BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS
DEPT. OF TELECOMMUNICATIONS AND MEDIA INFORMATICS

DETECTION, DIAGNOSIS AND CORRECTION OF FAULTS IN TELECOMMUNICATIONS SOFTWARE DEVELOPMENT BASED ON FORMAL METHODS

Zoltán Pap

Ph.D. Dissertation

Supervised by

Dr. Gyula Csopaki, Dr. Sarolta Dibuz and Dr. Katalin Tarnay
Department of Telecommunications and Media Informatics
Budapest University of Technology and Economics

Budapest, Hungary
2006

Zoltán Pap

Dept. of Telecommunications and Media Informatics

Budapest University of Technology and Economics[1]

[1]The reviews and the minutes of the Ph.D. Defense are available from the Dean's Office.

To my family

and friends

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CCITT	International Telegraph and Telephone Consultative Committee
CFSM	Communicating Finite State Machine
CUP	Constructor of Useful Parsers
c-use	Computational-use
CS	Characterizing Sequence
def	Definition
DFG	Data Flow Graph
D-method	Distinguishing Sequence Method
DS	Distinguishing Sequence
du-path	Definition-uses-path
EFSM	Extended Finite State Machine
ESO	Extra State Operator
FDT	Formal Description Technique
FIFO	First In, First Out
FSM	Finite State Machine
INRES	Initiator-Responder Protocol
ISO	International Organization for Standardization
ITU	International Telecommunication Union
ITU-T	ITU Telecommunication Standardization Sector
IUT	Implementation Under Test
JDK	Java Development Kit

LOTOS	Language of Temporal Ordering Specifications
MB	Megabyte
MHz	Megahertz
MSC	Message Sequence Chart
MSO	Missing State Operator
NFSM	Nondeterministic Finite State Machine
OO	Output Operator
OSI	Open Systems Interconnection
PC	Personal Computer
p-use	Predicate-use
SA	Simulated Annealing
SCR	Software Cost Reduction
SDL	Specification and Description Language
SDL/GR	Specification and Description Language - Graphical Representation
SDL/PR	Specification and Description Language - Phrase Representation
TRO	Transfer Operator
TTCN-2	Tree and Tabular Combined Notation version 2
TTCN-3	Testing and Test Control Notion 3rd Generation
TT-method	Transition Tour Method
UIO	Unique Input/Output Sequence
WAP	Wireless Application Protocol
WTP	Wireless Transaction Protocol

Abstract

Telecommunications software plays an increasingly important role for both business and wider society, since it is an indispensable brick in the foundation of today's communication infrastructure. To meet the requirements, formal methods-based software development is becoming prevalent, with testing and maintenance phases gaining in importance. The primary purpose of the work reported in this thesis is to investigate issues related to testing and maintenance, addressing problems in the field of fault detection, diagnosis and correction involving FSM and EFSM modeling techniques. First the feasibility of the fault diagnosis problem for finite state machines is studied. Next, a fault-based method is presented to improve test sets for systems defined using the Specification and Description Language. Finally, a novel approach to model patches is introduced and a new problem of optimizing patches is defined and analyzed.

Acknowledgements

Many people have contributed to the development of the ideas and methods presented in this dissertation over the last few years. First and foremost I wish to express my deepest gratitude to my advisors Gyula Csopaki, Sarolta Dibuz and Katalin Tarnay. This thesis would not have been possible without their constant support and guidance. Their thoughtfulness and optimism often kept me going even when progress was slow.

I would like to thank Gusztáv Adamis, my supervisor as a Masters student. He has taught me more than a couple things about the nature of research and teaching.

I would like to thank the Department of Telecommunications and Media Informatics at Budapest University of Technology and Economics for providing an environment in which it has been possible to freely pursue independent thoughts.

Many thanks to all the members of the Telecommunications Software Group for making it a stimulating and enjoyable place to work. I am indebted to my colleagues for their thoughtful comments, criticism, and encouragement provided throughout our collaborate work.

I would like to thank all my friends as well, who helped me to detach from work and relax at times I really needed it.

Finally, thanks to my family supporting me over the years.

Zoltan Pap
Budapest, 2006

Chapter 1

Introduction

Telecommunications software plays an increasingly important role for both business and wider society, since it is an indispensable brick in the foundation of today's communication infrastructure. As it becomes more and more significant, there are contradictory requirements these systems have to meet. On the one hand they have to provide more complex services with tightening time limits, on the other hand they have to be reliable, since thousands of applications are built on them.

The increase in complexity of the software increases the probability that faults will be introduced into the system somewhere along the development lifecycle. Faults are permanent defects in the system causing errors, i.e., deviations from intended behavior. Such erroneous behavior may lead to the failure of the system, which is an observable effect outside the system boundary arising from an internal error. Thus, the risk of failures rises along with increasing complexity of the system.

Still, there is a strong need for reliable telecommunications software functioning as required, without failures. Developers have to devote growing effort to ensure that telecommunications systems are of the highest possible integrity, containing as few faults as possible. Furthermore, as product lifecycles shorten speed becomes an ever important factor; therefore the development process should be as short as possible to reduce time-to-market.

The most promising way to cope with these requirements is the use of formal methods for software development. Formal methods are languages and methodologies with well established mathematical background aiding the precise specification and implementation of software and other types of systems. They are often backed by tool support, and can be used not just to describe a system but to analyze its behavior as well. The initial work on formal methods started with the standardization for Open Systems Interconnection (OSI) in the early eighties. Special working groups studied the possibility of using formal specifications for the definition of the OSI protocols. Their work led to the proposal of three languages, Estelle [TC988], LOTOS (Language

of Temporal Ordering Specification) [ISO89] and SDL (Specification and Description Language) [IT00], called formal description techniques (FDTs). The mathematical foundations for two of these languages, Estelle and – the most widely used – SDL, are provided by the (extended) finite state machine-based modeling technique.

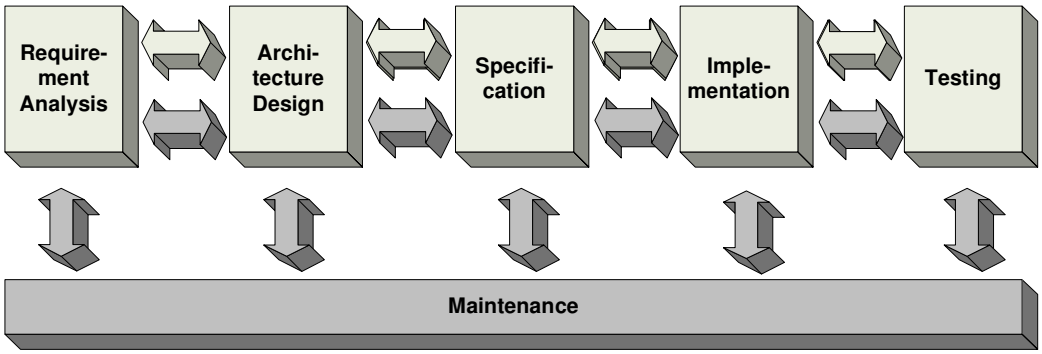


Figure 1.1: Software development lifecycle

As in the case of other systems, a telecommunication software development should follow a lifecycle including different phases such as requirements analysis, architecture design, specification, implementation, testing and maintenance illustrated in Figure 1.1. The fundamental idea behind formal methods-based software development is that a precise specification of the system can support this lifecycle in a number of ways, and thus help create a higher quality product.

- Precise and unambiguous formal specifications can be applied for the standardization of systems. They can help attain interoperable systems even if components are coming from many different vendors.

- Formal specifications can be used to properly understand the system at an early phase, possibly revealing any design problems or incomplete features.

- Formal specifications may serve as a driving force behind the development. The implementation process may be aided either by the refinement of the specification or by direct code generation from the specification.

- A formal specification can be used to support the testing process. It may help, for example, the tester to distinguish right and wrong behavior of an implementation, i.e., to derive correct oracles to check the results of tests. A formal specification could also be used to generate tests that are likely to be effective in detecting faults.

This thesis concentrates on the testing and maintenance phases of the telecommunication software development lifecycle, and considers systems specified using finite state machine (FSM) and extended finite state machine (EFSM) modeling techniques.

Software testing is an important and extremely expensive part of the software development process. Although advances in computer science help improve the quality of software, studies argue that testing is still time-consuming and accounts for around fifty percent of the total development cost. Thus, for economical reasons, it is crucial to make the testing process more efficient in time consumption, cost and quality. The most promising approach to reach this objective is the automation of testing activities. In particular, automation is an especially important challenge for the initial phase of testing, the test set development.

Maintenance is a phase of the software development lifecycle after deployment of the software into the field. It is concerned with changes to the software to correct faults and deficiencies found after deployment, either through additional testing or field usage. It is also used to modify or add functionality to comply with changes of the requirements. While for some systems the only solution for maintenance is to recall distributed implementations, in the case of software systems patches – or updates – are the most widely used means to change the behavior of a system. There are many benefits of employing patches. It is a faster and less expensive approach than recalls, enabling developers to constantly follow user demands and quickly respond to them.

1.1 The Purpose and Structure of the Thesis

The primary purpose of the work reported in this thesis is to investigate issues related to testing and maintenance. Our contributions to the topic includes three main aspects that are organized around the notion of faults. The aspects include fault detection, fault diagnosis and the correction of faults. The thesis consists of seven chapters. The structure of the thesis is shown in Figure 1.2.

After a short discussion on the motivation behind the thesis in Chapter 1, the fundamental aspects of FSM-based implementation and testing are outlined in Chapter 2. The first part of Chapter 2 is concerned with the FSM model; some key concepts, such as the equivalence, minimization and completeness assumptions are explained. In the second part of Chapter 2 a framework of testing problems is presented, with the emphasis on the two most important black box testing problems: fault detection – or conformance testing – and fault diagnosis. Related concepts like conformance relations and fault models are discussed as well. In all, the main objective of Chapter 2 is twofold: it lays down a common ground for the further studies in the rest of the thesis, and it provides a detailed insight into the state of the art in FSM-based specification and testing.

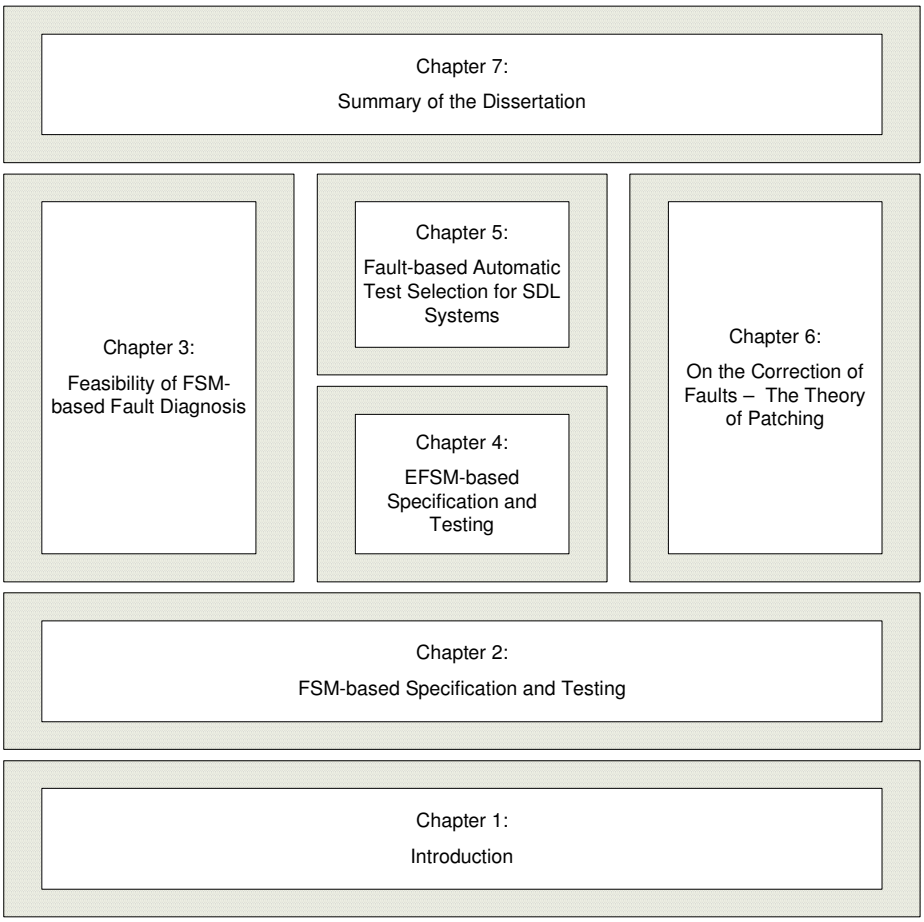


Figure 1.2: The structure of the thesis

Chapter 3 presents our contributions to the problem of fault diagnosis, i.e., localization of difference(s) between an implementation and a specification in systems modeled by finite state machines. The primary purpose of the investigations is to clarify the feasibility of the problem, concentrating on the case of a single transition or output fault in an FSM. It is shown that – contrary to the statements in the literature – it is not always possible to precisely diagnose (locate) a single fault in a finite state machine, as reduced implementation machines with different single faults may have the same observable behavior. We analyze the reasons for the problem and determine a set of sufficient conditions for the guaranteed exact localization of a single fault. Based on the analytical results an algorithm is elaborated for the fault diagnosis problem. If it is possible, the method exactly locates the difference between the implementation and the specification. In case exact localization is infeasible, it provides the minimal set of all potential single faults.

The purpose of Chapter 4 is to complete the specification and testing framework of Chapter 2 by introducing the extended finite state machine model. The first part of Chapter 4 details the EFSM model and some related basic concepts. The second part introduces the formal description techniques and discusses the two FDTs based on the EFSM model: Estelle and SDL. Finally the third part of the chapter presents the conformance testing framework for EFSM-based systems.

Chapter 5 presents the second aspect of our contributions: an algorithm to improve test sets for systems defined using the Specification and Description Language. Our approach utilizes ideas of mutation analysis to automate all steps of the test selection process. We propose a set of mutation operators that were designed considering the specialties of SDL and the need of automation. The algorithm provides the basis of the Test Selector tool developed at the Budapest University of Technology and Economics. The final part of the chapter presents the results of test selection experiments performed on well-known protocols using the tool.

Chapter 6 introduces the third aspect of our contributions. It studies the problem of patching, i.e., modifying an existing system, which – among other things – may be used to correct faults of a system after its deployment. We define a set of edit operators based on a traditional FSM fault model, and argue that sequences of edit operations can be considered as models of patches defining modifications to an FSM system. A new problem is introduced referred to as the optimal patch – or optimal update – problem. Given a requirement for a change, it is concerned with finding the optimal patch – minimal cost edit operations – modifying the system accordingly. A complexity analysis of the problem is conducted showing that it is unlikely to have a polynomial time solution for it. Finally we discuss how the problem can be transformed to a state-space search problem that can be approximated with existing heuristic algorithms.

Chapter 7 gives a summary of the thesis.

Chapter 2

FSM-based Specification and Testing

The specification of a system can be given either in a formal or an informal way. The use informal methods such as natural languages are typical at early phases of the design process. The main advantage of these descriptions is that they are easy to understand, and do not require knowledge of any formalism. Informal methods, however, lack the precision required to create unambiguous and terse specifications. They do not provide any support for correctness or completeness checking. These shortcomings of informal methods led to the introduction of various formal methods to aid the design and implementation process. These include Petri nets, formal grammars, high-level programming languages, process algebras, and hybrid and semi-formal languages. One of the most common ways of modeling the behavior of both hardware and software systems is using finite state machines.

2.1 Finite State Machines

A finite state machine (FSM) – also known as sequential machine or finite automaton – represents a "behavior", i.e., a regular sequential function transforming input sequences into output sequences. A sequential function is regular if at any stage the output symbol depends only on the sequence of input symbols which have been already received.

FSMs have been widely used to model systems in various areas. These include sequential circuits [FM71], some types of programs [ASU86] (in lexical analysis, pattern matching etc.), and more recently communication protocols [Hol90].

The theory of FSMs was mainly developed in the 1950s [Huf54] [Mea55] [Moo56]. General finite state machine theory and applications can be found in books on discrete mathematics [DDQ78] [Pra76] and digital switching theory [Pra67].

2.1.1 The FSM Model

A finite state machine contains a finite number of states and has finite sets of input and output symbols. One input produces at most one output which is either determined entirely by the FSM's state (Moore machine) or jointly by its input and previous state (Mealy machine). A Moore machine is therefore known as a state-assigned FSM and a Mealy machine is known as a transition-assigned FSM. The two types of machines can be converted into each other. In the remaining part of the thesis we will concentrate on transition-assigned FSM's.

Definition 2.1.1. A finite state machine M is a quintuple

$$M = (I, O, S, \delta, \lambda)$$

where

- I is the finite set of input symbols,
- O is the finite set of output symbols,
- S is the finite set of states,
- δ: $S \times I \rightarrow S$ is the transition function,
- λ: $S \times I \rightarrow O$ is the output function.

For further discussion let us denote the number of states, inputs, and outputs by $n = |S|$, $p = |I|$, and $q = |O|$, respectively.

FSM A is said to be strongly connected if, for each pair of states (s_j, s_l), there exists an input sequence which takes A from s_j to s_l.

If the transition function and the output function are defined for all state-input combinations, the FSM is said to be completely specified (or completely defined).

The machine in Definition 2.1.1 is deterministic, since for every (s, i) there is at most one transition going to a unique next state and producing a unique output determined by the transition and output functions, respectively. Machines that may have more than one possible transition for a given (s, i), that may go to different next states, or that produce different outputs are nondeterministic finite state machines (NFSMs) [Sta72].

A nondeterministic finite state machine (NFSM) A is a 4-tuple (I, O, S, h) where

- I is the finite set of input symbols,
- O is the finite set of output symbols,
- S is the finite set of states,

- h: $D \rightarrow 2^{O \times S}$ is a behavior function where $D \subseteq S \times I$ is the specification domain and $2^{O \times S}$ is the set of all subsets of the set $O \times S$.

An FSM (or NFSM) can be represented by a state transition diagram, a directed edge-labeled graph whose vertices correspond to the states of the machine and whose edges correspond to the state transitions. Each edge is labeled with the input and output associated with the transition. Suppose that the machine is currently in state s_3 and upon input c the machine moves to state s_2 and outputs 1. This transition can be written in a form $s_3 \xrightarrow{c/1} s_2$.

We extend the transition function δ and output function λ from input symbols to finite input sequences (strings) I^* as follows: For a state s_1, an input sequence $x = i_1, ..., i_k$ takes the machine successively to states $s_{j+1} = \delta(s_j, i_j), j = 1, ..., k$ with the final state $\delta(s_1, x) = s_{k+1}$, and produces an output sequence $\lambda(s_1, x) = o_1, ..., o_k$, where $o_j = \lambda(s_j, i_j), j = 1, ..., k$. The input/output sequence $i_1 o_1 i_2 o_2 ... i_k o_k$ is then called a trace of M. Note that if an FSM is deterministic then all its traces are deterministic, because there are no transitions with different next states and/or outputs for the same state-input combination.

We can extend the transition and output functions from a single state to a set of states: if Q is a set of states and x an input sequence, then $\delta(Q, x) = \delta(s, x) | s \in Q$, and $\lambda(Q, x) = \lambda(s, x) | s \in Q$.

We say that machine M has a reset capability if there is an initial state $s_0 \in S$ and an input symbol $r \in I$ that takes the machine from any state back to s_0. That is, $\delta(s_j, r) = s_0$ for all states $s_j \in S$. The reset is reliable if it is guaranteed to work properly in any implementation machine M', i.e., $\delta'(s'_j, r) = s'_0$ for all states $s'_j \in S'$, otherwise it is unreliable. Note that reset r is an input symbol as well. Thus, if M has reset then M is considered to be strongly connected if all the other states can be reached from the initial state s_0.

We say that a machine M has a status message if there is a special status input, and upon receiving this input the machine outputs its current state and stays there. Machines with status messages are observable at any state. We say that the status message is reliable if it is guaranteed to work in any implementation machine, otherwise it is unreliable.

2.1.2 FSM Equivalence

Finite state machines may contain redundant states. State minimization is a transformation into an equivalent state machine to remove redundant states.

Two states are equivalent written $s_j \cong s_l$ if and only if for every input sequence the machine will produce the same output sequence regardless of whether s_j or s_l is the starting state. In other words, for all input sequences $x \in I^*$, $\lambda(s_j, x) = \lambda(s_l, x)$.

(Note that their succeeding states for a particular input sequence are also pairwise equivalent).

Two states, s_j and s_l are distinguishable (inequivalent), if there exists a finite input sequence x which when applied to FSM M causes different output sequences starting in either state. In other words, $\exists x \in I^*$, $\lambda(s_j, x) \neq \lambda(s_l, x)$. Such an input sequence is called a separating sequence of the two inequivalent states. If the shortest such sequence is of length k then (s_j, s_l) are k-distinguishable. A FSM M is reduced (minimized), if no two states are equivalent, that is, each pair of states $(s_j,\ s_l)$ are distinguishable.

There is a number of equivalence relations proposed in the literature. Some examples of equivalence relations are strong bisimilarity, weak bisimilarity [Mil89], testing equivalence [Tre94] and trace equivalence [RHB97]. They are, however, all the same for completely specified and deterministic machines, and they are only different in the case of more general machines like nondeterministic machines.

Machine equivalence is an equivalence relation on all FSMs with the same input and output sets. Two completely specified deterministic FSMs M and M' are equivalent if and only if for every state in M there is at least one corresponding equivalent state in M', and vice versa.

A homomorphism from M to M' is a mapping ϕ from S to S' such that for every state $s \in S$ and for every input symbol $i \in I$, it holds that $\delta'(\phi(s), i) = \phi(\delta(s, i))$ and $\lambda'(\phi(s), i) = \lambda(s, i)$ [Moo56]. If ϕ is a bijection, then it is called an isomorphism. In this case M and M' must have the same number of states, and they are identical except for a renaming of states. Two machines are called isomorphic if there is an isomorphism from one to the other. Two isomorphic FSMs are equivalent, but the converse is not true in general.

In each equivalence class there is a reduced machine with the minimal number of states. In an equivalence class, any two reduced machines have the same number of states; furthermore, there is a one-to-one correspondence between equivalent states, which gives an isomorphism between the two machines. That is, the reduced machine in an equivalence class is unique up to isomorphism.

2.1.3 FSM Minimization

The minimization of deterministic finite state machines is one of the central problems of FSM theory, and has been studied since the late 1950s. The problem is to find the unique (up to isomorphism) minimal deterministic FSM which is equivalent to a given deterministic finite automaton.

The problem can be solved in polynomial time, and there are many algorithms for it. Most of them rely on determining sets of equivalent states in an automaton. Clearly, every state in an equivalence set, for all inputs, produces the same output

and goes to next states that are in the same equivalence set. Thus, after sets of equivalent states are identified, the minimal equivalent machine can be easily created by projecting each set to a single state.

One of the most naive approaches to determine sets of equivalent states is to compute the equivalence relation for pairs of states chosen in an arbitrary order. We look at each pair of states (s_1, s_2) in the FSM. If s_1 produces different outputs from s_2 upon any input i, we mark them non-equivalent. For each state pair (s_1, s_2) not marked, for each input i, find the state pair $(\delta(s_1, i), \delta(s_2, i))$. If $(\delta(s_1, i), \delta(s_2, i))$ are marked non-equivalent for any i, mark (s_1, s_2) non-equivalent. Iterate until no more marking is possible.

A slightly improved approach uses a well known algorithm that splits states successively into equivalent blocks [Koh78]. Instead of going through all pairs of states, the algorithm divides states of an FSM into blocks. Similarly to the previous approach, states are first split based on their outputs. States s_1 and s_2 are placed in the same block if and only if $\lambda(s_1, i) = \lambda(s_2, i)$ for all i. Each block is then further split into sub-blocks according to the transitions of the states they contain. States s_3 and s_4 are placed in the same sub-block if and only if their next states $\delta(s_3, i)$ and $\delta(s_4, i)$ are in the same block for all i. Iterate until no more splitting is possible. When the algorithm terminates the blocks contain sets of equivalent states. In each iteration we examine all p inputs for each of the n states. There are no more than $n - 1$ rounds of splitting, since there are n states. Thus, the time complexity of the state partitioning algorithm is $O(pn^2)$.

The best known algorithm for minimization was elaborated by Hopcroft [Hop71]. It has a time complexity of $O(pn\log\, n)$.

2.1.4 Completeness Assumptions

An FSM is said to be incompletely defined if there are state-input combinations for which the transition function and/or the output function is not defined. There are three main approaches proposed in the literature to define behavior in this case. These are called completeness assumptions.

Let us consider a state s_x for which no transition and output functions are defined upon receiving input i_y.

Completeness Assumption 1 According to this approach, we add a unique output symbol Υ_o to the set of output symbols O and a unique state Υ_s to the set of states S denoting the null (error) output symbol and state, respectively. Then, for (s_x, i_y) the transition and output functions are defined as follows:

$\delta(s_x, i_y) = \Upsilon_s$

$\lambda(s_x, i_y) = \Upsilon_o$

This approach was first proposed in [SvB82].

Completeness Assumption 2 According to this approach, only a unique null output symbol Υ_o is introduced, and is used if the output function is not defined for a state-input pair. In case a transition function is undefined, however, it is assumed that the FSM remains in its present state. Thus, for (s_x, i_y) the transition and output functions are defined as follows:

$\delta(s_x, i_y) = s_x$

$\lambda(s_x, i_y) = \Upsilon_o$

This approach was first introduced by Sabnani et al. in [SD88].

Completeness Assumption 3 This approach interprets undefined transition and output functions as "don't care". In other words, for (s_x, i_y)

$\delta(s_x, i_y) = s_k$, where s_k is any state in S

$\lambda(s_x, i_y) = o_l$, where o_l is any output in O

Any partially specified machine augmented with completeness assumption 1 or 2 can be regarded as a completely specified, deterministic FSM. Thus, using these assumptions incompletely specified machines present no new problems. Note that the notion of strong conformance was proposed based on these first two approaches.

Approach 3, on the other hand, introduces non-determinism into the system behavior. This imposes new challenges in case of many testing problems. Weak conformance testing is based on completeness assumption 3 and requires algorithms different from the ones for completely defined machines.

2.2 Testing Finite State Machines

Most of the initial research on FSM theory was focused on the state minimization process which is closely related to system design and implementation, but the demand of system reliability motivated research into the problem of testing finite state machines to ensure their correct functioning. The research started in the 60s motivated mainly by automata theory and sequential circuit testing. Moore introduced the theoretical framework for FSM based testing in his paper [Moo56]. Later Hennie established the foundations of transition checking [Hen64]. Then, after being abandoned for a while, the research into FSM based testing has been resumed since the late 80s due to its application to conformance testing of communication protocols [LY96].

A testing problem involves a machine about which we are missing some information, and we have to deduce the missing information only by its I/O behavior, i.e., by introducing sequences of inputs and by observing the given outputs.

There are a number of testing problems for finite state machines, which can be categorized into two main types. In case of the first type of problem, it is presumed that the tester exactly knows the behavior – the state diagram – of the machine, but misses information on the actual state of the machine. In the second type of problem the machine under test is a black box, i.e., its behavior is not known.[1]

2.3 Testing Problems for Machines with Known Behavior

Here a tester is given a machine with known behavior, but in an unknown state, and the problem is to acquire information on the state of the machine based solely on I/O behavior. The inherit difficulty of these type of problems comes from the limited controllability and observability of FSMs. Limited controllability means that the FSM cannot be directly put into a desired state. Limited observability prevents the external tester from directly observing the state of the FSM.

There are two main approaches to the problem: both apply a test sequence first, then one tries to determine the final state of the machine after the test, while the other identifies the initial state before the test. The homing sequence and the synchronizing sequence problems follow the first approach. State identification and state verification are concerned with identifying the initial state before the test. Besides being interesting on their own, algorithms for these problems are also useful in solving black box testing problems like conformance testing.

2.3.1 Synchronizing and Homing Sequences

An input sequence is called a homing sequence if it is possible to determine the final state of an FSM after applying the sequence to it and observing the given outputs, regardless to the initial state before the test. Formally $x \in I^*$ is homing iff for all states $s_1, s_2 \in S$, $\delta(s_1, x) \neq \delta(s_2, x) \Longrightarrow \lambda(s_1, x) \neq \lambda(s_2, x)$. Clearly, only reduced machines have homing sequences, since it is impossible to distinguish equivalent states using any input sequence. On the other hand, every reduced machine has a homing sequence, and it can be created in polynomial time. Polynomial time algorithms, however, do not construct the shortest possible homing sequence. Moreover, the problem of finding the minimal length homing sequence turns out to be NP-hard [Epp90].

An input sequence is called a synchronizing sequence if it takes a machine to the same final state, regardless of the initial state or the outputs. Formally $x \in I^*$ is synchronizing iff $|\delta(S, x)| = 1$. Synchronizing sequences are special types of homing

[1]For practical reasons, however, it is usual to make some assumptions on the machine to be tested; these assumptions will be discussed later in detail. (See Section 2.4.1)

sequences, and not all reduced machines have them. There are polynomial time algorithms to determine if a machine has a synchronizing sequence and to construct one if it is possible. Finding the minimal length synchronizing sequence, however, is NP-hard.

2.3.2 State Identification and Verification

The other types of problems, state identification and state verification, try to provide information on the initial state of the machine before the test. The major difference between the two problems is that while the former tries to identify the state without making any assumptions about what it might be, the latter assumes that the machine is in a given state s_j and tries to verify it. An input sequence that solves the state identification problem is called a distinguishing sequence (DS). One that solves the state verification problem is called a unique input output (UIO) sequence.

State identification is obviously the harder problem, as a distinguishing sequence is a special UIO sequence which is able to verify all states of a machine. It has been shown that neither state identification nor state verification is always solvable [Gil61], i.e., there are machines for which no UIO – and thus no DS – sequence exists. It has also been proven that both the distinguishing sequence and the UIO sequence problems are PSPACE-complete [LY94].

2.4 Black Box FSM Testing Problems

In a black box testing problem a tester is given a machine with unknown behavior and has to deduce some missing information based on its I/O behavior. The two most important black box testing problems are conformance testing and fault diagnosis. In both cases a tester is given a black box implementation FSM $Impl$ and the complete description of another machine $Spec$, the specification machine.

Conformance testing (or fault detection) The problem is to determine if $Impl$ conforms to $Spec$.

Fault diagnosis (or fault localization) Given a specification machine $Spec$ and an implementation machine $Impl$ the problem is to locate the difference(s) between the two machines.

It is clear that fault diagnosis is a more complex problem than conformance testing. While in the case of conformance testing the tester "only" has to determine whether there are difference(s) between the specification and the implementation, fault diagnosis addresses the more complex problem of locating the difference(s) between the system specification and an implementation if they are found to be different.

2.4.1 The Conformance Testing Problem

Conformance testing – often simply called testing – is arguably the most important and widely applied testing approach. Conformance testing, in essence, is the process of running a system (executing a program) with the intent of finding faults. It is done by means of test sequences defining input sequences to stimulate the IUT (Implementation Under Test), and the corresponding correct output sequences. A faulty implementation is said to be detected if its execution against a test sequence distinguishes its external behavior (or output) from what is expected.

It should be emphasized that conformance testing – or testing in general – is not the process of showing that the system contains no faults. "Testing can show the presence of bugs, but never their absence" – as Dijkstra formulated it in a famous aphorism in [Dij72]. Systems typically accept an infinite number of input sequences, behave accordingly, and provide infinite number of output sequences. At the same time, the test set has to be finite for practical reasons. Thus, testing can usually only show that the system behaves correctly in some cases, and can only be used to establish some confidence that the system behaves as specified.

In FSM-based conformance testing the specification is given as a finite state machine *Spec* and the problem is to determine if the implementation machine $Impl$ conforms to *Spec*. In contrast to the general case a very important property of FSM-based conformance testing is that under certain well understood conditions it is possible to generate tests that do determine correctness. In other words, it is possible to create finite test sequences (complete test sets) that show if a black box implementation is equivalent to the specification machine, and thus behaves correctly for all – infinite – input sequences. There are a number of algorithms to create such test sequences. Before explaining these algorithms in detail we discuss the necessary assumptions.

The assumptions that are usually made in the literature for the test to be at all possible can be divided into three categories:

- Assumptions about the specification machine
- Assumptions about what correctness (conformance) is – conformance relations
- Assumptions about the implementation machine – fault models

2.4.2 Assumptions About the Specification Machine

There are various types of assumptions in the literature about the specification machine. All of the basic FSM conformance testing algorithms creating complete tests are working with completely specified, deterministic machines and are assuming that the specification FSM is strongly connected and reduced [LY96]. If the specification

machine is not strongly connected, and if in the testing experiment the implementation machine starts at a state that cannot reach some other states, then the test will not be able to verify all states of the machine, thus, it cannot be complete. The reason for requiring the specification machine to be reduced is that by testing one cannot distinguish equivalent machines anyway as they have the same I/O behavior. This requirement is not restrictive as we can always minimize the specification machine if it is not reduced.

A further trivial assumption is that the implementation machine does not change during the experiment, and has the same input I and output O alphabet as *Spec*.

Some of the algorithms use some additional assumptions, for example requiring that the specification machine has reliable/unreliable reset or reliable/unreliable status message.

2.4.3 Conformance Relations

A tester should have a set of rules to determine if the behavior of a given implementation conforms to the corresponding specification. Such a set of rules is called a conformance relation. There are several candidates for conformance relations in the literature. The two main types of relations are preorders and equivalences, but there are some others not belonging to them like **conf** [Bri88].

Preorder relations are reflexive and transitive. That is, they guarantee that any specification conforms to itself. Moreover, if a specification system conforms to a high-level specification, and an implementation conforms to the specification then the implementation also conforms to the high-level specification according to the transitivity property. There are a number of preorders defined in the literature. The most important are trace preorder [RHB97] and testing preorder [Tre94].

Equivalence relations are a special and more restrictive case of preorders, as they are not just reflexive and transitive but symmetric as well. All equivalences except bisimilarity are defined as the symmetric closure of a preorder. The most important types of equivalence relations have already been discussed in Section 2.1.2. They partition the set of possible systems into equivalence classes. Within an equivalence class any system conforms to any other system.

In the context of completely specified deterministic FSMs, equivalence relations are mainly used as conformance relations. Since all equivalence relations are the same, the conformance relation is simply the equivalence relation between FSMs. The conformance testing problem for completely specified deterministic FSMs can be stated as follows. We are given a black box implementation machine $Impl$ and the complete description of a specification machine *Spec*. Determine whether $Impl$ is equivalent to *Spec*.

2.4.4 Fault Models

A practical approach to testing should avoid working directly with hardware and software failures because of their large number and complexity. Instead, possible faults of the system should be considered, i.e., defects causing erroneous behavior which may lead to a failure of the system. A fault model is a hypothetical model of what types of faults may occur in an implementation.

Fault models are in principle used to limit the number of possible implementations considered for constructing or analyzing a test set [PvBY96], i.e., they are a way to express a tester's assumptions on the implementation machine to be tested. A fault model defines all the faults that are to be found, and the tester only considers implementations that are mutants of the specification according to that fault model.

A fault model is also useful to determine the efficiency of a test set [PvBY96] [SL88] [DS88]. Efficiency is usually measured in terms of the number of faults detected by the test set, compared to the total number of faults according to the fault model. This measure is called fault coverage.

Fault models for software systems and FSMs specifically have extensively been studied, see for example [SL89] [Pet01]. There are various fault models proposed for FSMs in general [BDD+96] [KPY99]. Most FSM fault models are "structural", i.e., define operations modeling possible alterations – faults – of the specification machine. Faulty implementation machines are considered mutations of the specification according to the operations defined by the fault model.

For completely specified and deterministic machines the most widely accepted model was defined by Chow [Cho78]. It includes the following three types of faults:

- Operation (Output) fault – for a given state and input, the implementation provides an output different from the one specified by the output function.
- Transfer fault – for a given state and input, the implementation enters a different state than specified by the transition function.
- Extra state / missing state – adding / removing a state.

A significant property of Chow's model is that it defines atomic faults, i.e., faults that are not a composition of other faults. Moreover, Chow's fault types do not create nondeterministic machines, i.e., the set of deterministic finite state machines with a given set of input and output symbols I and O is closed under Chow's faults.

Bochmann et al. [BDD+96] complemented Chow's model by a fourth type of fault: Extra transition - created by adding a transition. The faults in this model are atomic as well, however in most cases this kind of fault results in a nondeterministic FSM.

2.4.5 FSM Conformance Testing Methods

Here we will discuss some of the main FSM-based test generation methods, their complexity and their fault detection capability (fault coverage). Note that for fault coverage analysis the single fault assumption is used, i.e., only single faults are considered to evaluate efficiency. This assumption is necessary do to the fact that multiple faults may create equivalent machines that can not be detected. Single faults, however, can not result in equivalent machines if the specification machine is reduced.

All methods discussed here work on reduced, completely specified and deterministic machines, assuming FSM equivalence as a conformance relation and considering a fault model with transfer and output faults (some of the methods – or their extensions – also consider extra/missing states).

All of the different methods have the same fundamental structure: a tester has to make sure that all of the transitions of the FSM are implemented correctly according to the specification machine.[2]

To test all of the transitions one has to create a test sequence, which for every transition $s_i \xrightarrow{i_k/o_l} s_j$ takes the machine to state s_i inputs i_k, and verifies both the output function (check that the output generated is o_l) and the transition function (check that the next state is s_j). Such a test sequence is called a checking sequence. Given a completely specified deterministic FSM *Spec* with n states, a checking sequence for *Spec* is an input sequence x that distinguishes *Spec* from all other machines with n states. That is, any implementation machine $Impl$ with at most n states not equivalent to *Spec* produces an output different from *Spec* on input x.

There are a number of conformance testing methods developed for constructing checking sequences. They mainly differ in their assumptions about the specification machine and in the way of checking the next state of the FSM after a transition by distinguishing the expected state from any incorrect alternative.

In FSM-based conformance testing typically one of the following approaches is used to check the state of the machine.

Separating family of sequences A separating family of sequences for FSM M is a collection of n sets $Z_i, i = 1, ..., n$ of sequences (one set for each state) such that for every pair of states s_j, s_l there is an input sequence x that separates them, that is, $\exists x \in I^*$, $\lambda(s_j, x) \neq \lambda(s_l, x)$. Moreover, x is a prefix of some sequence in Z_j and some sequence in Z_l. Z_j is called the separating set of state s_j, and the elements of Z_j its separating sequences. Every reduced (completely specified, deterministic) FSM has a separating family of sequences.

[2]In the case of partially defined machines the unspecified transitions should behave according to the given completeness assumption. In the case of strong conformance testing – completeness assumption 1 or 2 in Section 2.1.4 – machines can be regarded as a completely specified FSMs.

Characterizing sequences Characterizing sequences (CS) for FSM M are a set of input sequences (characterizing set or W-set) that can distinguish between every pair of states of M. A characterizing set is a special case of the separating family of sequences where all the sets Z_i are identical. For every reduced and completely specified finite state machine a characterizing set can be created with no more than $n-1$ sequences of length less than n. A method for constructing minimal characterizing sets can be found in [Gil62].

UIO sequences A UIO sequence y_j for a state s_j of an FSM M is a sequence able to verify if the machine is in state s_j, i.e., $\lambda(s_j, y_j) \neq \lambda(s_k, y_j)$ for any $s_k \in S$. (See Section 2.3). Only reduced FSMs may have UIO sequences, and nearly all (but not all) reduced FSMs have UIO sequences for each state. A set of UIO sequences is is a special case of the separating family of sequences where all the sets are singletons – if they do not violate the prefix condition of the separation property.

Distinguishing sequences A distinguishing sequence of FSM M is an input sequence z which is able to identify any state of M, i.e., $\lambda(s_j, z) \neq \lambda(s_k, z)$ for every pair s_j, s_k. (See Section 2.3). A distinguishing sequence is a special case of the separating family of sequences where all the sets are identical and singletons. Only reduced FSMs may have distinguishing sequence, however not every reduced machine has a distinguishing sequence. A distinguishing sequence can be considered as an unreliable status message.

Status messages A status message of FSM M is a special status input, and upon receiving this input the machine outputs its current state and stays there. The status message is reliable if it is guaranteed to work in any implementation machine. (See Section 2.1.1).

TT-method

The transition tour method (TT-method or T-method) [NT81] guarantees that a test is generated which traverses all the transitions in the specification machine. It requires the specification machine to be strongly connected. The TT-method is only able to detect all the output faults but it provides no guarantees on detecting transfer faults. That is, the test set created with the TT-method is not complete with respect to a fault model consisting of transfer and output faults. The problem of generating a minimum-cost test sequence using the transition tour method is equivalent to the Chinese Postman problem in graph theory, i.e., the problem asking for the shortest tour of a graph which visits each edge at least once. There are polynomial-time algorithms for the Chinese Postman problem for connected graphs.

D-method

The distinguishing sequence method [Gon70] assumes that the specification FSM *Spec* has a distinguishing sequence x. It checks the implementation machine $Impl$ in two phases:

Phase 1 The first phase checks whether all the states of the specification $s_i, i = 1...n$ are correctly implemented by $Impl$ using a sequence $x \cdot \tau(t_1, s_2) \cdot x \cdot \tau(t_2, s_3) \cdot ... \cdot x \cdot \tau(t_n, s_1) \cdot x$, where "$\cdot$" is the string concatenation operator, $t_i = \delta(s_i, x)$, and $\tau(s_i, s_j)$ is a transfer sequence taking *Spec* from s_i to s_j. This sequence takes the $Impl$ through all of its states and checks all of them, applying the distinguishing sequence x.

Phase 2 The second phase checks every transition. If the FSM $Impl$ is in state s_m, transition $s_i \xrightarrow{i_k/o_l} s_j$ is checked by sequence $\tau(s_m, s_{i-1}) \cdot x \cdot \tau(s_{i-1}, s_1) \cdot i_k \cdot x$. The machine is not directly taken from s_m to s_i since faults could alter the ending state s_i. Instead, sequence $\tau(s_m, s_{i-1}) \cdot x \cdot \tau(s_{i-1}, s_1)$ is used, taking $Impl$ to s_{i-1}, verifying it, then using the already checked $\tau(s_{i-1}, s_1)$ sequence to take the machine to state s_1.

The D-method is guaranteed to detect any output and/or transfer faults. The length of the checking sequence is polynomial in the size of the FSM *Spec* and the length of the distinguishing sequence x. A problem is, however, that in practice very few FSMs actually possess a distinguishing sequence.

Note that machines with unreliable status messages can be tested with the D-method, using the status message as the distinguishing sequence. For machines with reliable status message the problem becomes much simpler: A checking sequence can be generated by constructing a covering path of the transition diagram of *Spec* and applying the status message at each state visited.

W-method

For machines with reliable reset Chow has proposed a method in [Cho78] often referred to as the "W-method". Chow's method is based on a characterizing set or W-set (but any separating family of sets could be used as well).

A P-set of an FSM *Spec* is any set of input sequences such that for every transition of *Spec* from state s_j to state s_k on input i_x, there are input sequences y and $y \cdot i_x$ in P such that y takes the machine from the initial state into state s_j. A way to construct such a set is to build a testing tree of the transition diagram of the specification machine. Each branch of the testing tree is labeled with an input symbol.

A testing tree T for finite state machine *Spec* can be built inductively as follows. Let the root node of T be labeled by the initial state of *Spec*. This is the first level.

Let us assume that we already built T to level k. The $(k+1)^{th}$ level is created by examining the nodes of level k. A node at the k^{th} level is terminated if a node labeled by the same state has already appeared at a level j, $j \leq k$ as a non-terminal node. Otherwise, for each of the state's transitions in *Spec* create a branch in T labeled with the corresponding input symbol, and with successor node labeled by the corresponding next state. As a result the testing tree includes each transition of the machine exactly once. The P-set consists of all partial paths of the testing tree, i.e., all consecutive branches from the root of the tree to a node (a state). There are $np+1$ partial paths with a maximum length of n.

The set of test sequences – parts of the checking sequence – are created by the concatenation of the sets of sequences P and W. The checking sequence starts at the initial state (first a reset input is applied) and consists of no more than pn^2 test sequences of length less than $2n$ interposed with reset. Thus the total complexity of the algorithm is $O(pn^3)$, where $p = |I|$ and $n = |S|$.

The W-method is guaranteed to detect any output and/or transfer faults. It can also be extended to detect any extra state faults (if the extra states are reachable from the initial state). Let us consider that specification machine *Spec* has n states and implementation machine $Impl$ has m states. If $m > n$, that is, if the implementation machine has more states than the specification, then – according to Chow – we have to use a Z set instead of a W set for test sequence generation. A Z set can be created by extending the W set the following way [Cho78]:

$$Z: \; W \; U \; I \cdot W \; U \; \ldots \; U \; I^{n-m} \cdot W$$

Where "U" is the union operator, "$\cdot$" is the string concatenation operator and I is the input alphabet. The checking sequence is then created by the concatenation of the sets of sequences P and Z.

Further FSM Test Generation Methods

Wp-method Chow's W-method has been improved to the so called Wp-method in [FvBK+91]. In this method, the state checking phase relies on subsets of the characterization set W instead of using the whole for each state, i.e., it is using a minimal separating family of sets.

UIO-method The Unique Input Output method or UIO-method was proposed in [SD88]. The method assumes that the specification FSM *Spec* has reliable reset capability, and that all of its states have UIO sequences $x_s \forall s \in S$. The method itself is quite similar to the W-method. For every transition $s_i \xrightarrow{i_k/o_l} s_j$ a test sequence $r \cdot \tau(s_0, s_i) \cdot i_k \cdot x_j$ is generated, where r is the reset input, $\tau(s_0, s_i)$ denotes some input sequence that takes the machine from the initial state to s_i

and x_j is the UIO sequence for state s_j. The UIO-method does not guarantee that all output and transfer faults are revealed. In [VCI89] an improvement to the original UIO method was proposed called the UIOv-method. Here an additional step has been applied to verify the states when reset transitions are present. The UIOv-method provides full fault coverage for both output and transfer faults. The length of the generated checking sequences is polynomial in the size of the FSM *Spec* and the length of the UIO sequences. Note that UIO-sequences do not have a polynomial upper bound [LY94], but in many applications FSMs do have short UIO-sequences for their states [SD88].

Identifying sequences All of the previous methods assumed some special property of the specification FSM like the existence of status message, distinguishing sequence, UIO sequence or reset capability. For general machines without any of these properties a test generation method was proposed in [Koh78]. The method creates so called identifying sequences for all states of FSM *Spec* based on the state's separating set.[3] With the identifying sequences a checking sequence is generated in two phases similarly to the D-method. First every state is verified using the identifying sequences, then all of the transitions are checked. The identifying sequences based method provides full fault coverage for both output and transfer faults. The length of an identifying sequence, however, increases exponentially with the number of separating sequences of a state. As a state may have up to $n-1$ separating sequences, the length of the generated checking sequence is in general exponential in n, where $n = |S|$.

2.4.6 Fault Diagnosis

Conformance testing – in general – provides the means to check whether a black box system behaves according to its specification. Accordingly, the objective of FSM-based conformance testing is to detect any faults, i.e. to find any differences between the specification and the black box implementation machines.

Fault diagnosis – in contrast – addresses the more complex problem of locating the difference(s) between the system specification and an implementation if they are found to be different. A solution to this problem has various applications [LY96]. One of the most important is the correction of an implementation so that it conforms to its specification.

Much research has been done concerning fault diagnosis for different formalisms like EFSMs [EFPYvB03], non deterministic FSMs [GDvB92], and CFSMs (communicating finite state machines) [GvBD93a] [GDvB93]. Most of the research, however, focused on deterministic finite state machines.

[3] Every reduced (completely specified, deterministic) FSM has a separating family of sequences.

All papers on fault diagnosis for finite state machines are considering a fault model with two types of faults – output faults and transition faults – but are using different restrictions on the cardinality of faults.

There are two papers on fault diagnosis for deterministic FSMs using the assumption that the implementation contains only one – transition or output – fault. There is a heuristic procedure presented for the diagnosis of single faults in finite state machines in [GvB92]. An exact fault localization procedure is reported by D. Lee and K. Sabnani capable of locating a single fault in a finite state machine in [LS93]. Limiting the number of differences between the specification and the implementation to a single fault, both of these papers guarantee the precise localization of the difference.

Other contributions consider the case of multiple faults. Procedures for diagnosing multiple faults in FSMs and CFSMs were reported as well [GvBD93b] [EFYvB01]. These algorithms are not always able to locate the multiple faults of the implementation [EFYvB01]. The multiple fault diagnosis method for FSMs only guarantees the correct diagnosis of certain configurations of faults in an implementation, which are characterized by a certain type of independence of the different faults [GvBD93b].

Both single fault detection algorithms [GvB92] [LS93] use the same assumptions – the specification FSM is deterministic, completely specified, strongly connected and reduced with reliable reset capability – and have the same fundamental structure.

Given a specification FSM *Spec* and a black box implementation machine $Impl$, first a fault detection (or conformance testing) experiment is conducted to detect the presumed single fault in $Impl$. Both papers apply complete checking sequences that consist of test sequences interposed with reset (generated for example by the W-method). Since the checking sequence is complete with respect to a fault model consisting of transfer and output faults, at least one of the input sequences will detect the fault, i.e., at some transition it will provide an output different from the one expected. This is called a "symptom" in [GvB92]. A symptom at a transition indicates either that the output function of the given transition is faulty, or that an earlier transition fault may have lead the system to an incorrect state. Thus, we have a set of possibly faulty transactions, which is then reduced as far as possible – the two methods use different approaches to do that.

As a second step, additional testing is necessary to exactly identify the fault based on the results of the first step. The method in [GvB92] proposes the use of a "limited characterization set" at each considered transition, which is able to distinguish between the possible faulty next states and the correct next state. The complexity of the algorithm is $O(nL_sL_c^3)$, where n is the number of states in the specification machine *Spec*, L_s is the number of the initial test sequences and L_c is the length of the longest test case.

The paper [LS93] suggests two solutions for the second step. The brute force approach relies on the creation of a mutant machine for each possible fault by modifying

Spec according to the supposed fault. Then a checking experiment (conformance testing) is conducted on $Impl$ with respect to each candidate machine. The complexity of the brute force algorithm is $O(pn^5)$, where p is the number of inputs at each state ($p = |I|$) and n is the number of states in the specification machine *Spec* ($n = |S|$).

The fast algorithm, on the other hand, uses cross-verification to rule out the wrong candidate machines. Given two candidate machines an input sequence x is generated, such that the two machines have different output responses to x. Then x is applied to $Impl$. According to the outputs produced by the implementation, either one or both of the candidate machines can be ruled out. Finally, in a confirmation phase one checking experiment is conducted on $Impl$ with respect to the remaining candidate machine. The complexity of the fast algorithm is $O(pn^3 \log n)$, where $p = |I|$ and $n = |S|$.

2.5 Conclusion

In this chapter we introduced the fundamental concepts and notations of finite state machine-based system modeling, including state and machine equivalence, isomorphism and minimization. We presented the most important types of testing problems for finite state machines, and gave an overview of state-of-the-art test generation methods and techniques solving them.

Chapter 3

Feasibility of the FSM-based Fault Diagnosis Problem

In this chapter we study the problem of FSM-based fault diagnosis, i.e., localization of difference(s) between an implementation and a specification in systems modeled by finite state machines (see Section 2.4.6), and analyze the feasibility of the problem. We concentrate on the diagnosis of a single transition or output fault in an FSM. We show that implementation machines with different single faults may have the same observable behavior, and consequently – contrary to the statements found in the literature – it is not always possible to precisely diagnose a single fault in a finite state machine.

We determine a set of sufficient conditions for the guaranteed exact localization of a single output or transfer fault. Based on the analytical results we give an algorithm, a modified version of Lee's procedure [LS93], for the fault diagnosis problem. If it is possible, the method exactly locates the difference between the implementation and the specification. If exact localization is infeasible it provides the minimal set of all potential single faults.

3.1 Failure of Exact Fault Diagnosis in FSMs

We show that even in the least complex case it is not always possible to solve the fault localization problem. That is, even when considering the strictest assumptions – a single fault in a finite state machine (investigated by Ghedamsi et al. [GvB92] and Lee et al. [LS93]) – there are some situations where the exact localization of the fault cannot be assured.

For the rest of this chapter we will consider a specification finite state machine $Spec = (I, O, S, \delta, \lambda)$. We denote the number of states, inputs, and outputs by $n = |S|$, $p = |I|$, and $q = |O|$, respectively as defined in section 2.1.1. We also consider

implementation machines $Impl_a = (I, O, S', \delta', \lambda')$, $Impl_b = (I, O, S'', \delta'', \lambda'')$ and so on, with the same input and output sets, and the same number of equally labeled states. We use the term "same states" written $s'_j = s''_j$ for states that are labeled alike in different machines. Of course, these states are not necessarily equivalent, written $s'_j \cong s''_j$.

Obviously, without any assumptions fault diagnosis and conformance testing are impossible problems [LY96] (see Section 2.4.1). We use the assumptions usually made in the literature. We deal with reduced, completely specified and deterministic specification machines. We assume FSM equivalence as a conformance relation and consider a fault model with transfer and output faults. Furthermore, we concentrate on systems with reliable reset capability, and assume that there is only one difference – an output or a transfer fault – between an implementation and the specification machine.

All previous works on the diagnosis of a single fault in a FSM – [GvB92] [LS93] – used the same assumptions, and they claim to provide methods to precisely locate the single fault.

We show that – contrary to the statements found in the literature – it is not always possible to precisely diagnose a single fault in a finite state machine, not even considering the assumptions above.

Proposition 3.1.1. *The exact localization of a single – transition or output – fault in a finite state machine cannot always be guaranteed, not even considering a deterministic, completely specified, strongly connected and reduced specification machine with reliable reset capability.*

Proof. Take specification machine *Spec* and two implementation machines: $Impl_a$ differing from *Spec* by a single fault $Fault_a$, and $Impl_b$ differing from the specification by a single fault $Fault_b$. $Fault_a$ and $Fault_b$ are different faults. Evidently, neither $Impl_a$ nor $Impl_b$ can be equivalent to *Spec*, since *Spec* is deterministic, completely specified, strongly connected and reduced. $Impl_a$ and $Impl_b$, however, may be equivalent to each other despite the fact that the faults they contain differ. In this case it is impossible to decide between the faults, i.e., it is impossible to exactly locate the fault. □

The next simple example demonstrates the situation.

Example 3.1.2. Take specification machine *Spec* shown in Figure 3.1. The set of input symbols is $I = \{a, r\}$, where r is the reset input, the set of output symbols is $O = \{1, 2\}$ and the set of states is $S = \{s_0, s_1, s_2\}$ where s_0 is the initial state. Specification machine *Spec* is deterministic, completely specified, strongly connected and reduced. Note that the (reliable) reset transitions are omitted on the figure for the sake of perspicuity.

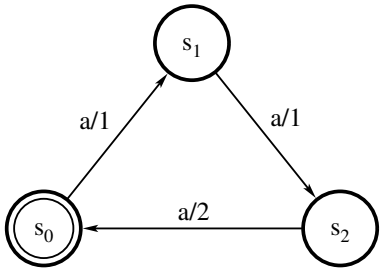
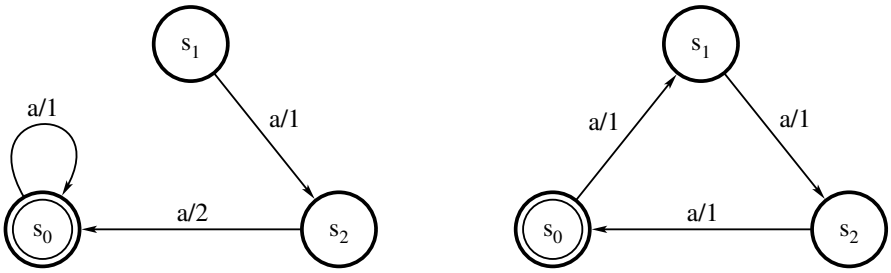


Figure 3.1: Specification machine *Spec*.



(a) $Impl_a$ (b) $Impl_b$

Figure 3.2: Faulty implementation machines: (a) $Impl_a$ contains a single transfer fault at state s_0, (b) $Impl_b$ contains an output fault at s_2.

Let us consider two implementation machines $Impl_a$ on Figure 3.2(a) and $Impl_b$ on Figure 3.2(b) with the same input and output alphabet as *Spec*.

The difference between $Impl_a$ and *Spec* is a single transfer fault at state s_0, $Fault_a : \delta'(s'_0, a) = s'_0$ instead of s'_1. In case of $Impl_b$ the difference is a single output fault at s_2, $Fault_b : \lambda''(s''_2, a) = 1$ instead of 2.

The two implementation machines are equivalent, as they both produce the same trace for every input string. Thus, it is impossible to distinguish between them and therefore between the two faults. Or, to formulate more precisely, it is impossible to distinguish among any faulty implementation machines belonging to the same equivalence class, and therefore among the faults that they contain.

3.2 Conditions for Guaranteed Fault Diagnosis

We determine a set of sufficient conditions for the guaranteed exact localization of a single output or transfer fault. That is, we analyze when two (or more) implementation machines, each differing from the specification by a single dissimilar fault, cannot be equivalent. Note that we still consider the assumptions made in the previous section, therefore the specification FSM *Spec* is deterministic, completely specified, strongly connected and reduced with reliable reset.

First we show that it is always possible to distinguish two different output faults if the specification machine has reliable reset capability.

Lemma 3.2.1. *Suppose that the specification machine under consideration is deterministic, completely specified, strongly connected and reduced with reliable reset capability. Two implementation machines, each differing from the specification by a single and dissimilar output fault, cannot be equivalent, thus any two output faults can be distinguished.*

Proof. Let us consider two implementation machines $Impl_a$ and $Impl_b$, both differing from the specification *Spec* by a single dissimilar output fault. We reset the machines and start to explore the state-space of the two implementations and the specification in parallel. Clearly, until we reach a faulty transition in one of the machines, for any input string x, the traversed states – and the output sequences – are the same in the two implementations and the specification: $\delta'(s'_0, x) = \delta''(s''_0, x) = \delta(s_0, x)$, and $\lambda'(s'_0, x) = \lambda''(s''_0, x) = \lambda(s_0, x)$. When we traverse a faulty transition in one of the implementations (let's say $Impl_a$) with an input string y, we find an inconsistency between $Impl_a$ and *Spec*: $\lambda'(s'_0, y) \neq \lambda(s_0, y)$. However, since $Fault_a \neq Fault_b$ the output of $Impl_b$ at the given transition cannot be equivalent to the output of $Impl_a$. Therefore, $\lambda'(s'_0, y) \neq \lambda''(s''_0, y)$, i.e., $Impl_a$ and $Impl_b$ are inequivalent, and input sequence y can distinguish them. □

Note that the statement made in Lemma 3.2.1 only holds if the specification machine has reliable reset capability. We demonstrate a counter-example of Lemma 3.2.1 in case the specification machine does not have a reliable reset capability.

Example 3.2.2. Take specification machine *Spec* shown on Figure 3.3

The set of input symbols is $I = \{a\}$, the set of output symbols is $O = \{1, 2\}$ the set of states is $S = \{s_1, s_2, s_3, s_4\}$.

Specification machine *Spec* is deterministic, completely specified, strongly connected and reduced. Take the two implementation machines $Impl_a$ in Figure 3.4(a) and $Impl_b$ in Figure 3.4(b) with the same input and output alphabet as *Spec*.

The difference between $Impl_a$ and *Spec* is a a single output fault at state s_3, $Fault_a$: $\lambda'(s'_3, a) = 2$ instead of 1. In the case of $Impl_b$ the difference is a single

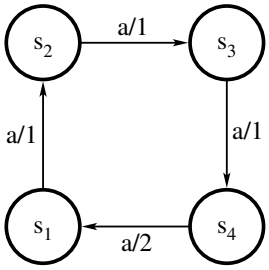
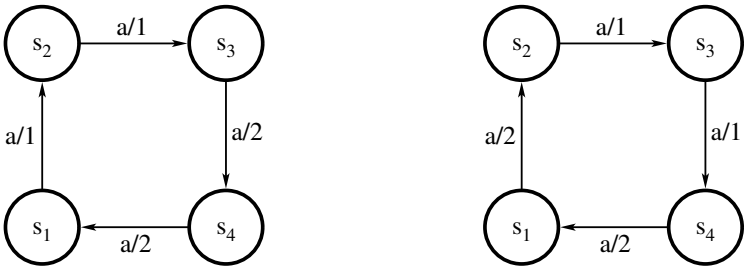


Figure 3.3: Specification machine *Spec* without reliable reset capability.



(a) $Impl_a$ (b) $Impl_b$

Figure 3.4: Faulty implementation machines: (a) $Impl_a$ contains a single output fault at state s_3, (b) $Impl_b$ contains an output fault at s_1.

output fault at s_1, $Fault_b : \lambda''(s_1'', a) = 2$ instead of 1. The two implementation machines are clearly equivalent.

Next we show that it is always possible to distinguish a single output and a single transfer fault if the faulty machines are reduced.

Lemma 3.2.3. *Suppose that the specification machine under consideration is deterministic, completely specified, strongly connected and reduced with reliable reset capability. Two implementation machines, one differing from the specification by a single output fault, the other by a single transfer fault cannot be equivalent if the implementation machines are reduced.*

Proof. Let us consider two implementations $Impl_a$ and $Impl_b$. One of the implementations (let's say $Impl_a$) contains a transfer fault, the other ($Impl_b$) an output fault. The two implementation machines are reduced therefore they must be isomorphic to be equivalent. That is, there has to be a one-to-one mapping ϕ from S' to S'' such that for every state s' in S' and for every input symbol i in I, $\delta''(\phi(s'), i) = \phi(\delta'(s', i))$ and $\lambda''(\phi(s'), i) = \lambda'(s', i)$ should hold.

Let's say the output fault in $Impl_b$ is at s''_x. Since the output of one of its transitions has changed, s''_x has to map to an other state (say s'_y) of $Impl_a$ where $\lambda''(s''_x, i) = \lambda'(s'_y, i),\ \forall i \in I$. However, this mapping cannot be one-to-one, as there is no output fault in $Impl_a$, and therefore $Impl_a$ has one less state with the same output characteristic as s''_x. Thus, there is clearly no one-to-one mapping ϕ fulfilling $\lambda''(\phi(s'), i) = \lambda'(s', i),\ \forall s' \in S',\ \forall i \in I$. □

Finally we show that it is always possible to distinguish two different single transfer faults if the faulty machines are reduced.

Lemma 3.2.4. *Suppose that the specification machine under consideration is deterministic, completely specified, strongly connected and reduced with reliable reset capability. Two implementation machines, each differing from the specification by a single and dissimilar transfer fault, cannot be equivalent if the implementation machines are reduced.*

Proof. Let us assume the following situation:

$Fault_a$ in $Impl_a$: $(s_c \xrightarrow{i_l/o_f} s_d) \Rightarrow (s'_c \xrightarrow{i_l/o_f} s'_e)$

$Fault_b$ in $Impl_b$: $(s_u \xrightarrow{i_m/o_g} s_v) \Rightarrow (s''_u \xrightarrow{i_m/o_g} s''_w)$

First, take the special case when the two faults are applied to the same transition in the two implementations, i.e., $c = u$ *and* $l = m$. In this case the outputs are the same ($f = g$) but the next states are not ($e \neq w$) because $Fault_a$ and $Fault_b$ are dissimilar. In this case there is only one difference between $Impl_a$ and $Impl_b$, and therefore, if the implementation machines are reduced ($s'_e \ncong s'_w$), we can certainly find an input sequence distinguishing them for example using Chow's method [Cho78]. Let y be a separating sequence distinguishing states s'_e and s'_w. We apply an input sequence (say x) corresponding to the path of the tree from the initial state to s'_c, input i_l and then apply y. This input sequence $x \cdot i_l \cdot y$ certainly distinguishes $Impl_a$ and $Impl_b$, thus the implementation machines cannot be equivalent.

Now take the general case when the two faults are applied to different transitions. Let's reset the machines and start to explore the state-space of the two implementations and the specification in parallel. Until we reach a faulty transition in one of the machines for any input string the traversed states – and the output sequences – are the same in the two implementations and the specification. Let's say we first encounter $Fault_a$ in $Impl_a$ with an input string x, i.e., with x we reach the state s'_c in $Impl_a$, s''_c in $Impl_b$ and s_c in *Spec*. If we input i_l after x, $Impl_a$ will transit to s'_e, *Spec* to s_d and $Impl_b$ to s''_d. Let Y be the set of all separating sequences distinguishing states s_e and s_d. Any input sequence $x \cdot i_l \cdot y_j$ where $y_j \in Y$ will clearly distinguish $Impl_a$ and *Spec*. Any of these input sequences will distinguish $Impl_a$ and $Impl_b$ as well, except if all $y_j \in Y$ starting from s''_d in $Impl_b$ traverse $Fault_b$ making $\lambda(s'_0, x \cdot i_l \cdot y_j)$ and $\lambda(s''_0, x \cdot i_l \cdot y_j)$ consistent for all $y_j \in Y$. For that, in *Spec* all

separating sequences distinguishing states s_e and s_d starting from s_d traverse transition (s_u, i_m); and if $\delta(s_u, i_m) = s_v$ then s_e and s_d are separable, if $\delta(s_u, i_m) = s_w$ then they are not separable. From that it follows that s''_e and s''_d in $Impl_b$ are not separable. Thus, the implementation machines cannot be minimal – we reached a contradiction. □

Theorem 3.2.1. *Suppose that the specification machine under consideration is deterministic, completely specified, strongly connected and reduced with reliable reset capability. Two implementation machines, each differing from the specification by a single and dissimilar fault, cannot be equivalent if the implementation machines are reduced.*

Proof. The proof follows from Lemmas 3.2.1, 3.2.3 and 3.2.4. □

The theorem shows that if there is only one difference between an implementation and a specification and the implementation is minimal then it is unique; no other fault can induce the same change in behavior. Thus it is possible to identify the given fault. Note that Lemmas 3.2.3 and 3.2.4 and Theorem 3.2.1 are studying the properties of the implementation FSM. As the implementation machine is black box, these results can not be directly utilized, but they can be used to improve the efficiency of the modified fault diagnosis algorithm.

3.3 Exact Algorithm for Fault Diagnosis

We give an algorithm – a modification of Lee's method [LS93] – for the localization of single transfer or output faults in finite state machines. We incorporate the analytical results of Section 3.2 to quickly verify if the first fault candidate the algorithm identifies is certainly the only possible one. If it is, then the fault is diagnosed and the algorithm terminates; otherwise the algorithm moves on and provides the minimal set of all potential single faults.

Let us consider a specification finite state machine *Spec* and an implementation *Impl* to be diagnosed. The algorithm is made up of two steps:

Step 1 Conformance testing is used to determine if there is difference between the specification and the diagnosed implementation.

Step 2 Localization of the fault.

3.3.1 Step 1: Detection of the Fault

For Step 1 of the algorithm a checking sequence needs to be constructed. There are a number of methods to generate checking sequences that are complete with respect to a fault model consisting of transfer and output faults (see Section 2.4.5).

In our algorithm we create the checking sequence using the W-method proposed by Chow for machines with reliable reset [Cho78]. It consists of no more than pn^2 test sequences of length less than $2n$ interposed with reset. We apply the checking sequence to the specification and to the diagnosed implementation. If we do not find an inconsistency of the observed outputs then we conclude that the implementation machine is equivalent to the specification, thus there is either no fault in the implementation or there are more than one; end of the algorithm. If we find a difference we move on to Step 2.

3.3.2 Step 2: Localization of the Fault

During conformance testing an inconsistency was found between the specification and the diagnosed implementation. Thus, there is at least one of the pn^2 test sequences (say x) detecting the fault, i.e. $\lambda(s_0, x) \neq \lambda'(s'_0, x)$. Let us assume that the earliest inconsistency between $\lambda(s_0, x)$ and $\lambda'(s'_0, x)$ is at the k^{th} output symbol where $1 \leq k \leq 2n$. Let's say that the first k elements (inputs) of x carry the specification machine from s_0 to $s_1, s_2, ..., s_k$, where these $k + 1$ states may or may not be different.

We assume $Impl$ has only a single output or transfer fault. In case the diagnosed implementation machine contains an output fault, x has to traverse the fault at the k^{th} transition. If $Impl$ contains a transfer fault, then x has to traverse the fault during the first $k - 1$ transitions.

Note that if there is more than one test sequence detecting the fault, we may use any of them for the localization of the fault (for practical reasons we should choose the shortest sequence). If multiple test sequences detect the fault, we might as well check if the set of possibly faulty transitions can be narrowed. For each test sequence detecting the fault, we determine the transitions it traverses in the specification machine prior to the first inconsistency. Trivially, in Step 2, we only have to consider the intersection of these traversed transitions.

In the algorithm we consider two cases. First we presume that the fault in $Impl$ is an output fault and verify if it is a potential candidate. If the verification succeeds, then we try to confirm whether it is the only potential candidate. If it is the only one, then we located the fault and end the algorithm. Otherwise we move on and presume that the fault could be a transfer fault, and similarly analyze each possibility. If at the end we do not find any potential candidates we conclude that there is more than one fault in $Impl$.

Step 2.1 – Output Fault

We assume that the fault in $Impl$ is an output fault, i.e., $\lambda(s_{k-1}, x_k) \neq \lambda'(s'_{k-1}, x_k)$ where x_k is the k^{th} input of x. For the verification we modify *Spec* according to the supposed fault: we change the output symbol at state s_{k-1} upon input x_k to the faulty output symbol $\lambda'(s'_{k-1}, x_k)$. We denote the modified specification C_1. We conduct a checking experiment (conformance testing) on $Impl$ with respect to C_1.

C_1, however, is not necessarily minimal. To use Chow's method for checking sequence generation, we first have to minimize machine C_1 and get $C_{1\ reduced}$. Let m be the number of states of reduced machine $C_{1\ reduced}$. If $m < n$, that is, if the reduced conjectured machine has less states than the specification, then according to Chow we have to use a Z set instead of a W set for test sequence generation. The checking sequence is then created by the concatenation of the sets of sequences P and Z.

If we find that $Impl$ conforms to C_1, we conclude that C_1 is a potential candidate. Then we try to confirm if it is the only possible candidate. For that we simply have to check if the reduced machine $C_{1\ reduced}$ has an equivalent number of states to the specification. If it has, we conclude that C_1 is certainly the only potential candidate, and therefore we exactly located the fault in $Impl$, end of algorithm.

If $Impl$ conforms to C_1, but the reduced machine has less states than the specification, we conclude that C_1 is a potential candidate, store it in the set of potential candidate machines PC, and proceed to the following step.

If $Impl$ does not conform to C_1 we proceed to the following step.

Step 2.2 – Transfer Fault

A transfer fault can occur in one of the first $k-1$ transitions, i.e., $\delta(s_j, x_{j+1}) \neq \delta'(s'_j, x_{j+1})$ where $j = 0, ..., (k-2)$. We assume that the fault occurs in the j^{th} transition and verify each assumption in turn. On input x_{j+1} at state s_j the implementation machine is supposed to transit to s_{j+1}. But instead $Impl$ transits to s_r, where s_r can be any of the $n-1$ states except the right state s_{j+1}. We verify each possibility in turn.

In each turn we modify *Spec* according to the supposed fault: we create candidate machine C_{l+1} where l is the cardinality of the set PC, by changing the next state symbol of the given transition to the supposed wrong state s_r. We minimize the candidate machine and conduct a checking experiment on $Impl$ with respect to $C_{l+1\ reduced}$.

If we find that $Impl$ conforms to $C_{l+1\ reduced}$, we conclude that C_{l+1} is a potential candidate. If PC is not empty ($l \geq 1$), we store C_{l+1} in PC, conclude that the exact localization of the fault is not possible, and move on to the next turn.

If $l = 0$ then we try to confirm if it is the only possible candidate. We simply

check if the reduced machine $C_{l+1\ reduced}$ has an equivalent number of states to the specification. If it has, we conclude that C_{l+1} is certainly the only candidate, and therefore we exactly located the fault in $Impl$, end of algorithm.

If $l = 0$ and $Impl$ conforms to C_{l+1}, but the reduced machine has less states than the specification, we conclude that C_{l+1} is a potential candidate, store it in set PC, and proceed to the next turn.

If $Impl$ does not conform to C_{l+1} we move on to the next turn.

For each assumed transition there are $n-1$ possible next states. Thus, there are no more than $2n^2$ turns. At the end of the last turn there are three possibilities:

- If $l = 0$, we conclude that there is more than one fault in $Impl$, end of algorithm.
- If $l = 1$, there is only one potential candidate, therefore we exactly located the fault in $Impl$, end of algorithm.
- If $l > 1$, we conclude that the exact localization of the fault is not possible, and PC is the set of all potential candidates, i.e., we determined the minimal set of all potential single faults, end of algorithm.

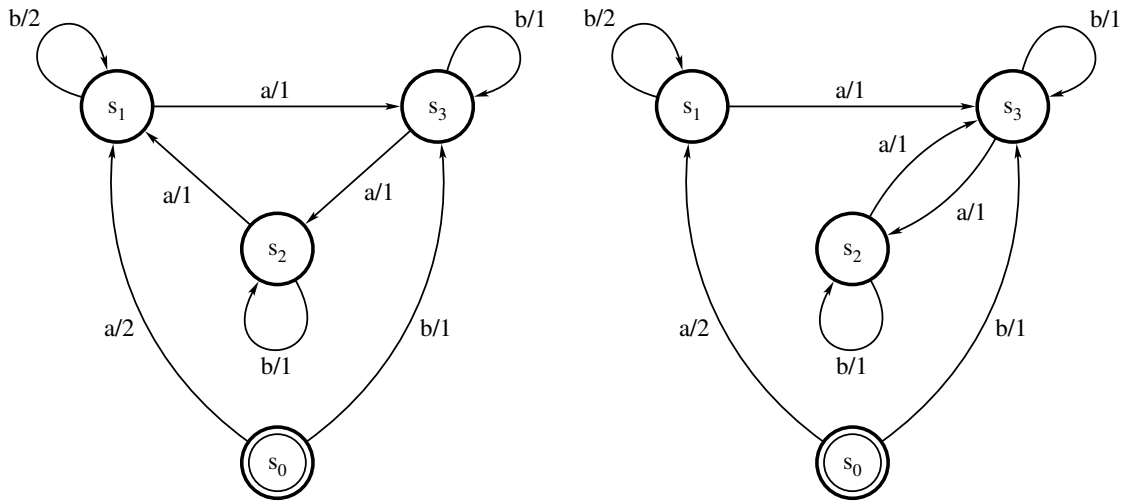
The complexity of the algorithm – in the worst case – is the same as the complexity original method [LS93]. The first step, the detection of the fault, is a checking experiment with a run time $O(pn^3)$ where $p = |I|$ and $n = |S|$. Step 2.1 – assuming that the fault is an output fault – consists of a single checking experiment with a run time $O(pn^3)$. In step 2.2 – assuming that the fault is an output fault – there are at most $2n$ possibly faulty transitions to consider as $2n$ is the maximal length of a test sequence. For each transition there are $n-1$ possible wrong next states. Thus, the second part of the second step consists of no more than $2n^2$ checking experiments with a run time $O(pn^3)$ each. Thus, the total complexity of the algorithm is $O(pn^5)$.

Example 3.3.1. We use an example to demonstrate the algorithm given above. Take the specification machine *Spec* shown on Figure 3.5(a) The set of input symbols is $I = \{a, b, r\}$, where r is the reset input, the set of output symbols is $O = \{1, 2\}$ and the set of states is $S = \{s_0, s_1, s_2, s_3\}$ where s_0 is the initial state. Reset transitions are again omitted in the figure.

The implementation machine in Figure 3.5(b) contains a single transfer fault at state s_2. This transfer fault is to be located using the algorithm.

For the detection of the fault (step 1 of the algorithm) we need to construct a checking sequence. Since our emphasis is not on checking experiments, we omit the details. A P-set of *Spec* can be constructed based on a testing tree.

$$P: \ \{r, ra, raa, rab, rb, rba, rbb, rbaa, rbab\}$$



(a) Specification machine *Spec* (b) Implementation machine *Impl*

Figure 3.5: Faulty implementation machine $Impl$ contains a single transfer fault at state s_2.

The characterizing set (W-set) of *Spec* is:

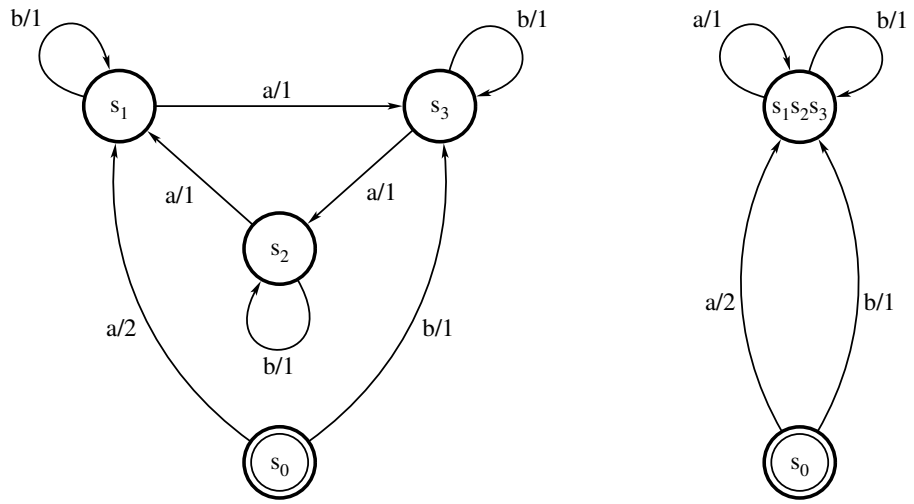
$$W : \{ab, b\}$$

By concatenating P and the characterizing set we get a basic test set of the checking sequence, interposed with reset. Obviously, if a prefix of a sequence can detect a fault then the whole sequence also can. Thus, we can remove all the sequences that are prefix of other sequences. As a result we get the following test set:

$$\{raaab, raab, rabab, rabb, rbaaab, rbaab^*, rbabab^*, rbabb, rbbab, rbbb\}$$

We execute the test set on $Impl$. The test sequences marked with * detect the fault. We use the shortest sequence – $rbaab$ – for the rest of the algorithm. Note that we can not narrow the set of possibly faulty transitions, because in *Spec* sequence $rbabab$ traverses all transitions that $rbaab$ does.

If applied to *Spec*, $rbaab$ produces the output sequence 1112, and if applied to $Impl$ we get 1111. That is, the fourth outputs are different ($k = 4$). First we presume that the fault in $Impl$ is an output fault (occurring at the fourth transition). Since the sequence $rbaab$ carries *Spec* from s_0 to s_3, s_2, s_1, s_1, we change the output at state s_1 input b from 2 to 1. We get the machine C_1 on Figure 3.6(a).



(a) C_1 (b) $C_{1\ reduced}$

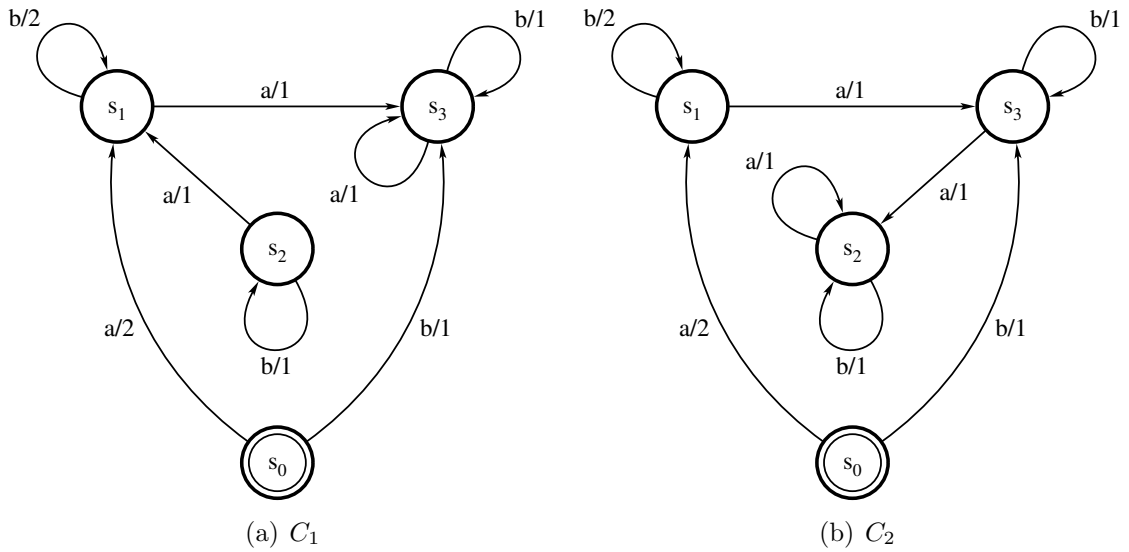
Figure 3.6: Conjectured machine C_1 with an output fault at state s_1 (a), and C_1 after minimization (b).

We reduce C_1 and get the machine $C_{1\ reduced}$ on Figure 3.6(b) We conduct a checking experiment (conformance testing) on $Impl$ with respect to $C_{1\ reduced}$. We find that the two machines are not equivalent (for example rab finds the difference), therefore, we move on and presume that the fault is a transfer fault occurring in one of the first three $(k-1)$ transitions.

We first conjecture that the first transition goes to a different state than specified. We have three possibilities: at state s_0, on input b the machine goes to s_0, s_1 or s_2 instead of s_3. We build the conjectured machines and verify them in turn. Omitting the details, we find that none of the machines conforms to $Impl$ (rba, rbb and $rbab$ rule out the possibilities respectively).

We move on and conjecture that the fault is at the second transition: at state s_1, on input a the machine goes to s_0, s_1 or s_3 instead of s_2. After building the machines and conducting the checking experiments we rule out the first two possibilities with sequences $rbaa$ and $rbab$ respectively. We also find that the third conjectured machine (C_1 in Figure 3.7(a)) conforms to $Impl$. Since the set of potential candidate machines PC is empty, we try to confirm if it is the only possible candidate. We find that after minimization C_1 has less states than the specification. Thus, we conclude that C_1 is a potential candidate, store it in set PC, and proceed to the next turn.

We conjecture that the fault is at the third transition: at state s_2, on input a



(a) C_1 (b) C_2

Figure 3.7: Conjectured machines C_1 and C_2

the machine goes to s_0, s_2 or s_3 instead of s_1. The sequence $rbaaa$ rules out the first possibility, but the other two conjectured machines – C_2 (Figure 3.7(b)), and C_3 (Figure 3.5(b)) – conform to $Impl$. Since PC is not empty, we know that the exact localization of the fault is not possible and store both machines in PC. As $k = 4$, a transfer fault may only occur at the first three transitions, therefore we have reached the end of the algorithm.

As a result we conclude that the exact localization of the fault is not possible. PC is the set of possible faulty machines C_1, C_2 and C_3 including all potential single faults.

3.3.3 Fast Algorithm

The paper [LS93] proposes an improvement to the original algorithm, called the fast algorithm, by introducing a more efficient method to rule out invalidly presumed transfer faults in Step 2.2 of the algorithm. The fast algorithm can be modified as well to cope with the problem described in Section 3.1.

Step 2.2 – Fast algorithm

We conjecture that the fault is a transfer fault. There are at most $2n$ possibly faulty transitions to consider as $2n$ is the maximal length of a test sequence. For each

transition there are $n-1$ possible wrong next states. Thus, there are no more than $2n^2$ conjectured machines. In the fast algorithm, instead of conducting a checking experiment on each of the $2n^2$ conjectured machines, we use cross-verification to rule out the wrong candidate machines.

We take two candidate machines in each round. First we minimize each candidate machine. There are two possibilities. If the two machines are equivalent then we mark them as belonging to the same equivalence class, and move on to the next round. Otherwise we generate an input sequence x, such that the two machines have different output responses to x. Then we apply input sequence x to $Impl$. According to the outputs produced by the implementation either one or both of the candidate machines can be ruled out. If the outputs produced by $Impl$ are the same as the output response of one of the candidate machines upon input sequence x, then we keep that candidate machine (and all machines marked to be in the same equivalence class). Otherwise we discard both candidate machines.

At the end there are three possibilities:

1. If no candidate machines are left, we conclude that there is more than one fault in $Impl$, end of algorithm.

2. If there is a single candidate machine left, we conduct a checking experiment on $Impl$ with respect to the remaining candidate machine. If the two machines are equivalent, then we exactly located the fault in $Impl$, end of algorithm.

3. If an equivalence class is left with more than one candidate machine, we again conduct a checking experiment on $Impl$ with respect to the remaining machine equivalence class. If it passes the experiment, we conclude that the exact localization of the fault is not possible, but we have we determined the minimal set of all potential single faults, end of algorithm.

The minimization of each candidate machine has an overhead of $O(pn \log n)$. The cross-verification of a pair of machines takes $O(pn)$ time. Since we conduct minimization and cross-verification on $2n^2$ conjectured machines, the total complexity of the fast algorithm is $O(pn^3 \log n)$.

3.4 Conclusion

In this chapter we studied the problem of fault diagnosis. The scope of fault diagnosis is beyond the scope of the fault detection (or conformance testing) problem. While the latter is concerned with determining if there are difference(s) between the behavior of the specification and the implementation machines, the former also tries to identify and locate the difference(s).

We concentrated on the diagnosis of a single transfer or output fault in an FSM. Clearly, the problem cannot be exactly solved if there are two or more equivalent implementation machines, each differing from the specification machine by a single dissimilar fault. We showed that implementation machines with different single faults may have the same observable behavior. Thus in general it is not possible to guarantee the exact localization of a single fault in a finite state machine.

We analyzed under what circumstances the exact localization of a single output or transfer fault can be guaranteed. That is, we determined a set of sufficient conditions when two (or more) implementation machines, each differing from the specification by a single dissimilar fault, cannot be equivalent. We incorporated the analytical results into an algorithm for the fault diagnosis problem. In case it is possible, the algorithm exactly locates the difference between the implementation and the specification, and when the exact localization is not possible, it provides the minimal set of all potential single faults.

Chapter 4

EFSM-based Specification and Testing

Finite state machines – in theory – could be used to model any finite state system as they are. In practice, however, typical software specifications contain variables and operations on the variables. Such systems can be modeled more appropriately by a technique supporting these features.

4.1 The EFSM Model

Extended finite state machines – or EFSMs – are an extension of the basic FSM technique by adding support for the use of variables.

Definition 4.1.1. An extended finite state machine (EFSM) is a 5-tuple:

$$M = (I, O, S, \vec{v}, T)$$

where

- I is the finite set of input symbols,
- O is the finite set of output symbols,
- S is the finite set of states,
- $\vec{v}$ is the finite set of variables,
- T is the finite set of transitions.

Each transition $t \in T$ is a 6-tuple:

$$t = (s_j, i, P, A, o, s_k)$$

where $s_j \in S$ is the start state of the transition, $i \in I$ is an input, P is a predicate on the variables, A is an action on the variables, $o \in O$ is an output and $s' \in S$ is the next state.

Given that the machine is in state s_j with variable values $\vec{v}_1$, upon input i the machine follows transition $t = (s_j, i, P, A, o, s_k)$ if for $\vec{v}_1$ $P(\vec{v}) = TRUE$. Then the machine outputs o, changes the variable values to $\vec{v}_2$ according to the action $\vec{v}_2 := A(\vec{v}_1)$ and transits to state s_k.

A combination of a state and variable values $[s, \vec{v}]$ is called a configuration. Initially, the configuration of the machine is represented by the initial state $s_0 \in S$ and by the initial variable values $\vec{v}_{init}$.

An EFSM M is deterministic if for every configuration $[s, \vec{v}]$ upon any input i there is at most one transition to follow. That is, for every (s, i) the sets of variable values for which $P = TRUE$ are mutually disjoint.

An EFSM M is said to be completely specified if for every $[s, \vec{v}]$ upon any input i there is at least one transition to follow.

An EFSM (with finite variable value domains) is in essence a compact representation of an FSM. Given an EFSM M, such that each variable of M has a finite number of values, there is a finite number of configurations $[s, \vec{v}]$ of M. Then we can simply create an equivalent FSM with configurations as states. As a special case, if the variable set is empty and all predicates P are $TRUE$, then an EFSM becomes an ordinary FSM.

A reachability graph is a directed graph including all the configurations reachable from the initial configuration as nodes, and all the transitions between them as edges. That is, a reachability graph is a state machine with either finite or infinite number of states. Clearly, a state of the EFSM – sometimes called a control state – may appear in several nodes of the reachability graph (with different variable values), and each transition of the EFSM may appear multiple times as edges in the reachability graph. Each path of the reachability graph from the initial configuration represents an I/O sequence and, if the EFSM is deterministic, every different I/O sequence follows a unique path.

4.2 EFSM-based Specification Languages

With the increasing importance of formal methods in the telecommunication software development lifecycle, many different formal description techniques have been proposed to specify such systems. In the early eighties, special working groups were

established on "Formal Description Techniques" with the purpose of studying the possibility of using formal specifications for the definition of the Open Systems Interconnection (OSI) protocols. Their work led to the proposal of three languages, Estelle [TC988], LOTOS (Language of Temporal Ordering Specifications) [ISO89] and SDL (Specification and Description Language) [IT00]. These languages are called formal description techniques (FDTs). In contrast to most programming languages not only a formal syntax was defined for the FDTs, but also a formal semantics defining the meaning of any valid specification. The formal semantics is essential for the construction of tools aiding the development of implementations.

While the foundations of LOTOS are provided by an algebraic calculus for communicating systems [Mil80], both other FDTs are based on the extended finite state machine model. SDL has been developed within ITU-T (International Telecommunication Union Telecommunication Standardization Sector) – formerly CCITT (International Telegraph and Telephone Consultative Committee) – since 1972, initially for the description of switching systems. Estelle was developed within ISO (International Organization for Standardization) for the specification of communication protocols and services. Both these languages, however, have a much broader scope of potential application.

In this chapter we first introduce the two EFSM-based FDTs, with the emphasis on the more widely accepted SDL, then we discuss the state of the art in conformance test generation methods for EFSM systems.

4.2.1 Estelle

Estelle is based on the extended finite state model and the Pascal Programming language. In Estelle a system is specified as a structured hierarchy of communicating modules. Communication is realized through exchanges of messages among the modules via bidirectional channels between their ports (called interaction points). The messages are stored in an infinite FIFO (First In, First Out) queue and processed according to the conditions, priorities and delays associated with the transitions of the related module. Communication is asynchronous, and the EFSM model allows the specifications to be non-deterministic.

The Estelle system – as a whole – is dynamic, meaning that both the hierarchy of modules and the structure of links may change over time. A state of the complete system is called a global situation, and consists of the states of single modules along with the connections and hierarchical relationships between them.

In Estelle, the behavior of each module is modeled as an EFSM. The specification of a module is divided into two parts, a header definition and a body definition. A header definition characterizes the external visibility of a module in terms of the set of interaction points and exported variables through which the module can be accessed.

A body definition characterizes the internal behavior of a module in terms of an EFSM transition system. A module with no transition specified is called inactive, otherwise it is active.

4.2.2 Specification and Description Language

SDL [IT00] is a well-accepted world standard supported by the ITU-T and ISO. SDL is widely used in the telecommunications industry for the description of telecommunication protocols, but it may be used in other fields as well. Typically complex, event-driven, real-time and communicating systems can be effectively described in SDL.

One of the strengths of SDL is its unambiguous semantics, making SDL specifications suitable for standardization. Moreover, SDL systems can be – either fully or semi automatically – transformed into executable code, therefore SDL can also be considered as a high-level implementation language.

An SDL specification models a system as a set of extended finite state machines that are executed in parallel. The EFSMs communicate asynchronously through well defined paths – called channels – using signals. Channels may be either unidirectional or bidirectional, and may transmit signals either with or without delay.

All of the EFSMs have their own memory for storing their variables and state information, and all of them contain a FIFO buffer of infinite length – a queue – for the incoming signals. The length of the queue increases by one as an input arrives and the input is added to the end of the queue. The length of the queue decreases by one as an input is processed. The input to be processed is not necessarily the first element of the queue. The SDL-specific "save" mechanism allows to save a specific input if it is the fist element of the queue, and process the next element in turn.

Signals may have parameters in SDL. A reception of a parameterized signal can be viewed simply as an input and a joint action assigning the new values (before the rest of the transition).

SDL provides syntactic shorthand notations for describing synchronous communication in the form of remote procedure calls, which are modeled implicitly by the exchange of two signals.

An SDL specification consists of a number of diagrams which, in combination, describe the structure and behavior of the system. The most important ones are:

State diagrams State diagrams describe the dynamic behavior of the agents using the Extended Finite State Machine (EFSM) model. The state machines are extended, since variables and timers can be defined.

Agent diagrams Agent diagrams are the building blocks of SDL, and comprise variables, procedures, state diagrams, and contained agents.

Package diagrams Package diagrams contain type definitions that can be imported by other SDL documents.

Procedure diagrams Procedure diagrams define behavior that can be reused by other agents.

There are two kinds of agents, blocks and processes. They differ in the degree of concurrency. In blocks, the state machine of the agent itself and the state machines of its embedded agents execute in parallel. In processes, the state machines are executed in an alternating manner. The "system" is the outermost block in an SDL specification that communicates with the environment, and contains all the other system components.

SDL is object oriented. Agent types, data types, procedures and signals can be specialized by inheritance. Inheritance has the following properties:

- Only single inheritance is allowed.
- Only parts with the keyword "virtual" in their definitions may be redefined.

An EFSM in a state diagram cannot only be triggered by incoming signals, but by timer expiration as well. Each state diagram may have an arbitrary number of local timers. Timers may be set and reset while executing a transition. The set operation activates a timer and has the expiration time as its parameter. Reset stops a timer. If a timeout occurs, a signal with the name of the timer is placed into the input queue, i.e., a timeout may be considered and handled as a simple input.

SDL uses two types of representation of its components, textual (SDL/PR) and graphical (SDL/GR) representation. While the former enables easy parsing of SDL descriptions for programs, the latter enables easier readability for humans. Figure 4.1 presents a system description in graphical and textual representation.

4.3 Testing EFSM-based Systems

In this section we are focusing on the conformance testing – or simply testing – problem for extended finite state machines. That is, we are concerned with the creation of test sets checking whether a black box implementation conforms to its specification given as an EFSM. According to the literature we are considering deterministic EFSMs with fixed initial state, i.e., with reliable reset capability.

Clearly, the problem is an extension of the FSM-based conformance testing problem discussed in Section 2.4.1. Moreover, as we already pointed out previously in this chapter, an EFSM with finite variable value domains is in essence a compact representation of and FSM, and an equivalent FSM may be created with configurations as states. Thus, the EFSM-testing problem could in theory be reduced to the problem

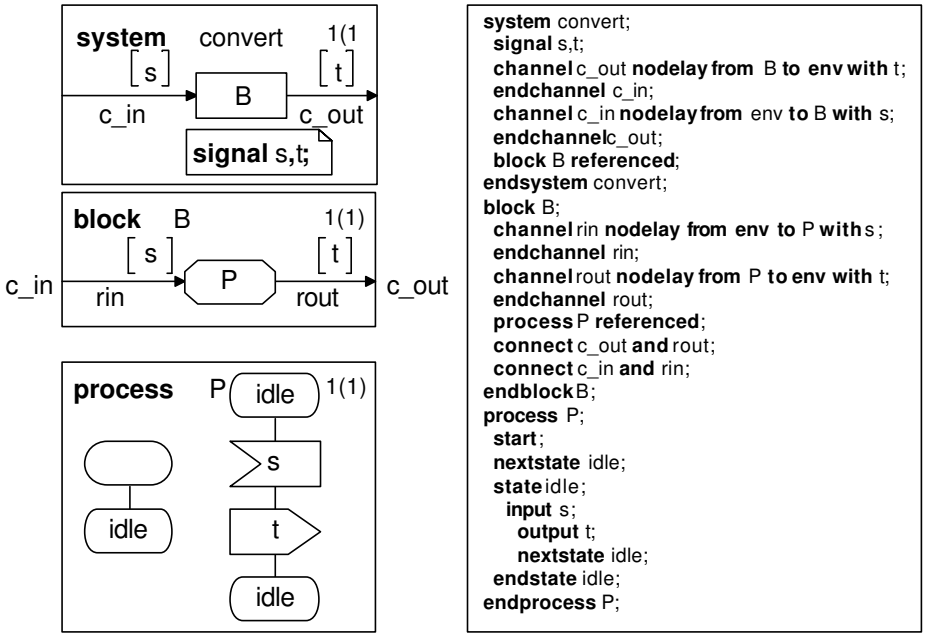


Figure 4.1: An SDL system in graphical and textual representation

of FSM-based testing. For most systems, however, this approach turns out to be impractical, as even for small EFSMs the number of configurations is so large that FSM test generation algorithms would create unacceptable test sets. Furthermore, if the EFSM has a single variable with unbounded domain, the number of configurations becomes infinite. This is the well known "state explosion problem" [LY96]. Consequently FSM-based conformance testing methods may not be directly applied to generate tests for EFSM systems.

EFSM-based testing is mostly based on heuristics, and does not have the well established theoretical background that the FSM-based algorithms have. Test generation methods for EFSMs have limited fault detection capabilities [PBG04], and are incomplete, i.e., they can only show the presence of faults, not their absence. Thus, testing can only be used to establish some confidence that the system behaves as specified.

Formal testing methods typically break an EFSM system into its "control" portion, and its "data" portion. The control part is essentially the underlying FSM modeling the control flow of a system. The data part includes parameters, variables, predicates, and operations reflecting the data flow of the system. Based on this separation, there are two main types of EFSM-based testing approaches. While some algorithms divide the testing of the control and the data portion, others propose to test the two jointly.

4.3.1 Separate Control and Data Flow Testing

EFSM-based testing involves the testing of both the control portion and the data portion of a system. A range of test generation algorithms handles these two parts as distinct elements that are independently tested.

Control Flow Testing

To test the control part of an EFSM system, traditional FSM conformance testing methods are applied. These methods have been discussed in Section 2.4.5. Applying FSM-based test generation methods directly to the control part of an EFSM system may result in non-executable test sequences because of the existence of predicates and conditional statements. To solve this problem, so called normal form specifications have been proposed [SvBC87], clearly separating the control and data flow aspects of an EFSM. In a normal form EFSM specification, all control flow aspects are incorporated into the FSM part, i.e., transitions do not contain branching statements or loops.

Data Flow Testing

The data part of an EFSM system includes parameters, variables, predicates, and operations. Data flow testing attempts to check the operation of these data objects, and aims to create test sequences for that purpose. Data flow testing is usually based on a data flow graph (DFG). A data flow graph $G = \langle N, E \rangle$ is a directed graph with the nodes $n \in N$ representing the functional units of a program and the edges $E = N \times N$ representing the flow of data objects. The functional unit could be a statement, a transition, a procedure or even a program [BDA96].

Any variable v of a system can be accessed in one of the following three ways:

Definition (*def*) A new value is assigned to v, that is, v appears at the left hand side of an assignment statement in the transition. Note that a value assignment may also occur on the reception of a parameterized signal, which can be viewed as the reception of a simple input and a joint action assigning new values to variables (before the rest of the transition).

Computational use (*c-use*) The variable v is used within an expression on the righthand side of an assignment statement or in an output statement.

Predicate use (*p-use*) The variable v affects the control flow, i.e., it is used in a predicate.

Definitions and computational uses may only occur in the nodes of a DFG while predicate uses only appear in its edges.

The global use of a variable v is the use of v whose definition occurs in some other transition, otherwise it is a local use. A global *def* of variable v is a definition of v for which there is a global use in some other transition, otherwise it is a local *def*.

For a given DFG G:

- $def(n_j)$ is the set of variables for which node n_j contains a global *def*.
- *c-use*(n_j) is the set of variables for which node n_j contains a global *c-use*.
- *p-use*(n_j, n_k) is the set of variables for which edge (n_j, n_k) contains a *p-use*.

A path $(n_1, n_2, ..., n_j, n_k)$ is a *def-clear-path* with respect to a variable v if $n_2, ..., n_k$ do not contain *def* of v.

A path $(n_1, n_2, ..., n_j, n_k)$ in a DFG is a *du-path* (definition-uses) with respect to a variable v if $v \in def(n_1)$ and either *c-use*(n_k) or *p-use*(n_j, n_k), and $(n_1, n_2, ..., n_j, n_k)$ is a *def-clear-path* with respect to variable v.

A number of heuristic coverage criteria were constructed for the selection of paths in a DFG [RW85]. The most important criteria are:

all defs All global variable definitions must be tested with either a computational or predicate use.

all p-uses All global variable definitions must be tested with all successive predicate uses.

all c-uses All global variable definitions must be tested with all successive computational uses.

all uses All global variable definitions must be tested with all successive computational and predicate uses.

all du-paths All global variable definitions must be tested with all successive computational and predicate uses using all possible paths (all *du paths*).

Obviously testing the data flow using all possible input values would provide the most thorough analysis of the operations of data objects, but in practice the input domain is usually too large for exhaustive testing to be possible. Instead, the usual procedure is to select a small subset of values which is in some sense representative of the entire input domain.

4.3.2 Joint Control and Data Flow Testing

A number of test generation algorithms attempt to generate test sequences for the EFSM as a whole, without breaking the problem down into control and data flow testing. These algorithms typically define some heuristic notion of coverage with a related metric, and try to create test sequences that provide the best coverage according to that metric. There are two main types of coverage criterion, specification and fault coverage. The former is concerned with covering all the transitions (symbols) of the EFSM specification with the test sequences. The latter introduces a fault model and tries to generate tests that uncover the most faults according to that model.

Specification Coverage

Specification coverage-based test generation algorithms build on the observation that a practical criterion of coverage is to execute each transition in the specification EFSM at least once. Obviously, this criterion is heuristic in a sense that it provides no guarantees as to fault detection capabilities. The approach is, however, widely accepted and used. Groz et al. discuss in [GCR96] that the coverage of extended transitions of an EFSM corresponds well to the preferred coverage implied in the conformance testing standard ISO9646. Specification coverage-based algorithms provide the basis for many commercial tools available [EGH+97].

The criterion of covering all transitions of an EFSM can be formulated as a graph-theory problem on the reachability graph[1] of the given machine. Let us assign a unique color to all the transitions of the EFSM, and carry these colors over to the induced transitions in the reachability graph. The problem of constructing a test sequence executing each transition at least once turns to the problem of finding a path in the reachability graph covering all colors. Although this problem does not seem too complex, it has been proven that the obviously easier reachability problem[2] is undecidable if the variable domains are infinite and PSPACE-complete otherwise [HU79]. It has also been proven that finding a minimal test set covering all transitions of an EFSM is NP-complete.

Specification coverage-based algorithms mostly conduct reachability analysis to provide test preambles to reach specific configurations of the EFSM enabling the transitions to be tested. Most algorithms, however, do not check the tail state of transitions, thus faults in an implementation may easily be undetected. Some methods try to address this problem by using special input sequences to check the tail state of the transitions – or more specifically the tail configuration reached with the given test [PBG04].

[1] For the definition of reachability graphs see Section 4.1

[2] In the reachability problem we only want to determine if a control state is reachable from the initial state.

Fault Coverage – Mutation Testing

Fault coverage-based testing – or mutation testing – methods follow a different strategy for evaluating and generating test sets. They consider a fault model, usually defined in terms of the specification's elements or by a set of mutation operators. The fault model is used to derive faulty implementation machines – mutants – from the specification. Then the notion of coverage is based on the mutant detection capability, i.e., the number – or percentage – of mutants the test set is able to identify.

A mutation testing system consists of three components: the system specification, a fault model (mutation operators) and an oracle analyzing the test results [SFSM00]. Each mutation operator is a template of an atomic syntactic change. By applying the operators systematically to the specification a set of mutants can be generated, where each mutant system is a small syntactic variation of the original.

The oracle is a person or a program used to distinguish the original from the mutant systems by their interaction with the environment. An oracle can distinguish mutants from the original if they produce different output. However, some of the mutants generated using the operators may be semantically equivalent to the original system. That is, a mutant and the original may compute the same function for all possible inputs. These systems are called equivalent mutants.

The basic ideas of mutation testing were established in the 1970s. Mutation analysis has initially been used for code-based software verification and validation [MLS78], but later it has also been applied to some simple specifications [AB99], for example SCR (Software Cost Reduction) [Hen01] description of software.

A fundamental assumption of the traditional code-based mutation analysis is the competent programmer hypothesis [ABM98], declaring that competent programmers tend to write nearly "correct" programs. That is, programs written by experienced programmers may not be correct, but they will differ from the correct version by some relatively simple faults such as an off-by-one fault. DeMillo's [MLS78] "Coupling effect" proposes that a test data set that highlights simple faults in programs is also sensitive enough to uncover more complex faults [Off92]. Based on these assumptions, code-based mutation analysis algorithms apply only small syntactic changes exactly one at a time to produce mutants [Kuh92]. The rationale is that if a test set can distinguish a program from its slight variants, the test set is exercising the program adequately.

A key obstacle to the practical use of Mutation Testing is its high computational cost, mainly due to the large number of possible mutants. It has, however, been shown in the literature that using a representative mutant set including only a small percentage of the mutants may be a useful heuristic, as a high mutant detection ratio against this representative mutant set would also indicate a high mutation score against the full set of mutants [OLR+96].

Recent research has showed that mutation testing can effectively be applied to

various formalisms as well. Major research has been done by Fabbri et al. in this field. They defined mutation operator sets and presented case studies for different formalisms. A mutation model for finite state machines was developed and manually applied to a CLASS 0 ISO transport protocol specification in [FMDM94]. They used the mutation analysis criterion to evaluate the adequacy of the tests produced by standard finite state machine test generation methods, the W and the TT methods (see Section 2.4.5). They also proposed the application of mutation testing for validating Estelle specifications [SFSM00]. They presented mutation analysis of Petri-Nets [FMM+96] and Statecharts [FMSM99] as well. Ammann and Black applied mutation analysis to model checking [ABM98] [BOY00] [AB99].

4.4 Conclusion

In this chapter the fundamental concepts and notations of extended finite state machine-based system specification and testing were introduced. We discussed the necessity of extending the finite state model with variables, and presented the EFSM model in detail – along with an overview of the two EFSM-based FDTs, Estelle and SDL. The final part of the chapter focused on the conformance testing problem based on the EFSM model, and detailed the most important state-of-the-art test generation approaches.

Chapter 5

Automatic Fault-based Test Selection for SDL Systems

The main purpose of the chapter is to give an algorithm to reduce test sets for systems defined in SDL without sacrificing coverage, thus improving the efficiency of conformance testing. Our approach utilizes ideas of mutation analysis to automate all steps of the test selection process.

For most telecommunication systems, automatic test generation algorithms – or the manual effort of multiple human experts – result in huge number of test sequences. There might be thousands of test sequences in the given test set. In such cases it is impractical to execute all of the tests, as it takes a significant amount of time to run each test. A possible solution is to reduce the test set by test selection.

The test selection problem can be defined as follows [LH01]: given a large set of test sequences, we want to select a minimal subset of tests to execute without sacrificing fault coverage.

We give an algorithm for automatic test selection building on the ideas of mutation analysis. The selection criteria are provided by mutant systems generated using a special set of mutation operators for SDL specifications. The proposed mutation operators are based on Chow's widely accepted FSM fault model (see Section 2.4.4), and induce atomic changes to a system. Furthermore, they can be applied automatically to any SDL specification, and include operators for SDL-specific features as well. Based on the algorithm, a Test Selector tool has been developed at the Budapest University of Technology and Economics. We have conducted several experiments on well-known protocols using the tool. Later in this chapter we will examine empirical data acquired from the experiments.

5.1 Mutation Operators Proposed for SDL Systems

Much research has been done concerning mutation operators for different formalisms, but no operator set has been defined for SDL systems. Fabbri et al. defined mutation operator sets for finite state machines in [FMDM94], for Petri-Nets in [FMM+96] and for Statecharts in [FMSM99]. The closest study was on operators for specifications written in Estelle [SFSM00]. Although both Estelle and SDL are based on the EFSM model, we have decided to build on the traditional FSM fault model defined by Chow [Cho78], and create a mutation operator set by extending his model. The rationale of the decision was that existing operators – including the ones defined in [SFSM00] – did not meet three very important requirements. Our intent was to include

- atomic operators,
- automatically applicable operators,
- operators for SDL-specific mechanisms,

in the operator set.

Atomic Operators

As previous studies on fault models suggested the use of atomic operators, we decided to apply as small changes to a system as possible. It is also in accordance with the principles of the mutation analysis technique [MLS78]. By using atomic operators we could keep the set of mutation operators as small as possible, and avoid including redundant ad-hoc operators like "origin state exchanged" along with the operator "tail state exchanged" as in [SFSM00].

Automatically Applicable Operators

While previous studies proposed operators that may be applied manually, our fundamental motivation was to create a fully automatic tool for test selection. Therefore, we had to define operators that can be applied automatically to any SDL specification. One of the main problems was the handling of complex predicates and data structures. To solve the problem, non-Boolean predicates are automatically replaced with a sequence of Boolean predicates in the specification, and operators are applied on them. Note that these transformations are done automatically and do not change the behavior of the system.

Operators for SDL-specific Mechanisms

To create a mutation operator set that is appropriate for SDL, we had to take SDL-specific features into account. In our approach, timeouts are handled as simple inputs, and accordingly the input mutation operator mutates them. To test timer transitions, timeout events are made controllable from the environment. That is, whenever a test sequence reaches a timeout, a corresponding "timeout" signal is sent directly to the owner process of the timer from the environment, and after its consumption the corresponding timer transition is executed. During the test execution, a timeout in the test sequence explicitly indicates that the tester will have to wait for the duration of the timer.

Furthermore, other SDL-specific features – like parameterized signals, the "save" mechanism and the implicit consumption of inopportune signals (essentially SDL's completeness assumption[1]) – also had to be dealt with.

5.1.1 The Proposed Mutation Operators

According to the considerations discussed previously, we defined six types of mutation operators for the SDL model:

- next state operator,
- input operator,
- output operator,
- action operator,
- predicate operator,
- save operator.

Henceforth, let us consider an SDL state diagram with the set of states S, inputs I, outputs O, variables $\vec{v}$ and transitions T, where each transition $t \in T$ is a 6-tuple: $t = (s_j, i, P, A, o, s_k)$. Note that there may be more than one output in a transition (o_1 ... o_n instead of o, imposing no problem for the current method). Let $i \in I$ denote the input to be processed, which is not necessarily the first element of the queue; and let the $\Omega()$ function represent the syntactic change applied.

[1]SDL essentially uses completeness assumption 2 in Section 2.1.4, i.e., if an input arrives for which no transition is defined then it is implicitly consumed meaning that the input is taken from the queue and the system remains at the given state.

Operator 1 (Next State)**.** *Mutating the next state in transition*
$t \in T$*:* $t = (s_j, i, P, A, o, \Omega(s_k))$*,* $\Omega(s_k) = s_l \in S$*,* $s_l \neq s_k$*. We replace the next state symbol, that is, we lead the system to a wrong state.*

Operator 2 (Input)**.** *Input mutations are only used to test the implicit consumption of inopportune inputs.* $t \in T$*:* $t = (s_j, \Omega(i), P, A, o, s_k)$*. Assigning the transition of input* $\Omega(i) = i_{inopp}$ *(*$i_{inopp} \in I$*, but* $i_{inopp} \notin I'$*, where* $I' \subseteq I$ *is the set of inputs, which have explicit transitions defined at the given state* s_j*) to the existing transition branch of input* $i \in I$*. This mutation effectively includes a transition branch for the input* i_{inopp} *that was implicitly consumed previously. Using this mutation, the processing of inopportune inputs (valid input arriving at wrong time) can be inspected.*

Operator 3 (Output)**.** *The mutation of one of the output events in the transition* $t \in T$ *is:* $t = (s_j, i, P, A, \Omega(o), s_k)$*,* $\Omega(o) : \Omega(o_y) = o_w \in O$*,* $o_w \neq o_y$*. For outputs with parameters (*$o \in O \times \vec{v}$*), we create a special output operator deleting the parameter of the given output:* $\Omega(o, v) = o$*), where* $v \in \vec{v}$*.*

Operator 4 (Action)**.** *It is difficult to define a general mutation operator for actions* $t = (s_j, i, P, \Omega(A), o, s_k)$*, because of the presence of abstract data types. Our practical solution is the deletion of an action statement.*

Operator 5 (Predicate)**.** *As we have transformed all non-Boolean predicates to a sequence of Boolean predicates, we only concentrate on the mutation of Boolean predicates. According to our previous analytical investigations [KPC02] we only use stuck-at operators. Setting the predicate to be stuck-at-true (*$\Omega(p) := true$*,* $p \in P$*) or stuck-at-false (*$\Omega(p) := false$*,* $p \in P$*).*

Operator 6 (Save)**.** *Save is one of the SDL-specific features that must be taken into account, therefore we have defined an operator for the save feature. This operator removes the save statement.*

As mentioned previously, inputs and outputs may have parameters ($i \in I \times \vec{v}$). The reception of a parameterized input is essentially a simple input and a joint action assigning the new values (before the rest of the transition). Therefore, we handle the mutation of parameterized inputs as the mutation of the implicit action assigning the new values. In this case, $\Omega(i)$ becomes $(\Omega(a))$, where $a \in A$ and is modified according to the action mutation.

According to the results of Offut et al. [OLR+96], our mutant creation strategy builds a representative mutant set (see Section 4.3.2), not an exhaustive one. That is, we apply operators systematically to all parts an SDL system, but only create some mutants for each element of the specification, instead of all possible mutants for the given symbol. Moreover, the operators themselves were elaborated with this strategy in mind, as their definitions already restrict the number of mutants they may create.

Action and save operators, for example, produce a single mutant for each action and save symbol in the SDL system, respectively. Next state, input and output operators on the other hand could be used to construct multiple mutants for a given symbol; the actual number can be set by parameters. For predicates, both stuck-at-true and stuck-at-false mutants are generated.

Table 5.1 demonstrates some examples of the application of the mutation operators on the INRES (Initiator-Responder Protocol).

Table 5.1: Example of the mutation operators proposed for SDL

Operators	Original	Mutant
State	`NEXTSTATE wait;`	`NEXTSTATE connected;`
Input	`INPUT ICONreq;`	`INPUT CC;`
Output	`OUTPUT CC;`	`OUTPUT DT;`
Action	`TASK counter := 1;`	`/* Missing */`
Predicate	`DECISION sdu!id = CC;`	`DECISION NOT(sdu!id = sdu!id);`
Save	`SAVE IDATreq (d);`	`/* Missing */`

5.2 Algorithm for Test Selection

In this section we describe the proposed test selection algorithm in detail. It requires the SDL specification of the system to be tested, and assumes the existence of a set of initial test sequences. Since our objective is to test a deterministic module of an SDL system at a time, we are only interested in testing (sub)systems with deterministic behaviors, i.e., at a given configuration of the EFSM and upon an input there is at most one transition from that state which is executable.

The SDL specification must be given in textual (SDL/PR) form. There are practical and widely used tools (for example Telelogic Tau [AB03]) to assist the specification process, and to create textual representations from graphical system descriptions.

Initial test sequences must be given as Message Sequence Charts (MSCs) [IT99] [GHN93]. They might be created either using an automatic test generation algorithm or manually. There are automatic test generation tools (for example the Autolink tool of the Telelogic Tau) creating MSC test sequences based on state space exploration algorithms. This test set is subject to optimization, meaning that a subset of test sequences is selected with equivalent coverage (mutant detection ratio). The final MSC test set can later be translated to different test description languages for example TTCN-2 (Tree and Tabular Combined Notation) [ISO96].

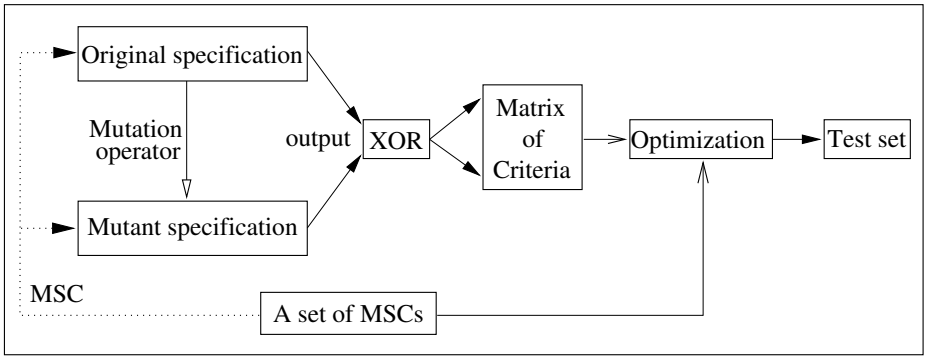


Figure 5.1: Test selection algorithm based on mutation analysis

Figure 5.1 presents our algorithm.

Let the matrix of criteria $\mathbf{C}$ be an $N \times M$ matrix of Boolean values, where N equals the number of mutants created for the given system according to our mutant creation strategy, and M is the cardinality of the initial test set.

Algorithm 5.2.1 (Selecting test sequences from an existing set of M test sequences)**.**

0. $i := 1$.

1. Apply a mutation operator to the system, that is, create a mutant specification (the i^{th} mutant).

2. Run the test set on the mutant specification and check for inconsistency.

3. Create a row vector $\mathbf{C}_i$ (the i^{th} row of the $\mathbf{C}$ matrix).

   - Let $\mathbf{C}_{ij}$ be 0, if the j^{th} test sequence is not able to detect the i^{th} mutant.
   - Let $\mathbf{C}_{ij}$ be 1, if the j^{th} test sequence is able to detect the i^{th} mutant.

   In other words, mark the test sequences killing the given mutant.

4. Repeat steps 1-3 for $i = 1$ to N, i.e., for all mutants.

5. Acquire the matrix of criteria, where rows represent the mutants and columns represent the test sequences in the original set. The matrix describes for each test sequence the mutations detected by the given test sequence.

6. Select an optimal test set from the original set, using an optimization method.

Mutants \ Test Cases								
	1	1	0	0	0	1	0	0
	0	1	1	0	0	1	1	1
	1	0	0	0	1	0	0	1
	1	1	0	0	0	1	1	0
	0	0	0	0	0	0	0	0
	0	1	1	1	1	0	1	0
	0	0	1	0	0	1	0	0

Figure 5.2: A sample matrix of criteria

The last step of Algorithm 5.2.1 is an optimization problem: we are trying to find the minimal number of tests with the highest possible mutant coverage. Note that here a "one is enough" coverage model is a natural choice, i.e., we consider a mutant covered if at least one of the test sequences killing it is in the final test set. For an overview of exact and heuristic algorithms for the optimization problem see [Csö01].

Clearly, since we are selecting – not generating – test sequences, the quality of the final test set depends on the quality of the initial test set. That is, the maximal coverage of the final test set – by any coverage metric – can not be higher than the coverage of the initial set. The aim of the test selection procedure is not to improve coverage compared to the initial set, but to select a subset of tests without sacrificing fault coverage. Therefore, to get good mutant coverage, the initial test set of the algorithm has to be large enough – the sufficient number of test sequences varies from protocol to protocol.

5.3 Comparison with Related Work

The number of contributions dealing with EFSM test generation and selection is constantly growing, nonetheless the problems are difficult and – except for some trivial cases – cannot be exactly solved, i.e., only the presence of faults can be shown, not their absence. For an overview of the state of the art in EFSM test generation see Section 4.3.

Test generation and test selection are closely related terms, and are often used as synonyms in the literature. Most of the papers on test selection – like [BAKD01] – are in fact describing test generation algorithms. We, however, follow the approach of Lee et al. [LH01] and consider test selection as a separate task minimizing an existing test set. There are a number of papers on test selection – as we define it – but none of them is based on mutation analysis ideas. Most papers study test selection for more simple formalisms, mostly for FSMs [YCL98] [PvB95].

The most relevant research was done by Lee et al. in [LH01]. They consider

deterministic EFSM models and use the coverage of edges in the reachability graph as selection criteria to remove any redundant test sequences. Although the presented results have a very clear theoretical background, they are for most systems impractical, as even for small EFSMs the number of configurations – and thus the size of the reachability graph – is too large.

Obviously, our algorithm augmented with any (for example random) state-space exploration approach may be viewed as a test generation method composed of two steps. The first step is the derivation of a number of test sequences from the specification covering a large enough part of the state space, and the second is the selection of the test sequences to be included in the final test set. Therefore – although this research focuses on the test selection problem – we should also analyze how our algorithm compares to existing EFSM test generation methods.

Since our algorithm is a fault-based test selection method, it can be compared directly to other mutation testing approaches (see Section 4.3.2). All these algorithms consider a fault model, i.e., a set of mutation operators, and the notion of coverage is based on the fault (mutant) detection capability. Mutation testing approaches have been initially applied to software code, but research has showed that mutation testing can effectively be applied to various formalisms as well. Major work has been done by Fabbri et al. in this field. They defined mutation operator sets and presented case studies for different formalisms. Their most relevant research is "Mutation testing applied to Estelle specifications" [SFSM00], since Estelle is based on the EFSM model similarly to SDL.

There are a number of differences and improvements in our approach compared to previous research. The most important are:

Mutation operators We are using a special set of mutation operators developed based on Chow's FSM fault model [Cho78] for SDL. For detailed discussion on our operators and their background see Section 5.1.

Test selection algorithm We propose an algorithm to optimize existing test sets by selecting a subset of the initial tests with equivalent (mutant) coverage. Moreover, a key objective of our research is the automation of the test selection process. Previous studies, in turn, used mutation operators only as an aid for the manual test creation (generation) process. The testing strategy in [SFSM00] simply encompasses the application of an operator and the manual creation of a test detecting the mutant.

Tool and experiments The testing strategy described in [SFSM00] was applied to the Alternating-bit Protocol manually. No tool has been developed to aid the process, although the authors found that "The results obtained manually indicated the need for a testing tool to support the application of Mutation

Testing". Our algorithm, on the other hand, has been designed to work automatically, and a tool has been developed based on it. We conducted experiments on the INRES and Conference protocols using the tool.

Most EFSM test generation methods use the specification itself as a coverage criterion, and require the test to execute each transition in the specification EFSM at least once (see Section 4.3). Many of the commercially available test generation tools for SDL (for example TVEDA [GR98], the Autolink tool [AB03] and the TestComposer tool [SEG00]) are based on this approach [GJK99].

To compare our algorithm with these types of methods, we have to analyze its coverage considering a specification-based coverage metric. Since we apply mutation operators systematically to all parts of the specification, the specification coverage of the optimized test set will be equivalent to the coverage of the initial set, given that at least one mutation is observable in each transition of the EFSM. This condition is trivially fulfilled if each transition contains an output, which is true for virtually all practical protocols. That is, our algorithm does not reduce the coverage of a test set assuming a specification coverage metric, but reduces its size (if possible). Obviously, if the coverage of the initial test set (the one we try to optimize) is good enough to find at least one mutant (of any type) in each transition of the EFSM, then the final test set will be complete with respect to specification coverage. Experience shows that this criterion is usually realized even in case of relatively low mutant detection ratios.

5.4 The Test Selector Tool

To implement the automatic test selection procedure, a Test Selector tool has been developed in Java based on our algorithm. The main dialog can be seen in Figure 5.3. The tool consists of several components indicated by rectangles in Figure 5.4.

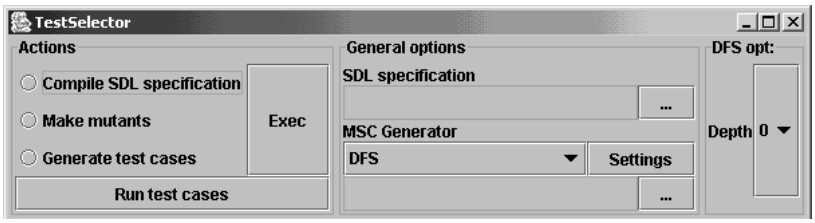


Figure 5.3: The main dialog window of the Test Selector tool

The mutant generator takes the SDL/PR (textual representation) description original specification as an input. It implements the mutation operators defined in Section 5.1, and creates mutant specifications in SDL/PR format.

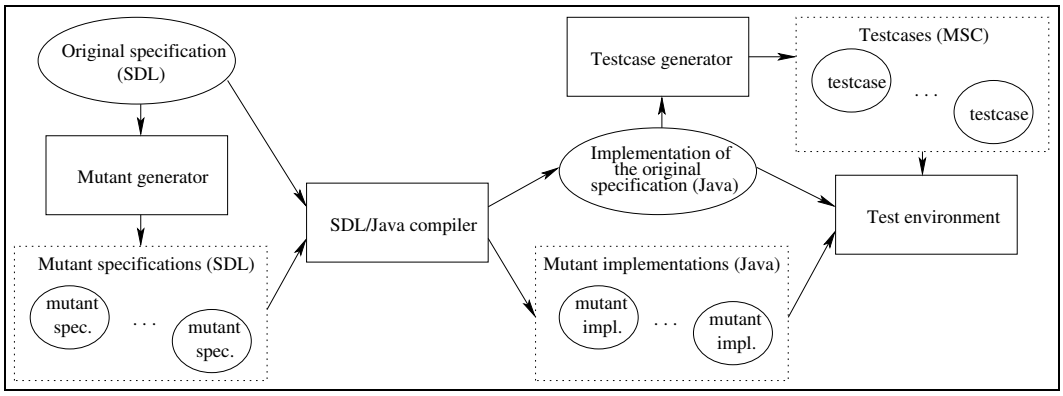


Figure 5.4: Components of the Test Selector tool

The SDL/Java compiler generates Java code from the SDL/PR description of both the mutants and the original system. The specification is modified automatically before the code generation, so that timers are made controllable, and decisions with multiple branches are transformed to series of Boolean decisions. Though the timer transformations do change the behavior of the system, for the presented method these changes are irrelevant.

Although it is not an integral part of the test selection algorithm, we implemented a testcase generator module to provide initial test sets for our experiments in MSC form. The input and output signals and the timers are extracted from the SDL specification, and a transition system is constructed from them [Tre00]. The transition system is explored randomly to a given depth and an MSC sequence is created in textual form. The exploration depth and the number of tests to be generated may be set by parameters.

The test environment is the core of the tool (see Figure 5.5). This component executes the test sequences against the implementations, sends messages according to the MSC test sequences, and evaluates the messages received. The test environment outputs the Boolean matrix of criteria, which is the subject of optimization.

Several components – the mutant generator, the compiler, and the test environment – are based on a parser. To generate these parsers we used the lexical analyzer JLex [Ber00] and the parser generator CUP (Constructor of Useful Parsers) [Hud00]. For the compilation from SDL to Java we defined mappings between the two languages, and built a Java code generator on top of the parser. In order to keep the produced code small and clean a run-time library has been created that provides a framework for the implementations. During the compilation the SDL specification goes through several stages. A preprocessor filters out timer actions (set, reset), creates special channels to make them controllable, and unfolds non-Boolean decisions. The remaining parts are left untouched. This is followed by the code generation

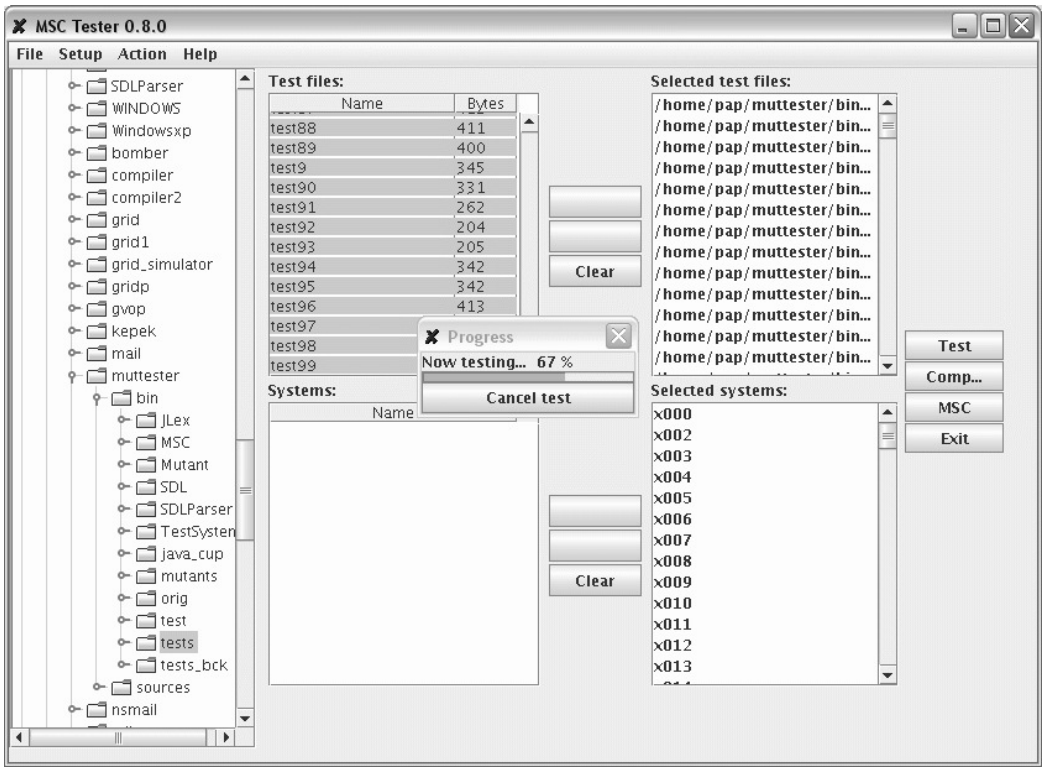


Figure 5.5: The graphical user interface of the test environment

phase, when the parser processes the specification and the code generator exports Java source. In the last step the SDL compiler calls the Java compiler and transforms the sources to classes.

In contrast to the compiler, the test environment treats the MSCs as scripts and executes them without producing Java code. Generic script languages for instance Perl, Javascript operate this way. The structure of the MSC is sequential and there are no conditions, therefore we can skip the code generation phase. During the test sequence execution, whenever a timeout is reached, a corresponding input signal is sent through the previously added timer channels.

Since the environment communicates with the implementation, we had to solve the problem of controllability. Thus, an interface has been created which can be used to send signals to the examined system and to receive the outputs. The interface has built-in FIFO channels to forward messages, and the most important control functions are available as well.

5.5 Empirical Analysis

We have conducted several experiments on protocols such as INRES, the conference protocol, and the Wireless Transaction Protocol (WTP) of the WAP (Wireless Application Protocol) protocol stack (see our papers [J0] [C4] [C6]). In the following we will discuss some of these experiments in detail.

5.5.1 Test Selection Applied to the INRES Protocol

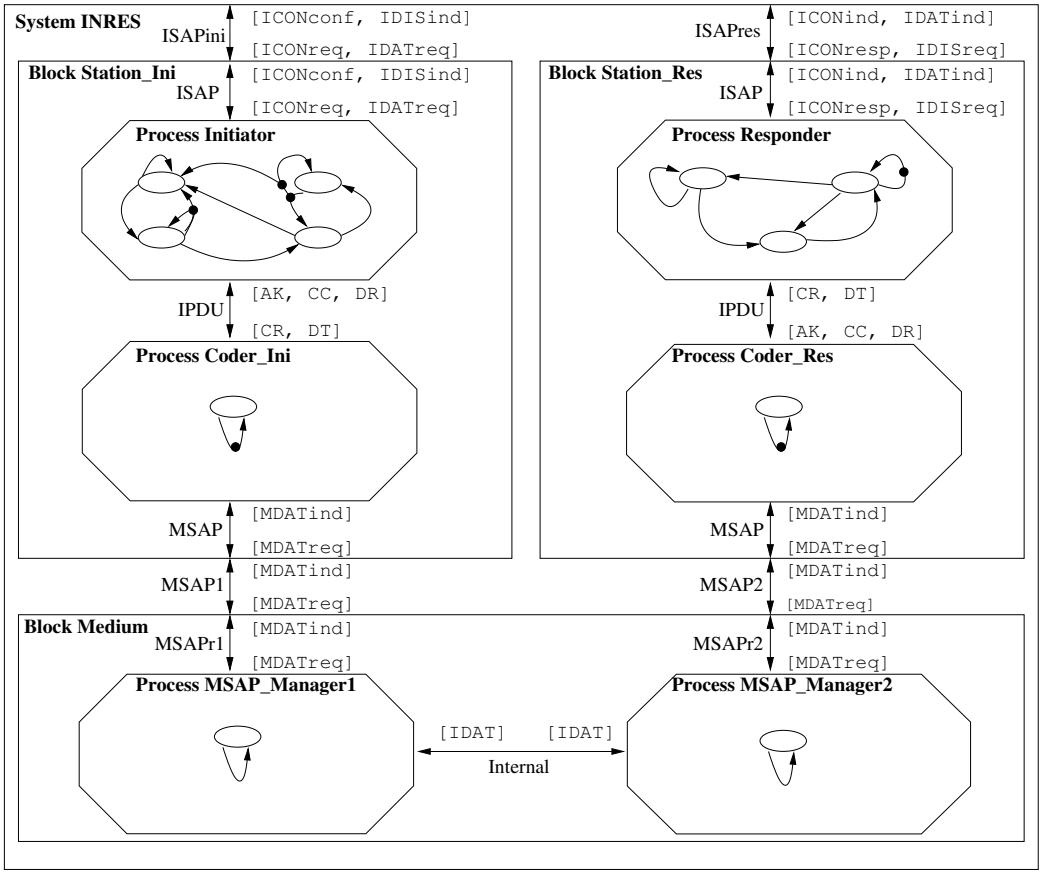


Figure 5.6: The INRES system

We used the well-known telecommunications protocol INRES [Hog91] to investigate the presented algorithm. It is a system widely used for experimental purposes as it includes all the typical properties of protocols in general. INRES is a connection-oriented protocol built on the OSI concept, which operates between two protocol entities: the Initiator and the Responder. These protocol entities communicate over

a Medium service. The SDL specification of the system was created using Telelogic Tau [AB03]. The structure of the system and the states of the processes are shown in Figure 5.6. (The transitions of the processes are shown schematically in the figure; dots represent decisions. Inputs, outputs and actions are not represented. For more details see [Hog91].)

We constructed 113 mutant specifications for the INRES protocol using the mutant generator module.

In the first type of experiment we used the testcase generator module to create random initial MSC test sets. In two experiments 54 and 92 test sequences were produced. Using 54 initial test sequences, the resulting test set contained only 14 test sequences. In the case of 92 initial tests, the optimized set included 18 test sequences. Table 5.2 presents the mutant coverage (percentage of mutants the test set is able to identify) detailed for mutant types, the overall mutant coverage, and the overall structural coverage (percentage of SDL symbols in the system executed by the test set) observed in the experiments.

Table 5.2: Data from the INRES case study with random initial test set

	Detailed Mutant Coverage (%)						Coverage (%)	
	next st.	input	output	action	predic.	save	mutant	struct.
54 Initial Tests	46%	91%	69%	42 %	50%	0%	61%	71%
92 Initial Tests	38%	100%	78%	58%	38%	0%	68%	81%

The experiments verified that the initial and final test sets have equivalent mutant detection ratio – an obvious result as the algorithm guarantees it. More interestingly, the experiments also showed that the initial and final test sets have equivalent coverage according to the most prevalent metric – the structural coverage. It is a very promising result, considering that the size of the test set was reduced by 74% in the first and by 80% in the second experiment.

Human experts evaluated both the final test set after the selection process and the test sequences that were eliminated. It was found that the reason for the relatively low mutant detection ratio was the inadequacy of the random initial test set; the coverage could be somewhat improved by increasing the number of the initial tests. The majority of test sequences were stuck at a relatively small part of the state space, and only a few tests turned out to be useful. Note that this result is according to expectations, as random test generation is not a too sophisticated approach. We want to emphasize, however, that initial test set generation is not an integral part of the proposed algorithm; it is only used for demonstration purposes.

The structural coverage reached was 71% and 81% in the first and second experiment, respectively. For comparison we created tests for the INRES system using the automatic test generation solution of Telelogic Tau [AB03]. We used the tree walk option of the Autolink tool to produce test sequences. Setting the time limit to 5 minutes, the Autolink tool reached 88% symbol coverage.

The real strength of our algorithm showed in the second type of experiment when the approach was used to select tests from a large number of test sequences created manually by multiple experts. The initial set contained 43 test sequences. Many of these sequences were quite complex, and all were based on the knowledge of human specialists. The final test set created during the experiment contained only three test sequences, but these three alone were able to kill all of the mutants (provided 100% mutant detection ratio). Manual evaluation revealed that the selected tests were in fact very good and provided 100% structural coverage.

5.5.2 Test Selection Applied to the Conference Protocol

The conference protocol is a multicast chat box protocol [BFV+99]. A conference is a session of the protocol in which a group of users can participate by exchanging messages with other users that can change dynamically. A user is initially only allowed to perform a join to enter a conference. After performing a join, the user may send or receive a message. To stop participating in the conference, the user can issue a leave at any time after a join. After that, another join primitive can be performed, starting a new participation in a conference. Different conferences can exist at the same time, but each user can only participate in at most one conference at a time. Messages may get lost or may be delivered out of sequence but are never corrupted.

We conducted a test selection experiment on the conference protocol and got results similar to those acquired in case of INRES. We generated 217 test sequences, and selected a subset of 29 tests – a 87% decrease – based on a mutant set including 123 mutants. Table 5.3 shows detailed results of the experiment.

Table 5.3: Data from the conference protocol case study

Mutation Operator	Generated Mutants	Detected Mutants	Detection Ratio
State	7	6	86%
Input	6	6	100%
Output	7	7	100%
Action	62	47	76%
Predicate	20	16	80%
Save	21	8	38%

As the data indicates the algorithm in this case worked well even with a random initial test set, and provided a final test set with 73% mutant and 100% structural coverage. This result is mainly do to with the simplicity of the protocol and the relatively large number of initial test sequences.

5.5.3 Performance

Another fundamental question of the empirical analysis is how it is going to work out in real life in terms of performance, what kind of hardware we are going to need, and how much time the test selection procedure will take. The experiments were conducted under the Windows 2000 operating system, on a Pentium II Celeron 333 MHz PC with 192 MB of memory. Table 5.4 shows the running times of the individual modules. The two results in the Test Selection field for the INRES protocol apply to the random experiments with 54/92 initial tests respectively.

Table 5.4: Execution times on a Pentium II Celeron 333 MHz PC with 192 MB of memory

Protocol	Mutant Generation	Code Generation & Compilation	Test Selection
INRES	1.5 min.	18 min.	9/23 min.
Conference Protocol	1.8 min.	26 min.	31 min.

The generation of the initial test sequence set is not indicated in the table as it is not an integral part of the algorithm; however it did not take significant amount of time. The most time was spent compiling the mutant systems to Java, because the javac compiler of the JDK (Java Development Kit) is quite slow. The time required for the actual testing of the mutant systems depends on the number of mutant systems, on the number of test sequences in the initial set, and on the complexity of the individual test sequences. The memory requirement of the tool was considerable.

These performance results are acceptable considering the speed problems of the Java language, and the low-end testing hardware used. Based on these results we can draw the conclusion that using modern hardware and a faster performing programming language, the mutation analysis method and tool can be applied to complex protocols.

5.6 Conclusion

Testing is a vital part of the telecommunications software development process. In practice the creation of test sets is usually a very time consuming manual process. Although several computer aided test generation methods have already been developed, most of them result in large and redundant test sets that need to be optimized by test selection.

We described how mutation analysis, a fault-based method, could be applied to automate all steps of the test selection process for SDL specifications. For this purpose we applied a special set of mutation operators created considering the requirements of automation and the specialties of SDL. We presented an algorithm for test selection using the operators. Based on the method a Test Selector tool has been developed, and was used for case studies. Both the selection process and the generation of the initial test set are done automatically.

As a future objective, an interesting possibility is the extension of our method to be adaptive. That is, we intend to make some experiments where the initial test and operator sets change on the fly.

Chapter 6

On the Correction of Faults – The Theory of Patching

In this chapter the problem of patching is studied. It is concerned with changes to the software to correct faults and deficiencies found after deployment either through additional testing or field usage. It is also used to modify or add functionality to comply with changes of the requirements. We consider systems modeled as finite state machines (FSMs), and define edit operators for them based on a traditional fault model. We argue that sequences of edit operations can be considered as models of patches defining modifications to an FSM system, and utilize recent results in graph matching theory as mathematical foundations. A new problem referred to as the optimal patch – or optimal update – problem is introduced. Given an FSM M modeling the behavior of an existing system and an other machine M' modeling a new design, find an optimal patch, i.e., the edit operations changing M to M' that are minimal according to a given cost function associated with the edit operations. A through complexity analysis of the problem is conducted, concluding that it is unlikely to have a polynomial time solution. We show that the problem can easily be transformed to a state-space search problem, for which many heuristic approximation algorithms have been developed and published in the literature.

6.1 Background

The ultimate goal of any system designer is to create a perfect implementation, i.e., an implementation that completely satisfies the needs of the target users. Essentially, the purpose of almost all research efforts in computer science, including requirements capture, design, implementation and testing methodologies, is to make it easier – or make it possible – to reach this objective.

Still, practice shows that the creation of perfect systems is virtually impossible.

Even if one could create error-free implementations, the constant change in user demands and other environmental factors inhibits the creation of an ideal system. This is the main reason why each and every developer faces a tough question; "When is the system good enough?", and has to make the decision of continuing the development of the system or putting it on the market. The significance of this decision lies in the fact that it is much more difficult and expensive to change a system after it is on the market and distributed among users. For some systems the only solution is to recall all distributed implementations if a serious problem emerges. In the case of software systems, however, patches – or updates – are the most widely used means to change the behavior of a system. The key benefit of employing patches is that one has to only distribute the difference – the modifications to the system – instead of replacing the whole. This approach provides an inexpensive way to cope with problems after the release of the system. Moreover, it enables developers to constantly follow user demands and quickly respond to them.

The practical importance and extensive use of patches motivates the current research trying to model and optimize patches. We consider finite state machine (FSM) models of systems. Finite state machines represent behavior and have been widely used to model systems in various areas. See Section 2.1 for detailed discussion of the FSM modeling technique.

We show that FSM fault models (see Section 2.4.4) may be considered as edit operators, and as such they can be used to model patches. We introduce a new problem referred to as the optimal patch – or optimal update – problem. The problem can be stated as follows: given an FSM M modeling the behavior of an existing system and an other machine M' modeling a new design, find an optimal patch, i.e., the edit operations changing M to M' that are minimal according to a given cost function associated with the edit operations. The mathematical foundations of this problem are provided by graph matching theory. Therefore before presenting the main results, we give a brief overview of this theory.

6.1.1 Graph Matching

Graph representations are widely used for dealing with structural information in different domains. When graphs are used to represent structured objects, then the problem of measuring the similarity of the objects turns into the problem of computing the similarity of graphs, which is also known as graph matching. Recently, much theoretical research has been devoted to this problem, motivated by its importance for many applications in the fields of pattern recognition [NB04], shape analysis [CCG+98] and data mining [WWS+02] among others.

There are two main kinds of graph matching: exact and inexact graph matching. Exact graph matching has been studied since the 1970s and includes problems like

graph isomorphism [Mil77], subgraph isomorphism, and maximum common subgraph [GJ79]. Given two (labeled/un-labeled, directed/undirected) graphs G and G' the problem is to find if the two graphs – or parts them – are isomorphic.

In real world applications like pattern recognition, however, there is often no perfect match between the two objects represented by the graphs. Thus, the inexact graph matching problem does not attempt to search for the exact way of matching vertices of a graph with vertices of the other, but to find the best matching between them. For that, graph edit operations like deletion, insertion, or substitution (i.e. label change) of nodes and edges are defined. In practical applications, some edit operations may have more importance than others. Therefore, often costs are assigned to the individual edit operations.

Given a set of edit operations and their costs, graph edit distance computation in its most general form means to find a sequence of edit operations transforming one graph into the other with minimum cost [WZC95]. Clearly, two machines are more similar if this cost is lower [Bun00].

In our own research we utilize the ideas of inexact graph matching to find an optimal patch between two finite state machines. We show that there are some natural edit operators for FSMs derived from fault models. The FSM edit operators are different from the ones considered in case of graph matching, and some special constraints have to be imposed to ensure the correctness of the automata. We adopt the idea of cost functions and try to find an optimal patch considering them.

6.2 Modeling and Optimizing Patches

For the rest of this chapter, we will focus on deterministic machines, and consider two FSMs identical if they are isomorphic – a natural consideration since Mealy machines are in principle edge-labeled directed graphs for which the actual labeling of the states is irrelevant.

6.2.1 Modeling Patches

Fault models have extensively been studied in the field of testing (see Section 2.4.4). They are in principle used to limit the number of possible implementations considered for constructing or analyzing a test set. A test set – in theory – should verify a conformance relation on a set of infinite input sequences, but on the other hand a test set must be finite to be practical. A way to cope with this problem is to introduce a certain fault model which defines all the faults that are to be found, and only consider implementations that are mutants of the specification according to that fault model.

There are various fault models proposed for FSMs in general. For completely specified and deterministic machines the most widely accepted model was defined by Chow [Cho78]. It includes the following three types of faults:

- Operation (Output) fault – for a given state and input, the implementation provides an output different from the one specified by the output function.
- Transfer fault – for a given state and input, the implementation enters a different state than specified by the transition function.
- Extra state / missing state – adding / removing a state.

A significant property of Chow's model is that it defines atomic faults, i.e., faults that are not a composition of other faults.

Next we show that Chow's fault types with some extension constraints can be considered as edit operators. For our further discussions we use a common completeness assumption[1] to handle incompletely specified machines: add a unique output symbol Υ_o to the set of output symbols O and a unique state Υ_s to the set of states S denoting the null (error) output symbol and state, respectively.

We define types of edit operators based on Chow's fault model. Consider an operator changing M to M'. A transfer operator is $TRO : \delta(s,i) = s_1 \rightarrow s_2$,[2] where $s_1, s_2 \in S, S' = S$ (S includes Υ_s). An output operator is $OO : \lambda(s,i) = a \rightarrow b$, where $a, b \in O$ (O includes Υ_o). An extra state operator is $ESO : a,\ S' := \{S \cup a\}$. A missing state operator is $MSO : a,\ S' := \{S \setminus a\}$.

We impose the following two constraints on the edit operations:

1. An extra state operator implies the addition of state s with null transition $\delta(s,i) = \Upsilon_s$ and null output functions $\lambda(s,i) = \Upsilon_o$ for all $i \in I$, i.e., it is an addition of a "blank state".
2. A missing state operator can only be applied to state s if s has no incoming transitions and all of its transition and output functions are $\delta(s,i) = \Upsilon_s$ and $\lambda(s,i) = \Upsilon_o$ respectively for all $i \in I$, i.e., only blank states with no incoming transitions may be removed.

Consider an edit operation e applied to finite state machine M_1 resulting in FSM M_2; this is written $M_1 \Rightarrow M_2$ via e. Let E be a sequence $e_1, e_2, ..., e_k$ of edit operations. E changes FSM M to FSM M' if there is a sequence of machines $M_0, M_1, ..., M_k$ such that $M = M_0$, $M' = M_k$, and $M_{i-1} \Rightarrow M_i$ via e_i for $1 < i < k$.

[1] Completeness assumption 1 in Section 2.1.4.

[2] For notational simplicity, we use $\delta(s,i) = s_1 \rightarrow s_2$ to represent a transfer operator instead of $[\delta(s,i) = s_1] \rightarrow [\delta'(s',i) = s_2']$, where M is the original and M' the edited machine. Similar notation is used for output operators as well.

Note that each operator e has an inverse e^{-1}, for example the inverse of $\lambda(s,i) = a \rightarrow b$ is $\lambda(s,i) = b \rightarrow a$. There are operators e^0 introducing no change to the system (creating an isomorphic machine), i.e., $M_1 \Rightarrow M_1$ via e^0, for example e^0: $\lambda(s,i) = a \rightarrow a$.

There are some important properties of Chow's fault model – and the edit operators based on it – we should emphasize here. Clearly, the set of deterministic finite state machines with a given set of input and output symbols I and O is closed under the edit operations defined above. Moreover, for any two deterministic FSMs M_1 and M_2 there is always a sequence of edit operations E changing M_1 to M_2, i.e., to a machine isomorphic to M_2. It is also easy to show that the previous statement does not hold if any of the three operator types is omitted.

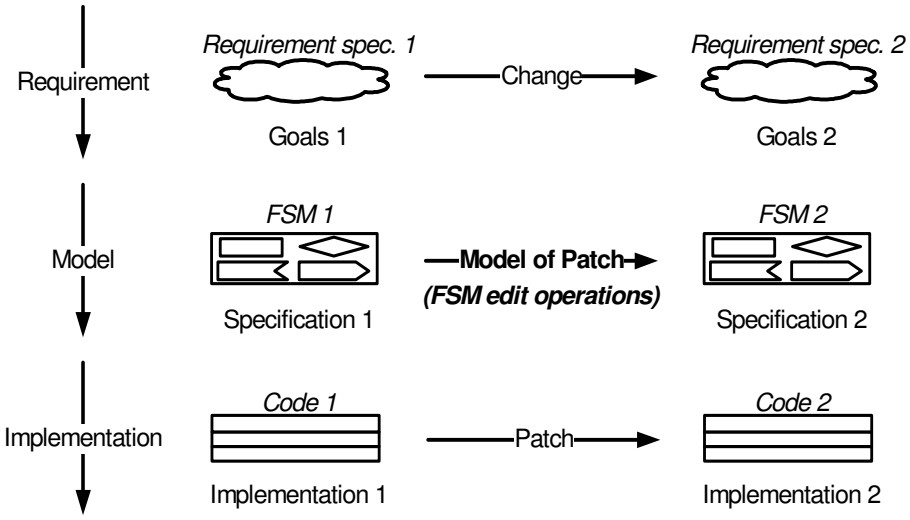


Figure 6.1: Modeling patches

Sequences of edit operations, therefore, can be considered as models of patches defining modifications – changes – to an FSM system (See Figure 6.1). Note that there are infinitely many possible sequences of edit operations – patches – between two finite state machines. Our objective here is to find the "best" among them. For that we first refine our model by assigning costs to edit operations.

6.2.2 The Optimal Patch Problem

Costs may indicate any practical property of the given operation. Let ρ be a cost function that assigns a nonnegative real number $\rho(e)$ to each type of edit operation. We constrain ρ to be a distance metric. That is, it satisfies the following three properties: $\rho(e) \geq 0$ and $\rho(e^0) = 0$ (nonnegative definiteness); $\rho(e) = \rho(e^{-1})$

(symmetry); $\rho(e_{13}) \leq \rho(e_{12}) + \rho(e_{23})$ for any three operations with the following property: $M_1 \Rightarrow M_2$ via e_{12}, $M_2 \Rightarrow M_3$ via e_{23} and $M_1 \Rightarrow M_3$ via e_{13} (triangle inequality). We extend ρ to a sequence of edit operations $E = e_1, e_2, ..., e_k$ by letting $\rho(E) = \sum_{i=1}^{k} \rho(e_i)$.

Now we can define the optimal patch problem for finite state machines. Let us consider an FSM M modeling the behavior of an existing system already implemented and distributed among users. For some reason the need arises to modify the behavior and a new design is created, modeled by M'. The problem is to determine the optimal patch updating the system, i.e., the edit operations changing M to M' that are minimal according to a given cost function.

As our costs are defined as distance metrics, this problem can be stated as finding the (edit) distance between two machines M and M', where the distance between M and M' is defined to be the minimum cost of all sequences of edit operations that change M to M':

Definition 6.2.1. $dist(M, M') = min\{\rho(E) \mid E$ is a sequence of edit operations changing M to $M'\}$

The definition of ρ makes *dist* a distance metric as well. By finding the distance between two machines, we also find an optimal patch changing one machine to the other.

Note that the solution to optimal patch problem (finding the edit distance) can be viewed as a measure of similarity for two FSM systems. Thus, it may have different uses such as assessing the effort of a developer or detecting plagiarism.

6.3 Transformations

Unfortunately, it is not convenient to work directly with sequences of edit operations as there are infinitely many ways of changing a given system to the other. Thus we introduce a related term we call transformation. For two machines M and M' let a transformation T be a one-to-one mapping between subsets of states of the two machines.

Definition 6.3.1. A transformation T between two FSMs M and M' is a set of pairs of states (s, s') where $s \in S_{sub} \subseteq S$, $s' \in S'_{sub} \subseteq S'$ and for any pair (s_1, s'_1) and (s_2, s'_2) in T, $s_1 = s_2$ if and only if $s'_1 = s'_2$ (state one-to-one).

Informally, transformations describe ways to change one deterministic FSM to an other. If a state s of M is mapped to a state s' in M', then s should be transformed to s' by the patch.

We assign sequences of edit operations to transformations.

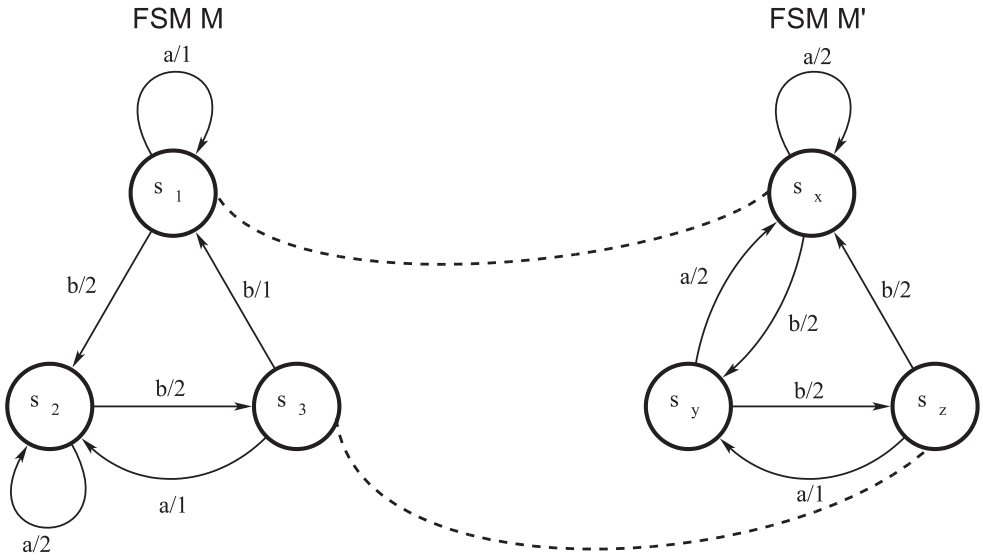


Figure 6.2: A transformation T from FSM M to M'. A dashed line between s in M and s' in M' indicates the pair $(s, s') \in T$. The states of M not touched by a dashed line are to be deleted and the states of M' not touched are to be inserted.

Algorithm 6.3.2 (Sequence of edit operations corresponding to a transformation)**.** *Take a transformation T from M to M'. Let $A \subseteqq S$ and $A' \subseteqq S'$ be the set of states of M and M', respectively, in T. Let $B \subseteqq S$ and $B' \subseteqq S'$ be the set of states of M and M' not in T.*[3] *$B = \{S \setminus A\}$, $B' = \{S' \setminus A'\}$. The edit operations corresponding to T are the following:*

- *Extra state operators: For each state in B' create a corresponding new blank state $s \in S_{new}$ in M. Now there is a bijection (say τ) between $\{S_{new} \cup A\}$ and S' (all states of M').*
- *Output operators:*
  - *For each state $s \in \{S_{new} \cup A\}$ of M for every input $i \in I$ change outputs $\lambda(s, i)$ to $\lambda'(\tau(s), i)$.*[4]
  - *For each state $s \in B$ of M for every input $i \in I$ change outputs to Υ_o.*
- *Transfer operators:*

[3] *For simplicity we use the term a state is (not) in T meaning that the given state does (not) occur in any of the state-pairs of T.*

[4] *Note that $\rho(e^0) = 0$, for example $\rho(\lambda'(s, i) = a \rightarrow a) = 0$.*

  - *For each $s \in \{S_{new} \cup A\}$ of M for every input $i \in I$ change the transitions $\delta(s, i)$ to $\tau^{-1}(\delta'(\tau(s), i))$.*
  - *For each state $s \in B$ of M for every input $i \in I$ change transitions to Υ_s.*

- *Missing state operators: Remove the – now blank – states $s \in B$ of M.*

By definition of isomorphism the resulting FSM is isomorphic to M and has the set of states $\{S_{new} \cup A\}$. The algorithm is clearly polynomial time with a complexity of $O(pn)$, where $n = |S|$ and $p = |I|$.

Example 6.3.3. Consider the example in Figure 6.2. Transformation T between M to M' is $\{(s_1, s_x), (s_3, s_z)\}$, and the corresponding edit operations are the following (we omit the description of $\rho(e^0)$ edit operations):

Extra state operators: a new blank state (s_{new_1}) needs to be inserted corresponding to s_y.

Output operators: creating outputs for the blank state: $\lambda(s_{new_1}, a) = \Upsilon_o \rightarrow 2$, $\lambda(s_{new_1}, b) = \Upsilon_o \rightarrow 2$; removing outputs of s_2: $\lambda(s_2, a) = 2 \rightarrow \Upsilon_o$, $\lambda(s_2, b) = 2 \rightarrow \Upsilon_o$; other output changes: $\lambda(s_3, b) = 1 \rightarrow 2$, $\lambda(s_1, a) = 1 \rightarrow 2$.

Transfer operators: creating transitions for the blank state: $\delta(s_{new_1}, a) = \Upsilon_s \rightarrow s_1$, $\delta(s_{new_1}, b) = \Upsilon_s \rightarrow s_3$; removing transitions of s_2: $\delta(s_2, a) = s_2 \rightarrow \Upsilon_s$, $\delta(s_2, b) = s_3 \rightarrow \Upsilon_s$; other transition changes: $\delta(s_3, a) = s_2 \rightarrow s_{new_1}$, $\delta(s_1, b) = s_2 \rightarrow s_{new_1}$.

Missing state operators: removing state s_2 (s_2 is not in T).

After the application of these edit operations we get an FSM isomorphic to M' – see Figure 6.3.

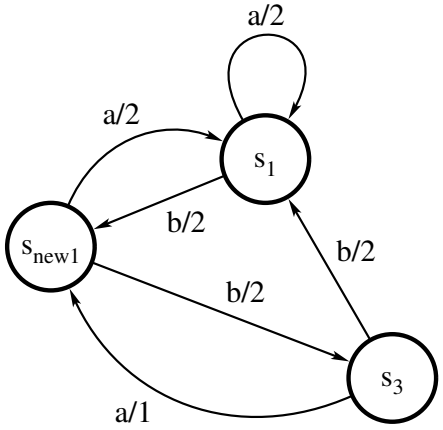


Figure 6.3: FSM M after the edit operations according to the transformation

Let T be a transformation from FSM M to M'. Then we can define a cost of T:

Definition 6.3.4. $\rho(T) = \sum_{i=1}^{k} \rho(e_i)$ where e_i are the edit operations corresponding to T (according to Algorithm 6.3.2).

6.3.1 Transformations and Edit Operations

Clearly, there are infinitely many sequences edit operations fulfilling a given transformation T – with different costs. Figure 6.4 shows the cardinality of transformations and patches for a given change.

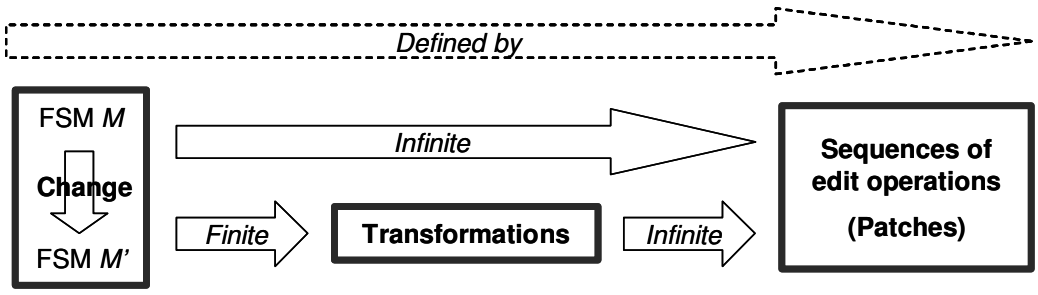


Figure 6.4: Cardinality of transformations and patches for a given change

We have to prove that the sequence of edit operations we assigned to T by Algorithm 6.3.2 is not an arbitrary choice, but the best possible sequence of edit operations for T with the lowest cost. With that we clarify the relation of transformations and edit distance, and show that the cost of the minimal transformation is equal to $dist$, i.e., a minimal cost transformation provides an optimal patch.

Transformations can be composed. Take a transformation T_1 from M to M' and a transformation T_2 from M' to M''.

Definition 6.3.5. $T_1 \circ T_2 = (s, s'')|\exists s' \in S'$ such that $(s, s') \in T_1 \wedge (s', s'') \in T_2$.

$T_1 \circ T_2$ is a transformation as well, by the definition of transformations.

Lemma 6.3.6. $\rho(T_1 \circ T_2) \leq \rho(T_1) + \rho(T_2)$.

Proof. Let $B \subsetneqq S$ and $B'' \subsetneqq S''$ be the set of states of M and M'' not in $T_1 \circ T_2$ respectively. For any $s \in S$ and $s'' \in S''$ three cases may occur. 1: $(s, s'') \in T_1 \circ T_2$; 2: $s \in B$; 3: $s'' \in B''$.

Let us only consider missing/extra state operators first. In the first case there are no extra/missing state operators on either side, thus the statement is true. In the second and third cases there are either the same extra/missing state operators on both sides; or on the left side there is $MSO : s$, $ESO : s_{new}$ and on the right

side $MSO : s$, $ESO : s_{new'}$, $MSO : s_{new'}$, $ESO : s_{new}$. Here the triangle inequality property of the cost function again ensures that the statement stands.

Next, let us consider output operators. For all (s, i) each case corresponds to a single output operation $\lambda(s, i) = a \rightarrow b$ on the left side, and at least two operations $\lambda(s, i) = a \rightarrow c_i$, ..., ..., $\lambda(s', i) = c_j \rightarrow b$ where $a, b, c_i, c_j \in O$. Thus, the triangle inequality property of the cost function again ensures that the statement stands.

It can be similarly shown that the statement also stands for transfer operators. Therefore, the lemma is proved.

□

Lemma 6.3.7. *For any sequence of edit operations $E = e_1, e_2, ..., e_k$ changing FSM M to M'' there is a transformation T from M to M'' such that $\rho(T) \leq \rho(E)$. Conversely, for any transformation T there exist a sequence of edit operations E where $\rho(E) = \rho(T)$.*

Proof. We prove the first part by induction on k. The base case is $k = 1$, so let's consider a single operation changing M to M'. For any (single) e^0 operation there is a zero cost transformation since in this case M is isomorphic to M'. For any other single output, transfer or extra state operation a transformation containing state pairs $(s_i, s'_i) \forall s_i \in S$ satisfies the inequality; and for any single missing state operator $MSO : s_j$ a transformation $(s_i, s'_i) \forall s_i \in \{S \setminus s_j\}$ satisfies the inequality. For the general case, let E_1 be the sequence $e_1, e_2, ..., e_{k-1}$ of edit operations. By induction hypothesis, there exists a transformation T_1 such that $\rho(T_1) \leq \rho(E_1)$. Let T_2 be the transformation for e_k. By Lemma 6.3.6, we have that $\rho(T_1 \circ T_2) \leq \rho(T_1) + \rho(T_2) \leq \rho(E_1) + \rho(e_k) \leq \rho(E)$.

For the second part simply take the sequence of edit operations E corresponding to transformation T according to Algorithm 6.3.2. E satisfies the equality $\rho(E) = \rho(T)$, thus the lemma is proved. □

Lemma 6.3.7 proves that our algorithm is not an arbitrary choice, it provides the best possible sequence of edit operations (with the lowest cost) for a given transformation. With that we can now state the connection between distance and transformations.

Theorem 6.3.1. *$dist(M, M') = min\{\rho(T) \mid T$ is a transformation from M to $M'\}$*

Proof. The proof follows from Lemma 6.3.7. □

Theorem 6.3.1 proves that the problem of finding the optimal patch (*dist*) can be solved by finding the minimum cost transformation. Transformations, therefore, provide a more constructive way to work with the optimal patch problem, and will be used in further discussions.

6.4 Complexity Analysis

We will show that finding $dist(M, M')$ is NP-complete, by reducing the (Perfect) Tripartite Matching problem to it. Tripartite Matching was among the original few problems that were shown to be NP-complete by Karp [Kar72]. The (Perfect) Tripartite Matching problem – or "3-Dimensional Matching" – can be stated as follows: Given 3 sets X, Y, Z with $|X| = |Y| = |Z| = n$, and a set W of triples $W \subseteq X \times Y \times Z$. Decide whether there exists a perfect matching W' such that: $|W'| = n$, and every element of X, Y and Z occurs exactly once in a triple in W'.

Theorem 6.4.1. *Finding $dist(M, M')$ is NP-complete.*

Proof. For the proof we reduce Tripartite Matching to finding the distance between two FSMs. Tripartite Matching is NP-complete and thus this reduction would indeed prove the theorem.

Let us consider any instance of Tripartite Matching: X, Y, Z and $W \subseteq X \times Y \times Z$ where $|X| = |Y| = |Z| = n$ and $|W| = q$. Without loss of generality we assume that $q > n$. Let us construct two FSMs M and M' the following way. Let I be $\{i_1, ..., i_n\}$, and create an output corresponding to each element of X, Y and Z (and the null output), O: $\{o_{x_1}, ..., o_{x_n}, o_{y_1}, ..., o_{y_n}, o_{z_1}, ..., o_{z_n}, \Upsilon_o\}$.

Create $3q$ states (outside of the null state) to construct M (see Figure 6.5(a)):

$s_{w1_x}, s_{w1_y}, s_{w1_z}, ..., s_{wq_x}, s_{wq_y}, s_{wq_z}$, i.e., three states for each triple $w \in W$. Create transitions the following way:

For each state s_{wi_x} for all $i_j \in I$: $s_{wi_x} \xrightarrow{i_j/o_{x_l}} s_{wi_y}$;

for each state s_{wi_y} for all $i_j \in I$: $s_{wi_y} \xrightarrow{i_j/o_{y_l}} s_{wi_z}$;

for each state s_{wi_z} for all $i_j \in I$: $s_{wi_z} \xrightarrow{i_j/o_{z_l}} s_{wi_z}$;

where o_{x_l}, o_{y_l} and o_{z_l} are corresponding to the elements of X, Y and Z, respectively, given by w_i.

To construct M' create $3n$ states (outside of the null state) corresponding to the elements of X, Y and Z (see Figure 6.5(b)):

$s_{x1}, ..., s_{xn}, s_{y1}, ..., s_{yn}, s_{z1}, ..., s_{zn}$. Create transitions:

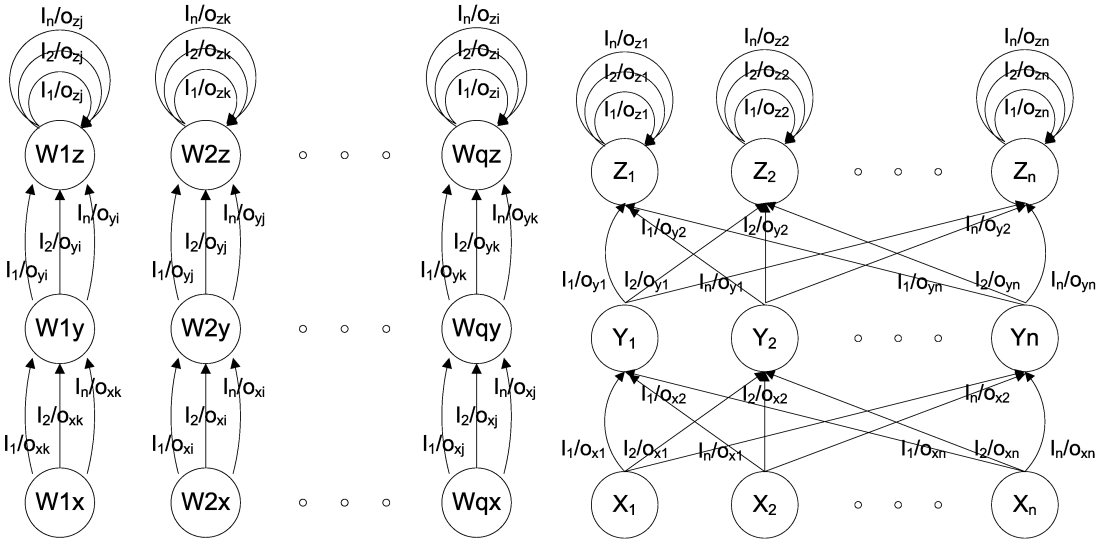
For each state s_{xi} for all $i_j \in I$: $s_{xi} \xrightarrow{i_j/o_{x_i}} s_{yj}$;

for each state s_{yi} for all $i_j \in I$: $s_{yi} \xrightarrow{i_j/o_{y_i}} s_{zj}$;

for each state s_{zi} for all $i_j \in I$: $s_{zi} \xrightarrow{i_j/o_{z_i}} s_{zi}$. Note that the outputs correspond to the elements of X, Y and Z.

Let us assume that all edit operations have unit cost. Let T be the minimum cost transformation, i.e., $\rho(T) = dist(M, M')$

A perfect tripartite match exists if and only if $dist(M, M') = 3 * (q - n) * (n * (\rho(TRO) + \rho(OO)) + \rho(MSO)) + 2n * (n-1) * \rho(TRO) = 3(q-n)(2n+1) + 2n(n-1) =$



(a) M (b) M'

Figure 6.5: Finite state machines M and M'. For clarity the state names on the figures only include the indexes. In the given case $w_1 = (x_k, y_i, z_j)$; $w_2 = (x_i, y_j, z_k)$ etc.

$f(q, n)$. In this case T includes $3n$ state pairs and defines the match between an n element subset of W and the corresponding elements of X, Y and Z.

For any T' with less then $3n$ state pairs: $\rho(T') \geq f(q, n) + (n + 2)$. For any T' where not all of the state pairs are corresponding elements of X, Y and Z: $\rho(T') \geq f(q, n) + n$ (output operators). For any T' with $3n$ corresponding state pairs but not defining a perfect tripartite match, or if a perfect tripartite match does not exist at all: $\rho(T') \geq f(q, n) + 1$. □

The result indicates that we must turn to heuristics to define an efficient algorithm approximating *dist* as there is no polynomial time algorithm for *dist* unless P = NP.

6.5 Heuristics

We show how to turn the optimal patch problem into a state-space search problem, which can be approximated with existing heuristic algorithms. Our focus here is on the formulation of the problem, not on the heuristics themselves. The discussion of different approximation algorithms, their comparative performance analysis, parameter settings, etc. are beyond our scope; it would take a whole thesis to thoroughly

investigate these problems.

Consider two FSMs M and M', and create a state-space where each state corresponds to a transformation T between M and M'. Let σ_T represent the state of the state-space corresponding to transformation T, and associate the cost of T to the given state σ_T: $\rho(\sigma_T) = \rho(T)$.

States σ_{T_1} and σ_{T_2}, corresponding to transformation T_1 and T_2 respectively, are neighbors if T_2 can be obtained from T_1 by adding or removing one pair of states in the mapping defined by T_1. Neighboring states can be reached from one another by one move in the state space. A walk is a sequence of moves between states. Clearly the state space is connected, that is, for each pair of states σ_{T_i} and σ_{T_j} there is a walk from σ_{T_i} to σ_{T_j}.

With that we turned the problem of finding $dist(M, M')$ to finding the the lowest cost among all states of the state-space.

There are many heuristic algorithms for this global optimization problem, i.e., for locating a good approximation to the global optimum of a given function in a large search space. One of the most widely used techniques is simulated annealing (SA) invented independently by S. Kirkpatrick et al. in 1983 [KGV83], and by V. Cerny in 1985 [Cer85]. SA has been successfully applied to different areas [AKvL97], and has attracted significant attention since it is suitable for large scale optimization problems, especially in the case a desired global minimum is hidden among many local minima.

The concept and name of SA is taken from thermodynamics, and is based on the manner in which liquids freeze or metals recrystalize in the process of annealing. In the annealing process a high temperature disordered melt is slowly cooled so that the system at any time is approximately in thermodynamic equilibrium. As cooling proceeds, the system becomes more ordered and approaches a "frozen" ground state. In the simulated annealing method, each state of the search space is analogous to a state of the physical system, and the function to be minimized is interpreted as the internal energy of the system in that state. Therefore the goal is to bring the system from an arbitrary initial state to a state with the minimum possible energy.

The SA algorithm takes an initial – usually random – state at temperature $Temp$ and with energy En, where En is the value of the function to be minimized, thus in our case $En(\sigma_T) = \rho(\sigma_T) = \rho(T)$. Holding $Temp$ constant, SA randomly chooses a neighbor of the current state σ and counts the change of energy $\Delta En = En(\sigma') - En(\sigma)$. If ΔEn is negative then SA moves to state σ'. If ΔEn is positive then SA still may move to σ' with a probability given by the Boltzmann factor $e^{\Delta En/Temp}$. This processes is then repeated a number of times to give good sampling statistics for the current temperature, and then the temperature is decremented. The entire process is repeated until a frozen state is achieved at $Temp = 0$.

Since SA probabilistically accepts uphill moves (moves to a state with higher

energy), the algorithm can escape from a local minimum. Initially, when the temperature is very high ($Temp \to \infty$) the algorithm will always accept any uphill move, and the simulated annealing algorithm will simplify to a type of a random search algorithm. Conversely, when the temperature is very low ($Temp \to 0$), the algorithm will always reject any uphill move, and SA will simplify to a form of a hill-climbing algorithm [HJ61]. It can be shown that – for any given finite problem – the probability that the simulated annealing algorithm terminates with the global optimal solution approaches 1 as the annealing schedule is extended.

6.6 Conclusion

Our concern in this chapter was the correction of faults found in a system, or to be more specific, the modeling and optimization of patches – changes in the behavior – considering systems given as finite state machines. Our main contributions presented here are the following:

- we defined edit operators for FSMs based on a traditional fault model, and argued that they can be used to model patches changing one system to the other;
- we introduced the new problem of finding the optimal patch, i.e., the minimal cost edit operations changing an FSM M to M' according to a given cost function;
- we defined transformations and showed their connection to sequences of edit operations;
- we conducted a complexity analysis of the problem proving that it is in fact NP-complete;
- we discussed how to turn the optimal patch problem to a state-space search problem that can be approximated with existing heuristic algorithms.

It has to be emphasized that the solution to the optimal patch problem may have some additional practical uses beyond optimizing patches. By solving the problem one finds the edit distance of two FSMs that can also be viewed as a measure of similarity for the two systems. Therefore the approach may be used for different purposes as well, such as assessing the effort of a developer or detecting plagiarism.

In this chapter we have concentrated on deterministic finite state machines. As a continuation of the current research our intention is to extend our definitions and algorithms to the non-deterministic case. Furthermore we will try to utilize our approach

for different formalisms, most importantly for EFSM-based modeling techniques that are closer to practical applications.

Chapter 7

Summary of the Dissertation

The objective of this dissertation was to investigate some fault detection, diagnosis and correction problems concerning the testing and maintenance phases of the telecommunication software development lifecycle. Our focus was on a development process supported by formal methods involving FSM and EFSM modeling techniques.

Motivation and background were discussed in Chapter 1. Chapter 2 sketched a through picture of the state of the art in FSM-based specification and testing, and provided the definitions of basic terms and notations used throughout the thesis. Our contributions to the problem of fault diagnosis were presented in Chapter 3. Chapter 4 was used to introduce the specification and testing framework for the extended finite state machine modeling technique. Chapter 5 presented the second aspect of our contributions, investigating the use of a fault-based technique – mutation analysis – for automatic conformance test selection based on SDL specifications. Chapter 6 introduced our contributions to the problem of patching, i.e., modifying an existing system, which – among other things – may be used to correct faults of a system after its deployment.

In the following we give a summary of the main contributions of our work.

7.1 Feasibility of the FSM-based Fault Diagnosis Problem

In Chapter 3, FSM-based fault diagnosis was studied, concentrating on the feasibility of the problem. We considered the diagnosis of a single transition or output fault in an FSM, and used the standard assumptions applied previously by other studies in the literature. The main results of the chapter can be summarized by the following:

- We showed that – contrary to the statements found in the literature – it is not always possible to precisely diagnose a single fault in a finite state machine. We

demonstrated that implementation machines with different single faults may be equivalent and thus have the same observable behavior despite the fact that the faults they contain differ. In this case it is impossible to decide between the faults, i.e., it is impossible to exactly locate the fault.

- We analyzed the conditions necessary to guarantee the exact localization of a single output or transfer fault in an FSM. That is, we determined a set of sufficient conditions when two (or more) implementation machines, each differing from the specification by a single dissimilar fault, cannot be equivalent. Assuming a deterministic, completely specified, strongly connected and reduced specification with reliable reset capability, we proved that:
  - it is always possible to distinguish two machines with different output faults,
  - it is always possible to distinguish a single output and a single transfer fault if the faulty machines are reduced,
  - it is always possible to distinguish two different single transfer faults if the faulty machines are reduced.

  Thus, two implementation machines, each differing from the specification by a single and dissimilar fault, cannot be equivalent if the implementation machines are reduced.
- An algorithm for the fault diagnosis problem was given incorporating our analytical results. If it is possible, the algorithm exactly locates the difference between the implementation and the specification, and when the exact localization is not possible, it provides the minimal set of all potential single faults. We provided two slightly different versions of the algorithm, complete with complexity figures.

The results presented in Chapter 3 were published in [C1].

7.2 Automatic Fault-based Test Selection for SDL Systems

In Chapter 5 we gave an algorithm to improve test sets for systems defined using the Specification and Description Language. Our approach utilized ideas of mutation analysis to automate all steps of the test selection process.

- We elaborated a special set of mutation operators extending the traditional FSM fault model defined by Chow. Our intent was to only include:

  - atomic operators
  - automatically applicable operators,
  - operators for SDL-specific mechanisms,

  in the operator set. Moreover, our operators and mutant creation strategy was defined to create a representative mutant set instead of an exhaustive one according to previous studies in the literature.

- We presented an algorithm for automatic test selection using the operators. It requires the SDL specification of the system to be tested in textual (SDL/PR) form, and assumes the existence of a large (but finite) set of test sequences given as Message Sequence Charts. This test set is subject to optimization, meaning that a subset of test sequences is selected with equivalent coverage (mutant detection ratio). Based on the method a fully automatic Test Selector tool has been developed, and was used for case studies on well-known protocols. We studied the fault detecting capability of our algorithm in comparison to some existing methods both analytically and empirically.

The results presented in Chapter 5 were published in [J0] [J4] [C4] [C6].

7.3 On the Correction of Faults – The Theory of Patching

In Chapter 6 our concern was to study the problem of of patching, i.e., modifying the an existing system, which – among other things – may be used to correct faults of a system after its deployment. The main purpose of the study was to elaborate the means of modeling changes in FSM systems, and to create a framework for optimizing patches.

Our main contributions presented here are the following:

- Proposition of edit operators for FSMs based on Chow's traditional fault model. We argued that sequences of edit operations can be used to define modifications changing any deterministic system to an other and as such they can be used to model patches for finite state machines. Furthermore, the model of patches was refined with the assignment of costs to edit operations.

- Introduction of a new problem aiming to find the optimal patch – the minimal cost edit operations – changing an FSM M to M' according to a given cost function. We emphasized that the solution to the optimal patch problem may have some additional practical uses beyond optimizing patches. By solving the

problem one finds the edit distance of two FSMs that can also be viewed as a measure of similarity for the two systems. Therefore the approach may be used for different purposes as well.

- Introduction of transformations that are used as an alternative way of describing changes. For two machines a transformation is a one-to-one mapping between subsets of states of the two machines. We defined an algorithm determining the sequence of edit operations corresponding to a transformation. We proved that the algorithm is not an arbitrary choice; it provides the lowest cost sequence of edit operations for a given transformation. Thus, we showed that the problem of finding the optimal patch can be solved by finding the minimum cost transformation.

- Complexity analysis of the optimal patch problem. We proved that the problem is NP-complete, by reducing the (Perfect) Tripartite Matching problem to it. This result indicates that heuristic approaches are needed to efficiently approximate the result. We discussed how to turn the optimal patch problem to a state-space search problem that can be approximated with existing heuristic algorithms.

The results presented in Chapter 6 were published in [C0].

Bibliography

[AB99] P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of the Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE 99)*, pages 239–248, 1999.

[AB03] Telelogic AB. Telelogic Tau. http://www.telelogic.com/products/tau/index.cfm, 2003.

[ABM98] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the IEEE International Conference on Formal Engineering Methods*, pages 46–54, 1998.

[AKvL97] E. H. Aarts, J. Korst, and P. J. v. Laarhoven. *Local Search in Combinatorial Optimization*, chapter 4, Simulated Annealing. John Wiley and Sons, 1997.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BAKD01] C. Bourhfir, E. Aboulhamid, F. Khendek, and R. Dssouli. Test cases selection from SDL specifications. *Computer Networks*, 35(6):693–708, 2001.

[BDA96] C. Bourhfir, R. Dssouli, and E. M. Aboulhamid. Automatic test generation for EFSM-based systems. Technical Report 1043, Department IRO, University of Montreal, 1996.

[BDD+96] G. v. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo. Fault models in testing. In *R. Gotzhein and J. Bredereke editors, FORTE, volume 69 of IFIP Conference Proceedings*, pages 163–178. Kluwer, 1996.

[Ber00] E. Berk. JLex: A lexical analyzer generator for Java. Princeton University, http://www.cs.princeton.edu/appel/modern/java/JLex/, 2000.

[BFV+99] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Proceedings of the* 12^{th} *International Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.

[BOY00] P.E. Black, V. Okun, and Y. Yesha. Mutation operators for specifications. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering, Proceedings ASE 2000*, pages 81–88, 2000.

[Bri88] E. Brinksma. A theory for the derivation of tests. In *Proceedings of the 8th International Conference on Protocol Specification, Testing and Verification*, pages 63–74, 1988.

[Bun00] H. Bunke. Graph matching: Theoretical foundations, algorithms, and applications. In *Proceedings of Vision Interface 2000, Montreal*, pages 82–88, 2000.

[CCG+98] V. Cantoni, L. Cinque, C. Guerra, S. Levialdi, and L. Lombardi. 2-D object recognition by multiscale tree matching. *Pattern Recognition*, 31(10):1443–1454, 1998.

[Cer85] V. Cerny. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, pages 41–51, 1985.

[Cho78] T. Chow. Testing software design modelled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.

[Csö01] T. Csöndes. *Conformance Test Suite Optimization*. PhD thesis, Budapest University of Technology and Economics, 2001.

[DDQ78] P. J. Denning, J. B. Dennis, and J. E. Qualitz. *Machines, Languages and Computation*. Prentice Hall Professional Technical Reference, 1978.

[Dij72] E. W. Dijkstra. Notes on structured programming. In *O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, Structured Programming. Academic Press*, pages 1–82, 1972.

[DS88] A. Dahbura and K. Sabnani. An experience in estimating fault coverage of a protocol test. In *Proceedings of the IEEE INFOCOM: International Conference om Computer Communications*, pages 71–79, 1988.

[EFPYvB03] K. El-Fakih, S. Prokopenko, N. Yevtushenko, and G. v. Bochmann. Fault diagnosis in extended finite state machines. In *Testing of Communicating Systems - Proceedings of 15th IFIP International Conference, TestCom*, 2003.

[EFYvB01] K. El-Fakih, N. Yevtushenko, and G. v. Bochmann. Diagnosing multiple faults in communicating finite state machines. In *Proceedings of Formal Techniques for Networked and Distributed Systems, FORTE 2001, IFIP TC6/WG6.1 - 21st International Conference on Formal Techniques for Networked and Distributed Systems*, pages 85–100, August 2001.

[EGH+97] A. Ek, J. Grabowski, D. Hogrefe, R. Jerome, B. Koch, and M. Schmitt. Towards the industrial use of validation techniques and automatic test generation methods for SDL specifications. In *Proceedings of the Eighth SDL Forum*, pages 245–259, 1997.

[Epp90] D. Eppstein. Reset sequences for monotonic automata. *SIAM Journal on Computing*, 19(3):500–510, 1990.

[FM71] A. D. Friedman and P. R. Menon. *Fault Detection in Digital Circuits.* Prentice-Hall, 1971.

[FMDM94] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, and P. C. Masiero. Mutation analysis testing for finite state machines. In *Proceedings of the ISSRE'94 - Fifth International Symposium on Software Reliability Engineering*, pages 220–229, California, USA, 1994.

[FMM+96] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and E. Wong. Mutation testing applied to validate specifications based on Petri nets. In *Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII*, pages 329–337, London, UK, UK, 1996. Chapman & Hall, Ltd.

[FMSM99] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on Statecharts. In *Proceedings of the ISSRE'99 - 10th International Symposium on Software Reliability Engineering*, pages 210–219, Florida, USA, 1999.

[FvBK+91] S. Fujiwara, G. v. Bochmann, F. Khendec, M. Amalou, and A. Ghedamsi. Test selection based on finite state model. *IEEE Transactions on Software Engenieering*, 17:591–603, 1991.

[GCR96] R. Groz, O. Charles, and J. Renévot. Relating conformance test coverage to formal specifications. In *IFIP TC6/ 6.1 international conference on formal description techniques IX/protocol specification, testing and verification XVI on Formal description techniques IX : theory, application and tools*, pages 195–210, London, UK, UK, 1996. Chapman & Hall, Ltd.

[GDvB92] A. Ghedamsi, R. Dssouli, and G. v. Bochmann. Diagnostic tests for single transition faults in non-deterministic finite state machines. In *Proceedings of the IFIP TC6/WG6.1 Fifth International Workshop on Protocol Test Systems V*, 1992.

[GDvB93] A. Ghedamsi, R. Dssouli, and G. v. Bochmann. Diagnosing distributed systems modeled by communicating finite state machines. *Revue Reseaux et Informatique Repartie*, 3(4):343–363, 1993.

[GHN93] J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by MSCs. In *Proceedings of the 6th SDL Forum*. North Holland, 1993.

[Gil61] A. Gill. State-identification experiments in finite automata. *Information and Control*, 4:132–154, 1961.

[Gil62] A. Gill. *Introduction to the Theory of Finite State Machines*. McGraw-Hill, New-York, 1962.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of* NP*-Completeness*. W. H. Freeman & Co., New York, 1979.

[GJK99] R. Groz, T. Jéron, and A. Kerbrat. Automated test generation from SDL specifications. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99 The Next Millenium, 9th SDL Forum, Montréal, Québec*, pages 135–152. Elsevier, June 1999.

[Gon70] G. Gonenc. A method for the design of fault detection experiments. *IEEE Transactions on Computers*, C-19:551–558, 1970.

[GR98] R. Groz and N. Risser. Eight years of experience in test generation from FDTs using TVeda. In *FORTE X / PSTV XVII '97: Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE X) and Protocol Specification, Testing and Verification (PSTV XVII)*, pages 465–480, London, UK, UK, 1998. Chapman & Hall, Ltd.

[GvB92] A. Ghedamsi and G. v. Bochmann. Test result analysis and diagnostics for finite state machines. In *Proceedings of the 12th International Conference on Distributed Systems*, 1992.

[GvBD93a] A. Ghedamsi, G. v. Bochmann, and R. Dssouli. Diagnosis for single transition faults in communicating finite state machines. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS'93), Pittsburgh, USA*, May 1993.

[GvBD93b] A. Ghedamsi, G. v. Bochmann, and R. Dssouli. Multiple fault diagnostics for finite state machines. In *Proceedings of the IEEE INFOCOM: International Conference om Computer Communications*, March 1993.

[Hen64] F. C. Hennie. Fault-detecting experiments for sequential circuits. In *Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110, November 1964.

[Hen01] K. L. Heninger. Specifying software requirements for complex systems: new techniques and their application. In *Software fundamentals: collected papers by D. L. Parnas*, pages 111–135, Boston, MA, USA, 2001. Addison-Wesley Longman Publishing Co., Inc.

[HJ61] R. Hooke and T. A. Jeeves. Direct search solution of numerical and statistical problems. *Journal of the ACM*, 8:212–229, 1961.

[Hog91] D. Hogrefe. OSI formal specification case study: The INRES protocol and service (revised). Technical Report IAM-91-012, Universitat Bern, Institut fur Informatik, 1991.

[Hol90] G. J. Holzmann. *Design and Validation of Protocols.* Prentice-Hall, 1990.

[Hop71] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford University, Stanford, CA, USA, 1971.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[Hud00] S. Hudson. CUP parser generator for Java. Princeton University, http://www.cs.princeton.edu/appel/modern/java/CUP/, 2000.

[Huf54] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of Franklin Institute*, 257:161–190, 1954.

[ISO89] ISO. Iso 8807: Information processing systems, Open Systems Interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour, 1989.

[ISO96] ISO/IEC. Information technology - OSI - conformance testing methodology and framework - part 3: The Tree and Tabular Combined Notation (TTCN)ISO/IEC IS 9646-3, 1996.

[IT99] ITU-T. Recommandation Z.120: Message Sequence Chart, 1999.

[IT00] ITU-T. Recommendation Z.100: Specification and description language, 2000.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[Koh78] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, New York, 1978.

[KPC02] G. Kovács, Z. Pap, and G. Csopaki. Automatic test selection based on CEFSM specifications. *Acta Cybernetica*, 15(4):583–599, 2002.

[KPY99] I. Koufareva, A. Petrenko, and N. Yevtushenko. Test generation driven by user-defined fault models. In *Proceedings of the Twelfth International Workshop on Testing of Communicating Systems IWTCS*, pages 215–233, 1999.

[Kuh92] D. R. Kuhn. A technique for analyzing the effects of changes in formal specifications. *The Computer Journal*, 35(6):574–578, 1992.

[LH01] D. Lee and R. Hao. Test sequence selection. In *Formal Techniques for Networked and Distributed Systems, FORTE 2001, IFIP TC6/WG6.1 - 21st International Conference on Formal Techniques for Networked and Distributed Systems, Cheju Island, Korea*, pages 269–284, 2001.

[LS93] D. Lee and K. Sabnani. Reverse-engineering of communication protocols. In *Proceedings of the IEEE International Conference on Network Protocols, California*, pages 208–216, 1993.

[LY94] D. Lee and M. Yannakakis. Testing finite state machines: state identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 1994.

[LY96] D. Lee and M. Yiannakakis. Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[Mea55] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical JournalJ*, 34:1045–1079, 1955.

[Mil77] G. L. Miller. Graph isomorphism, general remarks. In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 143–150. ACM Press, 1977.

[Mil80] R. Milner. A calculus for communicating systems. In *Volume 92 of Lecture Notes of Computer Science*. Springer-Verlag, 1980.

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MLS78] R. A. De Millo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[Moo56] E. F. Moore. *Automata Studies*, chapter Gedanken-experiments on sequential machines, pages 129–153. Princeton University Press, Princeton, N.J., 1956.

[NB04] M. Neuhaus and H. Bunke. An error-tolerant approximate matching algorithm for attributed planar graphs and its application to fingerprint classification. In *Proceedings of the 10th International Workshop on Structural and Syntactic Pattern Recognition, LNCS 3138*, pages 180–189, 2004.

[NT81] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. In *Proceedings of FTCS (Fault Tolerant Computing Systems)*, pages 238–243, 1981.

[Off92] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, January 1992.

[OLR+96] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[PBG04] A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in EFSM testing. *IEEE Transactions on Software Engineering*, 30(1):29–42, 2004.

[Pet01] A. Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In *MOVEP '00: Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, pages 196–205, London, UK, 2001. Springer-Verlag.

[Pra67] R. E. Prather. *Introduction to switching theory: A mathematical approach*. Allyn and Bacon Inc., 1967.

[Pra76] R. E. Prather. *Discrete Mathematical Structures for Computer Science*. Houghton Mifflin Co., Boston, MA, USA, 1976.

[PvB95] A. Petrenko and G. v. Bochmann. Selecting test sequences for partially-specified nondeterministic finite state machines. In Gang Luo, editor, *IWPTS '94: 7th IFIP WG 6.1 international workshop on Protocol test systems*, pages 95–110, London, UK, UK, 1995. Chapman & Hall, Ltd.

[PvBY96] A. Petrenko, G. v. Bochmann, and M. Yao. On fault coverage of tests for finite state specifications. *Computer Networks and ISDN Systems*, 29(1):81–106, 1996.

[RHB97] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[RW85] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985.

[SD88] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15(4):285–297, 1988.

[SEG00] M. Schmitt, M. Ebner, and J. Grabowski. Test generation with Autolink and TestComposer. *Proceedings of the 2nd Workshop of the SDL Forum Society on SDL and MSC - SAM'2000*, June 2000.

[SFSM00] S. R. S. Souza, S. C. P. F. Fabbri, W. L. Souza, and J. C. Maldonado. Mutation testing applied to Estelle specifications. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8*, Washington, DC, USA, 2000. IEEE Computer Society.

[SL88] D. Sidhu and T. Leung. Fault coverage of protocol test methods. In *Proceedings of the IEEE INFOCOM: International Conference om Computer Communications*, pages 80–85, 1988.

[SL89] D. P. Sidhu and T. K. Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, pages 413–425, April 1989.

[Sta72] P.H. Starke. *Abstract Automata*. American Elsevier Publishing Company, Inc, Amsterdam, 1972.

[SvB82] B. Sarikaya and G. v. Bochmann. Some experience with test sequence generation for protocols. In *Proceedings of the IFIP WG6.1 Second International Workshop on Protocol Specification, Testing and Verification*, pages 555–567. North-Holland, 1982.

[SvBC87] B. Sarikaya, G. v. Bochmann, and E. Cerny. A test design methodology for protocol testing. *IEEE Transactions on Software Engineering*, 13(5):518–531, 1987.

[TC988] ISO TC97/SC21. Estelle – a formal description technique based on an extended state transition model. international standard 9074, 1988.

[Tre94] J. Tretmans. A formal approach to conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 257–276. North-Holland, 1994.

[Tre00] J. Tretmans. Specification based testing with formal methods: A theory. In *Proceedings of the FORTE / PSTV 2000 – Tutorial Notes*, Pisa, Italy, October 10 2000.

[VCI89] S. T. Vuong, W. W. L. Chan, and M. R. Ito. The UIOv-method for protocol test sequence generation. In *Proceedings of 2nd IFIP Workshop on Protocol Test Systems (IWPTS'89)*, pages 161–175, 1989.

[WWS+02] X. Wang, J. T. L. Wang, D. Shasha, B. A. Shapiro, I. Rigoutsos, and K. Zhang. Finding patterns in three-dimensional graphs: Algorithms and applications to scientific data mining. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):731–749, 2002.

[WZC95] J. T. L. Wang, K. Zhang, and G. W. Chirn. Algorithms for approximate graph matching. *Information Sciences*, 82(1-2):45–74, 1995.

[YCL98] N. Yevtushenko, A. Cavalli, and L. Lima. Test suite minimization for testing in context. In *IWTCS: Proceedings of the IFIP TC6 11th International Workshop on Testing Communicating Systems*, pages 127–146, Deventer, The Netherlands, The Netherlands, 1998. Kluwer, B.V.

Publications

Journal papers

[J0] Gábor Kovács, **Zoltán Pap**, Gyula Csopaki and Katalin Tarnay. Iterative automatic test generation method for telecommunication protocols. *Computer Standards & Interfaces*, Accepted paper (Corrected proof in Press), 2005.

[J1] Gábor Vincze, **Zoltán Pap** and Róbert Horváth. Peer-to-Peer Based Distributed File Systems, *International Journal of Internet Protocol Technology (IJIPT)*, Accepted paper, 2005.

[J2] Gábor Vincze, **Zoltán Pap**, Róbert Horváth. Peer-to-peer alapú elosztott fájlrendszerek. *Hiradástechnika*, Volume LX, Pages 21-26, 2005. (in Hungarian).

[J3] Gusztáv Adamis, Róbert Horváth, **Zoltán Pap** and Katalin Tarnay. Standardized languages for telecommunication systems. *Computer Standards & Interfaces*, Volume 27, Number 3, Pages 191-205, March 2005.

[J4] Gábor Kovács, **Zoltán Pap**, Gyula Csopaki. Automatic Test Selection Based on CEFSM Specifications. *Acta Cybernetica*, Volume 15, Number 4, pages 583-599, 2002.

[J5] **Zoltán Pap**, Zoltán Rétháti, Róbert Horváth, Gusztáv Adamis. Standardized Event Pair Based Test Generation Method Using TSS&TP. *Acta Cybernetica*, Volume 15, Number 4, Pages 653-667, 2002.

[J6] Róbert Horváth, **Zoltán Pap**, Zoltán Rétháti. Development of Telecommunications Software Using Formal Languages. *Magyar távközlés selected papers 2000*, Pages 48-50, 2000.

[J7] Róbert Horváth, **Zoltán Pap**, Zoltán Rétháti. Távközlési szoftver fejlesztés formális nyelvek felhasználásával. *Magyar távközlés*, Volume X, Number 7, Pages 3-6, July 1999. (in Hungarian).

[J8] **Zoltán Pap**. IP alapú távközlés. *Magyar távközlés*, Volume X, Number 6, Pages 19-22 June 1999. (in Hungarian).

Conference papers

[C0] **Zoltán Pap**, Gyula Csopaki, Sarolta Dibuz. On the Theory of Patching, in *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods, SEFM 2005,* Pages 263-271, Koblenz, Germany, September 5-9, 2005.

[C1] **Zoltán Pap**, Gyula Csopaki, Sarolta Dibuz. On FSM-based Fault Diagnosis, in *Proceedings of the Testing of Communicating Systems, 17th IFIP TC6/WG 6.1 International Conference, TestCom 2005,* Pages 159-174, Montreal, Canada, May 31 - June 2, 2005.

[C2] Gusztáv Adamis, **Zoltán Pap**, Róbert Horváth. Introduction of Aspect-Oriented Methodology to Formal Description Techniques, in *Proceedings of the IFIP WG6.3 Workshop on Next Generation Networks & EUNICE'2003,* Balatonfüred, Hungary, September 8-10, 2003.

[C3] Gusztáv Adamis, **Zoltán Pap**, Róbert Horváth. Self-adaptive Testing of Communication Protocols, in *Proceedings of IWSAS 2003, International Workshop on Self-Adaptive Software,* Arlington, VA, USA, June 9-11, 2003.

[C4] Gábor Kovács, **Zoltán Pap**, Dung Le Viet, Antal Wu-Hen-Chang, Gyula Csopaki. Applying Mutation Analysis to SDL Specifications, in *Proceedings of the Eleventh SDL Forum "System Design",* Pages 269-284, Stuttgart, Germany, 1-4 July, 2003.

[C5] **Zoltán Pap**, Zoltán Rétháti, Gusztáv Adamis. Standardized Beta Test Subsequence Based Test Generation Method Using Formal Documents, in *Proceedings of the 4th Symposium on Wireless Personal Multimedia Communications - WPMC'01,* Aalborg, Denmark, September 9-12 2001.

[C6] Gábor Kovács, **Zoltán Pap**, Gyula Csopaki. Automatic Test Selection Method Applied to WAP, in *Proceedings of IFIP Workshop on IP and ATM Traffic Management WATM'2001 & EUNICE'2001,* Pages 115-121, Párizs, France, September 3-5, 2001.

[C7] **Zoltán Pap**, Zoltán Rétháti, Gusztáv Adamis. Alternative Test Generation Method Based on Atomic Test Cases, in *Proceedings of IFIP Workshop on IP and ATM Traffic Management WATM'2001 & EUNICE'2001,* Pages 107-114, Párizs, France, September 3-5, 2001.

[C8] Róbert Horváth, **Zoltán Pap**, Zoltán Rétháti. Protokoll fejlesztés és tesztelés formális nyelveken, in *Proceedings of Students Scientific Conference,* Budapest University of Technology and Economics, Budapest, Hungary, November, 1999.