 | 6; 6 | 0;180 | 4 |
| 8; 4 | 8;8 | 4; 4 | 1056;0 | 40 |
| 8; 2 | 6; 7 | 4; 3 | 720;0 | 0 |
| 12; 6 | 6; 6 | 6; 6 | 0;180 | 0 |
| | | **Total** | **3816** | **114** |

*Table* 6.4. *Experimental results based on the second diagnostic level*

It is clear that for 114 of the above 3816 experiments, the faulty transition of the faulty component machine in the system was not identified. Moreover, it seems that as the system gets larger, the ability to locate the faulty transition increases.

## 6.5 The Complexity of the Method

In order to calculate the complexity of each step of the diagnostic method, we first consider as variables the number of states $n_1$ in the specification (implementation) of $M_1$, and $n_2$ in the specification (implementation) of $M_2$. We also let $L_{TS}$ denote the number test cases in $TS$, and $L_{TC}$ the number of inputs in the longest test case. We also let $Tr_{M_1} = (n_1 (|X| + |Z|))$ be the number of transitions of $M_1$ and $Tr_{M_2} = (n_2 |U|)$ be the number of transitions of $M_2$, where $|X|$ and $|Z|$ are the cardinalities of the input alphabets $X$ and $Z$ of $M_1$, and $|U|$ is the cardinality of the input alphabets of $M_2$.

In the following subsection, we compute the algorithmic complexity of building a composed machine from a product machine, since it will be used in different steps of the method. Afterwards, in subsequent subsections, we compute the complexity of the different steps of the diagnostic method.

### 6.5.1 The Complexity of Building a Composed Machine

As mentioned before, a composed machine $M_1 \Diamond M_2$ is obtained from the product machine $M_1 \times M_2$ that describes the joint behavior of the component machines in terms of all actions within the system. A product machine can be obtained by performing reachability computation [Bra83]. The complexity of building such a machine is the complexity of building its states and transitions. There are two types of states in a product machine, *stable* and *transient* states. A stable state has empty input queues, and thus it is ready to accept an external input action. Accepting such an action, the system changes its current state from a stable to a transient state where it cannot accept any external action. The system returns to a stable state after it has produced an external output action. The number of stable states is bounded by $n_1 n_2$, and the number of internal states is bounded by $n_1 n_2 |U| + n_1 n_2 |Z|$. Therefore, the complexity of building these states is of order equals to $O(n_1 n_2) + (n_1 n_2 |U| + n_1 n_2 |Z|)$. In order to determine the number of transitions in a product machine, we know

that at each stable state, there are |X| external input actions that can be applied at that state. Moreover, we know, by assumption, that machine is live-lock free, i.e. it does not have a cycle labeled only with internal actions. Hence, for each of these external inputs, the machine, in the worst case, might traverse all the transient states of the system. Hence, the number of transitions in a product machine is bounded by $n_1 n_2 |X|(n_1 n_2 |U| + n_1 n_2 |Z|)$. Therefore the complexity of building a product machine is of order:

$$O(n_1 n_2) + O(n_1 n_2 |U| + n_1 n_2 |Z|) + O(n_1 n_2 |X|(n_1 n_2 |U| + n_1 n_2 |Z|)) =$$
$$O(n^2_1 n^2_2 |X| (|U| + |Z|))$$

A composed machine is obtained from a product machine by hiding all internal actions in the product machine, and pairing input with output actions. For each of the $n_1 n_2$ stable states of the product machine, and for each external input in the alphabet $X$, all transient states might be traversed. Therefore, the complexity of such a step is bounded by $O(n_1 n_2 |X| [n_1 n_2 |U| + n_1 n_2 |Z|]) = O(n^2_1 n^2_2 |X| (|U| + |Z|))$.

Therefore the overall complexity of building a composed machine is of order: $ComplexityComposed = O(n^2_1 n^2_2 |X| (|U| + |Z|))$.

### 6.5.2 The Complexity of Step-1 of the Method

Each input of each test case in the initial test suite is processed for generating the expected and the observed outputs or for collecting symptoms. The expected outputs are generated by running the test cases of $TS$ against the specification, i.e. Reference System = $M_1 \Diamond M_2$. The complexity of building this specification is ***ComplexityComposed***. We assume as elementary operation, with one unit cost, the execution of a single transition of the specification. Therefore, the complexity of running the test cases of $TS$ against the reference system is of the product of the number of test cases in $TS$ and the number of inputs in the longest test case, which is $O(L_{TS} L_{Tc})$. Therefore, the complexity of generating the expected outputs is of order, $ComplexityComposed + O(L_{TS} L_{Tc})$.

We assume as elementary operation, with one unit cost, the execution of a single transition of the implementation and the comparison between the observed and expected outputs. Therefore, the complexity running the test cases of *TS* against the implementation and the comparison between the outputs is the product of the number of test cases in *TS* and the number of inputs in the longest test case, which is $O(L_{TS} L_{Tc})$.

Therefore, the overall complexity of Step-1, is of order ***ComplexityComposed*** + $O(L_{TS} L_{Tc}) = O(n^2_1 n^2_2 |X| (|U| + |Z|)) + O(L_{TS} L_{Tc})$.

### 6.5.3 The Complexity of Step-2 of the Method

In order to determine the transitions of a fault-containing path of a test case of *TS* of machine $M_i$ for $i = 1, 2$, we insert the transitions traversed in Step-1 while executing this test case against the product machine into a corresponding fault-containing path set. This set would include, in the worst case, all the transitions of the corresponding machine (i.e. $Tr_{M_i}$). We assume an elementary operation, with one unit cost, the insertion of a transition in a fault-containing path set. Therefore, the complexity of generating these sets for all test cases in the *TS*, is of order $O(L_{TS} Tr_{M_i})$.

Moreover, in order to determine, for each machine in the system, the transitions that are suspected to be faulty, the intersection of the transitions of its fault-containing paths sets is formed. There are at most $L_{TS}$ conflict sets, each of which has at most $Tr_{M_i}$ elements. In order to find the intersection of these sets, we should consider all elements of one of them, one at a time. For each of these elements, $O(L_{TS})$ comparisons are required if the conflict sets are sorted. Therefore, intersecting all conflict sets requires at most $O(L_{TS} Tr_{M_i})$ comparisons. The best known algorithm which will sort a set of $Tr_{M_i}$ elements, makes at most $O(Tr_{M_i} \log Tr_{M_i})$. Hence, the complexity of intersecting the conflict sets is

of order, $O(L_{TS}\, Tr_{M_1}) + O(L_{TS}\, Tr_{M_1} \log Tr_{M_1}) + O(L_{TS}\, Tr_{M_2}) + O(L_{TS}\, Tr_{M_2} \log Tr_{M_2})$

$= O(L_{TS}\, Tr_{M_1} \log Tr_{M_1}) + O(L_{TS}\, Tr_{M_2} \log Tr_{M_2}) =$

$O(L_{TS}\, [Tr_{M_1} \log Tr_{M_1} + Tr_{M_2} \log Tr_{M_2}]) =$

$O(L_{TS}\, [(n_1(|X| + |Z|) \log(n_1(|X| + |Z|)) ) + (n_2|U| \log(n_2\, |U|))])$

Still in Step-2, in order to form the tentative diagnostic candidates of machine $M_i$, for each transition $T_k$ of $M_i$ in the set of transitions that are suspected to be faulty, a number of diagnostic candidates is formed each by computing and assigning to $T_k$ a possible fault (output or transfer) and by leaving all remaining transitions of $M_i$ unchanged. In the worst case the number of transitions $T_k$ in the conflict set is $Tr_{M_i}$. We note that the number of possible output faults for a transition of $M_1$ is $NumO_{M_1} = (|Y| + |U| - 1)$, where $|Y|$ and $|U|$ are the cardinalities of the output alphabet sets $Y$ and $U$ of $M_1$, and the number of possible output faults for a transition of $M_2$ is $NumO_{M_2} = (|Z| - 1)$, where $|Z|$ is the cardinality of the output alphabet set $Z$ of $M_2$. Moreover, the number of possible transfer faults for a transition $T_k$ of $M_i$ is $n_i - 1$, where $n_i$ is the number states of machine $M_i$. Therefore, the number of diagnostic candidates of machine $M_i$, $NumCandM_i = Tr_{M_i}$ $(NumO_{M_i} + (n_i - 1))$. Therefore, the overall complexity of generating the tentative diagnostic candidates of machine $M_i$ is, $O(NumCandM_i) = O(Tr_{M_i}(NumO_{M_i} + n_i - 1)) =$ $O(Tr_{M_i}[NumO_{M_i} + n_i])$. Consequently, the overall all complexity of generating the tentative diagnostic candidates of machines $M_1$ and $M_2$ is,

$O(Tr_{M_1}\, [NumO_{M_1} + n_1\, ]) + O(Tr_{M_2}\, [\, NumO_{M_2} + n_2] ) =$

$O(n_1(|X|+|Z|)[(|Y|+|U|-1) + n_1]) + O(n_2|U|[(|Z|-1) + n_2]) =$

$O(n_1(|X|+|Z|)[(|Y|+|U|) + n_1]) + O(n_2|U|[|Z| + n_2]) =$

$O(\, n_1|X||Y| + n_1|X||U| + n_1|Z||Y| + n_1|Z||U| + n^2{}_1|X| + n^2{}_1|Z| + n_2|U||Z| + n^2{}_2|U|\, )$

In order to eliminate from the tentative diagnostic candidates all candidates that do not succeed to explain all observations of the SUT, first, we form, for each of these candidates,

say $Tdiagc^{Mi}$, the behavior diagnostic candidate (or composed machine), $Tdiagc^{Mi} \Diamond M_j$ where $i, j = 1, 2$, and $i \neq j$. Therefore, the complexity of this first step is of order $O(NumCandM_i\ ComplexityComposed)$. Second, we generate for each behavior diagnostic candidate its expected outputs for all test cases of $TS$ with a complexity equals to the product of the number of these candidates, the number of test cases, and the length of the inputs of the longest test case $O(NumCandM_i\ L_{TS}\ L_{Tc})$. Third, for each behavior candidate, its expected outputs are compared with the observed ones, and this is also of complexity $O(NumCandM_i\ L_{TS}\ L_{Tc}) = O(NumCandM_1\ L_{TS}\ L_{Tc}) + O(NumCandM_2\ L_{TS}\ L_{Tc})$. Therefore, the overall complexity eliminating the tentative diagnostic candidates that do not explain all observations of the SUT is of order :

$$O(NumCandM_1\ ComplexityComposed) + O(NumCandM_2\ ComplexityComposed) +$$
$$O(NumCandM_1\ L_{TS}\ L_{Tc}) + O(NumCandM_2\ L_{TS}\ L_{Tc}) =$$
$$O([NumCandM_1 + NumCandM_2]\ [ComplexityComposed + L_{TS}\ L_{Tc}]\ ) =$$
$$O([n_1|X||Y| + n_1|X||U| + n_1|Z||Y| + n_1|Z||U| + n^2{}_1|X| + n^2{}_1|Z|) + n_2|U||Z| + n^2{}_2|U|]$$
$$[(n^2{}_1 n^2{}_2|X||U| + n^2{}_1 n^2{}_2|X||Z| + L_{TS}\ L_{Tc}]\ )$$
$$= O(n^3{}_1 n^2{}_2\ (|X|^2|Y||U| + |X|^2|U|^2 + |Z||Y||X||U| + |Z||U|^2|X| + |X|^2|Y||Z| + |X|^2|U||Z| + |Z|^2|Y||X|$$
$$+ |Z|^2|U||X|\ ) +$$
$$O\ (n^4{}_1 n^2{}_2\ (|X|^2|U| + |Z||X||U| + |X|^2|Z| + |X||Z|^2)\ ) +$$
$$O(n^2{}_1 n^3{}_2\ (|X||U|^2|Z| + |X||Z|^2|U|\ ) +$$
$$O(n^2{}_1 n^4{}_2\ |X||U|^2 + |U||X||Z|) +$$
$$O(L_{TS}\ L_{Tc}\ [n_1(|X||Y| + |X||U| + |Z||Y| + |Z||U|) + n^2{}_1(|X| + |Z|) + n_2|U||Z| + n_2{}^2|U|\ ]\ )$$

Therefore, the overall complexity of the sequential steps of Step-2 of the diagnostic method, is determined by the above complexity.

### 6.5.4 The Complexity of Step-3 of the Method

In order to compute the complexity of Step-3, we first note, as reported in [Ghe93b], that the complexity of generating a distinguishing test for two machines is of order $O(I\ n_3\ n_4)$, where $I$ is the number of input alphabets of the machines and $n_3$ is the number of states of

the first machine and $n_4$ is the number of states of the second one. In our case, these machines are two behavior diagnostic candidates. In the worst case, the number of states of a behavior candidate (or a composed machine), $Tdiagc^{Mi} \Diamond M_j$ where $i, j = 1, 2$, and $i \neq j$, equals to $n_i\, n_j$, where $n_i$ is the number of states of $Tdiagc^{Mi}$ which is equal to the number of states of machine $M_i$, and $n_j$ is the number of states of $M_j$. Therefore, the complexity of generating a distinguishing test for two behavior candidates is of order $O(I\, n_i\, n_j\, n_i\, n_j) = O(I\, n^2{}_i n^2{}_j) = O(I\, n^2{}_1 n^2{}_2)$. This equals after replacing $I$ by $|X|$ to $O(|X|\, n^2{}_1 n^2{}_2)$.

Moreover, in the worst case, no preliminary diagnostic candidates are eliminated in step-2, and all the diagnostic candidates are considered as behavior diagnostic candidates. Then, for each machine $M_i$ of the system, there could be $NumCandM_i$ behavior diagnostic candidates. In Step-3, for each machine of the system, at most $NumCandM_i - 1$ additional tests are needed, in order to reduce the number of the behavior diagnostic candidates to one. Hence, the complexity of the selection of distinguishing tests to distinguish between $NumCandM_i$ behavior diagnostic candidates, is bounded by $O((NumCandM_i - 1)\ I\ n_1\ n_2) = O((NumCandM_i) I\ n_1 n_2) = O((NumCandM_1) I\ n_1 n_2) + O((NumCandM_2) I\ n_1 n_2)$, which equals, after replacing the number of input alphabets $I$ by $|X|$ to :

$O(|X|\, n^2{}_1 n^2{}_2\ (NumCandM_1)) + O(|X|\, n^2{}_1 n^2{}_2\ (NumCandM_2)) =$

$O(|X|\, n^2{}_1 n^2{}_2\ [NumCandM_1 + NumCandM_2\ ])$

Also, for each of $NumCandM_i - 1$ additional tests, the complexity of running each test against the SUT and comparing its observed outputs with those expected by the corresponding two diagnoses, is bounded by the length of this test, which was shown in Lemma 4.1 in [Gil62] to be of at most $(n_3 + n_4 - 1) = (n_1\, n_2 + n_1\, n_2 - 1)$ . Therefore, the complexity of running and comparing these tests against the outputs of the SUT, is of order

$O((NumCandM_1 - 1)\ (n_1\, n_2 + n_1\, n_2 - 1)) + O((NumCandM_2 - 1)\ (n_1\, n_2 + n_1\, n_2 - 1)) =$

$O(\ n_1\, n_2\, NumCandM_1\ ) + O(\ n_1\, n_2\, NumCandM_2\ ) =$

$O(\ n_1\, n_2\, [NumCandM_1 + NumCandM_2]\ )$

Therefore, the overall complexity of the sequential steps of Step-3 is determined by the complexity of selecting distinguishing tests which is, in the worst case, the complexity of the selection of distinguishing tests to distinguish between $NumCandM_i$ behavior diagnostic candidates.

$O(|X| n^2_1 n^2_2 (NumCandM_1)) + O(|X| n^2_1 n^2_2 (NumCandM_2)) =$

$O(|X| n^2_1 n^2_2 [NumCandM_1 + NumCandM_2]) =$

$O(|X| n^2_1 n^2_2 [n_1|X||Y| + n_1|X||U| + n_1|Z||Y| + n_1|Z||U| + n^2_1|X| + n^2_1|Z| + n_2|U||Z| + n^2_2|U|])$

$=$

$O(|X| n^3_1 n^2_2 [X||Y| + X||U| + |Z||Y| + |Z||U|]) +$

$O(|X| n^4_1 n^2_2 (|X| + |Z|)) +$

$O(|X| n^2_1 n^3_2 |U||Z|) + O(|X| n^2_1 n^4_2 |U|)$

### 6.5.5 The Overall Complexity of the Method

The overall complexity of the method is determined by the complexity the most complex step of its three sequential steps. If we assume that $n_1 = n_2 = n$, and $|X| = |Y| = |U| = |Z| = k$, then the worst case complexity of Step-1 is of order $O(k^2 n^4) + O(L_{TS} L_{Tc})$, the worst case complexity of Step-2 is of order $O(k^3 n^6) + O(kn^2 L_{TS} L_{Tc})$, and that of Step-3 is of order $O(k^2 n^6)$. Therefore, the overall worst case complexity of the method is of order $O(k^3 n^6) + O(kn^2 L_{TS} L_{Tc})$.

# 7 Diagnosing Multiple Faults in Communicating Finite State Machines

## 7.1 Introduction

In this chapter, we propose a new diagnostic method [Elf01] for the case that the system specification and implementation are given in the form of two communicating finite state machines and at most a single component machine may have multiple faults. The method enables us to decide whether it is possible to identify the faulty machine in the system, once faults have been detected in a system implementation. If this is possible, it also provides tests for identifying the faulty component machine. Two examples are used to demonstrate the different steps of the method. The method can be used for locating the faulty transitions within a faulty component machine.

This chapter is organized as follows. Section 7.2 comprises the necessary definitions for FSMs. In Section 7.3, the diagnostic problem is discussed, while the method to solve the problem is presented in Section 7.5. In Section 7.4, the different steps of the method are illustrated by a working example. Section 7.6 includes another application example, and Section 7.7 comments on the complexity of the method. Further related research work is discussed in the following chapter.

## 7.2 Finite State Machines

In this section, we recall from Chapter 2 some definitions related to the FSM model. Moreover, we present some definitions that will be used in the method presented in subsequent sections.

A non-deterministic finite state machine (FSM) is an initialized non-deterministic Mealy machine which can be formally defined as follows. A *non-deterministic finite state machine* $A$ is a 6-tuple $(S, I, O, h, D_A, s_1)$, where $S$ is a finite nonempty set of $n$ states with $s_1$ as the initial state; $I$ and $O$ are input and output alphabets; $D_A$ is a specification domain which is a subset of $SxI$; and $h$: $D_A \rightarrow 2^{SxO} \backslash \varnothing$ is a behavior function where $2^{SxO}$ is

the set of all subsets of the set $S \times O$. The behavior function defines the possible transitions of the machine. Given present state $s_i$ and input symbol $i$, each pair $(s_j,o) \in h(s_i,i)$ represents a possible transition to the next state $s_j$ with the output $o$. This is also written as a transition of the form $s_i \xrightarrow{i/o} s_j$. If $D_A = S \times I$ then $A$ is said to be a *complete* FSM; otherwise, it is called a *partial* FSM. In the complete FSM we omit the specification domain $D_A$, i.e. complete (non-deterministic) FSM is 5-tuple $A = (S, I, O, h, s_1)$. If for each pair $si \in D_A$ it holds that $|h(s,i)| = 1$ then FSM $A$ is said to be *deterministic*. In the deterministic FSM $A$ instead of behavior function $h$ we use two functions, transition function $\delta$: $S \times I \rightarrow S$ and output function $\lambda$: $S \times I \rightarrow O$. i.e. a complete deterministic FSM is a 6-tuple $M = (S, I, O, \delta, \lambda, s_1)$.

**Definition 7.1: A sub-machine of a non-deterministic FSM**

FSM $B = (S', I, O, g, s_1)$, $S' \subseteq S$, is a *sub-machine* of complete FSM $A = (S, I, O, h, s_1)$ if for each pair $si \in S' \times I$, it holds that $g(s,i) \subseteq h(s,i)$. Similar to [Kou99], we further denote $Sub(A)$ the set of all deterministic sub-machines of FSM $A$. We also use the definition of a deterministic path in FSM $A$ since only such paths can occur in its deterministic sub-machines.

**Definition 7.2: A deterministic path in a non-deterministic FSM**

A *deterministic path* $P$ of FSM $A$ starts at the initial state and has no transitions with different next states and/or outputs for the same state-input combination.

As usual, function $h$ can be extended to the set $I^*$ of finite input sequences. Given state $s \in S$ and input sequence $\alpha = i_1 i_2 ... i_k \in I^*$, output sequence $o_1 o_2 ... o_k \in h(s,\alpha)$ if there exist states $s_1 = s, s_2, \ldots, s_k, s_{k+1}$ such that $(s_{j+1}, o_j) \in h(s_j, i_j), j = 1, \ldots, k$.

We let the set $h^o(s,\alpha) = \{\gamma | \exists s' \in S\ [(s', \gamma) \in h(s, \alpha)]\}$ denote the *output projection* of $h$, while denoting $h^s(s,\alpha) = \{s' | \exists \gamma \in Y^*\ [(s', \gamma) \in h(s, \alpha)]\}$ the *state projection* of $h$. Input/Output sequence $i_1 o_1 i_2 o_2 ... i_k o_k$ is called a *trace* of $A$ if $o_1 o_2 ... o_k \in h^o(s_0, i_1 i_2 ... i_k)$.

We also use the notation $h^s{}_\gamma(s,\alpha)$ to denote the set $\{s' | (s',\gamma) \in h(s, \alpha)]\}$ of all states where the sequence $\alpha$ can take FSM $A$ from the initial state with the output response $\gamma$. For certain $(s, \gamma)$ the set $h^s{}_\gamma(s,\alpha)$ may be empty.

**Definition 7.3: Observably reachable state**

If the set of all states where the sequence $\alpha$ can take the FSM $A$ from the initial state with the output response $\gamma$ (i.e. the set $h^s{}_\gamma(s,\alpha)$) has the unique state $s$ then we say the state $s$ is *observably reachable* from the initial state via the trace $\alpha/\gamma$.

We recall that given states $s_i$ and $s_j$ of a complete FSM $A$, states $s_i$ and $s_j$ are *equivalent*, written $s_i = s_j$, if for each input sequence $i_1 i_2 ... i_k \in I^*$, it holds that $h(s_i, i_1 i_2 ... i_k) = h(s_j, i_1 i_2 ... i_k)$. If states $s_i$ and $s_j$ are not equivalent then they are *distinguishable*, written $s_i \neq s_j$. Given a complete FSM $A$, sequence $\alpha \in I^*$ such that $h(s_i, \alpha) \neq h(s_j, \alpha)$ is said to *distinguish* states $s_i$ and $s_j$. An FSM $A$ with pair-wise distinguishable states is called a *reduced* FSM.

We also recall that complete FSMs $A = (S,I,O,h,s_1)$ and $B = (T,I,O,g,t_1)$ are *equivalent*, written $A = B$, if their sets of traces coincide. It is well known, given a complete deterministic FSM $A$, there always exists a reduced FSM that is equivalent to $A$.

## 7.3 Diagnosis Problem

### 7.3.1 Diagnosis Problem Statement

Let $M_1$ and $M_2$ be complete deterministic FSMs representing the specifications of the components of the given system of two communicating finite state machines while $M'_1$ and $M'_2$ are their corresponding implementations. We propose an adaptive method for diagnostic test derivation. If the implementation system does not pass a given test suite, our algorithm enables us to decide whether it is possible to identify the faulty component of the given system under the assumption that multiple faults may occur only in one of

the implementations $M'_1$ or $M'_2$. Furthermore, if this is possible, the algorithm enables to decide whether it is possible to locate the faulty transitions within the faulty component, and if possible it locates them. Moreover, the algorithm draws the conclusion "*Faults cannot be captured by the assumed fault model*" if it has been detected that the implementation at hand has faults that cannot be captured by the assumed fault model.

The diagnostic method can be used for the case where multiple transfer or output faults may occur in one of the implementation component FSMs $M'_1$ or $M'_2$. However, for simplicity of presentation, hereafter, we assume that only output faults may occur.

### 7.3.2 An Overview of the Diagnosis Approach

Let $RS = M_1 \Diamond M_2$ be a specification system while $TS$ is a conformance test suite. If the composition $M'_1 \Diamond M'_2$ of the implementations $M'_1$ and $M'_2$ of the given system produces unexpected output responses to the given test suite $TS$, then the composition $M'_1 \Diamond M'_2$ is not equivalent to $RS$, i.e. either $M'_1$ or $M'_2$ is a faulty implementation. Our objective is to determine whether $M'_1$ is not equivalent to $M_1$ while $M'_2$ and $M_2$ are equivalent, or vice versa.

In order to determine whether the output responses of $TS$ can be produced when FSM $M'_2$ has output faults and $M'_1$ is equivalent to its specification, we derive the FSM $(M_2)_a^{out}$ by adding new (faulty) transitions with all possible outputs to each transition of $M_2$, and then we combine the obtained non-deterministic FSM with $M_1$. We call the obtained non-deterministic machine, $M_1 \Diamond (M_2)_a^{out}$, the Fault Function (FF) of the embedded component (or *FF-Embedded*). Similarly, we derive the FF of the context $M_1$, *FF-Context* = $(M_1)_a^{out} \Diamond M_2$, to determine whether the output responses of $TS$ can be produced when FSM $M'_1$ has output faults and $M'_2$ is equivalent to its specification. Fault functions were introduced in [Pet92] to represent in a concise way all mutants of a given FSM with a given type of implementation errors.

The set of all deterministic sub-machines of $M_1 \Diamond (M_2)_a^{out}$ (or $(M_1)_a^{out} \Diamond M_2$) includes all the implementations $M_1 \Diamond M'_2$ (or $M'_1 \Diamond M_2$), where output faults may be present in the implementation of component FSM $M_2$ (or $M_1$). However, the set also includes superfluous sub-machines that do not correspond to a composition of any possible deterministic component machines. This is due to the fact that while deriving the composition, we do not take into consideration that for a specific state-input combination $(s,i)$, one and only one output is possible in a deterministic implementation.

Since our implementation system is deterministic we do not take into account non-deterministic paths of machines $M_1 \Diamond (M_2)_a^{out}$ and $(M_1)_a^{out} \Diamond M_2$. Moreover, we also remove from $M_1 \Diamond (M_2)_a^{out}$ and $(M_1)_a^{out} \Diamond M_2$ any behavior that does not agree with the observed outputs to the applied test suite *TS*. In Section 7.4, we describe the algorithm that removes from machine $M_1 \Diamond (M_2)_a^{out}$ (or $(M_1)_a^{out} \Diamond M_2$) sub-machines whose output responses to the test suite do not agree with those obtained by applying the test suite *TS* to the SUT.

If the SUT is equivalent to a deterministic sub-machine of $M_1 \Diamond (M_2)_a^{out}$ then the faulty machine is $M_2$, and if it is equivalent to a deterministic sub-machine of $(M_1)_a^{out} \Diamond M_2$, then the faulty machine is $M_1$. However, if the SUT is equivalent to a deterministic sub-machine of $M_1 \Diamond (M_2)_a^{out}$ and to a deterministic sub-machine of $(M_1)_a^{out} \Diamond M_2$, then the faulty machine cannot be identified. This is due to the fact that there are some possible faults in $M_1$ and some possible faults in $M_2$ that cause the same observable behavior of the SUT. If none of the above cases applies, we conclude that the implementation has faults that are not captured by the considered fault model.

In order to draw one of the above conclusions, we should have test cases such that by observing the output responses of the SUT to these test cases, we can distinguish the SUT and sub-machines of $(M_1)_a^{out} \Diamond M_2$ and of $M_1 \Diamond (M_2)_a^{out}$. If the machines $(M_1)_a^{out} \Diamond M_2$ and

$M_1 \Diamond (M_2)_a^{out}$ have no equivalent deterministic sub-machines then there exists a so-called *distinguishing set* (written as $Dis_{Set}$) of input sequences [Kou00] such that given the set of deterministic output responses to these input sequences, we always can determine whether the machine under test is a sub-machine of $(M_1)_a^{out} \Diamond M_2$ or of $M_1 \Diamond (M_2)_a^{out}$. The algorithm for deriving such a distinguishing set is proposed in [Kou00]. We illustrate this algorithm in Section 7.4. In other cases, it may happen that there are sub-machines of $(M_1)_a^{out} \Diamond M_2$ and others of $(M_1)_a^{out} \Diamond M_2$ that are equivalent, but these sub-machines are not equivalent to the SUT. In this case, the observed outputs to the given test suite are insufficient to distinguish between these sub-machines and the SUT. Therefore, more test cases must be generated and added to the test suite. This can be done by breaking *FF-Embedded* and *FF-Context* into sub-machines, and by comparing each pair of the obtained sub-machines. Each time when two obtained sub-machines become distinguishable, their distinguishing test is added to the test suite. This enables the elimination of all sub-machines whose output responses to a distinguishing test are different from those observed by applying this test to the SUT. We break the machines by fixing some transitions as deterministic transitions, i.e. by reducing the non-determinism of Fault Functions. In the worst case, we can come up with an explicit enumeration of all faulty machines. However, as the considered examples show, we usually need to fix only a small number of transitions in order to draw a conclusion.

The details of the diagnostic method are presented in Section 7.5. Meanwhile, in the following section we present its constituent algorithms illustrated by a working example.

### 7.3.3 Working Example

As an example, we consider the two machines $M_1$ and $M_2$ shown in Fig.7.1. The set of external inputs is $X=\{x_1, x_2\}$, the set of external outputs is $Y=\{y_1, y_2, y_3\}$, the set of internal inputs is $U=\{u_1, u_2\}$, and the set of internal outputs is $Z = \{z_1, z_2\}$. Their corresponding reference system $RS = M_1 \Diamond M_2$ is shown in Fig. 7.2.

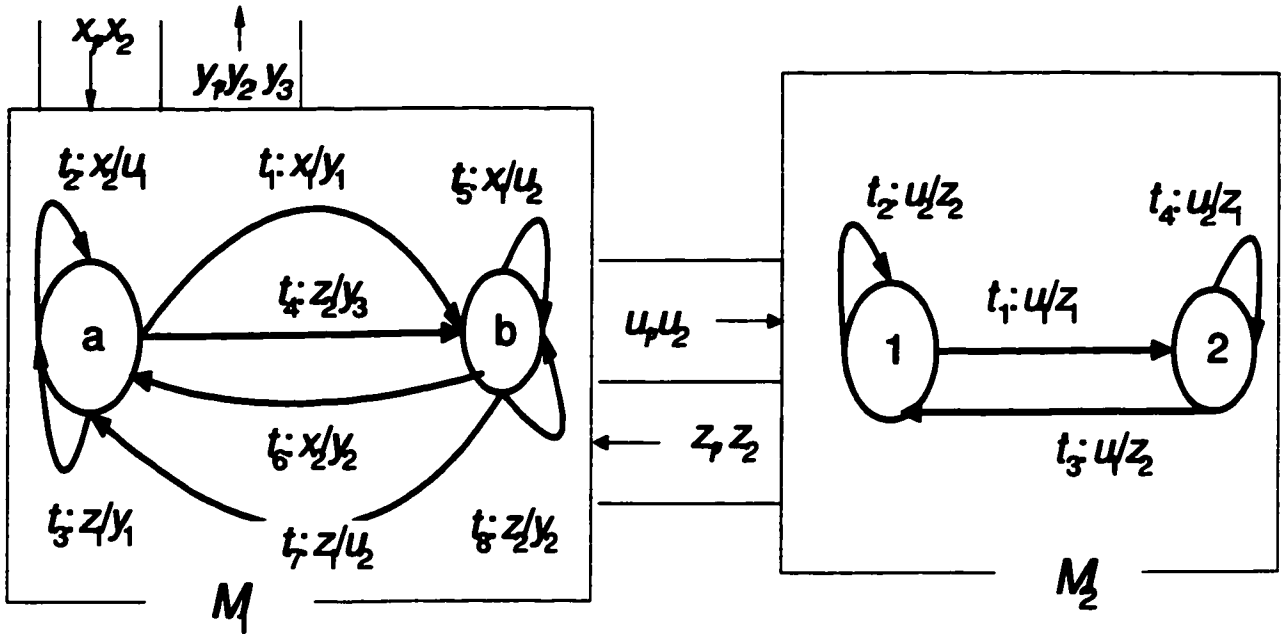


*Figure* 7.1. *A system of two ComFSMs* $M_1$ *and* $M_2$

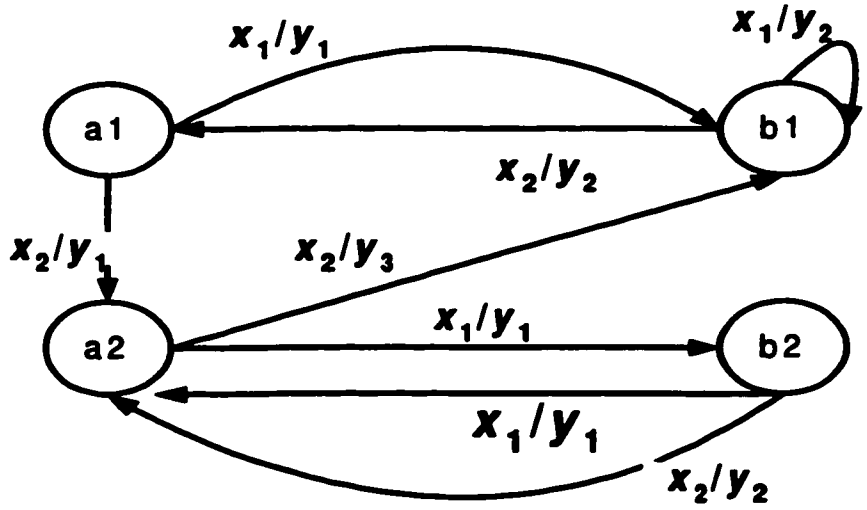


*Figure* 7.2. *Reference System* ($M_1 \Diamond M_2$) *of the* $M_1$ *and* $M_2$ *of Fig.* 7.1

A given complete test suite *TS* derived from *RS* is *TS* = {$x_1x_1x_1$, $x_1x_2x_2$, $x_2x_1x_1x_2$, $x_2x_1x_2x_2$, $x_2x_2x_1$}. *TS* is derived using the method presented in [Fuj91], and it detects any complete FSM $M'_1 \Diamond M'_2$ that is not equivalent to *RS*, under the assumption that FSMs $M'_1$ and $M'_2$ have up to two states. The set of expected output responses to the *TS* is as depicted in Table 7.1 below.

| *tc#* | *tc1* | *tc2* | *tc3* | *tc4* | *tc5* |
|---|---|---|---|---|---|
| Inputs | $r\,x1\,x1\,x1$ | $r\,x1\,x2\,x2$ | $r\,x2\,x1\,x1\,x2$ | $r\,x2\,x1\,x2\,x2$ | $r\,x2\,x2\,x1$ |
| Expected Outputs | $y1\ y2\ y2$ | $y1\ y2\ y1$ | $y1\ y1\ y1\ y3$ | $y1\ y1\ y2\ y3$ | $y1\ y3\ y2$ |

*Table 7.1. Test cases and their expected outputs*

Let us assume that the composition $M'_1 \Diamond M'_2$ of the implementation component FSMs $M'_1$ and $M'_2$ produces unexpected output responses as follows:

$y1y3y3$ to $x1x1x1$; $y1y1y3y2$ to $x2x1x1x2$ and $y1y3y3$ to $x2\,x2\,x1$ **(1)**

Thus, the composition $M'_1 \Diamond M'_2$ is not equivalent to *RS*, i.e. either $M'_1$ or $M'_2$ is a faulty component implementation.

Figures 7.3 and 7.4 include the *FF-Embedded* machine, $M_1 \Diamond (M_2)_a^{out}$, and the *FF-Context* $,(M_1)_a^{out} \Diamond M_2$, obtained as described in Section 7.3.2. In this example, the *FF-Context* is derived under an assumption that in the faulty context implementation, external outputs can only be replaced with external outputs and internal outputs can only be replaced with internal outputs, respectively. This is done in order to have a simple and more readable working example.

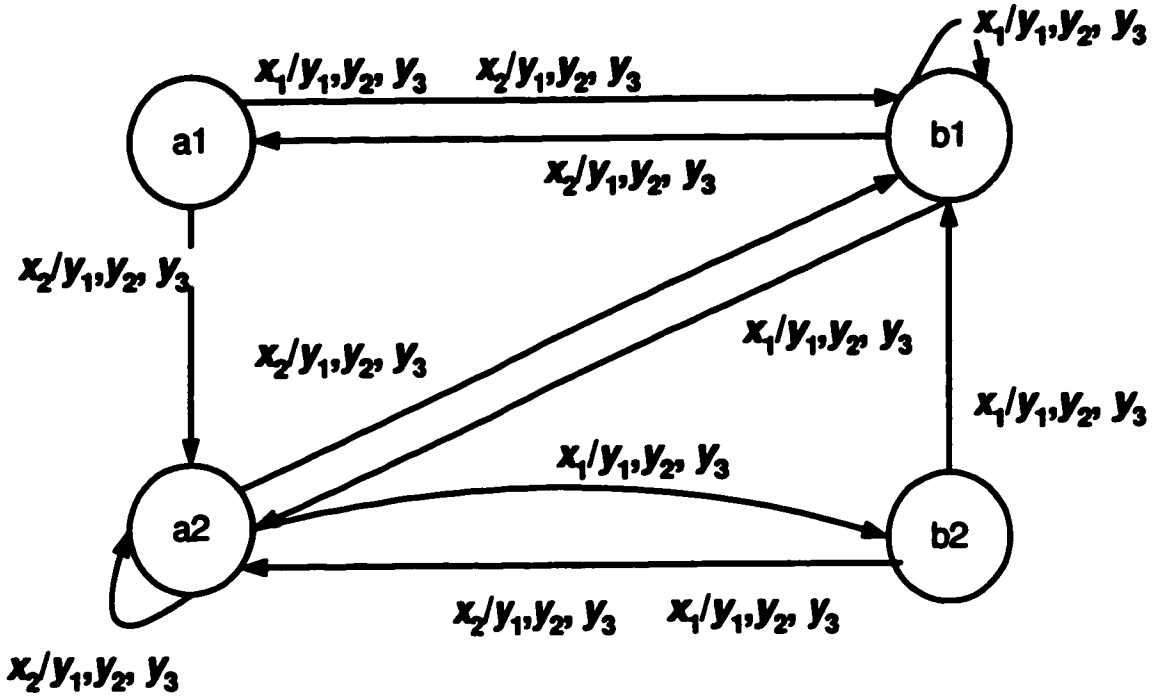


*Figure 7.3. Fault function of Embedded Component*

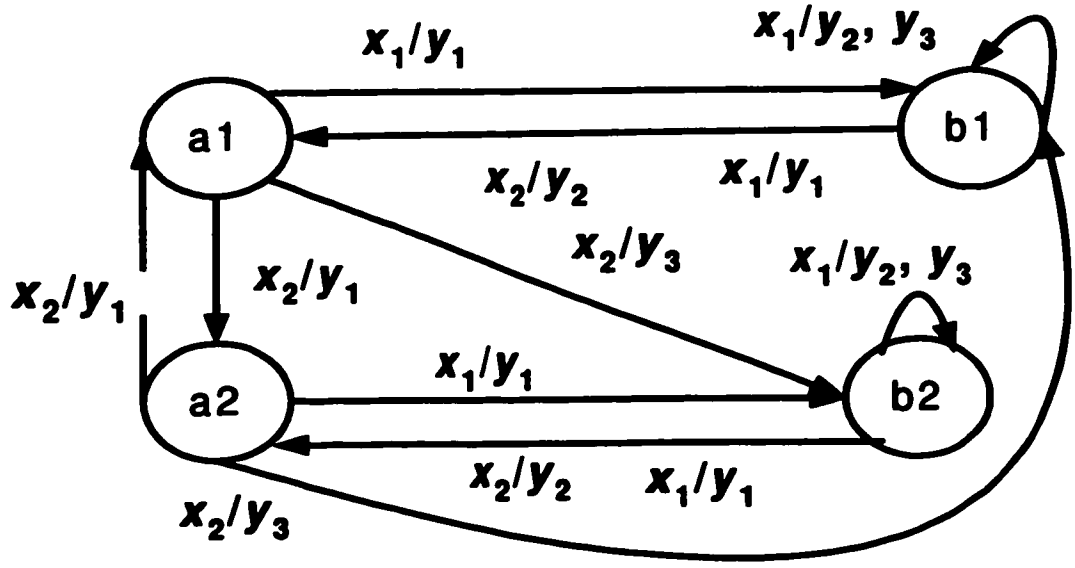


*Figure 7.4. Fault Function of Context*

## 7.4 Distinguishing Non-deterministic FSMs

In this section, we present the different algorithms of the test suite derivation method, given in the subsequent section, for identifying a faulty component FSM.

### 7.4.1 Removing Sub-machines of a Non-deterministic FSM

The following algorithm is used to remove from $(M_1)_a^{out} \Diamond M_2$ (and from $M_1 \Diamond (M_2)_a^{out}$) sub-machines whose output responses to the test cases of *TS* disagree with those obtained by applying these test cases to the SUT.

Given a complete non-deterministic FSM $A = (S,I,O,h,s_0)$ and a set $V$ of deterministic sequences over alphabet $(IO)^*$, the algorithm returns a smallest sub-machine $A^r$ of $A$ which has the property that each sub-machine of $A$ that comprises $V$ as a subset of its traces is a sub-machine of $A^r$. We note that in our case, the input parts of the sequences of the initial set $V$ are those of the given test suite *TS*, and the output parts are those observed by applying *TS* to the SUT.

**Algorithm 7.4.1.** Removing from FSM $A$ sub-machines that do not match $V$

**Input:** A complete non-deterministic FSM $A = (S, I, O, h, s_0)$ and a set $V$ of deterministic sequences over alphabet $(IO)^*$

***Output*:** The smallest subFSM $A^r = (S, I, O, h, s_0)$ of A that contains each sub-machine of $A$, that includes $V$ in the set of its traces, if such a submachine exist.

**Step-1.** Given FSM $A_1 = A$ and the set $V$ of sequences over alphabet $(IO)^*$, we derive the tree $Tree_1$ of all deterministic paths through $A_1$ labeled with sequences of $V$. Assign $i$:=1 and go-to Step-2.

**Step-2.** If there exists a sequence in $V$ such that no path in $Tree_i$ is labeled with this sequence, then there is no sub-machine in $A$ such that $V$ is a subset of the set of traces of that sub-machine ($A^r$ does not exist, end of Algorithm 7.4.1). Otherwise, we build a machine $A_{i+1}$ which is a sub-machine of $A_i$ as follows. For each observably reachable node in $Tree_i$, we copy into the corresponding state in $A_{i+1}$ the outgoing transitions from this node. If there are several observably reachable nodes in $Tree_i$ with the same label, we copy for the corresponding state in $A_{i+1}$ only the matching transitions at all of these nodes (same input/output/next-state values). If there are no matching transitions in $Tree_i$, then there is no sub-machine in $A$ such that $V$ is a subset of the set of traces of that sub-machine ($A^r$ does not exist, end of Algorithm 7.4.1). For each state in $A_i$, that labels only non-observably reachable nodes in $Tree_i$, we copy all the outgoing transitions from machine $A_i$ into $A_{i+1}$. If no transition in the machine $A_i$ has been changed (i.e. $A_{i+1}=A_i$), then we have $A^r = A_i$ (End of Algorithm 7.4.1). Otherwise, go-to Step-3.

**Step-3.** At this step we trim $Tree_i$ in order to obtain $Tree_{i+1}$ using the machine $A_{i+1}$ obtained by Step-2 above. For each path in $Tree_i$ that has a node where the output and/or next node of the transition do not match machine $A_{i+1}$, remove this transition and its sub-tree. If all outgoing transitions from some node for an appropriate input have been removed, we remove the incoming transition to this node. If all transitions from the root node for an appropriate input have been removed, then there is no sub-machine in $A$ such that $V$ is a subset of the set of traces of that submachine ($A^r$ does not exist, end of Algorithm 7.4.1). If the trees $Tree_i$ and $Tree_{i+1}$ coincide, then the machine $A^r = A_{i+1}$ is derived (End of Algorithm 7.4.1). Otherwise, increment $i$ by 1 and go back to Step-2.

As an example, we consider $Tree_1$ in Fig. 7.5 generated for the fault function of the context, i.e. $A_1 = (M_1)_a^{out} \Diamond M_2$ in Fig. 7.4, using Step-1 and *TS*. We do not include in $Tree_1$ the paths that do not match the observed output of *TS*. For example, for all test cases in the *TS* that start with the input $x_2$, the observed output for $x_2$ is $y_1$. Therefore, the paths of $(M_1)_a^{out} \Diamond M_2$ that start from the initial state $a_1$ by a transition labeled with a label

other than $x_2/y_1$ are not included in $Tree_1$. Moreover, we do not include in $Tree_1$ any non-deterministic path. For example, we do not include the path $a_1 \xrightarrow{x_1/y_1} b_1 \xrightarrow{x_1/y_3} b_1 \xrightarrow{x_1/y_3} a_1$.

In $Tree_1$, the root node $a_1$ is observably reachable through the empty sequence. Therefore, in Step-2, we copy the outgoing transitions of that node into $A_2$ (shown in Fig. 7.6), i.e. transitions, $a_1 \xrightarrow{x_1/y_1} b_1$, $a_1 \xrightarrow{x_2/y_1} a_2$ and $a_1 \xrightarrow{x_2/y_1} b_1$. Moreover, in $Tree_1$, starting from the root node $a_1$, the node $b_1$ is observably reachable through the sequence $x_1/y_1$. Therefore, in Step-2, we copy the outgoing transitions of $b_1$ from $Tree_1$ into $A_2$. Nodes $a_2$ and $b_2$ in $Tree_1$ are only non-observably reachable. Therefore, we copy from $A_1$ in Fig. 7.4 the outgoing transitions from states $a_2$ and $b_2$ into $A_2$.

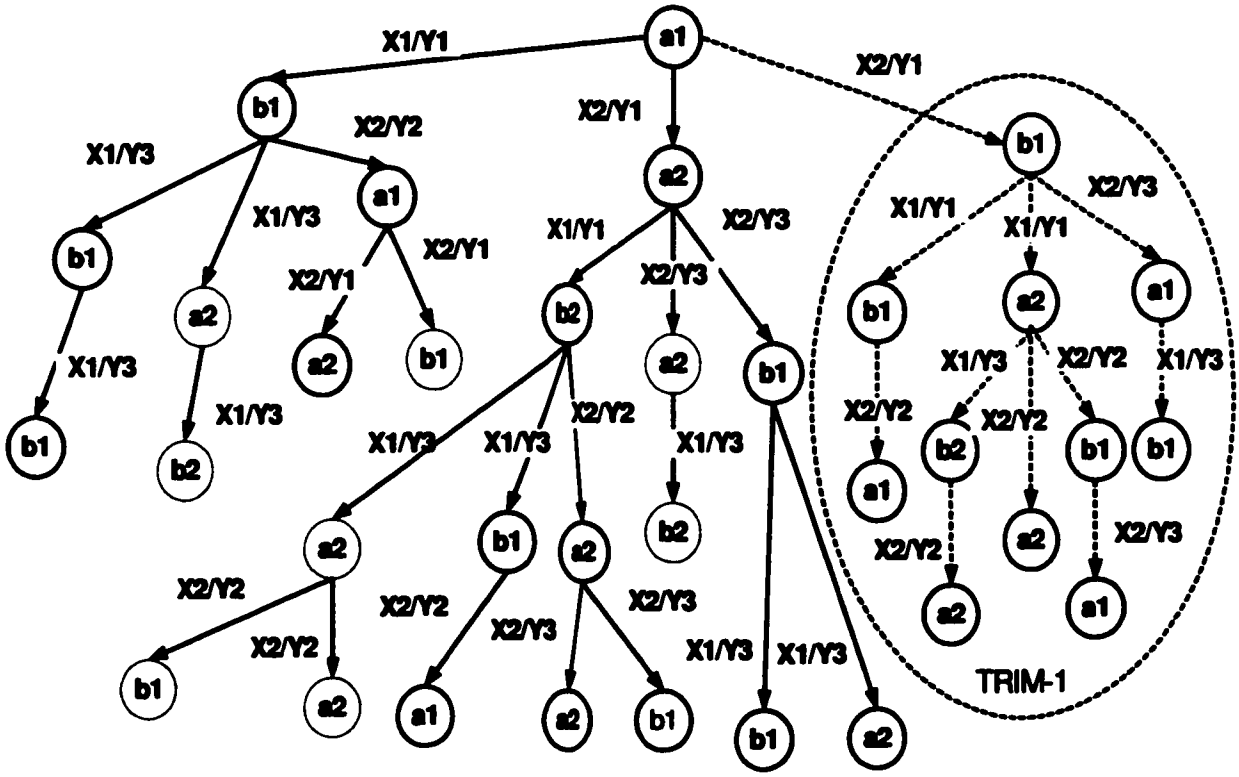


*Figure* 7.5. $Tree_1$ obtained by removing non-deterministic paths from $(M_1)_a^{out} \Diamond M_2$ of Fig.7.4

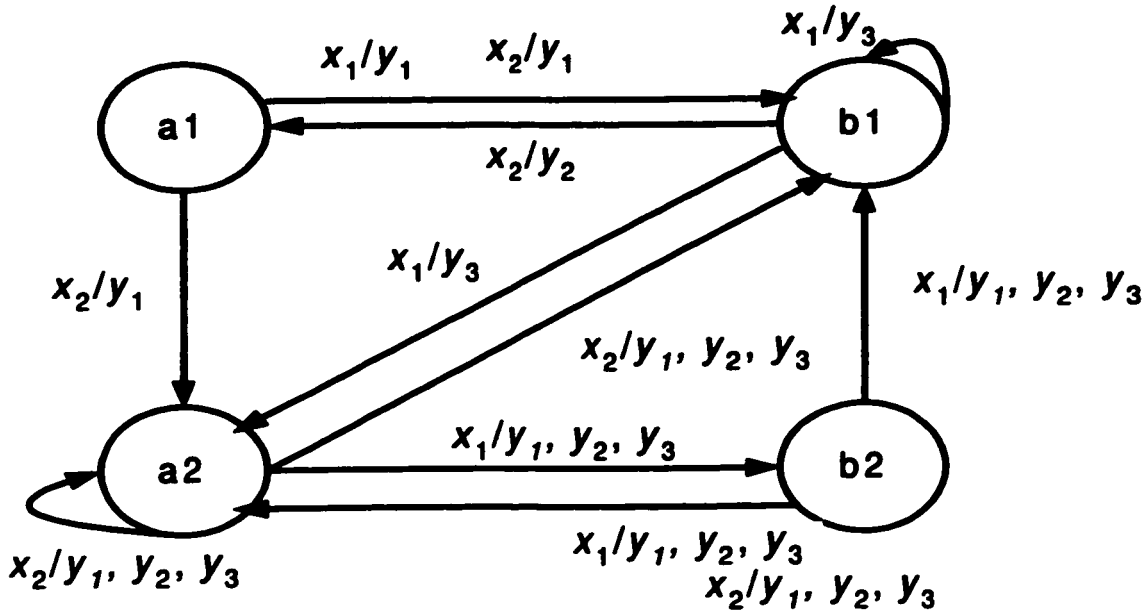


*Figure* 7.6. *Machine* $A_2$ *that correspond to* $Tree_1$ *depicted in Fig.* 7.5

Afterwards, using $A_2$ in Step-3, we consider in $Tree_1$, starting from the root node $a_1$, the outgoing transitions from the node $b_1$ that is reached through the sequence $x_2/y_1$. We notice that all its outgoing transitions do not match $A_2$. Therefore we trim the sub-tree of this node, and since all outgoing transitions for inputs $x_1$ and $x_2$ from this node are removed, we remove its incoming edge, and we get $Tree_2$ shown in Fig. 7.7, which is equal to $Tree_1$ except that the shaded area TRIM-1 is removed.

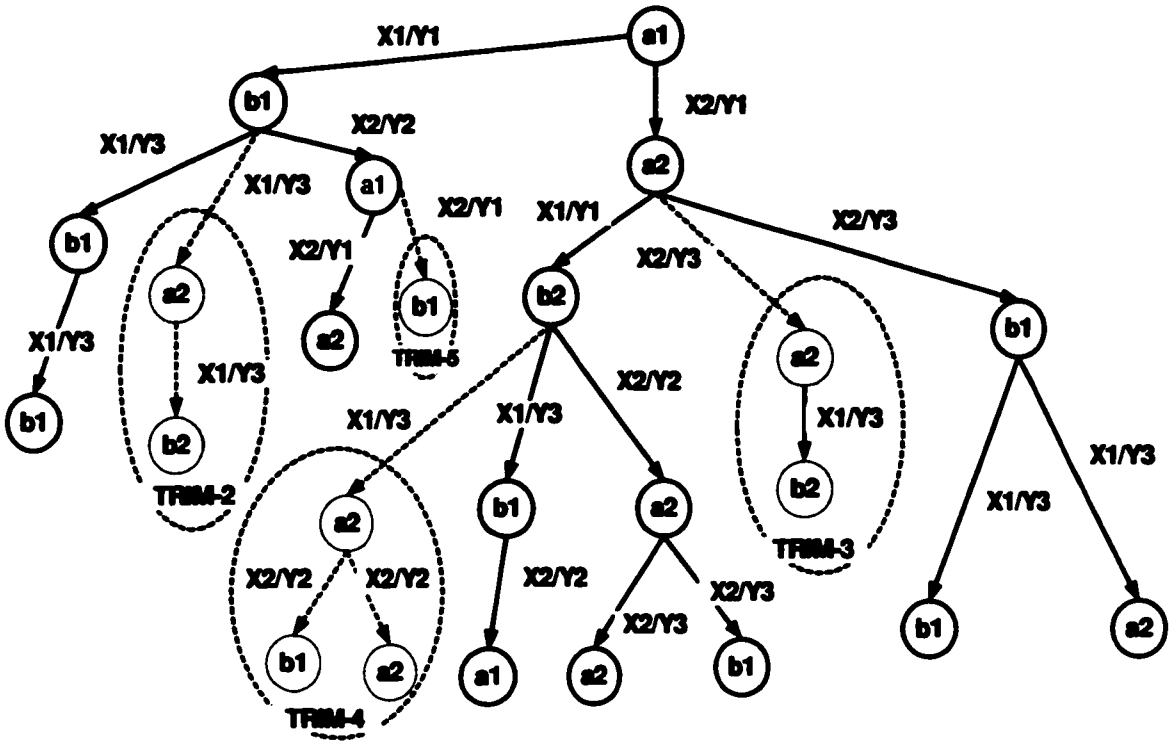


*Figure 7.7. Tree*$_2$

Back to Step-2, by considering *Tree*$_2$, the root node $a_1$ is observably reachable through the empty sequence. Moreover, starting from the root node $a_1$, the ending nodes $a_1$, $b_1$, and $a_2$ are observably reachable through the sequences $x_1/y_1x_2/y_2$, $x_1/y_1$, and $x_2/y_1$, respectively, and node $b_2$ is observably reachable through the sequence $x_2/y_1x_1/y_1$. Therefore, for these nodes, we copy their matching outgoing transitions from *Tree*$_2$ into $A_3$ in Fig.7.8.

Using the same reasoning we trim from *Tree*$_2$ the shaded areas of TRIM-3, TRIM-4, and TRIM-5 and we obtain *Tree*$_3$ depicted in Figure 7.9.

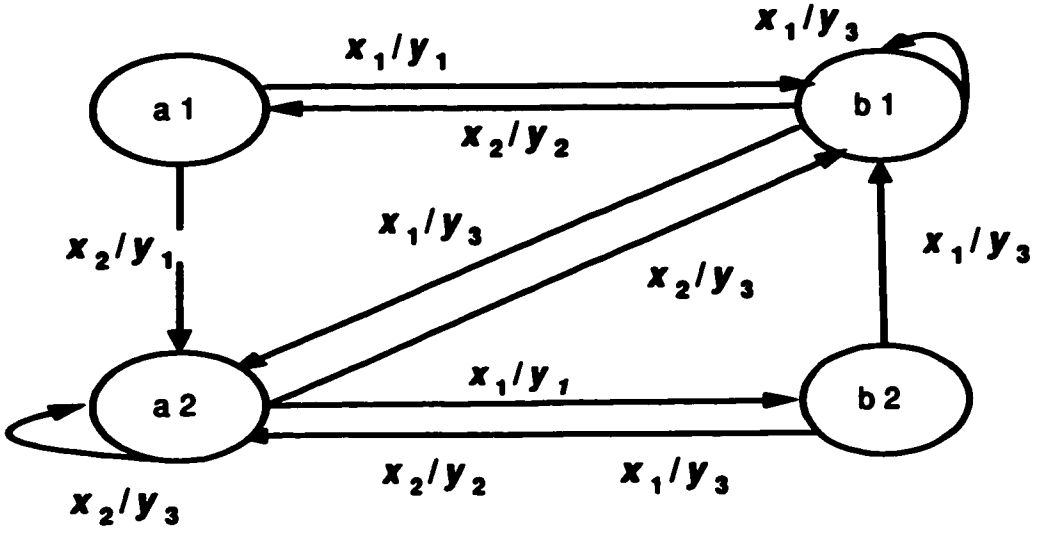


*Figure 7.8. Machine A3 that correspond to Tree2*

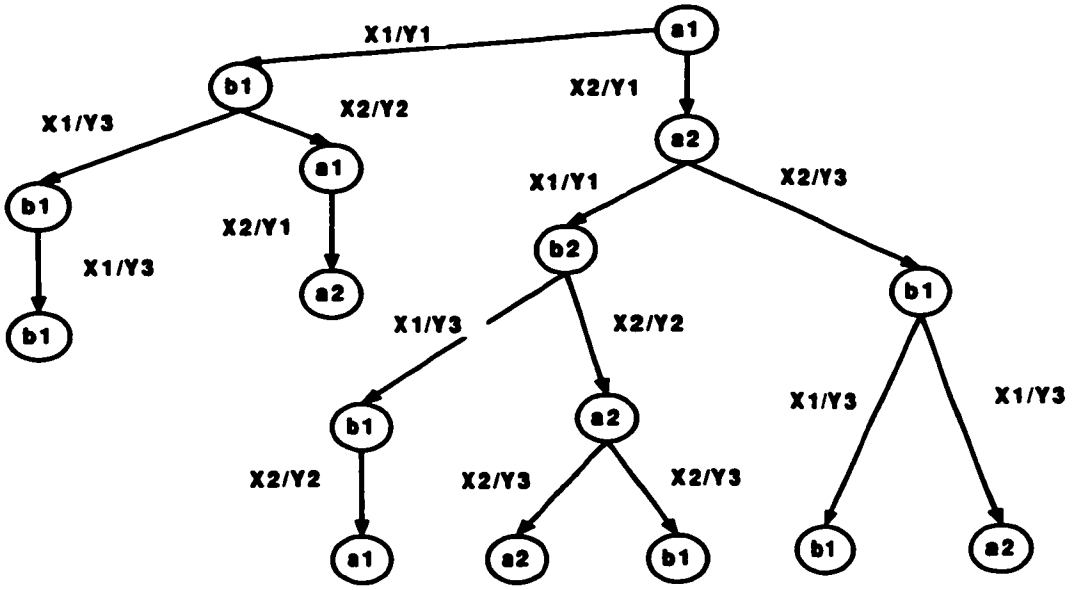


*Figure 7.9. Tree3*

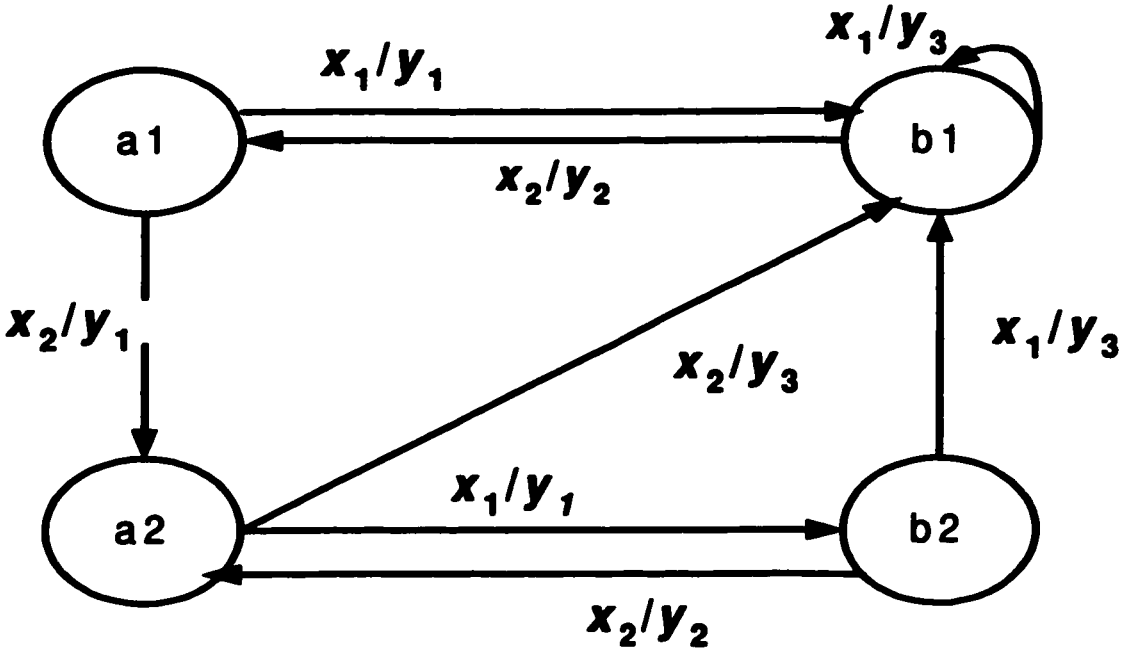


Figure 7.10. *Machine A4 that correspond to Tree3*

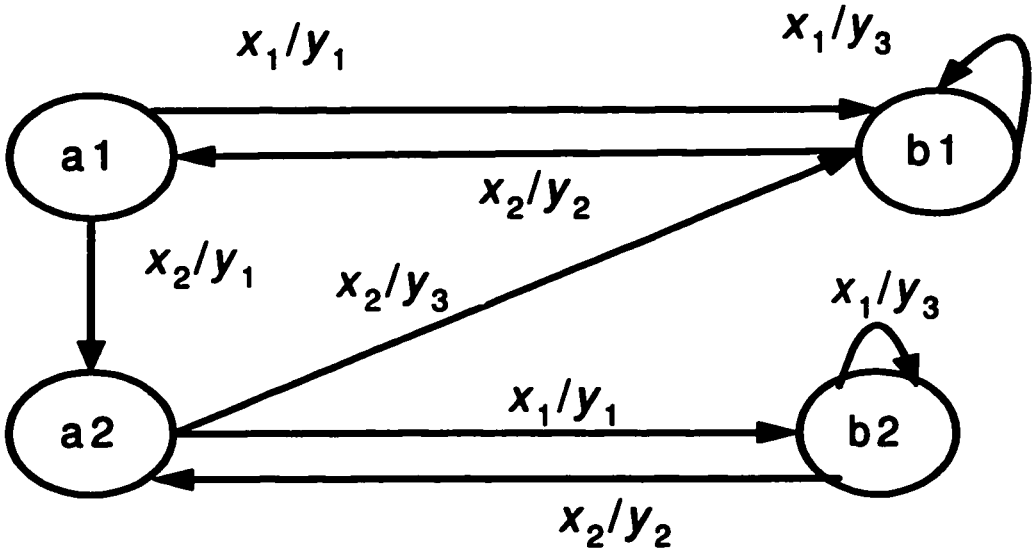


*Figure 7.11 Machine for the embedded component*

Back to Step-2, by considering *Tree3*, we notice that all nodes are observably reachable. We copy all the matching outgoing transitions of these nodes and obtain the final machine *A4* shown in Fig. 7.10.

We apply Algorithm 7.4.1. to the *FF-Embedded*, i.e. for $M_1 \Diamond (M_2)_a^{out}$ of Fig. 7.3, and we obtain the machine shown in Fig. 7.11.

**Theorem 7.1.** Given a complete non-deterministic FSM $A = (S, I, O, h, s_0)$ and a set $V$ of deterministic sequences over alphabet $(IO)^*$, if there exist a sub-machine $B$ of $A$ that has $V$ as a subset of its traces, then the Algorithm 7.4.1 derives an FSM $A^r$ which includes $B$ as submachine. Otherwise, the algorithm finds that there is no such submachine.

**Proof of Theorem 7.1:** In Step-2 of Algorithm 7.4.1, if there exists no deterministic path in $Tree_i$ labeled with some sequence $\alpha/\beta \in V$, then there is no deterministic sub-machine of $A$ that can produce the observed output sequence $\beta$ to the input sequence $\alpha$, i.e. there is no sub-machine $A^r$ in $A$ such that $V$ is a subset of traces of $A^r$. Moreover, In Step-3, if all the transitions from the root node in $Tree_i$ have been removed, then there is no deterministic sub-machine $A^r$ in $A$ such that $V$ is subset of traces of $A^r$. The transitions of $A$ are changed only during Step-2 for the states that label observably reachable nodes of the corresponding tree. For this reason, it is enough to show that each sub-machine $B$ of $A_i$ that has $V$ as a subset of its traces, is a sub-machine of $A_{i+1}$. Let state $s$ label an observably reachable node of $Tree_i$ that is reachable through some sequence $\alpha/\beta \in V$. Any sub-machine of $A_i$ that has $V$ as a subset of its traces must reach state $s$ after applying the sequence $\alpha$. Therefore, all transitions from the state $s$ of $B$ match transitions from this node, i.e. $B$ is a sub-machine of $A_{i+1}$.

### 7.4.2 Distinguishing the Sets of Deterministic Sub-machines of Two Non-deterministic FSMs

If the two sub-machines of the FSMs *FF-Context* and *FF-Embedded* obtained after using the above procedure for removing behaviors that do not match observed outputs, do not have equivalent sub-machines then we can derive a distinguishing set $Dis_{Set}$ that allows us to recognize which of the component FSMs is faulty. This set can be constructed using the algorithm given in [Kou00].

Given two completely specified machines $A = (S,I,O,h,s_1)$ and $B = (T,I,O,g,t_1)$, we derive a so-called *s-product* $A \times_s B = (S \times T, I, O, H, D_{A \times B}, s_1 t_1)$, where the defined specification domain $D_{A \times B} \subseteq (S \times T \times I)$ is the set of triplets of states and defined inputs [Kou00].

Given the triplet $(sti) \in S \times T \times I$, if $h^o(s,i) \cap g^o(t,i) = \emptyset$ then $(sti) \notin D_{A \times B}$, i.e. a behavior of the *s*-product $A \times_s B$ is *undefined* at state $st$ under input $i$. Otherwise, for each $o \in h^o(s,i) \cap g^o(t,i)$, $H(st,i)$ has each pair $(s't',o)$, $(s',o) \in h(s,i)$, $(t',o) \in g(t,i)$.

State $st$ of the machine $A \times_s B$ is called 1-*undefined* if there is an undefined transition at the state for some $i \in I$, i.e. there exists an input $i \in I$ such that $(sti) \notin D_{A \times B}$, and thus, states $s$ and $t$ can be distinguished by such an $i$. The set $\{i\}$ is called a *distinguishing set* of the 1-undefined state $st$. State $st$ is said to be *k-undefined*, $k>1$, if there exists $i \in I$ such that each state of the set $h^s(st,i)$ is $l$-undefined for $l<k$. A distinguishing set of the $k$-undefined state $st$ is obtained concatenating the distinguishing set of each state of the set $h^s(st,i)$ with input $i$.

**Theorem 7.2.**[Kou00] (1) If there exists $k \geq 1$ such that the initial state of the $A \times_s B$ is $k$-undefined then any deterministic sub-machine of $A$ is distinguishable from that of $B$. (2) Given distinguishing set $Dis_{Set}$ of the $k$-undefined initial state of $A \times_s B$ and machines $P \in Sub(A)$ and $R \in Sub(B)$, the machines $P$ and $R$ have different output responses to the sequences in the set $Dis_{Set}$.

In other words, let machines $M_1 \Diamond (M_2)_a^{out}$ and $(M_1)_a^{out} \Diamond M_2$ be distinguished with the distinguishing set $Dis_{Set}$. Let also $P$ be a sub-machine of $M_1 \Diamond (M_2)_a^{out}$ or of $(M_1)_a^{out} \Diamond M_2$. Then by observing the output responses of $P$ to sequences in the set $Dis_{Set}$, we can always conclude whether $P \in Sub(M_1 \Diamond (M_2)_a^{out})$ or $P \in Sub((M_1)_a^{out} \Diamond M_2)$, i.e. the diagnostic problem is always solvable. Therefore, we derive, using the above algorithm, a distinguishing set $Dis_{Set}$ for the FSMs *FF-Context* and *FF-Embedded* in order to recognize a sub-machine that corresponds to the faulty SUT. As an example, we consider the tree shown in Figure 7.12, and the machines *FF-Context* (machine $A_4$ of Fig. 7.10) and *FF-Embedded* (Fig. 7.11) obtained after deleting sub-machines with traces that do not match observed output responses. The sequence $x_2x_1x_1x_2x_2$ distinguishes these machines. Therefore, if we apply the input $x_2$ after the input sequence $x_2x_1x_1x_2$ and the implementation at hand produces the output response $y_1$ to the tail input $x_2$, then we conclude that $M'_1$ is the faulty implementation. If the output $y_3$ is produced to the tail input $x_2$, then we conclude that $M'_2$ is faulty. If the implementation produces the output different from $y_3$ and $y_1$, then the implementation at hand has faults that cannot be captured by the assumed fault model.

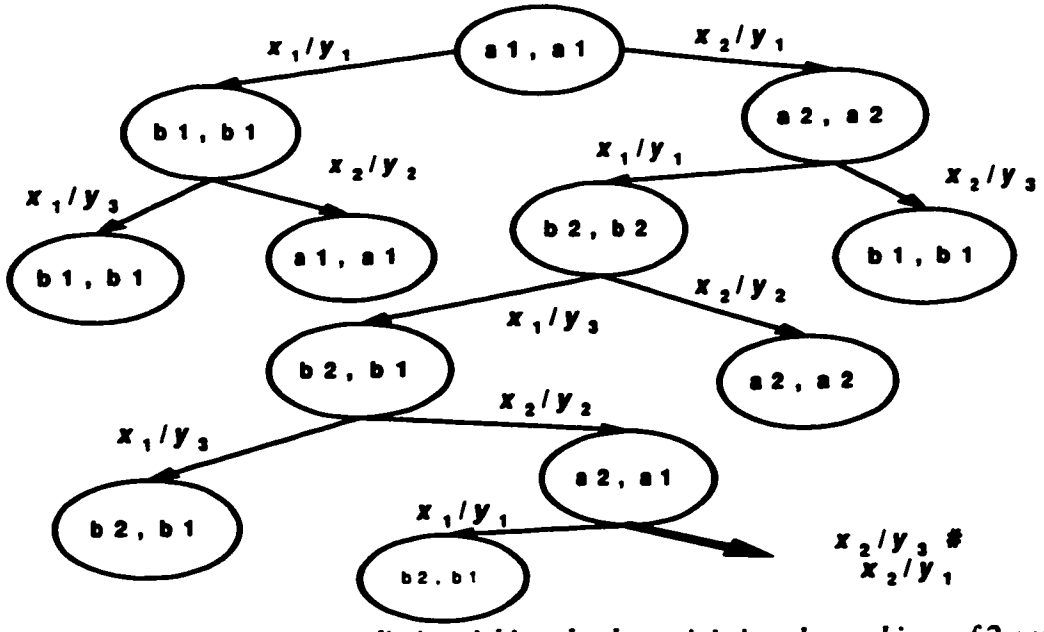


*Figure 7.12. A test sequence distinguishing the deterministic sub-machines of 2 non-deterministic FSMs*

### 7.4.3 Determining a Superfluous Sub-machine

Due to the considered fault model, the SUT $M$ is a sub-machine of $M_1 \Diamond (M_2)_a^{out}$ or $(M_1)_a^{out} \Diamond M_2$. However, we mentioned above that not each sub-machine of $M_1 \Diamond (M_2)_a^{out}$ (or $(M_1)_a^{out} \Diamond M_2$) can be obtained through output faults in the implementation of $M_2$ (or $M_1$). The reason is that we do not take into account deterministic and non-deterministic paths when combining the compact representation $(M_2)_a^{out}$ of all possible implementations of $M_2$ with $M_1$ (or the compact representation $(M_1)_a^{out}$ of all possible implementations of $M_1$ with $M_2$), and thus we may obtain superfluous sub-machines. Therefore, it may happen that the SUT $M$ is equivalent to a sub-machine of $M_1 \Diamond (M_2)_a^{out}$ and to a submachine of $(M_1)_a^{out} \Diamond M_2$, but there is no sub-machine $M'_2 \in (M_2)_a^{out}$ such that $M_1 \Diamond M'_2$ is equivalent to $M$. In this case, only the implementation of $M_1$ is faulty. Thus, we have the following problem.

Given FSM $M_1 \Diamond (M_2)_a^{out}$ and its sub-machine $M$, we must check whether there exists FSM $M'_2 \in (M_2)_a^{out}$ such that $M = M_1 \Diamond M'_2$. To solve the problem we can project sub-machine $M$ onto the set of states of $M_2$ and input and output alphabets of $M_2$. There exists FSM $M'_2 \in (M_2)_a^{out}$ such that $M = M_1 \Diamond M'_2$ if and only if the obtained FSM is deterministic. A sub-machine $M$ of $M_1 \Diamond (M_2)_a^{out}$ for which there is no $M'_2 \in (M_2)_a^{out}$ such that FSMs $M_1 \Diamond M'_2$ and $M$ are equivalent, is called a *superfluous* sub-machine.

## 7.5 Fault Diagnosis Algorithm

**Algorithm 7.5.1.** Recognizing a faulty component FSM

***Input***: Composition $M \cong M_1 \Diamond M_2$ of two FSMs $M_1$ and $M_2$, and the set $V = TS$ of sequences over alphabet $(IO)^*$,

***Output***: Verdict "*Component FSM* $M_1$ (or $M_2$) *is faulty*", or verdict "*Both* $M_1/M_2$ *could be faulty*" when there is a possible faulty implementation of $M_1$ and a possible faulty implementation of $M_2$ that cause the same observable behavior as the implementation at

hand, or verdict "*Faults cannot be captured by the assumed fault model*" if it has been determined that the implementation at hand has faults that can not be captured by the assumed fault model.

**Step-1.** Derive machines, $A_1 = (M_1)_a^{out} \Diamond M_2$ (Fault Function of $M_1$) and $A_2 = M_1 \Diamond (M_2)_a^{out}$ (Fault Function of $M_2$). Let the set $\mathfrak{R}_1$ be equal to $\{A_1\}$, and the set $\mathfrak{R}_2$ be equal to $\{A_2\}$.

**Step-2.** For each machine say $A_k$ in the sets $\mathfrak{R}_1$ and $\mathfrak{R}_2$, call Algorithm 7.4.1. to obtain the smallest sub-machine $A^r$ of $A_k$ which includes all sub-machines of $A_k$ that have $V$ as a subset of their traces. If such an $A^r$ exist, replace $A_k$ by $A^r$. Otherwise, remove $A_k$ from the corresponding set $\mathfrak{R}_1$ or $\mathfrak{R}_2$.

If the sets $\mathfrak{R}_1$ and $\mathfrak{R}_2$ are empty, then the implementation at hand has a fault that is not captured by the assumed fault model (End of diagnosis algorithm).

If the set $\mathfrak{R}_1$ (or $\mathfrak{R}_2$) is empty, then we conclude the other machine $M_2$ (or $M_1$) is faulty (End of diagnosis algorithm).

Otherwise, check, as described in [Kou00], whether there are two machines, say $A_i$ in $\mathfrak{R}_1$ and $A_j$ in $\mathfrak{R}_2$, that are distinguishable.

- If there exist such two machines, we obtain, using the algorithm given in [Kou00] and described in Section 7.4.2, the $Dist_{set}$ that distinguish them, and we apply the input sequences of this set to the SUT.
  - If $|\mathfrak{R}_1| = 1$ and $|\mathfrak{R}_2| = 1$, i.e. $\mathfrak{R}_1 = \{A_i\}$ and $\mathfrak{R}_2 = \{A_j\}$, then:
    - If the output responses of the SUT to the $Dist_{set}$ are different from those expected by both machines $A_i$ and $A_j$, then we conclude that the implementation at hand has faults that cannot be captured by the assumed fault model (End of diagnosis algorithm).
    - Else, if the output responses of the SUT are different than those expected by $A_j$ (or $A_i$), then we conclude that $M_1$ (or $M_2$) is the faulty machine (End of diagnosis Algorithm).

-- If $|\mathfrak{R}_1| > 1$ or $|\mathfrak{R}_2| > 1$, then after observing the output responses of the SUT to the sequences in $Dist_{set}$, we remove $A_i$ (or $A_j$) from the set $\mathfrak{R}_1$ (or $\mathfrak{R}_2$) if these responses are different than those expected by $A_i$ (or $A_j$). Then, we add the sequences in $Dist_{set}$ with the observed output responses to $V$, and we return return back to Step-2.

♦- If all the machines in $\mathfrak{R}_1$ are indistinguishable from those in $\mathfrak{R}_2$, then

--If $\mathfrak{R}_1$ and $\mathfrak{R}_2$ have only deterministic machines, then check whether all the machines in $\mathfrak{R}_1$ and in $\mathfrak{R}_2$ are superfluous as described above.

-If all the sub-machines in $\mathfrak{R}_1$ (or in $\mathfrak{R}_2$) are superfluous, then machine $M_2$ (or $M_1$) is faulty (End of diagnosis algorithm).

-If all the sub-machines in the sets $\mathfrak{R}_1$ and $\mathfrak{R}_2$ are superfluous, then the implementation at hand has a fault that can not be captured by the assumed fault model (End of diagnosis algorithm).

-Else, If there exist a sub-machine in $\mathfrak{R}_1$ and another in $\mathfrak{R}_2$ that are not superfluous, then we conclude that "Both_$M_1$/$M_2$ could_be_faulty". There is a possible faulty implementation of $M_1$ and a possible faulty implementation of $M_2$ that cause the same observable behavior as that of the implementation at hand

--Else, if the set $\mathfrak{R}_1$ (or $\mathfrak{R}_2$) has at least one non-deterministic FSM, then we break that machine into $k$ machines by fixing one of its non-deterministic transitions. Then, we replace that machine in the set $\mathfrak{R}_1$ (or $\mathfrak{R}_2$) with the obtained sub-machines, and we go back to Step-2.

### 7.5.1 Another Application Example

As another example of the diagnostic method, suppose that the SUT of implementation component FSMs $M'_1$ and $M'_2$ of specifications in Fig. 7.2 produces the unexpected output responses depicted in Table 7.2 below.

| *Tc#* | *tc1* | *tc2* | *tc3* | *tc4* | *tc5* |
|---|---|---|---|---|---|
| Inputs | *r x1 x1 x1* | *r x1 x2 x2* | *r x2 x1 x1 x2* | *r x2 x1 x2 x2* | *r x2 x2 x1* |
| Expected Outputs | *y1 y2 y2* | *y1 y2 y1* | *y1 y1 y1 y3* | *y1 y1 y2 y3* | *y1 y3 y2* |
| Observed Outputs | *y1* ***y1*** *y1* | *y1 y2 y1* | *y1 y1 y1 y3* | *y1 y1 y2 y3* | *y1 y3* ***y1*** |

*Table 7.2. Test cases and their outputs*

Thus, the composition $M'_1 \lozenge M'_2$ is not equivalent to $RS = M_1 \lozenge M_2$, i.e. either $M'_1$ or $M'_2$ is a faulty component implementation. In order to decide whether it is possible to identify the faulty machine in the system, in Step-1 of Algorithm 7.5.1 the *FF-Context* and the *FF-Embedded* are derived, i.e. machines $A_1 = (M_1)_a^{out} \lozenge M_2$ and $A_2 = M_1 \lozenge (M_2)_a^{out}$. These machines are already shown in Figures 7.4 and 7.5, respectively. However, throughout this example, we use a transition table representation of these machines as shown in Tables 7.3 and 7.4 below. We note that an FSM can be given in the form of a transition table, and as shown in Table 7.3, states and input symbols are used to name the rows and columns, respectively. A state/output pair $b_1$/ $y_{1\text{-}3}$ appeared at the location of row $x_1$ and column $a_1$ implies that there are transitions $a_1$-$x_1$/$y_1$->$b_1$, $a_1$-$x_1$/$y_2$->$b_1$, and $a_1$-$x_1$/$y_3$->$b_1$.

Initially we have the sets $\mathfrak{R}_1 = \{A_1\}$ and $\mathfrak{R}_2 = \{A_2\}$.

| | $a_1$ | $a_2$ | $b_1$ | $b_2$ |
|---|---|---|---|---|
| $x_1$ | $b_1$ / $y_{1\text{-}3}$ | $b_2$ / $y_{1\text{-}3}$ | $a_2$ / $y_{1\text{-}3}$<br>$b_1$ / $y_{1\text{-}3}$ | $a_2$ / $y_{1\text{-}3}$<br>$b_1$ / $y_{1\text{-}3}$ |
| $x_2$ | $a_2$ / $y_{1\text{-}3}$<br>$b_1$ / $y_{1\text{-}3}$ | $a_2$ / $y_{1\text{-}3}$<br>$b_1$ / $y_{1\text{-}3}$ | $a_1$ / $y_{1\text{-}3}$ | $a_2$ / $y_{1\text{-}3}$ |

*Table 7.3. Fault function of the Context, $A_1 = (M_1)_a^{out} \lozenge M_2$*

| | $a_1$ | $a_2$ | $b_1$ | $b_2$ |
|---|---|---|---|---|
| $x_1$ | $b_1$ / $y_1$ | $b_2$ / $y_1$ | $b_1$ / $y_2$-3<br>$a_1$ / $y_1$ | $b_2$ / $y_2$-3<br>$a_2$ / $y_1$ |
| $x_2$ | $a_2$ / $y_1$<br>$b_2$ / $y_3$ | $a_1$ / $y_1$<br>$b_1$ / $y_3$ | $a_1$ / $y_2$ | $a_2$ / $y_2$ |

*Table 7.4. Fault Function of the Embedded Component,* $A_2 = M_1 \Diamond (M_2)_a^{out}$

During Step-2, the algorithm calls Algorithm 7.4.1 in order to remove from the machines in the sets $\mathfrak{N}_1 = \{A_1\}$ and $\mathfrak{N}_2 = \{A_2\}$ the sub-machines that do not match the set $V$. For example, consider $Tree_1$ in Fig. 7.13 generated for the fault function of the context, i.e $A_1$ in Table 7.3, using Step-1 of Algorithm 7.4.1 and $TS$. In $Tree_1$, the root node $a_1$ is observably reachable through the empty sequence. Therefore, in Step-2, for the root node $a_1$ we copy its outgoing transitions into $A_2$ (shown in Table 7.5), i.e. we copy transitions $a_1 \xrightarrow{x_1/y_1} b_1$, $a_1 \xrightarrow{x_2/y_1} a_2$ and $a_1 \xrightarrow{x_2/y_1} b_1$. Moreover, in $Tree_1$, starting from the root node $a_1$, the ending node $b_1$ of transition $a_1 \xrightarrow{x_1/y_1} b_1$ is observably reachable through the sequence $x_1/y_1$. Therefore, in Step-2, we copy the outgoing transitions of $b_1$ from $Tree_1$ into $A_2$, i.e. we transitions $b_1 \xrightarrow{x_1/y_2} a_2$, $b_1 \xrightarrow{x_1/y_2} b_1$, and $b_1 \xrightarrow{x_2/y_2} a_1$. Furthermore, nodes $a_2$ and $b_2$ in $Tree_1$ are only non-observably reachable. Therefore, we copy from $A_1$ in Table 7.3 the outgoing transitions from states $a_2$ and $b_2$ into $A_2$.

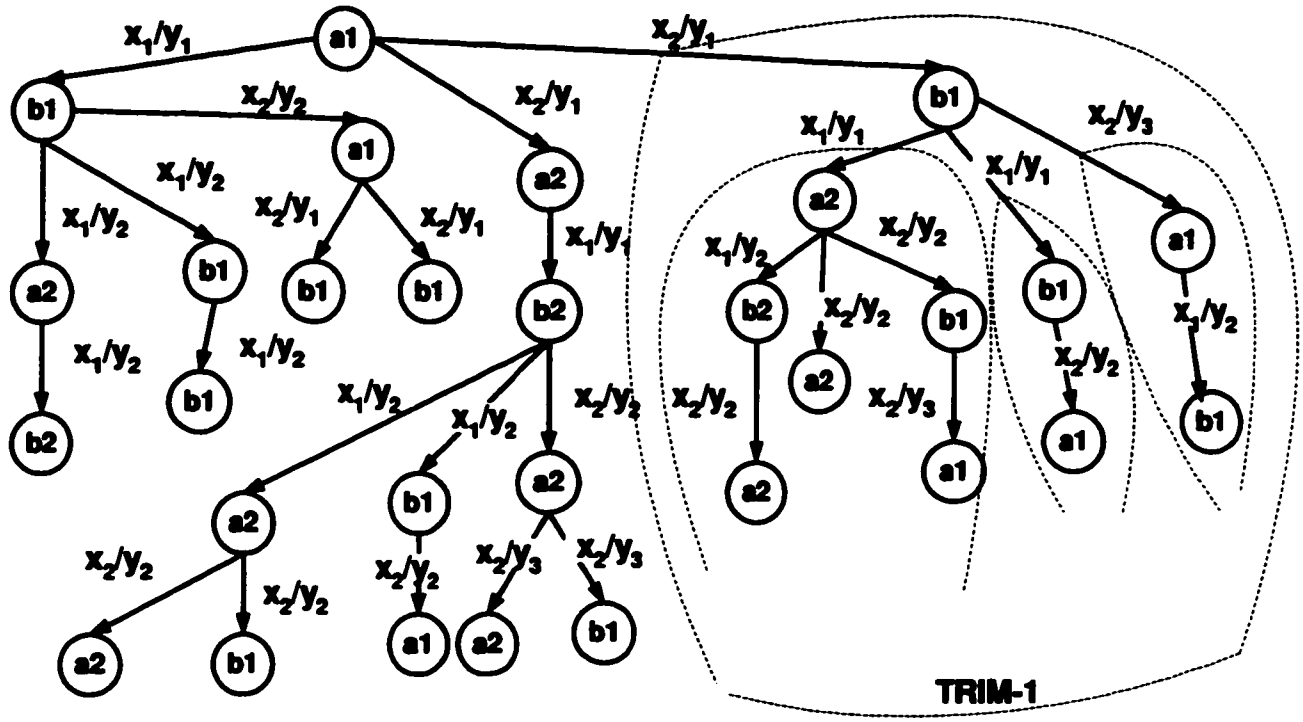


*Figure* 7.13. *Tree*1 *obtained by removing non-deterministic paths from* $B_1$ = $(M1)_a^{out} \Diamond M2$ *of Table* 7.3

| | $a_1$ | $a_2$ | $b_1$ | $b_2$ |
|---|---|---|---|---|
| $x1$ | $b1$ / $y1$ | $b2$ / $y1$-3 | $a2$ / $y2$<br>$b1$ / $y2$ | $a2$ / $y1$-3<br>$b1$ / $y1$-3 |
| $x2$ | $a2$ / $y1$<br>$b1$ / $y1$ | $a2$ / $y1$-3<br>$b1$ / $y1$-3 | $a1$ / $y2$ | $a2$ / $y1$-3 |

*Table* 7.5. *Machine* $A_2$

Afterwards, using $A_2$ in Step-3, we consider in *Tree*1, starting from the root node $a_1$, the outgoing transitions from the node $b_1$ that is reached by transition $a1 \xrightarrow{x_2/y_1} b_1$. We notice that all its outgoing transitions do not match $A_2$. Therefore we trim the subtree of this node, and since all outgoing transitions for inputs $x_1$ and $x_2$ from this node are removed, we remove its incoming edge, and we get *Tree*2 in Fig. 7.14, which is equal to *Tree*1 except that the shaded area TRIM-1 is removed.

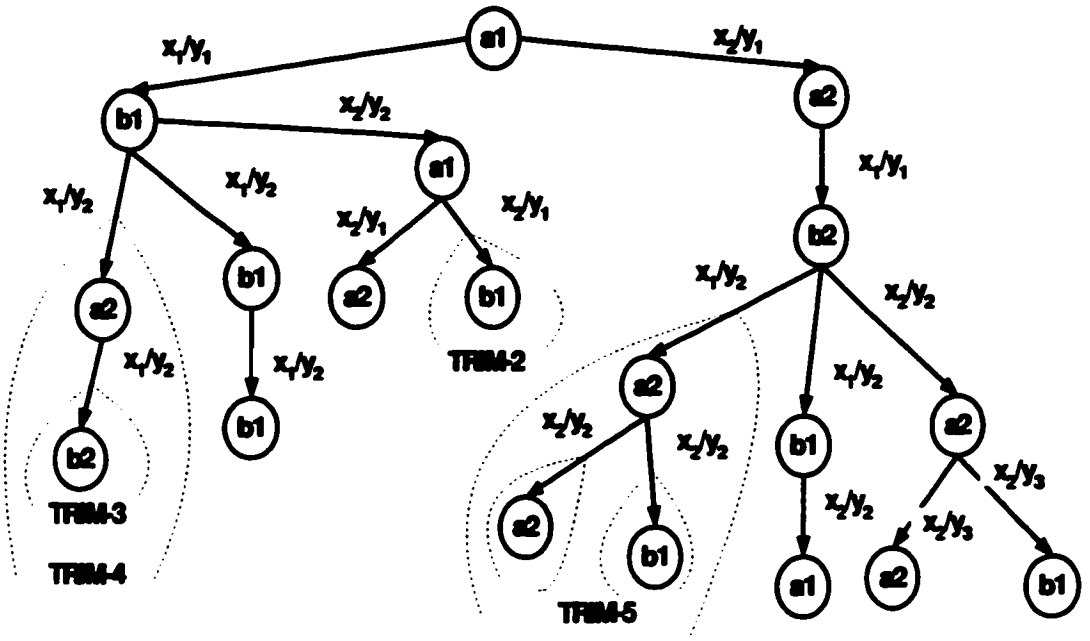


*Figure* 7.14. *Tree*2

Back to Step-2, by considering *Tree*2, we notice that the root node $a_1$ is observably reachable through the empty sequence. Moreover, starting from the root node $a_1$, the ending node $a_1$ of consecutive transitions $a_1 \xrightarrow{x_1/y_1} b_1 \xrightarrow{x_2/y_2} a_1$ is also observably reachable through the sequences $x_1/y_1x_2/y_2$. Therefore, for these nodes, we copy their matching outgoing transitions from *Tree*2 into $A_3$ in Table 7.6, i.e. we copy transitions $a_1 \xrightarrow{x_2/y_1} a_2$ and $a_1 \xrightarrow{x_1/y_1} b_1$. We also notice that starting from the root node $a_1$ the ending nodes labeled with $a_2$ of consecutive transitions $a_1 \xrightarrow{x_2/y_1} a_2 \xrightarrow{x_1/y_1} b_2 \xrightarrow{x_2/y_2} a_2$ and the transition $a_1 \xrightarrow{x_2/y_1} a_2$ are observably reachable through the sequences $x_2/y_1x_1/y_1x_2/y_2$ and $x_2/y_1$. Therefore, for these nodes, we copy their matching outgoing transitions into $A_3$, i.e. we copy transitions $a_2 \xrightarrow{x_2/y_3} a_2$, $a_2 \xrightarrow{x_2/y_3} b_1$, and $a_2 \xrightarrow{x_1/y_1} b_2$. Finally, starting from the root node $a_1$, node $b_1$ is observably reachable through the sequence $x_1/y_1$. Therefore, for that node we copy its outgoing transitions into $A_3$, i.e. we copy transitions $b_1 \xrightarrow{x_2/y_2} a_1$,

$b_1 \xrightarrow{x_1/y_2} b_1$, and $b_1 \xrightarrow{x_1/y_2} a_2$. Finally, starting from the root node $a_1$, node $b_2$ of the consecutive transitions $a_1 \xrightarrow{x_2/y_1} a_2 \xrightarrow{x_1/y_1} b_2$ is also observably reachable through the sequence $x_2/y_1x_1/y_1$. Therefore, we also copy all its outgoing transitions into $A_3$, i.e. we copy transitions $b_2 \xrightarrow{x_1/y_2} a_2$, $b_2 \xrightarrow{x_1/y_2} b_1$ and $b_2 \xrightarrow{x_2/y_2} a_2$.

| | $a_1$ | $a_2$ | $b_1$ | $b_2$ |
|---|---|---|---|---|
| $x_1$ | $b_1$ / $y_1$ | $b_2$ / $y_1$ | $a_2$ / $y_2$<br>$b_1$ / $y_2$ | $a_2$ / $y_2$<br>$b_1$ / $y_2$ |
| $x_2$ | $a_2$ / $y_1$ | $a_2$ / $y_3$<br>$b_1$ / $y_3$ | $a_1$ / $y_2$ | $a_2$ / $y_2$ |

*Table 7.6. Machine $A_3$*

Back to Step-3, using $A_3$ in Step-3, we consider in *Tree*$_2$, the outgoing transition from the node $a_1$ of the shaded area TRIM-2. We notice that this transition does not match $A_3$. Therefore, we trim the sub-tree of this node, and since all outgoing transitions for inputs $x_1$ and $x_2$ from this node are removed, we remove its incoming edge. Using the same reasoning we trim from *Tree*$_2$ the shaded areas of TRIM-3, TRIM-4, and TRIM-5 and we obtain *Tree*$_3$ in Fig. 7.15.

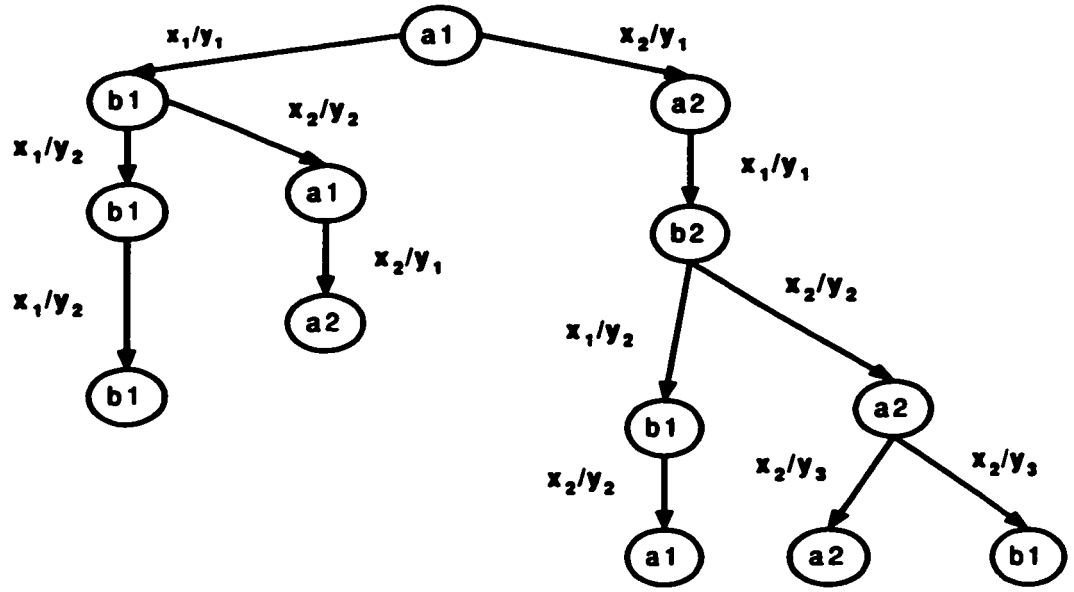


*Figure* 7.15. *Tree*3

Back to Step-2, by considering *Tree*3, starting from the root node $a_1$ is observably reachable through the empty sequence, and the ending node labeled with $a_1$ of consecutive transitions $a_1 \xrightarrow{x_1/y_1} b_1 \xrightarrow{x_2/y_2} a_1$ is also observably reachable through the sequence $x_1/y_1x_2/y_2$. Therefore, for these nodes, we copy their matching outgoing transitions into $A_4$, i,e we copy transitions $a_1 \xrightarrow{x_1/y_1} b_1$ and $a_1 \xrightarrow{x_2/y_1} a_2$. Moreover, starting from the root node $a_1$, the ending nodes labeled with $b_1$ of transitions $a_1 \xrightarrow{x_1/y_1} b_1$,$a_1 \xrightarrow{x_1/y_1} b_1 \xrightarrow{x_1/y_2} b_1$, and $a_1 \xrightarrow{x_2/y_1} a_2 \xrightarrow{x_1/y_1} b_2 \xrightarrow{x_1/y_2} b_1$ are observably reachable through sequences $x_1/y_1$, $x_1/y_1x_1/y_2$, and $x_2/y_1x_1/y_1x_1/y_2$, respectively. Therefore, for these nodes, we copy their outgoing matching transitions into $A_4$, i.e. we copy transitions $b_1 \xrightarrow{x_1/y_2} b_1$ and $b_1 \xrightarrow{x_2/y_2} a_1$. Furthermore, starting from the root node $a_1$, we notice that the ending node $a_2$ of consecutive transitions $a_1 \xrightarrow{x_2/y_1} a_2 \xrightarrow{x_1/y_1} b_2 \xrightarrow{x_2/y_2} a_2$ and the transition $a_1 \xrightarrow{x_2/y_1} a_2$ are observably reachable through the sequences $x_2/y_1x_1/y_1x_2/y_2$ and $x_2/y_1$. Therefore, for these nodes, we copy their matching outgoing transitions into $A_3$, i.e. we copy transitions

$a_2 \xrightarrow{x_2/y_3} a_2$, $a_2 \xrightarrow{x_2/y_3} b_1$, and $a_2 \xrightarrow{x_1/y_1} b_2$. Finally, starting from the root node $a_1$, we notice that the ending node $b_2$ of consecutive transitions $a_1 \xrightarrow{x_2/y_1} a_2 \xrightarrow{x_1/y_1} b_2$ is observably reachable. Therefore, we copy all its outgoing transitions into $A_4$ in Table 7.7, i.e. we copy transitions $b_2 \xrightarrow{x_1/y_2} b_1$ and $b_2 \xrightarrow{x_2/y_2} a_2$.

| | $a_1$ | $a_2$ | $b_1$ | $b_2$ |
|---|---|---|---|---|
| $x_1$ | $b_1$ / $y_1$ | $b_2$ / $y_1$ | $b_1$ / $y_2$ | $b_1$ / $y_2$ |
| $x_2$ | $a_2$ / $y_1$ | $a_2$ / $y_3$<br>$b_1$ / $y_3$ | $a_1$ / $y_2$ | $a_2$ / $y_2$ |

*Table* 7.7. *Machine* $A_4 = A^r_1$

Back to Step-3, we notice that each node in a path in *Tree*3 has its output and next node match machine $A_4$. Hence, Algorithm 7.4.1 terminates and returns $A^r_1 = A_4$ of Table 7.7 which replaces $A_1$ in $\mathfrak{R}_1$. i.e. $\mathfrak{R}_1$ becomes $\{A^r_1\}$.

Now, we apply Algorithm 7.4.1 for the *FF-Embedded*, i.e. for $A_2$ in Table 7.4. Using Step-1 and *TS*, we obtain *Tree*1 in Fig. 7.16. All nodes of *Tree*1 are observably reachable. Therefore, in Step-2 of Algorithm 7.4.1, we copy their outgoing transitions into $A_1$ in Table 7.8. Back to Step-3, we notice that each node in a path in *Tree*1 has its output and next node match machine $A_1$. Hence, Algorithm 7.4.1 terminates and returns $A^r_2 = A_4$ of Table 7.7 which replaces $A_2$ in $\mathfrak{R}_2$. i.e. $\mathfrak{R}_2$ becomes $\{A^r_2\}$.

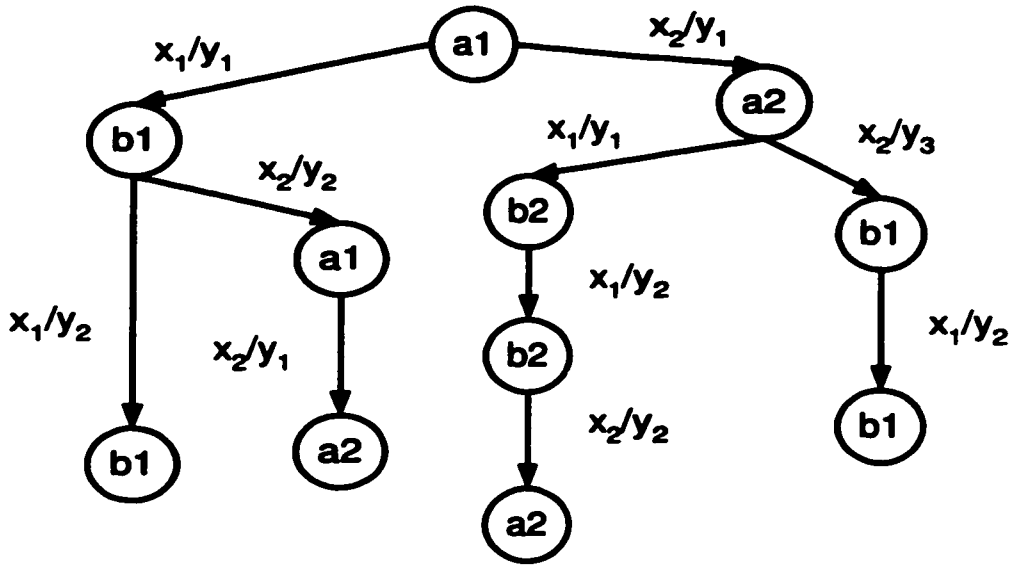


*Figure* 7.16. *Tree*1 *obtained by removing non-deterministic paths from* $M_1 \Diamond (M_2)_a^{out}$ *of Table* 7.4

| | $a_1$ | $a_2$ | $b_1$ | $b_2$ |
|---|---|---|---|---|
| x1 | $b_1$ / $y_1$ | $b_2$ / $y_1$ | $b_1$ / $y_2$ | $b_2$ / $y_2$ |
| x2 | $a_2$ / $y_1$ | $b_1$ / $y_3$ | $a_1$ / $y_2$ | $a_2$ / $y_2$ |

*Table* 7.8. *Machine* $A_1 = A'_2$

Back to Algorithm 7.5.1, we find, using the algorithm given in [Kou00] and described in Section 7.4.2 above, that the two machines obtained from the above steps, i.e. $A'_1 = A_4$ of Table 7.7 and $A'_2 = A_1$ of Table 7.8, are indistinguishable. Therefore, we break the non-deterministic machine $A'_1$ into the two machines say $B_3$ and $B_4$ shown in Tables 7.9 and 7.10, respectively. Therefore, the set $\mathfrak{R}_1$ becomes $\{B_3, B_4\}$.

Back to Step-2 of Algorithm 7.5.1, we find that the two deterministic machines $B_3$ and $B_4$ have no transitions that do not match the *TS*. Therefore, Algorithm 7.4.1 does not modify these machines. Afterwards, Algorithm 7.5.1 calls the algorithm defined in [Kou00] and described in Section 7.4.2 which generates, as shown in the tree in Fig. 7.17,

the distinguishing set $Dist_{set} = \{x2x2x1\}$ that distinguishes between the two machines $B3$ and $A'2$ in Table 7.8. Hence, if we apply the input $x1$ after the input sequence $x2$ $x2$ and the implementation at hand produces the output response $y1$ to the tail input $x2$ then we delete machine $A'2$ from the set $\mathfrak{R}2$, and we conclude that $M1$ is the faulty component, since the remaining machines $B3$ and $B4$ are in $\mathfrak{R}1$. Moreover, since the input sequence in the $Dist_{set}$ also distinguishes between $B3$ and $B4$, we notice that machine $B4$ can be eliminated since the output response of the SUT to the tail input $x1$ of the $Dist_{set}$ is not $y2$ as predicted by this machine. Thus, in this case, the faulty transition in the implementation at hand is that predicted by $B3$. However, if the output $y2$ is produced to the tail input $x2$ of the $Dist_{set}$, then we delete machine $B3$ from the set $\mathfrak{R}1$. In this case, the algorithm declares afterwards that the faulty machine can not be identified, since the remaining two machines in $B4$ in $\mathfrak{R}1$, and $A'2$ in $\mathfrak{R}2$, are indistinguishable, i.e, equivalent, and none of them is superfluous. Otherwise, if the SUT responds to the input sequence in the $Dist_{set}$ by an output response other than that expected by $B3$, $B4$ or $A'2$, i.e by an output other than $y1y3y1$ or $y1y3y2$, then the algorithm concludes that the SUT has a fault that cannot be captured by the assumed fault model.

| | $a1$ | $a2$ | $b1$ | $b2$ |
|---|---|---|---|---|
| $x1$ | $b1$ / $y1$ | $b2$ / $y1$ | $b1$ / $y2$ | $b1$ / $y2$ |
| $x2$ | $a2$ / $y1$ | $a2$ / $y3$ | $a1$ / $y2$ | $a2$ / $y2$ |

*Table 7.9. Machine $B3$ obtained from $A4$ of Table 7.7*

| | $a1$ | $a2$ | $b1$ | $b2$ |
|---|---|---|---|---|
| $x1$ | $b1$ / $y1$ | $b2$ / $y1$ | $b1$ / $y2$ | $b1$ / $y2$ |
| $x2$ | $a2$ / $y1$ | $b1$ / $y3$ | $a1$ / $y2$ | $a2$ / $y2$ |

*Table 7.10. Machine $B4$ obtained from $A4$ of Table 7.7*

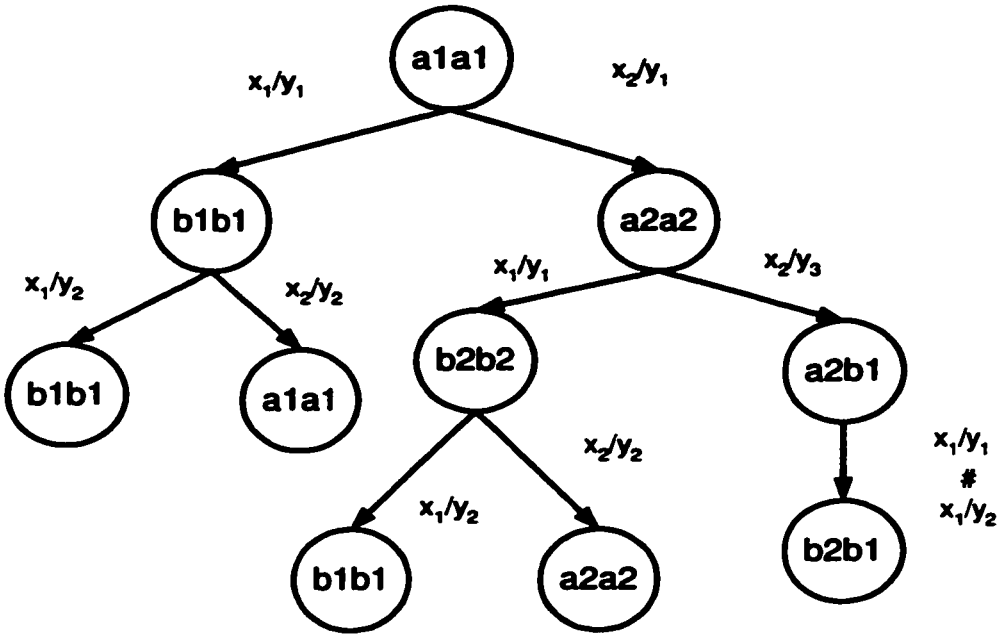


*Figure* 7.17. *A test sequence distinguishing the two deterministic FSMs* of $B_3$ of *Table* 7.9 and $A^r_2$ *of Table* 7.8

### 7.5.2 Locating the Faults within the Faulty Component

The proposed algorithm, Algorithm 7.5.1, decides whether it is possible to identify the faulty machine in a system of two *ComFSMs*. If this is possible, the same algorithm can be used (if necessary) to locate the faults within the faulty machine (if possible). The idea here is to locate the faulty implementation that corresponds to the faulty machine. This implementation is a sub-machine of $(M_1)_a^{out} \Diamond M_2$ when $M_1$ has a faulty implementation, or a sub-machine of $M_1 \Diamond (M_2)_a^{out}$ when $M_2$ has a faulty implementation. Therefore, the algorithm can be used to decide which sub-machine of $(M_1)_a^{out} \Diamond M_2$ (or $M_1 \Diamond (M_2)_a^{out}$) is equivalent to the system at hand, i.e. comparison between sub-machines of a given FSM can be added to the algorithm. We note that sometimes it is not possible to locate the faulty sub-machine. It is the case when the SUT has the same observable behavior for different output faults of the faulty machine.

## 7.6 On the Complexity of the Method

In general, the problem of diagnosing multiple faults is known to be exponentially complex [Lee93], since for an arbitrary number of faults, it becomes a machine identification problem. The machine identification problem (Moore, 1956) is known to be exponentially complex. However, as mentioned in [Lee93], for a small number of faults, the problem is still manageable; if there are a limited number of next state and output changes, then we have polynomial time algorithms for diagnosis.

In Algorithm 7.5.1, in the worst case, we have to enumerate all deterministic sub-machines of *FF-Context* and *FF-Embedded*. This happens when these sub-machines and their corresponding non-deterministic sub-machines are indistinguishable. In this case, the complexity of the algorithm depends mainly on the number of possible iterations of the algorithm multiplied by the number possible mutant sub-machines of *FF-Context* or *FF-Embedded*, which could be determined as follows:

Let $n$ denote the number of states of a given Fault Function (FF) machine, and $|I|$ denote the number of input alphabet $I$, and $|O|$ denote the number of the output alphabet of the machine. We note that for *FF-Context*, $n = n_1 n_2$, where $n_1$ is the number of states of machine $M_1$ and $n_2$ is the number of states of machine $M_2$, $|I| = |X| + |Z|$, where $X$ and $Z$ are the external and the internal input alphabets of $M_1$, and $|O| = |Y| + |U|$, where $Y$ and $U$ are the external and the internal output alphabets of $M_1$. For *FF-Embedded*, $n = n_1 n_2$, $|I| = |U|$ where $U$ is the number internal input alphabet of $M_2$, and $|O| = |Z|$, where $Z$ is the number of internal output alphabet of $M_2$.

The maximum number of transitions in a FF machine is $|n||I||O||n| = |n|^2|I||O|$, since for each state of the machine and for each input symbol, the machine may produce all outputs and transfer to all states. Therefore, the maximum number of transitions of

*FF-Context*, ***Trans_Context*** = $|n_1 n_2|^2$ (|X| + |Z|) (|Y| + |U|), and the maximum number of transitions of *FF-Embedded*, ***Trans_Embedded*** = $|n_1 n_2|^2$ |U| |Z|.

In the worst case, at the end of the first iteration of the algorithm, we may break *FF-context* and *FF-Embedded* each into $k$ machines, by fixing for each machine all outgoing non-deterministic transitions under an appropriate input at a given state. Then, $k$ equals |n||O|. Consequently, for *FF-Embedded*, $k$ equals $|n_1 n_2|$ |Y + U|, and for *FF-Embedded* $k$ equals $|n_1 n_2|$ |Z|. We put the obtained machines in the set $\mathfrak{R}_1$ and $\mathfrak{R}_2$, respectively, where these machines replace the existing machines $\mathfrak{R}_1$ = {*FF-context*} and $\mathfrak{R}_2$ = {*FF-Embedded*}. In the second iteration, in the worst case, we break each machine of the $k$ machines in the sets $\mathfrak{R}_1$ and $\mathfrak{R}_2$ into $k$ machines by fixing for these machines all outgoing non-deterministic transitions under an appropriate input at a given state. The obtained machines replace existing machines in the sets $\mathfrak{R}_1$ and $\mathfrak{R}_2$ and the cardinality of these sets becomes $k^2$. Using the same analysis, in the third iteration, the cardinality of the sets $\mathfrak{R}_1$ and $\mathfrak{R}_2$ becomes $k^3$. In the worst case, the last iteration of the algorithm occurs after generating as mentioned above all sub-machines of *FF-Context* and *FF-Embedded*. Consequently, for a FF machine, the last iteration happens after |n||I| iterations. Accordingly, in the worst case, the maximum number of iterations of the algorithm equals to the maximum of ($|n_1 n_2|$ (|X|+|Z|)) and ($|n_1 n_2|$ |U|). Moreover, at the iteration, in the worst case, the possible number of mutant machines in the set $\mathfrak{R}_1$ becomes $k^{n|I|} = k^{n_1 n_2 |X|+|Z|} = (n_1 n_2 |Y + U|)^{n_1 n_2 |X|+|Z|}$, and the possible number of the mutant machines in the set $\mathfrak{R}_2$ becomes $k^{n|I|} = k^{n_1 n_2 |U|} = (n_1 n_2 |Z|)^{n_1 n_2 |U|}$.

It is clear from the above analysis that in the worst case the diagnostic algorithm will not be applicable in practice. This is due to the fact that the number of mutant sub-machines of *FF-Context* and *FF-Embedded* is huge. However, the worst case occurs when after each iteration of the algorithm all the machines in the sets $\mathfrak{R}_1$ and $\mathfrak{R}_2$ are found to be indistinguishable. However, in order for all the machines in the set $\mathfrak{R}_1$ to be indistinguishable from those in the set $\mathfrak{R}_2$, there should exist for each pair of these

machines a sub-machine that makes this pair indistinguishable. Actually, we do not think that such a worst case would happen in practice. Moreover, for the first application example that we considered in this chapter, the algorithm converted to a solution only after one iteration where $|\mathcal{R}_1| = 1$ and $|\mathcal{R}_2| = 1$. Furthermore, for the second application example, the algorithm converted to a solution after two iterations where $|\mathcal{R}_1| = 1$ and $|\mathcal{R}_2| = 2$. However, according to the worst case complexity, the algorithm could iterate for 8 iterations, and the cardinality of the set $\mathcal{R}_2$ in the last iteration may become 104976.

We note that in the best case, the algorithm could converge to a solution only after one iteration. This happens when *FF-Context* and *FF-Embedded*, after removing those behaviors that do not match the observed output of the SUT, are distinguishable. The application of the distinguishing set to the SUT allows us to conclude which component machine is faulty. In this case, the difference between the worst-case complexity and the best case is at least of the order of the number of possible iterations of the worst case multiplied by the number of possible mutant sub-machines of *FF-Context* or *FF-Embedded*, i.e. $|n||I| k^{n|n}$, which is huge.

# 8 Conclusion and Further Research Work

Many methods have been developed for deriving tests for a system represented by a Finite State Machine (FSM) model. The purpose of these tests is to determine whether an implementation of the system conforms to (i.e., is correct with respect to) its specification. Usually a conforming implementation is required to have the same I/O behavior. In realistic applications, maintaining a system modeled by a given specification machine involves modifying its specification as a result of changes in the user requirements and designers implement incrementally these modifications. Testing the whole system implementation after each modification is considered expensive and time consuming. Therefore, it is important to generate tests that would only test the modified parts of the implementation that correspond to the modified parts of its specification. This would reduce the maintenance cost of such a system, which is about two-thirds of the cost of the software production [Sch92]. In the first part of this thesis, we presented test generation methods (called re-testing methods) that select tests (called re-tests) for testing the modified parts of the system specification, in order to check that these modifications were correctly implemented in the system implementation. Here we assume that the parts of the system implementation that correspond to the unmodified parts of the system specification are left intact. Moreover, we also reasonably assume that before modifying the system specification, its implementation was tested and found conforming to this specification. These methods are based on well-known test derivation methods called the W [Cho78], Wp [Fuj91], HIS [Pet93b], Unique-Input-Output UIOv- [Vuo89], and distinguishing sequence methods.

We note that although the re-testing methods are developed for the FSM model, it is expected that they could be useful for the Labeled Transition System (LTS) model as well, since testing approaches developed for FSMs have been transformed for the LTSs [Pet93a], [Tan95].

Based on the work presented in the first part of this thesis, it would be interesting to:

- Extend the re-testing methods for systems modeled as Extended Finite State Machines (EFSMs). EFSMs are used in formal description techniques ESTELLE [ISO87b] and SDL [SDL87] and also in UML Statecharts to specify not only the control part of a protocol, but also its data part. Therefore, the problem here is to find an appropriate way for re-testing both the control flow and the data flow parts of a modified EFSM. We believe that previous work on test derivation based on this model and on regression testing will help in solving this problem.

- Extend the re-testing methods for systems modeled as non-deterministic FSMs. All the three major specification languages for protocols, LOTOS, ESTELLE, and SDL and the UML support the description of non-determinism. Moreover, many test derivation methods [Luo94] [Pet93b] [Yev91] have been presented for systems modeled as non-deterministic FSMs. We believe that these methods could be used as a starting point for such a work.

- The distinguishing sequence re-testing methods presented in this thesis do not guarantee the derivation of minimal length re-testing (or re-checking) sequences. In order to minimize the lengths of these sequences some appropriate minimization algorithms have to be developed. The work presented in [Aho91] and [Ura97] could be a good starting point for such work.

As the purpose of conformance testing is to check whether an implementation conforms to its specification. An interesting complementary problem is to locate the differences between a specification and its implementation when the implementation is found to be nonconforming [Lee93]. In the second part of the thesis we address issues related to the fault diagnostic and localization problems. More specifically, we consider a system consisting of two communicating FSMs, called components. The system contains a context machine that communicates with the environment and an embedded machine.

The interaction between these two components is assumed to be hidden or unobservable. First, we reviewed a heuristic proposed in [Ghe93a] to locate a single fault within a given system of two communicating FSMs, once a fault has been detected in its implementation (called System Under Test (SUT)), and we showed that it is not always possible to locate such a fault. This happens when the SUT has the same observable behavior for a certain fault in an implementation of the context and another fault in the implementation of the embedded component. Accordingly, we presented a new two-level single fault diagnostic method for the fault localization of the given system [Elf99b] assuming as in previous related research work that the SUT has a single fault in one of its transitions (i.e. in one of its component machines). The first level (called *machine level diagnosis*) of the diagnostic method, decides if it is possible to identify the faulty component in the given system. If this is possible, the faulty component is identified, and if needed, at the second level (called *transition level diagnosis*) the method determine if it is possible to locate the fault within the faulty component. If that is possible, the method provides additional test cases to locate the fault. We note that sometimes it is not possible to locate the fault within the faulty component, since the same observable behavior of the SUT may be caused by different faults in the component implementation. At the end of the second part of the thesis, we presented a new multiple faults diagnostic method [Elf01] for the given system, for the case when the SUT may have multiple faults in at most one of its components. We note that for a system decomposed into components, it is usually assumed for high-level abstractions that the structure of the system is preserved in all possible implementations, while a component may be either faulty or operating correctly [Kle87]. The multiple faults method enables to decide if it is possible to identify the faulty machine in the system, once faults have been detected in a system implementation. If this is possible, it also provides tests for identifying the faulty component machine. Moreover, if desired, the method can be used for deciding if it is possible to locate the faults within the faulty machine, and if possible it can provide tests for locating these faults.

Based on the diagnostic methods presented in the second part of the thesis, it would be interesting to:

- Implement and experiment with the multiple diagnosis method in order to assess its applicability to small-size and large-size systems. According to the application examples presented in Chapter 7, the fault functions representing the fault domain of the given system are significantly reduced by Algorithm 7.4.1, using a given test suite and the observed behavior of the implementation for this test suite. It would be interesting to know if we obtain similar results for realistic small-size and large-size systems. Moreover, as done for the single fault diagnostic method, it would be interesting to determine how often a faulty component machine can be identified for various implementations of a given specification.

- Generalize the multiple diagnosis method for asynchronous systems that may have more than one message in transit, i.e. where the next external input may be submitted to the system before the system produces an external output to the previous external input. Asynchronous systems with more than one message in transit are widely used in many application areas such as telecommunication protocols.

- Extend the multiple faults diagnosis method for identifying the faulty component FSM within a system of two communicating FSMs when both component FSMs may be faulty. The idea here is to derive appropriate fault functions for this case.

- Extend the multiple diagnostic method for the case when an implementation component FSM can have more states that its specification. We think that the method presented in the thesis can be used for this case if all possible component implementations can be described by an appropriate Fault Function, i.e. all possible component implementations are deterministic sub-machines of an appropriate non-deterministic FSM.

- Use the method for the diagnosis of transfer and output faults when the system specification and implementation are given in the form of a single finite state machine. In this context all possible implementations are sub-machines of one appropriate fault function, and the algorithm starts by breaking this function into many machines in order to distinguish between them and the SUT.

# References

[Aho91] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar, (1991). "An optimization technique for protocol conformance test sequence generation based on UIO sequences and rural Chinese postman tours", *IEEE Trans. on Communications*, vol.39, 1991, pp. 1604-1615.

[Boc78] G. v. Bochmann, "Finite state description of communication protocols", *Computer Networks*, Vol. 2, 1978, pp. 361-372.

[Boc80] G. v. Bochmann, and C. A. Sunshine, "Formal methods in communication protocol design", *IEEE Trans. on Comm.*, Vol 28, 1980, pp 624-631.

[Boc86] G. v. Bochmann, "Recent development in protocol specification, validation and testing", *Journal of China Institute of Communications*, Vol.7, 1986, pp. 76-88.

[Boc90] G. v. Bochmann, "Protocol specification for OSI", *Computer Networks and ISDN Systems*, 18, 1990, pp. 167-184.

[Boc92] G.v. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo, "Fault models in testing", *IFIP Transactions, Protocol Testing Systems IV (Proc. of IFIP TC6 Fourth International Workshop on Protocol Test Systems,* 1991), Ed. by Jan Kroon, Rudolf J. Heijink and Ed Brinksma, North-Holland, 1992, 17-30.

[Boc94] G. v. Bochmann, A. Petrenko, "Protocol testing: review of methods and relevance for software testing", in *Proc. of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, USA, Ed. by Thomas Ostrand, 1994, pp. 109-123.

[Boc95] G. v. Bochmann, A. Petrenko, and M. Yao,(1995). 'Fault coverage of tests based on finite state models', *IFIP Transactions, Protocol Test Systems VII* (*Proc. of* IWPTS'94, *1994*), Ed. by T. Mizuno, T. Higashino and N. Shiratori, 1995, pp. 55-76.

[Bra83] D. Brand, and P. Zafiropulo, "On communicating finite state machines", *Journal of the ACM* , Vol. 30 No. 2, 1983, pp. 323-342.

[Bri90] E. Brinksma, et al., "A formal approach to conformance testing", in *IWPTS*'90, 1990, pp. 349-363.

[Cho78] T. S. Chow, "Test design modeled by finite-state machines", *IEEE Trans. SE*-4, No.3, 1978, pp. 178-187.

[Elf99b] K. El-Fakih, and G. v. Bochmann, "Locating a faulty machine in a system of communicating finite state machines", in *Proc. of the EEEL Workshop on Software Embedded Systems and Testing,* Maryland, USA, 1999, pp.75-80.

[Elf01] K. El-Fakih, N. Yevtushenko, and g. v. Bochmann, "Diagnosing multiple faults in communicating finite state machines", in *Proc. of the IFIP 21st International Conference on Formal Techniques for Networked and Distributed Systems* (FORTE2001), Cheju, KOREA , 2001, pp. 85-100.

[Elf02] K. El-Fakih , N. Yevtushenko, and G. v. Bochmann, "FSM based re-testing methods", in *Proc. of the IFIP 14th International Conference on Testing of Communicating Systems* (TestCom2002), Berlin, Germany, March 19-22, 2002, pp. 373-389. .

[Dav88] R. Davis, and W. Hamscher, "Model-based reasoning: Troubleshooting", in *Exploring Artificial Intelligence*, edited by Shrobe, H. E. and the American Association for Artificial Intelligence, 1988, pp. 297-346, Morgan Kaufman.

[ISO87a] ISO DIS8807, *LOTOS: a formal description technique*, 1987.

[ISO87b] ISO DIS9074, *Estelle: A formal description technique based on an extended state transition mode*, 1987.

[Fuj91] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, A., "Test selection based on finite state models", *IEEE Trans. SE*-17, No. 6, 1991, pp. 591-603.

[Gil62] A. Gill, *Introduction to the Theory of Finite-State Machines*, McGraw-Hill, 1962.

[Gen84] M.r. Genesereth, "The use of design descriptions in automated diagnosis", *Artificial Intelligence* 24(3), 1984, pp. 411-436.

[Ghe92a] A. Ghedamsi, and G. v. Bochmann, "Test result analysis and diagnostics for finite state machines', in *Proc. of the 12-th international conference on distributed systems*, Yokohama, Japan, 1992.

[Ghe92b] A. Ghedamsi, and G. v. Bochmann, "Diagnostic tests for finite state machines", TR No. 807, Univ de Montréal, Montréal, January 1992.

[Ghe92d] A. Ghedamsi, R. Dssouli, and G. v. Bochmann, "Diagnostic tests for single transition faults in non-deterministic finite state machines", in *Proc. of the* $5^{th}$ *international Workshop on Protocol Test Systems*, Montreal. September 28-30, 1992.

[Ghe93a] A. Ghedamsi, G. v. Bochmann, R. Dssouli, R., "Diagnostic tests for communicating finite state machines", in *Proc. of the 12th IEEE International Phoenix Conference on Communications*, Scottsdale, USA, March 1993.

[Ghe93b] A. Ghedamsi, G. v. Bochmann, and R. Dssouli, "Multiple fault diagnosis for finite state machines", in *Proc. of IEEE INFOCOM'93*, 1993, pp.782-791.

[Gon70] G. Gonenc, "A method for the design of fault detection experiments". *IEEE Trans. Computers*, Vol. C-19, No. 6, 551-558.

[Hsi71] E. P. Hsieh, "Checking experiments for sequential machines", *IEEE Trans. on Computers*, Vol. 20, No. 10, 1971, pp. 1152-1166.

[Hen64] F. C. Hennie, "Fault detecting experiments for sequential circuits", in *Proc. of 5th Annual Symposium on Switching Circuit Theory and Logical Design*, Princeton, 1964, pp. 95-110.

[Hoa85] A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[Kle87] J. de Kleer, and B.c. Williams, "Diagnosing multiple faults", *Artificial Intelligence*, 32(1), 1987, pp. 97-130.

[Kle89] J. de Kleer, B.c. Williams, "Diagnosing with behavioral models", in *Proceedings IJCAI*, Detroit-Michigan, 1989, pp. 1324-1330.

[Koh78] Z. Kohavi, *Switching and Finite Automata Theory.* New York, McGraw-Hill, 1978, p. 658.

[Kou99] I. Koufareva, A. Petrenko, and N. Yevtushenko, "Test generation driven by user-defined fault models", in *Proc. of the IFIP TC6 12th International Workshop on Testing of Communicating Systems*, Hungary, 1999, pp. 215-233.

[Kou00] I. Koufareva, *Using non-deterministic FSMs for test suite derivation*, Ph.D. Thesis, Tomsk State University, Russia, 2000 (In Russian).

[Lee93] D. Lee, and K. Sabnani, "Reverse engineering of communication protocols", in *Proc. of the ICNP*, October 1993, pp. 208-216.

[Lee94] D. Lee and M. Yannakakis "Testing finite-state machines: State identification and verification", *IEEE Trans. on Computers*, Vol. 43, No. 3, 1994, pp. 306-320.

[Luo94] G. Luo, A. Petrenko, G. v. Bochmann, "Test selection based on communicating nondeterministic finite state machines using a generalized Wp-method", *IEEE Trans.*, Vol. SE-20, No. 2, 1994.

[Mil80] R. Milner, *A Calculus of Communication Systems*, LNCS, 1980, p. 92.

[Mor90] L.J. Morell, "A theory of fault-based testing", *IEEE Trans. SE*-16, 8, 1990, pp. 844-857.

[Pet91] A. Petrenko, "Checking experiments with protocol machines", in *Proc. of the 4th Int. Workshop on Protocol Test Systems*, 1991, pp. 83-94.

[Pet92] A. Petrenko, and N. Yevtushenko, "Test suite generation for a FSM with a given type of implementation errors", in *Proc. of the 12th Int. Workshop on Protocol Specification, Testing and Verification*, 1992, pp. 229-243.

[Pet93a] A. Petrenko, G. v. Bochmann, and R. Dssouli, "Conformance relations and test derivation", in *Proc. of the IFIP International Workshop on Protocol Test Systems*, Pau, France, September, 1993, pp. 157-178.

[Pet93b] A. Petrenko, N. Yevtushenko, A. Lebedev, and A. Das, 'Nondeterministic state machines in protocol conformance testing', in *Proc. of the IFIP Sixth International Workshop on Protocol Test Systems*, France, pp. 363-378, 1993.

[Pet96a] A. Petrenko, G. v. Bochmann, and M. Yao, "On fault coverage of tests for finite sate specifications", *Computer Networks and ISDN Systems*, Vol. 29, 1996, pp. 81-106.

[Pet96b] A. Petrenko, N. Yevtushenko, G.v. Bochmann, and R. Dssouli, "Testing in context: Framework and test derivation" *Computer Communications Journal*, Special issue on protocol engineering, 1996, pp. 1236-1249.

[Rei87] R. Reiter, "A theory of diagnosis from first principles", *Artificial Intelligence* 32(1), 1987, pp. 57-96.

[Ray87] D. Rayner, "OSI conformance testing", *Computer Networks and ISDN Systems* 14, 1987, pp. 79-98.

[Sar82] B. Sarikaya, and G. v. Bochmann, "Some experience with test sequence generation for protocols', in *Proc. of the 2nd Int. Workshop on Protocol Specification, Testing and Verification*, 1982.

[Sab88] K. Sabnani, and A. Dahbura, "A protocol test generation procedure", *Computer Networks and ISDN Systems*, Vol. 15, No. 4, 1988, pp. 285-297.

[Sch92] S. Schach, *Software Engineering*, Boston:Aksen Assoc, 1992.

[SDL87] CCITT SG XI (1987) Recommendation Z.100, 1987.

[Sho76] E. H. Shortlife, *Computer-based Medical Consultations : MYCIN*, Elsevier, New-York, 1976.

[Sid89] D. P. Sidhu, and T. K. Leung, "Formal methods for protocol testing: a detailed study", *IEEE Trans.* SE-15, No. 4, 1989, pp. 413-426.

[Str89] P. Struss, and O. Dressler, "Physical negation - integrating fault models into the general diagnostic engine", in *Proc. IJCAI*, Detroit - Michigan, 1989, pp. 1318-1323.

[Tan95] Q. M. Tan, A. Petrenko and G. v. Bochmann, G. v., "Modelling basic LOTOS by FSMs for conformance testing", Publication #958, Département d'informatique et de recherche opérationnelle, Université de Montréal, February 1995.

[Ura92] H. Ural, "Formal methods for test sequence generation", *Computer Communications*, Vol. 15, No. 5, 1992, pp. 311-325.

[Ura97] H. Ural, X. Wu, and F. Zhang, "On minimizing the lengths of checking experiments", *IEEE Trans. on Computers*, Vol. 46, No.1, 1997, pp. 93-99.

[Vas73] M. P. Vasilevskii, "Failure diagnosis of automata", translated from Kibernetika, No.4, 1973, pp. 98-108.

[Vuo89] S. T. Vuong, W.W.L. Chan, and M.R. Ito, "The UIOv-method for Protocol test sequence generation", in *Proc. of the IFIP TC6 Second International Workshop on Protocol Testing Systems*, Ed. by Jan de Meer, Lothar Machert and Wolfgang Effelsberg, 1989, North-Holland, pp. 161-175.

[Wes78] C. H. West, "An automated technique of communication protocols validation", *IEEE Trans. on Comm.* 26, 1978, pp. 1271-1275.

[Yan90] B. Yang, and H. Ural, "Protocol conformance test generation using multiple UIO sequences with overlapping", *Computer Communication Review*, No. 4, 1990, pp. 118-125.

[Yan95] M. Yannakakis, and D. Lee, "Testing finite state machines', *Proc. of the 23d Annual ACM Symposium on Theory of Computing*, New Orleans, Louisiana, 1995, pp. 476-485.

[Yao93] M. Yao, A. Petrenko, and G. v. Bochmann, "Conformance testing of protocol machines without reset", in *Proc. of the* $13^{th}$ *IFIP Symposium on Protocol Specification, Testing and Verification*, Liege, Belgium, May, 1993, pp. 241-253.

[Yao95] M. Yao, *On the development of conformance test suite in view of their fault coverage*, Ph.D Thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal, Montréal, Canada, 1995.

[Yev90] N. Yevtushenko, and A. Petrenko, *Test derivation method for an arbitrary deterministic automaton*, Automatic Control and Computer Sciences, Allerton Press, Inc., USA, #5, 1990.

[Yev91] N. Yevtushenko, A. Lebedev, and A. Petrenko, "On the checking experiments with nondeterministic automata", *Automatic Control and Computer Sciences*, Allerton Press, Inc., N.Y., Vol.25, No.6, 1991.