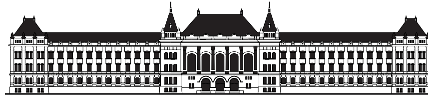


BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS
DEPT. OF TELECOMMUNICATIONS AND MEDIA INFORMATICS

SELECTIVE AUTOMATIC TEST GENERATION FOR EFSM FORMAL MODELS WITH FAULT AND STRING EDIT DISTANCE BASED METHODS

Gábor Kovács

Ph.D. Dissertation

Supervised by

Gyula Csopaki, Ph.D. and Katalin Tarnay, D.Sc.
High Speed Networks Laboratory
Dept. of Telecommunications and Media Informatics
Budapest University of Technology and Economics

Budapest, Hungary
2009

© Copyright 2009
Gábor Kovács
High Speed Networks Laboratory
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.

Contents

List of Tables vii

List of Figures viii

List of Abbreviations x

List of Symbols xii

Abstract xiv

Acknowledgements xv

1 Introduction 1

1.1 Objectives . . . 3
1.2 Structure of the thesis . . . 4

2 Preliminaries 6

2.1 Protocol engineering methodology . . . 6
2.1.1 Formalization and formal description techniques . . . 8
2.2 Formal modelling . . . 9
2.2.1 Finite state machine . . . 9
2.2.2 Extended finite state machines . . . 11
2.2.3 Input/output transition systems . . . 13
2.2.4 Input/Output Transition Systems (IOTSs) and Finite State Machine (FSM) traces . . . 13
2.3 Modelling and testing . . . 14
2.3.1 Conformance testing . . . 15
2.3.2 Test case, test suite . . . 15
2.3.3 Implementation relations, conformance relation . . . 15
2.3.4 Conformance relation formally . . . 16
2.3.5 Timed models . . . 17
2.4 Automatic test derivation for extended finite state machines . . . 18
2.4.1 Test generation for finite state machines . . . 18
2.4.2 State space exploration, search methods . . . 19

2.4.3 Simple search methods . . . 19
2.4.4 Search methods employed in test generation tools . . . 22
2.4.5 Functions for test generation and test set optimization . . . 23

3 Mutation based automatic test generation for Specification and Description Language (SDL) specifications 24
3.1 Background and related work . . . 24
3.1.1 Related work . . . 24
3.1.2 Mutation analysis . . . 26
3.2 Test generation for SDL models . . . 27
3.2.1 Algorithms for test generation . . . 28
3.2.2 The test generation framework and tool . . . 32
3.3 Empirical analysis . . . 35
3.3.1 Mutation operators for SDL systems used in the experiments . . . 35
3.3.2 SDL systems examined . . . 37
3.3.3 Test generation for the INRES protocol . . . 38
3.3.4 Test generation for the Conference Protocol . . . 39
3.3.5 Test generation for the WAP WTP . . . 39
3.3.6 Test generation for the Signalling System No. 7 (SS7) Message Transfer Part (MTP)2 . . . 40
3.3.7 Performance . . . 41
3.4 Conclusion . . . 42

4 String edit distance based test set optimization 43
4.1 Background and related work . . . 43
4.1.1 Related work . . . 43
4.1.2 String representation of traces . . . 44
4.1.3 Trace distance . . . 46
4.2 Maximizing the diversity in trace sets . . . 48
4.2.1 Finding the cardinality of the optimal subset . . . 48
4.2.2 Optimizing the test suite using the given density and cardinality . . . 49
4.3 Empirical analysis . . . 53
4.4 Conclusion . . . 55

5 Iterative test specification 56
5.1 Related work . . . 57
5.2 Iterative test generation algorithms . . . 57
5.2.1 Evolutionary algorithms in test generation . . . 57
5.2.2 Iterative test derivation . . . 57
5.3 Iterative test generation with fault based test suite metrics . . . 58
5.3.1 Updating a test set for the next cycle . . . 64
5.3.2 Comparison of the algorithms . . . 65
5.4 Iterative test generation with string edit distance based metrics . . . 70

5.5 Conclusion 73

6 Summary **74**
6.1 Mutation based automatic test generation for SDL specifications 74
6.2 String edit distance based test set optimization 75
6.3 Iterative test specification 76

Bibliography **78**

List of Tables

2.1 State transition table 11

3.1 Mutation Operators for SDL 37
3.2 Complexity of the systems examined 37
3.3 Data from the INRES case study 38
3.4 Data from the Conference Protocol case study 39
3.5 Data from the Wireless Application Protocol (WAP) Wireless Transaction Protocol (WTP) experiment 40
3.6 Data from the SS7 MTP2 experiment 41
3.7 Execution times ([hh:]mm:ss) 41

4.1 Signal sequence to character mapping for ISAP Manager Ini 46
4.2 Properties of the unfolded SDL processes 53
4.3 Results of the selection experiments 54
4.4 Execution times 54

5.1 The number of mutant systems and parameters for test generation 66
5.2 The number of test cases in the optimized suite 66
5.3 Fault detection ratio [%] 67
5.4 Execution times in form of hh:mm 67

List of Figures

1.1 Introducing an iterative cycle in the test generation 4

2.1 The protocol design methodology standardized in International Telecommunication Union (ITU) [IT97b] 7
2.2 Abstraction and concretization [PP05] 9
2.3 State transition graph of an FSM 11
2.4 State graph of a sample Extended Finite State Machine (EFSM) 12
2.5 Graph representation of a sample IOTS 14

3.1 Mutation technique in program/model based testing 26
3.2 Mutation based test generation 27
3.3 Mutation based test derivation 29
3.4 Mutation Analysis of SDL specifications Based on an Existing Test Set 31
3.5 A sample matrix of criteria 31
3.6 The main dialog 32
3.7 Components of the tool 33
3.8 The test selection dialog 34

4.1 INRES ISAP Manager Ini process [EHS97] 45
4.2 The state transition diagram of the FSM of the ISAP Manager Ini process from the sample INRES protocol in original form and after event to character mapping 45
4.3 The Σ test set of eight Message Sequence Charts (MSCs) generated with random walk. (IMI abbreviates `ISAP Manager Ini` from the INRES system) . 47
4.4 The **A** matrix and the bipartite graph G' constructed from it 51
4.5 The flow problem equivalent to the maximum distance k-cardinality matching 52

5.1 Functional architecture for fault based iterative test generation 61
5.2 Examples for the test suite fitness evaluation 63
5.3 Generating a new test case or a new postfix to a test case. Test case σ is the original test case to be modified. Test case σ'_a is a new test case. Test case σ'_b is the modification of σ such that events of σ are removed after the $\mathsf{fails_after}(\sigma, m)$ value (i.e. from the third event), and a new postfix is added. 64
5.4 Iterative fault detection experiments on sample systems 68

5.5 Iterative fault detection experiments on real-life systems 69
5.6 Transition coverage during the iterations 73

List of Abbreviations

ACT Asynchronous Communication Tree

ASN.1 Abstract Syntax Notation One

CCS Calculus of Communicating Systems

CEFMS Communicating Extended Finite State Machine

CFMS Communicating Finite State Machine

CSP Communicating Sequential Processes

CUP Constructor of Useful Parsers

EFSM Extended Finite State Machine

ETSI European Telecommunication Standardization Institute

FIFO First-In First-Out

FSM Finite State Machine

IETF Internet Engineering Task Force

IOTS Input/Output Transition System

ISO International Organization for Standardization

ITU International Telecommunication Union

IUT Implementation Under Test

LOTOS Language Of Temporal Ordering Specification

LTS Labelled Transition System

MDA Model Driven Architecture

MMS Multimedia Messaging Service

MSC Message Sequence Chart

MTP Message Transfer Part

OSI Open Systems Interconnection

PDU Protocol Data Unit

PSTN Public Switches Telephone Network

SDL Specification and Description Language

SDL/PR SDL Phrase Representation

SS7 Signalling System No. 7

TTCN-2 Tree and Tabular Combined Notation version 2

TTCN-3 Testing and Test Control Notation version 3

UIO Unique Input Output

UML Unified Modeling Language

WAP Wireless Application Protocol

WTP Wireless Transaction Protocol

List of Symbols

A	action(s) on variables
C	an alphabet of characters
E	set of edges of a graph
F	a set of features
G	a graph
I	set of input events
O	set of output events
M	a set of systems modelled either as FSM or as EFSM
N	a set of nodes of a graph
P	predicate(s) on variables
S	set of states
V	set of variables
T	set of transitions
TP	set of transition parts
X	set of input strings
Y	set of output strings
d	distance function
e	an edge of a graph
f	a feature
h	transition function of FSM
i	an input symbol
l	an edge-label of a graph
m	a system modelled either as FSM or as EFSM
$null$	the unobservable output event
o	an output symbol
s	a state
t	a transition
v	a variable
x	a string of input events

y	a string of output events
$\mathbf{A}$	a matrix of boolean values
$\mathbf{C}$	a matrix of boolean values
$\mathbf{D}$	distance matrix
$\mathbf{L}$	a vector structure
$\mathbf{M}$	a matrix
$\mathbf{s}$	a vector of boolean values
$\mathbf{\Phi}$	feature vs. test case matrix
$\mathcal{M}$	set of mutant machines
δ	next state function
ε	approximation threshold
η	an input or output or timeout event
ϑ	timeout action in the specification
θ	timeout action in the tester
λ	output function
μ	event to character mapping
σ	ordered string of input and output events, an IOTS
ϕ	an element of the feature matrix
ψ	a maximum pairing in a bipartite graph
ω	mutation operator
Θ	set of timers in the implementation
Λ	length bound on traces
Σ	set of test cases or set of IOTSs
Ω	set of all possible mutations

Abstract

A well known barrier in front of the widespread application of automated formal model based system development processes is the limited capability of generating adequate and compact abstract test sets automatically. It has been written down in several articles and books that, for systems of larger scale, guided random walk based strategies should be preferred. This dissertation proposes three solutions that can work as selective extensions to existing automatic test generation algorithms for system models expressed as EFSM or more specifically in SDL. The mutation analysis based approach automatically generates test cases that can detect a given set of faults defined by a fault model. The string edit distance based test selection algorithm gives the most compact subset of a test set for a given density parameter. Finally, an iterative extension to the test specification activity is proposed and investigated that can reduce the computation complexity of selective automatic test generation and help the test set maintenance in iterative development processes.

Acknowledgements

I have been fortunate to have the help and support of many people over the last several years. Without their help this dissertation would not have been completed.

First of all, I would like to express my deepest gratitude to my supervisors Gyula Csopaki, who has been supervising my work since my undergraduate studies, and Katalin Tarnay for their encouraging guidance and invaluable support.

I wish to thank the High Speed Networks Laboratory for the support of my research activities in the field of protocol engineering over so many years, the possibility of participating in the project MUSE and this long lasting patience. I would also like to acknowledge the help I received from the people at Department of Telecommunications and Media Informatics.

My special thanks to my past and present colleagues at the protocol engineering group for the fruitful collaborative work, and to the undergraduate students I supervised for their help in developing the software tools I use in the dissertation.

Gábor Kovács
Budapest, 2009

Chapter 1

Introduction

Telecommunication systems and hence telecommunication protocols being developed today are getting more and more robust. The increasing size and complexity addresses an increased challenge for the designers, developers and test engineers. The traditional approach for simplifying problems is creating models for the system under development, which can aid engineers during the whole life cycle of the project.

A model is a simplification of the system that hides away some details as requirements during an activity in the development process. Models are usually specified using a language that can have a graphical representation in the form of a set of diagrams. The model can be semi-formal or formal depending on that language. If the modelling language has a well established mathematical background, which provides the means to specify all selected properties in an unambiguous and concise way, we talk about formal modelling. If any part of the requirements is still expressed informally in natural language, the model is said to be semi-formal. Formal modelling enables the mathematical analysis of the created model, which has an all important advantage: early fault detection that reduces the cost of error correction.

Besides, a model that has mathematically been proven to reflect the selected subset of requirements can support implementation by generation of code skeletons with the specified behavior and provide the basis for testing and test generation. Creating lower level models during the design activity and the automation of these generation processes can significantly decrease the human resource needed for the implementation and test development. Due to the growing complexity and robustness of systems, the latter is is a crucial and increasingly expensive part of the – software – development process, where any reliable automation technique results in cost reduction.

Modelling can take place in different stages of the design and therefore play different roles [PP05]. The model can be common for system development and test development; in this case we are talking about model driven development. In case of model based testing, the model is automatically or manually built after the implementation to perform testing on the model and thus simplify the testing task. The most general approach is when separate models are created for the implementation and testing activities. This thesis considers the first scenario, model based development, which is in line with the SDL methodology

[IT97b], where the formal model is built from the requirements and used as a starting point for implementation and test generation.

Conformance testing, which provides the means to check whether the implementation works as required by the specification, is not intended to be exhaustive, and a successfully passed test suite does not imply a 100-percent guarantee. But it does ensure, with a reasonable degree of confidence, that the implementation is consistent with its specification. The easiest way to increase the level of confidence that the Implementation Under Test (IUT) conforms to the specification is to add more and longer test cases to a test suite. However, as the size of the test suite grows so does the execution time increase, because taking back the system under test to its initial state is costly. To reduce this time requirement more compact test suites must be generated manually or automatically.

The manual way of test case development is rather time consuming and requires the effort of many human experts. Test engineers use a textual specification as starting point, and formulate a model of that in their mind. Based on their experience, the knowledge of typical faults that developers may commit, they define a compact set of test purposes. The test purposes are then transformed into test cases that are likely to detect faults. This method is employed almost everywhere in the software and telecommunication industry. It has high cost for human resources, and takes as much time as the development. On the other hand it provides test suites of high quality.

The systematic test case development [GHN93] aims to decrease this cost of human resource. It assumes that a formal model is generated during the design phase of the product life cycle. The set of test purposes is determined by the formal model, for instance, a test purpose can defined for each transition in the model. The test purposes are systematically transformed into test cases. This approach does not require well-trained expert, so its human resource cost is lower, and still provides test suites of high quality. The weakness of this method is that it requires the existence of a formal model which has a high additional cost in the design phase of the life cycle. Another weakness is that as the abstraction level of the formal model increases so does the size of the generated test suite and therefore the time required for test execution grow.

The development of formal specifications allow the usage of computer aided automatic test generation methods. This way of test generation has reduces human resource cost at the test specification phase and requires much less time to produce a test suite. However, time must be invested to develop the formal specification and prove its correctness. The generated test suite is large and highly redundant and its quality is uncertain, so test selection has increased significance.

Research in the field of testing has established various computer-aided test generation methods [LY96, KPC01, WRQC08] and tools like Autolink [SKGH97], TestComposer, TorX [TB03], TGV [JJ02] or Phact [HFT00] for different EFSM formalisms. [1]. These tools typically have a sound mathematical background, but are often criticized for resulting in excessive number of test cases in industrial-size application. In case of complex system specifications they impose unacceptable time requirements for test execution. Two test

[1] A detailed survey on the available test generation tools is in [BFS05]

generation strategies are employed in these tools. One approach is to create an exhaustive suite by generating all possible test cases. The other approach assumes that huge number of test cases should provide a high level of confidence.

1.1 Objectives

Increasing the level of confidence of the automatically generated test suite significantly increases the size of the test suite and the execution time. The optimization of these contradictory parameters, execution time and level of confidence, address together an important challenge. Automatic test generation must, therefore, consider the generation-time detection and reduction of redundancies among the large number of test cases to be able to compete with manual methods.

Test suite reduction methods already exist and are used for preliminary selection of a subset of test cases for the test execution activity to reduce the execution time [HGS93, VAC92, CKS01, WP02]. Their aim to minimize the cardinality of the target test set without sacrificing its quality. The quality of the test set is measured with coverage metrics. However as shown in [HGS93] the selection problem is NP-hard, so approximate solutions are employed [CKS01, WP02].

The main goal of this thesis is to investigate issues related to automatic test generation: propose efficient methods and algorithms that can reduce a huge set of automatically generated test cases while preserving its quality, and support generation time maintenance and compression of test sets. Two methods are proposed that aid the EFSM model based automatic compression of test sets and algorithms to reduce the complexity of the whole test generation process.

The first method is based on mutation analysis [DMLS78], which has initially been used for code based software verification and validation. Recent research has shown that it can be applied to various formal models as well. The main focus is the automation of the test generation and selection process: an algorithm based on mutation analysis is proposed for automatically generating compact test sets. The selection criteria are provided by the mutant systems generated by means of mutation operators.

The second method utilizes the string edit distance-based test coverage metric to reduce MSC test sets. Existing string edit distance-based methods can decide if two test cases are redundant or not. The proposed solution can decide which of the two test cases should be removed from a test set. A two-step selection algorithm is proposed to find the minimum cardinality subset with the highest possible diversity. First, the minimum cardinality for a given approximation threshold is determined by reducing the distance based selection problem to an assignment problem in bipartite graphs. Then, a test set with the highest overall internal edit distance is selected from all subsets with that computed cardinality. It is shown that the algorithm runs in polynomial time of the size of the input test set and that it is independent of the size of the system.

The two methods referred above have been implemented in a Test Selector software tool that accepts an SDL specification of the system under test and a test set defined in MSC.

Empirical study has evaluated them against existing symbol coverage based test selection approaches by conducting experiments on the well-known INRES [EHS97] and Conference Protocol [HFT00].

The third solution focuses on the test generation from the methodological point of view. The test generation activity of the design process is considered to be composed of two separate tasks: test derivation and reduction. The derivation derives a test suite from the formal specification, the reduction makes that suite more compact. Instead of the standard waterfall model, an iteration cycle is constructed from test derivation and a separate reduction steps. Figure 1.1 shows this process. This technique reduces the total cost of test generation by deriving a small number of test cases, and selecting the adequate ones in each cycle, hence maintaining the test set at the price of finding only a close-to-optimal solution. In each iteration cycle, first the actual test suite is modified, and then reduced. Based on the test selection criteria, it is possible to define a metric for test suite evaluation and compare two test suites: the new one with the one of the previous iteration cycle, obtaining a better suite. This method is especially effective for test selection methods with high computation demands like the first, fault based solution, but it can co-work with string edit distance based metrics as well.

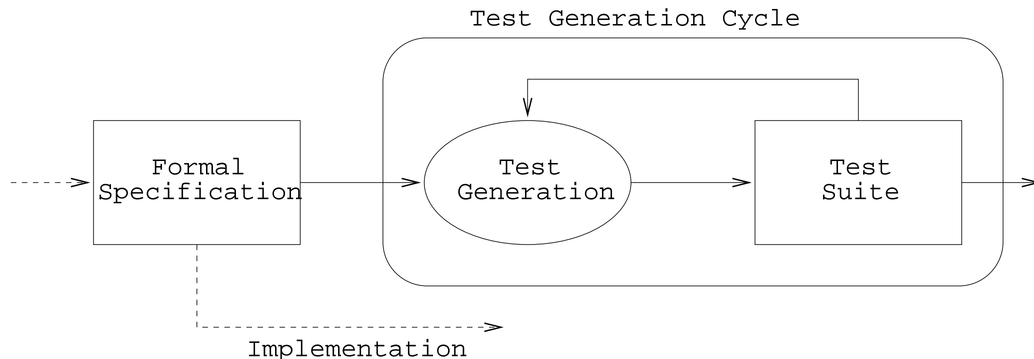


Figure 1.1: Introducing an iterative cycle in the test generation

1.2 Structure of the thesis

The thesis consists of six chapters. After this short introduction, fundamental notations, definitions, and terms are given in Chapter 2. Section 2.1 gives an overview of a design process, more specifically the one standardized in [IT97b], focusing on the activity of formalization and formal description techniques. In Section 2.2 three formalisms are defined: FSMs, EFSMs and IOTSs that are used later to give models of systems, testers and test cases. The next section (Section 2.3) gives a short summary of the fundamentals of model or specification based testing defined in the *Specification Based Testing with Formal Methods*

[Tre00] framework and its adaptation to the FSM/EFSM world. Finally in Section 2.4 automatic test generation methods for FSM and EFSM models are presented which are used in test generation tools.

In Chapter 3 I propose a mutation analysis based approach for automatic test generation that takes an SDL specification and a formal fault model as input and outputs a set of test cases in the form of MSCs. First, related work and mutation analysis are introduced in Section 3.1. Then, test generation and selection algorithms are proposed. The software tool we developed that implements this method is described in Section 3.2.2, and simulation based experiments made with that tool are evaluated in Section 3.3.

The extension to the string edit distance based methodology is presented in Chapter 4. First, a brief overview of the papers of Vuong and Feijs is given and assumptions and notations are introduced in Section 4.1. Section 4.2 describes the procedure of distance maximization among the test cases, an example illustrates each step of the method. Section 4.3 compares the string edit distance based test selection method with two other approaches; the fault based proposal and a coverage based solution using the sample SDL systems INRES and Conference Protocol.

The iterative approach for test generation is proposed in Chapter 5. In Section 5.2 two abstract iterative test generation methods are presented. Fault based or string edit distance based metrics derived from the solutions of Chapter 3 and Chapter 4 concretize the approach: the fault model based in Section 5.3 and the string edit distance based in Section 5.4. Then, in Section 5.3.2 results of fault based iterative test generation experiments are presented: different test derivation methods are compared through sample and standardized real-life protocols.

Chapter 6 concludes the thesis.

Chapter 2

Preliminaries

This chapter introduces basic definitions, formalism and terms used throughout the thesis. Section 2.1 gives an overview of a design process, more specifically the one standardized in [IT97b], focusing on the activity of formalization and formal description techniques. In Section 2.2 three formal abstractions are defined: FSM, EFSM and IOTS models that are used later to give models of systems, testers and test cases. The next section (Section 2.3) gives a short summary of the fundamentals of model or specification based testing defined in the *Specification Based Testing with Formal Methods* [Tre00] framework and its adaptation to the FSM/EFSM world. Section 2.4 first overviews the test generation for FSM models and discusses challenges of its practical application. Then EFSM based test generation and state space exploration strategies are introduced that try to tackle the problem of state space explosion, and the methods used in commercial tools are presented.

2.1 Protocol engineering methodology

"Communication protocols are the rules that govern the interaction between different components in a distributed system" – Holzmann [Hol91]. To be able to use different implementations of a specific protocol from different vendors, developers must have an agreement on the set of procedure rules. The agreement is settled by standardization bodies like ITU or European Telecommunication Standardization Institute (ETSI) or Internet Engineering Task Force (IETF) in the telecommunications industry, or International Organization for Standardization (ISO), which does not have a specific area in focus.

These behavior rules must be phrased unambiguously to avoid misinterpretation, so different implementations of different developers or vendors should be able to communicate with each other. Thus protocol engineering has a special importance.

The protocol engineering process, just like any other engineering process, consists of several well-separated activities [IT97b], though not all of them appear in all projects. The activities are done in different departments or even different institutes (the requirements are collected in standardization bodies, vendors implement the standard) that produce outputs for each other. This methodology has been developed in ITU, but its application domain is

not limited to the telecommunication sector.

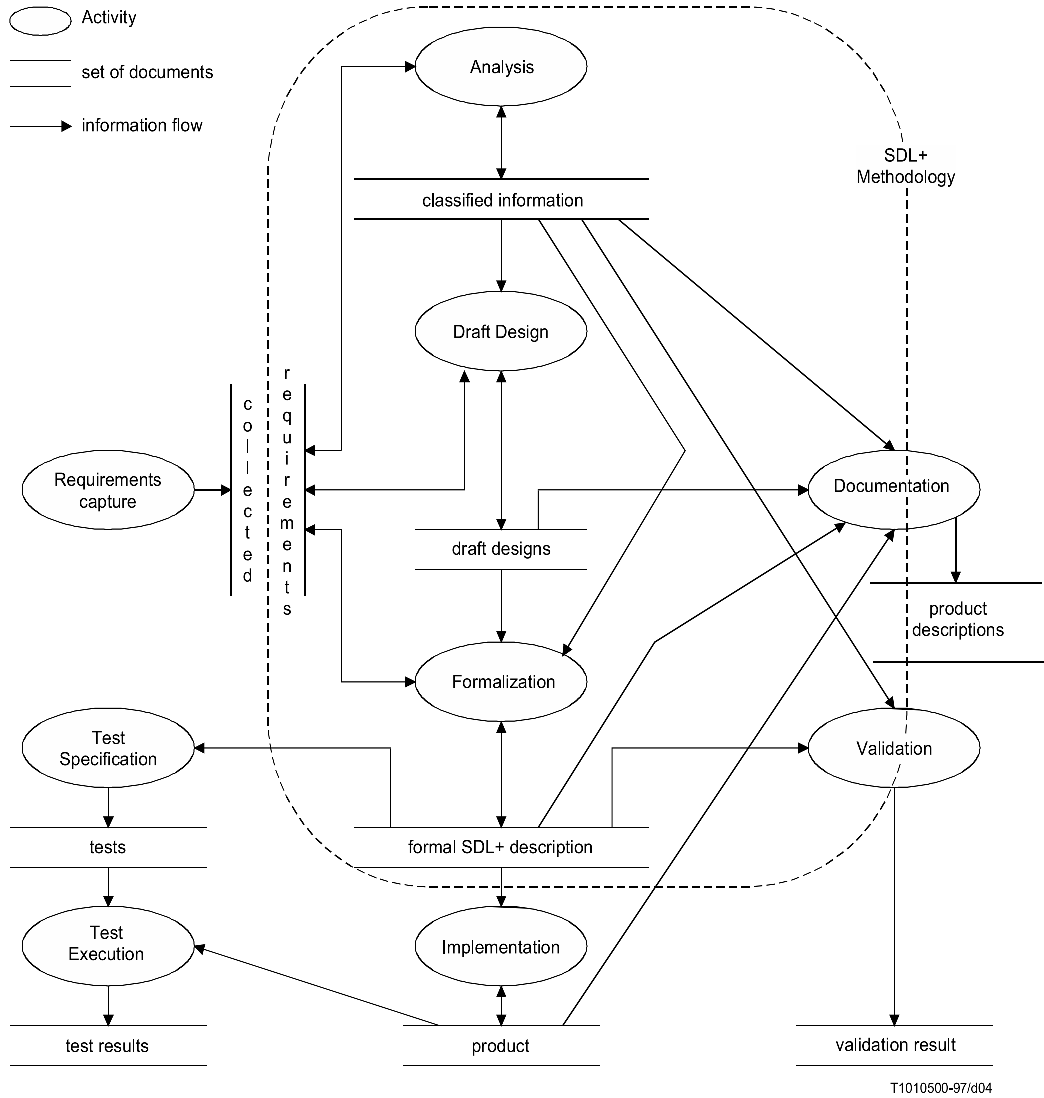


Figure 4-3/Suppl. 1 to Rec. Z.100 – The SDL+ methodology

Figure 2.1: The protocol design methodology standardized in ITU [IT97b]

The methodology shown in Figure 2.1[IT97b] starts with the requirement capture activity. After requirement classification and draft design a formal model is created that provides the basis for test specification, implementation and validation.

Requirements capture finds the ideas to be embodied in the system under development, the criteria that must be fulfilled by the application. In the analysis phase the requirements collected are explored and the information those contain is organized. Design creates a view of the model of a system and its environment, and focuses on the functions required to support the service. It is separated into two stages. First, the structured information is transformed into partial or partial informal specification in the draft design phase, then a formal system specification is produced. The validation activity evaluates that the formal model is complete and meets the criteria expressed by the requirements the system is based on. Test specification provides input for test execution in form of a set of tests for particular purposes such as testing the conformance of an implementation to a formal SDL description.

In the next sections the formalization that plays a central role in this methodology is discussed detail.

2.1.1 Formalization and formal description techniques

The formalization activity of the protocol engineering process takes place iteratively and in parallel with other closely related activities: requirement analysis and draft design [IT97b]. For some protocols these tasks are worked out within a standardization institute.

Protocol specifications are usually given informally in natural language, but a model given in standardized formal description techniques is available in some cases (like for SS7 MTP by ITU [IT97a]). Informal languages are easy to understand for the developers, but lack the capability of defining unambiguous and concise specifications. Formal specifications are on the contrary excellent means for system modelling, and provide the basis for validation and automatic test specification. However the generation of the formal model is very time consuming.

In the field of communication protocol modelling the theory of two major abstraction technique tracks that have been developed independently are used nowadays: FSM based modelling and process algebra. Three protocol specification languages based on these two tracks have been developed by two standardization bodies: SDL [IT00b], Estelle [ISO97] and Language Of Temporal Ordering Specification (LOTOS) [ISO89]. SDL that has been developed by ITU is based on the Communicating Extended Finite State Machine (CEFMS) concept, where machines are coupled fundamentally asynchronously. It has two equivalent representation forms: a flowchart like graphical and a programming language that is textual. Estelle developed by ISO uses the same concepts as SDL. LOTOS developed also within ISO based on the Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP) [1], models synchronous communication between processes. It is also called a process algebra, and has a programming language-like format. [2]

[1]The data types of LOTOS are derived from the ACT ONE language

[2]Though these three modelling languages can express both the behavior of the communicating peers and the message formats (protocol and service data units), the latter is usually given in Abstract Syntax

In the last decade the semi-formal Unified Modeling Language (UML) has become the dominant modelling language in software engineering. The concept of UML fits in the draft design activity of the design procedure in Figure 2.1. Formal properties cannot be checked, but it provides an excellent basis to generate code skeletons for object-oriented high level languages. There have been studies on supporting the test specification as well.

2.2 Formal modelling

Modelling is a mapping between a concrete and an abstract world. The model may exist formally or can be implicit and exist only in the minds of engineers. During modelling some of the requirements and environmental parameters are hidden away and replaced with (semi-)formal information. Such abstraction is shown in Figure 2.2 from [PP05]. When concretized the *input* i of the model on the right hand side for instance is replaced with $PSOVerifyDigSig(SigCA)$.

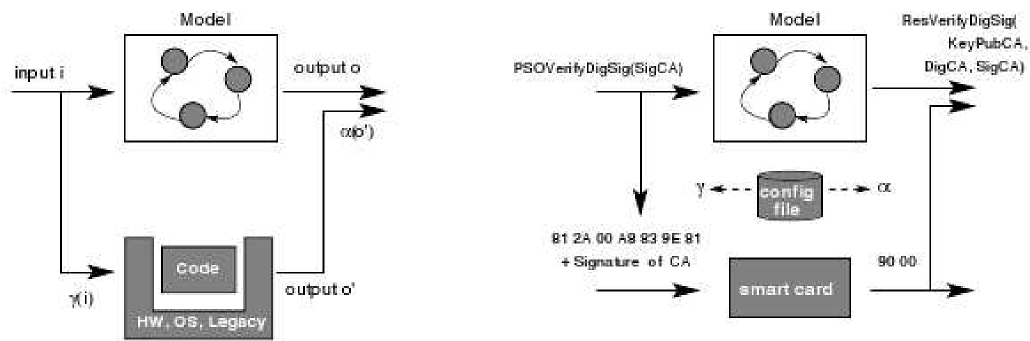


Figure 2.2: Abstraction and concretization [PP05]

This section focuses on formal abstractions: finite state machines, extended finite state machines and input/output transition systems.

2.2.1 Finite state machine

Wide area of systems can be modelled using the Finite State Machine (FSM) abstraction. Beside communication systems and protocols [Cho78] it is also used in hardware design for modelling sequential circuits, in compiler theory [AU72] for lexical analysis (for instance in tool JLex [BA96]) and in other fields of computing that require pattern recognition like artificial intelligence.

As the name indicates, such machines have a finite number of states. A state is a certain kind of memory that symbolizes assumptions the machine makes about its environment and defines the set of possible actions based on the history of previous events.

Notation One (ASN.1) [IT94].

Two representations of the FSMs exist: *Mealy machines* [Mea55] and *Moore machines* [Moo56]. While in Mealy machines the output event produced is determined by the actual state and the received input event, in Moore machines the output event depends only on the current state. It has been shown that the two representation forms can be transformed into each other. This thesis considers Mealy machines.

Definition 2.2.1 (Mealy Finite State Machine) *A Mealy FSM m is defined by a quintuple: $m = (S, I, O, \delta, \lambda)$[3], where S, I and O are the finite and non-empty sets of states, input and output events[4] (incoming and outgoing messages from the point of view of the protocol instance), respectively. The δ is the next state function: $\delta : S \times I \rightarrow S$ that determines the next state based on a state/input event pair. The λ is the output function: $\lambda : S \times I \rightarrow O$ that determines the output event based on a state/input event pair.*[5]

Machine m is *deterministic*, if $\forall s, s', s'' \in S, i \in I, o', o'' \in O : \delta(s, i) = s', \lambda(s, i) = o'$ and $\delta(s, i) = s'', \lambda(s, i) = o'' \Rightarrow s' = s'', o' = o''$, that is, for each state/input event pair there is exactly one next state, and always the same output event is produced upon the given input event in the given state.

Machine m is *complete*, if $\forall s \in S, i \in I : \exists s' \in S, o \in O : s' = \delta(s, i), o = \lambda(s, i)$, that is, for each state/input pair the next state is defined and output is produced. Otherwise, the machine is said to be incomplete. Nevertheless, for incomplete machines the *complete test assumption* [LPvB94] can be applied. Completeness can be achieved by introducing either implicit or undefined or forbidden transitions. This thesis considers implicit transitions: for each unspecified $s \in S, i \in I$ state/input event pair let $\delta(s, i) = s, \lambda(s, i) = null$, where $null \in O$ is the unobservable output event. This means that the next state remains the same as the actual state and no output event is produced and can be hence observed, as the transition is implicit.

An FSM can be represented by a state table or state transition graph; Table 2.1 and Figure 2.3 show an example. In the state table a row and a column select a state and input event pair and their intersection cell contains the next state and output event pair. The state transition graph is a directed node and edge labelled graph $G = \{S, E\}$, where states are mapped to vertices labelled with the state name, and the state transition function is mapped to directed edges such that $e = \langle s_i, s_j, l \rangle \in E$, where $s_i, s_j \in S$ and $l \in I \times O$.

The definition of transition and output functions can be extended to input sequences.

Definition 2.2.2 (Input sequences, output sequences [LY96]) *Let $X \subseteq I*$ be the set of strings of input symbols (input string hereafter), and let $Y \subseteq O*$ be the set of strings of output symbols (output string hereafter). Let the input string $x = i_1, \ldots, i_k \in X$ take machine m successively through the states $s_1, \ldots, s_{k+1}$ such that $s_{j+1} = \delta(s_j, i_j)$, $j = 1, \ldots, k$, and produce the output string $\lambda(s_1, x) = o_1, \ldots, o_k \in Y$, where $k \in \mathbb{N}, 1 \leq k < \infty$.*

[3] Some definitions include the initial state $s_0 \in S$ in the tuple.

[4] In compiler theory $O = \{ACCEPT, REJECT\}$, and the model is called a finite automaton or finite acceptor.

[5] The transition and output functions are sometimes used in the form of a single transition function too: $h : S \times I \rightarrow O \times S$.

	a	b
s_1	$s_1/null$	$s_2/2$
s_2	$s_2/2$	$s_3/2$
s_3	$s_2/1$	$s_1/1$

Table 2.1: State transition table

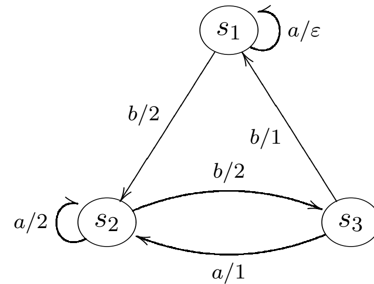


Figure 2.3: State transition graph of an FSM

Machine m is *strongly connected* if $\forall s_i, s_j \in S : \exists x \in X : \delta(s_i, x) = s_j$. Alternatively, it can be said that m is strongly connected if its transition graph is strongly connected.

2.2.2 Extended finite state machines

Expressing all details of a specification with the FSM model gets more and more difficult as the size of the protocol specification grows. Hence, an extended model, the Extended Finite State Machine (EFSM) model has been introduced, which already incorporates some programming language concepts.

The EFSM model extends the FSM model with additional state variables, actions, predicates and input and output event parameters. The state of the modelled machine is characterized by the control state – inherited from the FSM model – and the additional state variables. The transition function of the FSM model is extended with enabling predicates and actions on the variables. An enabling predicate is a conditional expression on the additional state variables that must be satisfied for the transition to execute. A transition action is a value assignment to an additional state variable.

Definition 2.2.3 (Extended Finite State Machine) *Formally an EFSM m is a quintuple: $m = (S, V, I, O, T)$ [LY96], where S, V, I, O and T are the finite sets of states, variables, input symbols, output symbols and transitions, respectively. Each transition $t \in T$ is an six-tuple: $T \subseteq (S \times V) \times I \times TP \times (S \times V)$. TP is the transition part that is $TP \subseteq (P(V_P) \times (A(V_A) \times O))*$, where $P : V_P \rightarrow \{\mathsf{true}, \mathsf{false}\}$ prescribes enabling conditions on $V_P \subseteq V$, $A : V_A \rightarrow V'_A$ defines assignment actions for $V_A \subseteq V$. $I \subseteq (I_{ID} \times V_o)$[6] and $O \subseteq (O_{ID} \times V_o)$[7], where $V_o \subseteq \underset{k<\infty}{\times} V$ [8], the ordered parameter list of finite length.*

The interpretation of the terms determinism and completeness is analogous to the FSM model. The completeness assumption can similarly be adapted for the EFSM model with the introduction of implicit transitions: a transition $t \in T$ is implicit if for $i \in I$ the transition part is empty, which means that neither the control state nor the state of variables changes.

[6]Like a function definition with formal parameters in programming languages.

[7]Like a function call with actual parameters in programming languages.

[8]$\underset{k}{\times}$ represents the Cartesian product relation of k sets.

An EFSM can be represented with a labelled directed graph $G = (\{S \cup P(V)\}, E)$, which has two types of vertices: ones labelled with control states (S) and ones labelled with predicates ($P(V)$). Directed edge $e = \langle n_1, n_2, l\rangle \in E$ is labelled the following way depending on the type of vertices it connects:

- if $n_1, n_2 \in S$, then $l \in \{I \times (A(V) \times O*)\}$,
- if $n_1, \in S$ and $n_2 \in P(V)$, then $l \in I$,
- if $n_1 \in P(V)$, then $l \in \{\{\mathsf{true}, \mathsf{false}\} \times (A(V) \times O*)\}$.

A sample state graph of an EFSM is given in Figure 2.4, which shows a part of the INRES Initiator [EHS97] SDL process.

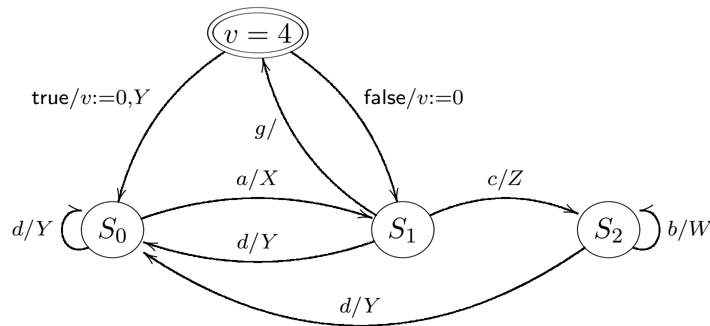


Figure 2.4: State graph of a sample EFSM

The next control state and output functions of the FSM model can be generalized to the EFSM context as follows:

Definition 2.2.4 (Generalized output function) *Let* $\mathsf{out} : M \times X \rightarrow Y$ *be the output function generalized for EFSMs, such that* $\mathsf{out}(m, x) = y$ *if EFSM* m *produces the* $y \in Y \subseteq O*$ *output sequence on the* $x \in X \subseteq I*$ *input sequence.*

Definition 2.2.5 (Generalized next state function) *Let* $\mathsf{after} : S \times X \rightarrow S$ *be the next state function generalized for EFSM, such that* $\mathsf{after}(s_i, x) = P$*, where* $P \subseteq S$ *is the set of possible next control states after executing the* $x \in X \subseteq I*$ *input sequence.*

Nevertheless, FSMs are still sufficient to model the control portion of a protocol specification; omitting state variables reduces the EFSM model to the FSM model. Fundamentally, an EFSM is a compact representation of an FSM. With the additional state variables where each is restricted to a finite domain it is always possible unfold an EFSM model into an FSM model by considering all possible combinations of control states and variable values. However, this may result in the well-known state space explosion problem.

2.2.3 Input/output transition systems

The communication of two or more processes, such as a test execution scenario, where the process of the IUT and the process(es) of the tester(s) exchange data with each other, can be effectively modelled with rendezvous interaction based formalisms: CCS and CSP. The interaction of subsystems can be modelled with the parallel composition of Labelled Transition Systems (LTSs).

In LTSs there is no distinction between input and output events. The interaction takes place if all participating LTSs have a transition defined for the current state for the given action.

IOTSs, which separates the label set into two disjunct sets: input action and output action sets, represent a subset of LTSs. IOTSs are therefore related to the process algebra track of formal design. They can be used in the FSM world to represent possible execution sequences or trees.

Definition 2.2.6 (Input/Output Transition System) *An Input/Output Transition System σ is a quintuple $\sigma = (S, I, O, T, s_0)$ [9], where S is the set of states, I is the set of input actions, O is the set of output actions, the set of transitions is $T \subseteq S \times \{I \cup O\} \times S$, and s_0 is the initial state.*

A transition is denoted the following way: $s_k \xrightarrow{e} s_{k+1}$, where $s_k, s_{k+1} \in S, e \in \{I \cup O\}$. A sequence of transitions called trace is: $s_k \overset{\sigma}{\Rightarrow} s_{k+n}$, where $s_k, s_{k+1} \in S, \sigma = e_1, e_2, \ldots, e_n, \forall i : e_i \in \{I \cup O\}$. A trace is executable if after the given sequence of events the specified system arrives in an existing state, otherwise it is non-executable. An executable or non-executable e event or σ sequence from state s_k are denoted respectively: $s_k \xrightarrow{e}$ and $s_k \overset{e}{\not\rightarrow}$ or $s_k \overset{\sigma}{\Rightarrow}$ and $s_k \overset{\sigma}{\not\Rightarrow}$.

If a tester is modelled as an IOTS, outputs must always be enabled: $\forall s \in S, \forall o \in O : s \xrightarrow{o}$. This is analogous to the completeness assumption for system specification in Section 2.2.1. The IOTS is deterministic, if $\forall s \in S, e \in \{I \cup O\} : s \xrightarrow{e} s', s \xrightarrow{e} s''$, then $s' = s'', s', s'' \in S$.

An IOTS can be modelled with a rooted, edge-labelled tree. [10] For example Figure 2.5 shows a sample IOTS [11]:

2.2.4 IOTSs and FSM traces

Taking the temporal ordering into account, the $x \in X \subseteq I*$ input and $y \in Y \subseteq O*$ output event strings (see the definition 2.2.2) of FSM m determine a σ IOTS trace such that $\sigma = i_1, o_1, \ldots, i_k, o_k$. Let the trace σ composed of the input string x and output string y be denoted with $\sigma_{<x,y>}$. Let $\sigma_{<x>} = x = i_1, \ldots, i_k \in X$ denote the input string extracted from $\sigma_{<x,y>}$ and $\sigma_{<y>} = y = o_1, \ldots, o_k \in Y$ denote the output string extracted from $\sigma_{<x,y>}$.

[9] The initial state s_0 is in some sources omitted from the definition

[10] Note that in labelled trees generally vertices are labelled instead of edges. Nevertheless, the transformation is straightforward.

[11] In Figure 2.5 the vertices are labelled with state names, but this labelling can be omitted

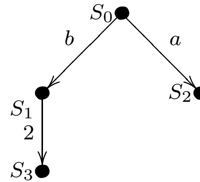


Figure 2.5: Graph representation of a sample IOTS

Let length of trace σ be the number of input and output events it contains: $n = \mathsf{length}(\sigma)$. Let the first l input and output events of σ be denoted by $\mathsf{substring}(\sigma, l)$, i.e. a prefix of l events. Let $\sigma[l]$ denote the lth event in σ.

2.3 Modelling and testing

Model validation, test specification and test execution are closely related activities in the protocol engineering process. In model based test specification it must be made sure that the model reflects the user requirements, that is, it is valid. As it is easier to use abstract, formalized models for any design activity, test cases are abstracted as well. Hence, model based test specification generates abstract test cases from a validated abstract model that expresses the explicit behavior of the system to be implemented.

Model based testing is an approach to simplify the test execution activity. A model is extracted from the implementation manually or automatically [PP05] and some system properties are checked on that model. In both cases (model based test specification and model based testing), traces of a model can be regarded as test cases: the inputs specify stimuli for the implementation, and the outputs specify the expected behavior of the implementation.

It has already been mentioned that models hide some details in the requirements of the system. If the basis of both the implementation and the test specification is a validated formal model, then both the automatic code generation (or model based development a.k.a. Model Driven Architecture (MDA)) tools and the assumptions on the environment must be tested [CHK00] as well. It is also well known that the more expressive the model is, the difficult it is to validate it. If a model is expressive it suits well the purposes of automatic implementation and test specification, but it takes more time to create such a model. Development resources must be moved from the implementation and test specification activities to the design activity.

The models of non-terminating systems like telecommunication protocols define infinite numbers of test cases of possibly infinite length. Hence the test case specification activity is basically a set of criteria for defining a representative subset of traces in order to reduce the total cost of testing.

2.3.1 Conformance testing

While model based test specification defines abstract test cases, to perform the test execution their concretization is necessary. Environmental assumptions and configuration settings are taken into account to find out if a protocol implementation complies with its specification. The purpose of conformance testing [IT95] is to decide whether the IUT is the implementation of the specification (user requirements). Service and functional behavior are tested in order to find logical errors and prerequisites for interoperability.

For a black box testing method, i.e., complete information is available about a specification, only the input/output behavior of the IUT can be observed. The internal operations within the implementation are not visible to the tester, thus the correctness must be determined from the input/output – and temporal – behavior.

2.3.2 Test case, test suite

The conformance testing is done by means of test cases. An abstract test case is a *"specification of the actions required to achieve a specific test purpose"* [IT95]. A test case is an element of the set of all possible traces, a test suite is a subset of this set.

Definition 2.3.1 (Test case) *Formally, a test case of machine m is a σ IOTS such that $\sigma = (S, I_t, O_t, T, s_0)$, where the $I_t = O$ is the input action set that is identical to the output alphabet of the specification machine m, $O_t = \{I \cup \Theta'\}$, where O is the output action set of the specification machine m and Θ' is the union of different timers[12] used in m and in the test system, and each last state of σ is labelled with either* true *or* false*.*

A test suite is built up from a hierarchy of test components. The test event is the smallest, indivisible unit of a test suite. Typically, it corresponds to sending or receiving a message and the operations for manipulating timers. The test step is a grouping of test events, similar to a subroutine or procedure in programming languages. The test case is the fundamental building block in a test suite. A test case tests a particular feature or function in the IUT. A test case has an identified test purpose and it assigns a verdict that depends on the outcome of the test case. A test group is simply a grouping of test cases. A test suite can range from a large number of test groups and test cases to a single test event contained in a test case.

2.3.3 Implementation relations, conformance relation

Several different *implementation relation*s have been defined between an implementation machine and a specification machine: different variations of bisimulations (strong, weak etc.), preorders (trace, testing, refusal), preorders with inputs and outputs, and different levels of conformance (e.g. conf, ioco, mioco) [Tre00].

The hypothesis of specification based testing is that implementation relations can be decided based on formal models. The fundamental assumption is that for each IUT there

[12]Timers are discussed in Section 2.3.5

exists a formal model such that the execution of a test case against the IUT provides the same verdict as model of the test execution, the observation made on the formal model. This means that in the formal framework the real test execution is modelled by the formal observation function.

In the FSM context test execution can be modelled with Communicating Finite State Machines, where one machine models the system under test and the other the tests [LY96]. From all possible combinations of states of the communicating machines, a composite machine, another FSM, can be constructed. On the other hand, it is possible to adapt the execution model of process algebra for the FSM world. It has already been shown that test derivation problems can be transferred from the FSM/EFSM context to process algebra [PvBD93, TPvB95].

2.3.4 Conformance relation formally

The formal framework of [Tre00] consists of several functions that can be interpreted in the FSM/EFSM context as follows. These functions take two arguments, a trace and a model machine. In the original article both the model and the trace are given as LTSs. In this thesis specifications and implementation models are modelled in FSM or EFSM models, and test cases are represented as IOTSs.

The *execution* function defines the execution of a trace against a machine:

Definition 2.3.2 (Execution) *The function* exec *is* $\Sigma \times M \rightarrow \{\mathsf{pass}, \mathsf{fail}\}$, *where* Σ *is a set of IOTSs and* M *is the set of FSMs with the same I/O alphabet. The execution of the sequence* $\sigma_{<x,y>} \in \Sigma$ *on FSM* $m \in M$ *is defined in [vBP94a] the following way. The input sequence is executable* [13], *that is, it returns* pass, *if* $\forall s_i \in S : \exists s_{i1}, s_{i2}, \ldots, s_{ik} \in S*$ [14] *and* $\exists y = o_1, \ldots, o_k \in Y$ *such that there is a sequence of transitions* $s_i \overset{i_1/o_1}{\rightarrow} s_{i1} \overset{i_2/o_2}{\rightarrow} s_{i2} \rightarrow \ldots \rightarrow s_{ik-1} \overset{i_{ik-1}/o_{ik-1}}{\rightarrow} s_{ik}$ *in machine* m.

The observation function returns the verdict of the execution.

Definition 2.3.3 (Observation) *Let* $\mathsf{obs} : \Sigma \times M \rightarrow \{\mathsf{fail}, \mathsf{pass}\}$ *be the observation function, where* M *is the set of systems with the same* I *and* O *sets,* Σ *is a set of test cases represented as IOTS. Let* $\forall \sigma \in \Sigma, m \in M : obs(\sigma, m) = \mathsf{fail}$, *if* σ *is observed to fail against the system* m. *If* m *is observed to pass* σ, *let this function return* pass.

The conformance implementation relation is determined an observer by means of a set of sequences. The next definition is the adaptation of conformance relation defined in [Tre00] to FSMs.

Definition 2.3.4 (Conformance relation) *Given two machines* A *and* B *with the same input event (and timer)* $I \cup \Theta$ *and output event* O *sets. Machine* A *is strongly connected.* B *conforms to* A *if and only if* $\forall \sigma_{<x,y>} \in \Sigma$, *where* $\sigma = (S, I, \{O \cup \Theta'\}, T, s_0)$ *and* $s_{A0} \overset{\sigma}{\Rightarrow}$ *in* A *and* $\forall i \in I, o \in O$ *the next two statements hold:*

[13] In the paper [vBP94a] it is called acceptable input sequence.

[14] Note that the completeness assumption makes the condition about the state sequence unnecessary.

- *if for machine B* $\lambda_B(\delta_B(s_{B0}, \sigma_{<x>}), i) = o$, *then* $\lambda_A(\delta_A(s_{A0}, \sigma_{<x>}), i) = o$ [15] *is possible for machine A, and*

- *if* $\sigma_{<x>}$ *is non-executable for machine B, it may be non-executable for machine A.*

The conformance implementation relation, which is called ioco in [Tre00] in the i/o case, permits the implementation to add optional features, which are not present in the specification, but the definition also states that all mandatory requirements must be expressed.

Based on the definitions above the abstract test suite Σ is said to be complete if it characterizes all conforming implementation models. If each implementation model machine $iut \in M$ conforms to the specification machine, $\forall \sigma \in \Sigma : obs(\sigma, iut) = \mathsf{pass}$, the implementation iut should pass Σ.

The refusal preorder implementation relation [Tre00] is stronger than the conformance relation. It requires that the two requirements of the conformance relation must hold not only for all sequences of the specification machine, but for all possible sequences that can be composed from all input, output and timeout events:

Definition 2.3.5 (Refusal preorder) *Given two machines A and B with the same input event (and timer)* $I \cup \Theta$ *and output event O sets. Machine A is strongly connected. B conforms to A if and only if* $\forall \sigma_{<x,y>} \in (I \cup O \cup \Theta)*$ *and* $\forall i \in I, o \in O$ *the next two statements hold:*

- *if for machine B* $\lambda_B(\delta_B(s_{B0}, \sigma_{<x>}), i) = o$, *then* $\lambda_A(\delta_A(s_{A0}, \sigma_{<x>}), i) = o$ [16] *must be possible for machine A, and*

- *if* $\sigma_{<x>}$ *is non-executable for machine B, it must be non-executable for machine A.*

2.3.5 Timed models

Timers in test traces have several purposes as introduced in [HKN01]. Timers on the one hand assure that test cases terminate when deadlocked because of unexpected behavior of the tested system. On the other hand, timers check timing constraints, and thirdly timers make the delaying of messages possible.

Delaying timers in testers has three purposes:

- they allow machine M to execute its current transition, that is, they slow down a too fast tester

- they check that machine M does not produce an output event for a given amount of time

[15] In case of EFSMs first both A and B are reset to the initial control state, then with the sequence $\sigma_{<x>}$ they are brought to states P_A and P_B, where it is checked if $\mathsf{out}(B, i) = o$ holds then $\mathsf{out}(A, i) = o$ is possible.

[16] In case of EFSMs first both A and B are reset to the initial control state, then with the sequence $\sigma_{<x>}$ they are brought to states P_A and P_B, where it is checked if $\mathsf{out}(B, i) = o$ holds then $\mathsf{out}(A, i) = o$ must hold

- they check if the output event produced by machine M is delayed too long

The test case ending and delaying timers appear only on the test system side. Timers for timing constraints, which are somehow related to the third type of delaying timers, are defined in the formal specification (cf. SDL timers) as well.

Timeout events for timing constraints in the specification machine M are the means for checking if an input event from its communicating peer is delayed for too long. Such timeouts can be triggered knowingly by the tester side during testing to check if the absence of inputs is handled adequately. This is achieved by not sending an input event to M for the given amount of time, that is, the tester side waits before producing the successive input event. Let such set of timers be denoted with $\Theta = \{\vartheta_1, \ldots, \vartheta_n\}$ [17] that is considered to be a special kind of input symbol set. It is in line with the SDL semantics where timers and timeout events in a process are treated as input signals sent by the process to itself. Hence, the input alphabet of the specification machine in the timed model is $I \cup \Theta$.

2.4 Automatic test derivation for extended finite state machines

2.4.1 Test generation for finite state machines

For FSMs complete test suites can be derived with the transition checking approach [Hen64]. This method assumes deterministic strongly connected machines with reliable reset capability ($\exists r \in I : \forall s_i \in S : \delta(s_i, r) = s_0$). For the undefined transitions, conventions must be introduced like the implicit transitions or undefined transition or forbidden transitions. Each transition that means each state pair of the machine yields a test case: first the machine is reset to its initial state, then with a sequence of observable input and output events it is brought to the starting state of that transition where the triggering input is applied and the corresponding output is observed. Finally it is verified with either status messages, separating sequences, distinguishing sequences, Unique Input Output (UIO) sequences, characterizing sequences or identifying sequences if the machine is in the correct end state [LY96, vBP94b].

It is always possible to unfold an EFSM into such an FSM, however the size of the state set of the FSM doubles[18] as the domain of any variable of the EFSM increases by a single bit. State identification problems are usually proportional to the third power of the number of states that makes these methods simply impractical: the generation complexity and the length and number of test cases grow exponentially as more and more details are added to the model. Nevertheless, the generated test set is complete and provides hundred percent assurance that the IUT conforms to the specification machine if that passes the test set.

[17] In the "formal framework for conformance testing" [Tre00] ϑ was denoted with θ, and θ was denoted with δ.

[18] Note that the number of states does not necessarily double due to the possible FSM minimization. Nevertheless the trend of growth is exponential.

2.4.2 State space exploration, search methods

Since unfolding EFSMs into FSMs is in most cases not a way to tackle the problem of automatic test generation, heuristic state space exploration method must be used. Search methods must explore a subset of the whole state space such that all main services of the system model are tested and the probability of finding an error must be better than the ratio of explored stated to all states [Hol91].

Such search methods are investigated in [Hol91]. Different strategies have been proposed to access only a part of the whole state space:

- depth bound: where the length of execution sequences is limited, so only a subset of behaviors can be analyzed
- scatter search: the selective search algorithm favors outputs over inputs such that deadlocks can more easily (with higher probability) be found
- guided search: the next transition is selected based on a dynamically evaluated cost function
- probabilistic search: transitions are explored in decreasing order of their probability of occurrence
- partial order: not all interleavings of concurrent events are investigated
- random selection: the next transition is selected at random

2.4.3 Simple search methods

This section defines simple search methods created from the heuristics mentioned above.

Random walk

"In random selection of successor states no effort is made to predict where likely errors are to be found." "This is not only the simplest technique to implement, but is also likely to produce the highest quality search." [Hol91] Besides, random selection based methods meet the requirements mentioned in the beginning of Section 2.4.3. Note that it is possible that we get trapped in a small portion of the system; for such cases guided random walk was suggested [LY96].

The next simple FSM based automatic test derivation algorithm (Algorithm 2.1) traverses successive transitions of a machine at random and has a depth bound as stop condition. Its input is an FSM, and it outputs a test case. It offers an arbitrary input for the machine and then it either accepts an output or output string from it and continues with offering the next input or selects and invalid output at random and returns. This procedure is applied until a certain stop condition (e.g. a certain number of transitions is reached) is fulfilled, and from the inputs and outputs, a test case is constructed.

Algorithm 2.1: Simple random trace derivation

```
    input:  FSM m = (S, I, O, T, s_0), stop condition
    output: a test case σ
1   data(s, s' ∈ S, i ∈ I);
2   s := s_0;
3   repeat
4       i := selectI(m);
5       s' := δ(s, i);
        /* Add this i to the end of σ */
6       σ[length(σ) + 1] := i;
7       switch random do
            /* Add either the o specified by m to the end of σ */
8           case
9               σ[length(σ) + 1] := λ(s, i));
            /* or add a random o specified by m to the end of σ */
10          case
11              σ[length(σ) + 1] := selectO(m);
12              return σ
13          endsw
14      endsw
15      s := s';
16  until stop = false ;
```

Example Consider the machine of Figure 2.3, where $S = \{s_1, s_2, s_3\}$, $I = \{a, b\}$, and $O = \{1, 2, \varepsilon\}$, where ε is the empty output string. Let its initial state be s_1. Let the procedure stop after two transitions. In the first cycle, let the selected input be $i = b$, $\sigma = b$. Let the rule in line 9 be chosen: $y = \lambda(s_1, b) = 2$, $\sigma = b2$. The new state is s_2. In the second cycle, let the selected input be $i = b$, $\sigma = b2b$. Let the rule in line 11 be chosen: $y = \lambda(s_1, b) \rightarrow 1$, an invalid output is added to the end of the trace and the process is terminated: $\sigma = b2b1$. This test case t will only pass if there is an invalid transition from s_2 to s_3 in the implementation.

Random sequence

Algorithm 2.2 constructs a test case by generating a random string of test events (input and output events). In this case the whole test case is random, the knowledge on the internal structure of the FSM is not used. According to the classification in Section 2.4.3, it is a random search combined with scatter search as it results in a deadlock on an undefined input event.

The algorithm is composed of two steps: first a random sequence σ is obtained from $\{I \cup O\}$, and then the number of correct events in σ checked against m. Construct the input string x and the output string y from $\sigma = \ldots, \eta_j, \ldots$ such that $x = \ldots, \eta_j, \ldots$, where η_j is an input test event and $y = \ldots, \eta_j, \ldots$, where η_j is an output test event, where $0 < j <= \mathsf{length}(\sigma)$. The sequences x and y store the input and output events, respectively.

To remove the "unusable" part of the random sequence, two functions are introduced. The observation function $\mathsf{obs}(\sigma, m)$ determines whether the machine m passes the test case σ. This means, if m can produce the output string y of σ prescribed by the input string x of σ, m passes σ, otherwise fails. The $\mathsf{fails_after}(\sigma, m)$ function gives the number of test events, after which the test case σ is observed to fail against the machine m.

Algorithm 2.2: Random event sequence

```
   input: FSM m
   output: test case σ
 1 data( η: is a test event (input or output) of m
 2 stop: is an extra symbol indicating that the algorithm should return);
 3 repeat
      /* Select an event e from the union of the sets of input and output
         events including a stop symbol.  */
 4    η := select(m);
 5    if stop then return;
 6    else
 7       σ[length(σ) + 1] := η
 8    endif
 9 until true ;
```

Example Consider again the FSM in Figure 2.3. The set of test events is $I \cup O = \{a, b, 1, 2\}$. Let the initial state be s_1. Let the sequence of events generated be: $\sigma = ab2$. This random event sequence passed the specification: it traverses the loop around s_1 initiated by input a, where no output is produced, then it traverses the transition from s_1 to s_2 initiated by b outputting 2. The function $\mathsf{fails_after}(\sigma, m)$ will return 3.

All sequences of a given length

The next algorithm (Algorithm 2.3) creates all test event strings up to a certain length Λ, there is no random selection of successor state, only a depth bound. Its input is an FSM, and it outputs a set of test cases of the given length. The number of test cases increases exponentially as Λ increases.

Algorithm 2.3: All sequences of a certain length

input: FSM m, a maximum length Λ
output: a test suite Σ

```
1  data( η: is an input or output of M
2  σ and σ': are test cases);
   /* Initialization */
3  Σ := ∅;
4  σ := ε, length(σ) = 0;
5  Σ := Σ ∪ {σ};
6  foreach σ ∈ Σ, length(σ) < Λ do
7      foreach η ∈ {I ∪ O} do
8          σ' := σ;
9          σ'[length(σ) + 1] := η;
10         Σ := Σ ∪ {σ'};
11     endfch
12     Σ := Σ \ {σ};
13 endfch
```

2.4.4 Search methods employed in test generation tools

Autolink [SKGH97] supports besides the brute force technique random walk, tree walk and guided walk based state space exploration. The latter is a semi-automatic method that lets the user select the next transition. The brute force method is like Algorithm 2.3 without the L length limit. The random walk algorithm implemented in that tool is like Algorithm 2.1 without lines 10-11.

The algorithm proposed in [Tre00] is implemented in TorX. It [TB03] operates in a very similar way as Algorithm 2.1, but instead of a random choices all input and output events are examined that results in an exhaustive test set. That algorithm constructs a tree that can be decomposed to a set of traces.

2.4.5 Functions for test generation and test set optimization

This section defines abstract functions that are used in the rest of the thesis in test generation and test selection algorithms.

Definition 2.4.1 generates a σ trace for the EFSM m by means of a state space exploration method.

Definition 2.4.1 (Test derivation) *Let* derive : $M \rightarrow \Sigma$, *where* m *is an EFSM* $m = (S, V, I, O, T)$ *and* σ *is an IOTS* $\sigma = (S', I, O, T')$ *with common* I *and* O *sets. The trace* σ *is generated with a search method of Section 2.4.2.*

Definition 2.4.1 updates a σ trace for the EFSM m by modifying its postfix using a state space exploration method.

Definition 2.4.2 (Incremental test derivation) *Let* derive_inc : $M \times \Sigma \rightarrow \Sigma$, *where* m *is an EFSM* $m = (S, V, I, O, T)$ *and* $\sigma = \alpha\beta, \sigma' = \alpha\beta'$ *are IOTSs with common* I *and* O *sets. Traces* σ *and* σ' *have the same* α *prefix. This function generates a new* β' *postfix for the input* σ *trace with a search method of Section 2.4.2.*

The abstract function in Definition 2.4.3 solves a possibly NP-hard selection problem. The result is stored in a boolean vector that marks the selected rows of the input matrix.

Definition 2.4.3 (Selection) *Let* select : $\mathbf{M} \rightarrow \mathbf{s}$, *where* $\forall i : s[i] \in \{0, 1\}$ *give the solution of the selection problem represented with the matrix* $\mathbf{M}$. *It results in the vector* $\mathbf{s}$, *where* $s_i = 1$ *if the element corresponding to the* i^{th} *row of* $\mathbf{M}$ *is selected by the optimization algorithm. Otherwise, let* $\mathbf{s}$ *contain* 0 *in row* i. *If the size of* $\mathbf{M}$ *is* $m \times n$, *then the size of* $\mathbf{s}$ *is* n.

The reduction function of Definition 2.4.4 keeps the rows of vector $\mathbf{L}$ that are marked in vector $\mathbf{s}$, and removes the rows which are not marked.

Definition 2.4.4 (Reduction) *Let* reduce : $\mathbf{L} \times \mathbf{s} \rightarrow \mathbf{L}$ *remove the unselected rows from* $\mathbf{L}$ *according to the selection vector* $\mathbf{s}$.

If $\mathbf{L}$ is a vector of vectors of size $m \times n$ and $\mathbf{s}$ is of size n, then $\mathbf{L}_r :=$ reduce$(\mathbf{L}, \mathbf{s})$ is a vector of vectors of size $m \times k$, where $k = \sum_{g=0}^{n} \mathbf{s_g}$.

The append function (Definition 2.4.5) copies the rows of the second matrix below the rows of the first, where both matrices have the same number of columns.

Definition 2.4.5 (Append) *Let* append : $\mathbf{M} \times \mathbf{M} \rightarrow \mathbf{M}$. *If* $\mathbf{M}_1$ *is of* $m_1 \times n$ *size and* $\mathbf{M}_2$ *is of* $m_2 \times n$ *size, then* $\mathbf{M}' = append(\mathbf{M}_1, \mathbf{M}_2)$ *is of* $m_1 + m_2 \times n$ *size such that the rows of* $\mathbf{M}_1$ *are from row* 1 *to row* m_1 *and the rows of* $\mathbf{M}_2$ *are from row* $m_1 + 1$ *to row* $m_1 + m_2$ *in* $\mathbf{M}'$.

Chapter 3

Mutation based automatic test generation for SDL specifications

This chapter proposes a method for automatic test specification. The input of the method is a specification given with the EFSM/SDL semantics. By means of the formal fault based technique called mutation analysis, abstract test cases are output in the MSC [IT00a] syntax.

The test generation method is built on four cornerstones:

- the test case derivation process must take place without any human intervention, that is, fully automatically
- the number of traces (hence test cases as well) included in the generated set must be limited
- though random subset of all traces, that is, walks have been shown to be "quite satisfactory" [TB03], the strategy should have a better fault detection ability or require less execution time than the same number of random traces
- test cases are generated from the black box perspective; the state space search algorithm does not use information related to the current state of the specification model

Section 3.1 gives an overview of related work and mutation analysis. A fault model for SDL and algorithms for automatic test generation and selection are given in Section 3.2. The method has been implemented in a software tool that is presented in Section 3.2.2. In Section 3.3 the method is investigated with simulation based results.

3.1 Background and related work

3.1.1 Related work

AutoLink [SKGH97] is a software component of the SDL specification and Tree and Tabular Combined Notation version 2 (TTCN-2) or Testing and Test Control Notation version 3

(TTCN-3) test case representation. This is supported by the Tau suite from Telelogic [Tau][1] developed in a joint project by the Institute for Telematics of the University of Lübeck and Telelogic. It provides automatic and semi-automatic test generation functions for the SDT Validator and Simulator. "In Autolink a test case is derived from a path" [SKGH97], where the path is interpreted as the trace in the current context (see Section 2.2.3 and Section 2.2.4). In that tool such paths can be derived several different ways, from brute force methods to more selective ones like manual navigation in the state space – see Section 2.4.4. The system model in Tau is considered to be given in SDL, and the paths are represented as MSCs that can be later transformed into any test notation (TTCN-2, TTCN-3). This is the starting point of the discussion of this chapter.

TorX [TB03] is an automatic test generation, test execution and test analysis tool built on the ioco theory of Tretmans and Brinksma [Tre00] that is based on process algebra semantics. This chapter uses that theory and applies that to SDL semantics. TGV [JJ02] is built on the ioco theory as well. A detailed summary and evaluation of available tools can be found in [BFS05].

A common weakness of these tools and the theory behind them is the lack of support for data portion testing, that is selecting the appropriate input parameter vectors. Test data selection for SDLs is addressed in [LY93, BCL99]. The method proposed unfolds the EFSM to an FSM, and assigns a color to a node or an edge depending on the variable V configuration. The test set generated should cover all colors in the graph of the FSM. Finding the minimum size set is shown to be NP-hard [LY93].

The mutation based technique has initially been used for code based software verification and validation. Research has shown that it can be applied to various formalisms as well to validate formal models. Probert and Guo studied mutation analysis in the context of Estelle in [PG91]. Wang and Liu proposed a white box method for creating test cases that detect given faults in an EFSM specification specified by state transition graphs, and demonstrated their algorithms on the Alternating Bit Protocol [WL93]. Major research has been done by Fabbri et al. in this field. For different formalisms, they defined mutation operator sets and presented case studies. The mutation analysis of Finite State Machines is introduced in [FMDM94]. They applied manually this technique to a transport protocol specification, using a method based on characterizing sets and transition tours [LY96]. They proposed the application of mutation testing for validating Estelle specifications [SMFLDS00]. Later in their independent research they extended their solution to SDL as well comparing the differences to our solution in [SMW04]. The starting point of their solution is the formal syntax of these modelling languages. They approach the problem from aspects well known from software engineering. Our approach however is an analytical one that uses the formal semantics of SDL and the theory of model based testing. They presented mutation analysis of Petri-Nets [FMM$^+$95] and Statecharts [FMSM99] as well. Ammann and Black applied mutation analysis to model checking [ABM98] [BOY00][AB99]. Hong and Lu proposed a framework based on the method proposed in this paper to test web services [HL05]. Most recently Wong and Choi proposed a similar approach to validate SDL specifications, the

[1]Telelogic was bought by IBM in 2008

fault model they used was not formalized [WRQC08]. Liang et al. extended our solution to test component-based systems [JHS+08].

Test generation methods for EFSM that do not use a fault model are proposed in several research papers like [BH89, BDA, DÜU04, Kim08]. Duale and Uyar unfold the EFSM into FSM, and perform test generation on that model [DÜU04]. Bromstrup and Hogrefe [BH89] and Kim [Kim08] introduce an alternative representation forms instead of EFSMs: Asynchronous Communication Tree (ACT) and event-based EFSM (EEFSM), and derive test cases from that model. Bourhfir et al. target the checking of loops in their paper [BDA].

3.1.2 Mutation analysis

The basic idea behind mutation analysis [DMLS78] [FWH97] is that by applying small syntactical changes, or mutations, at atomic level to the system exactly one at a time, faults are intentionally produced [Kuh92]. The rationale is that if a test set can distinguish a specification from its slight variations, the test set is exercising the specification adequately.

A mutation analysis system consists of three components: the original system, mutant systems and an oracle. A mutant system is a syntactically modified variant of the original; mutation operators, which represent a small syntactic change, create mutant systems from the original system. The oracle is a person or – in our case – a program to distinguish the original from the mutant by their interaction with the environment.

The architecture of traditional program-based mutation techniques [DMLS78] is in Figure 3.1. The idea is to find the appropriate values to be used in test cases. Given program code, faults are injected into that to create mutant programs, and the test cases are executed with different data configurations. When applied to model of a program as in [AB99] to perform model checking, the test cases are evaluated with the symbolic execution of the model.

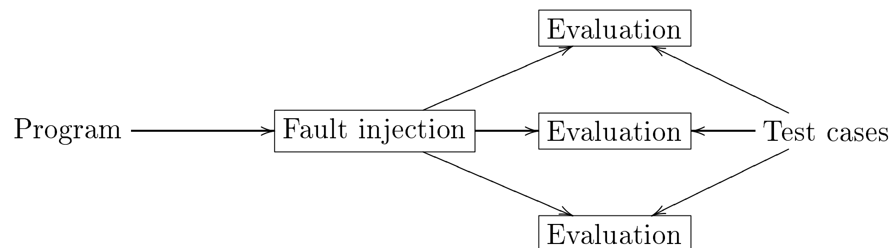


Figure 3.1: Mutation technique in program/model based testing

Mutation operators [Kuh99] [SMFLDS00] play a central role in a mutation analysis system as they define a fault model. Each operator is a type of atomic syntactic change that can be used to inject a certain fault into a correct system. Using these operators is practical for two reasons. On the one hand, they enable the formal description of fault types, on the other hand, operators make automated mutant generation possible. By applying the operators systematically to the specification a set of mutants can be generated.

3.2 Test generation for SDL models

Mutation analysis is based on the knowledge of the internal logic of a system (it is a white box method). In this thesis, mutation analysis is used for deriving adequate black box test cases based on SDL specifications.

Traditional program-based mutation analysis assumes the competent programmer hypothesis [ABM98]. In the current work, a similar *"competent specifier"* hypothesis is assumed stating, that the specifier of an SDL model is likely to construct a specification close to the requirements, and hence the test cases distinguishing syntactic variations of that specification are useful.

In recent work, only first-order faults are considered, i.e. exactly one mutation is applied at a time, because it is enough to detect one fault to assign a fail verdict to the implementation. In case of mutation testing of software this hypothesis could not be used. However, it is appropriate to use it in model based development scenario (see Figure 2.1) on a formal model available prior to the software implementation. On the one hand we have mathematic means to check the correctness of that specification. So, if we inject two or more faults to a specification we would either be able to find each fault separately or the specification would be indistinguishable from the correct one. In the latter case we simply do not generate a new test case. On the other hand while software changes frequently as new bugs appear and get corrected specifications change only after a redesign that requires the regeneration of test sets as well. Therefore as no changes are made, no new faults are uncovered.

The test generation framework is based on the conformance relation (ioco) of Definition 2.3.4. During test execution it is checked whether the implementation can produce the behavior of specification. If there is one fault in the implementation, that must fail the test, and we can deduce that those implementation is a correct implementation of a mutant specification, which has an extra fault injected.

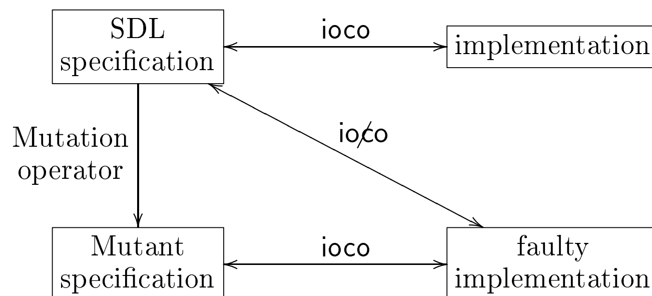


Figure 3.2: Mutation based test generation

Figure 3.2 shows this scenario. The ioco relation holds both between the valid SDL specification and the correct implementation and between the mutant SDL specification and the faulty implementation. Each mutation operator applies a modification to the SDL specification that adds and/or removes – possibly a set of – transitions and/or relabels

the transition part of a transition. From the design aspect this means that a requirement is either added and/or removed or modified. From the point of view of implementation, the faulty product that is in ioco relation with that mutant specification implements some optional features, but may not implement mandatory requirements from the original set of requirements. Because of these missing features the ioco relation between the original specification and the faulty system may not hold.

The relation between the original and mutant specifications is a refusal preorder implementation relation (Definition 2.3.5). According to the conformance relation

- mandatory requirements must be present in the mutant specification as well, and
- the optional requirements are considered only in the implementation and not in the specification, that is a mutant specification must not specify additional optional requirements.

These are two "must" rules as in case of refusal preorder. To prove that there is a refusal preorder between two systems, one must check all possible sequences. However, to prove that the refusal preorder relation does not hold, a single trace of the original specification must be found that is not present in the mutant specification. This provides the basis for automatic test generation.

Therefore, the aim is to generate such a test case for each mutant specification that can determine the absence of the refusal preorder relation, and that will inherently be able to check the conformance of an implementation. Test cases distinguish mutants from the original system only if they produce different output, hence some of the mutants generated using mutation operators may be semantically equivalent to the original system; in this case there is no way to detect the inequivalence. All equivalents should thus be ignored, but each non-equivalent should be considered during test generation.

3.2.1 Algorithms for test generation

We propose two algorithms for automatic test generation with different purposes, but they build on the same concept and use the same fault model. The technique comes from mutation analysis, the SDL specification is stimulated using inputs or timeout signals from the environment, and the outputs to the environment are checked for inconsistency. Both procedures require the SDL specification of a system. There are practical and widely used tools to assist the specification process, e.g. Telelogic Tau [Tau]. After the specification is completed, processing in both cases can be all automated. We represent test cases in MSC [IT00a] [GHN93].

Algorithm 3.1 generates abstract conformance test cases such that it ensures that mutant specifications are not in refusal preorder implementation relation with the original specification. Its architecture is shown in Figure 3.3. The input of this first algorithm is an SDL specification. As a result, we get a set of MSC test cases. Mutation operators of Section 3.3.1 are used for systematically generating mutant specifications. A state space search method from Section 2.4.3 is used to incrementally develop the distinguishing test

case and co-simulate the original and mutant specification. The generation is terminated if a difference is observed in their behavior.

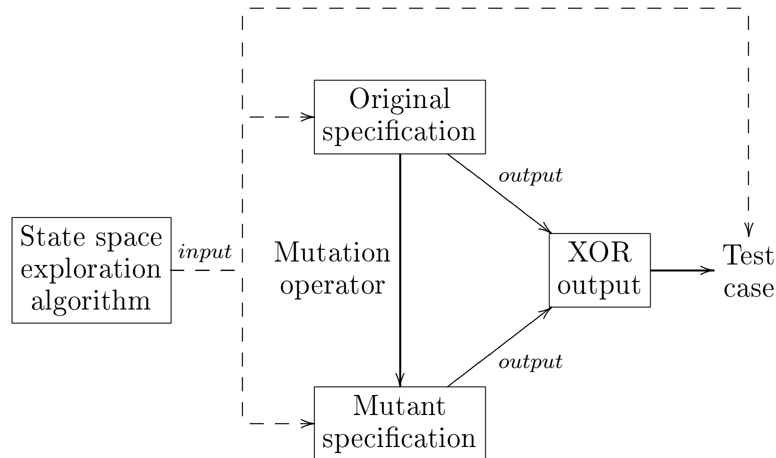


Figure 3.3: Mutation based test derivation

This algorithm consists of the following steps. For each statement of the SDL specification m (i.e. a start state, a triggering input, a transition part with outputs, predicates and actions, and a next state) all appropriate operators are applied automatically and m' is constructed. A search method is used to construct the σ transition system, that is an abstract conformance test case for m. The search is stopped in line 8 if a difference in the co-simulation of the original and mutant systems is found.

This algorithm must have a stop condition to break the infinite cycle of the search method. When this condition is met, the mutant is declared to be quasi-equivalent, that is, it could not have been decided if it is distinguishable from the original system. This condition can be for instance the reaching of a certain length of σ, exceeding a time limit, etc.

Algorithm 3.2 is in line with the original intention of mutation analysis: it is used for test selection. Its architecture is shown in Figure 3.4. It requires the SDL specification of the system to be tested, and assumes that a finite size test set exists (for example in a form of a set of MSC test cases). These test sets are usually created by the means of state space exploration algorithms exploring the specification of the system, but could also be developed manually. This set is subject to optimization, meaning that a subset of test cases is selected with equivalent coverage (mutant detection ratio). The algorithm uses one parameter that stores data for the optimization: let the matrix of criteria $\mathbf{C}$ be a two-dimensional matrix with boolean values.

The operation of the algorithm is the following. After the initialization, for all possible transition statements mutant systems are generated with all possible uses mutation operators. These mutant systems yield a vector of selection criteria for each abstract test case σ_j.

Algorithm 3.1: Derivation of test cases from a specification with mutation analysis

```
    input: m = (S,V,I,O,T), an SDL specification; stop condition
    output: Test suite Σ = {...,σ,...}, where σ is an IOTS.
 1  for t ∈ T do
 2      for ω ∈ Ω do
 3          Let m' = (S,V,I,O,T'), where t' = ω(t), t' ∈ T', t ∈ T;
 4          σ = null;
 5          repeat
 6              if stop then return;
                /* Let an arbitrary test generation method explore the state
                   space of M and construct σ incrementally */
 7              σ := derive(m);
 8          until out(m,σ) ≠ out(m',σ) ;
 9          Σ := Σ ∪ {σ};
10      endfor
11  endfor
12  return Σ
```

If difference is observed in the behavior of a mutant specification compared to the original specification, the corresponding element in **C** is set to 1. After this experiment, this **C** matrix is used in an optimization procedure to reduce the input test set Σ.

Some simplification of **C** can be done as shown in Figure 3.5

- If $\forall i : \mathbf{C}_j[i] = 0$, then σ_j did not find any of the mutants, and can thus be omitted together with the jth column.
- If $\forall j : \mathbf{C}_i[j] = 0$, it represents either that the mutant M_i is an equivalent, or no $\sigma \in \Sigma$ could find the difference (kill the mutant).
- If there are rows $\mathbf{C}_m$ and $\mathbf{C}_n$ in **C**, where $\forall j : \mathbf{C}_m[j] \leq \mathbf{C}_n[j]$, then the row $\mathbf{C}_m[j]$ is unnecessary.

There is an upper bound on the size of the test set derived by Algorithm 3.1 or reduced by Algorithm 3.2. The number of mutations created depends on the number of statements in the SDL specification and the number of mutations applicable to each statement. Therefore, the upper bound on the resulting test set is determined by the size of the SDL specification and the Ω fault model.

Both algorithms have their advantages and drawbacks. The first algorithm requires less computation and time than the second, but it does not provide enough data for any kind of optimization. To get good coverage, the initial test set of the second algorithm has to be large enough – the sufficient number of test cases varies from protocol to protocol. A major drawback of the second method is that it is quite computation-intensive. Parallel or grid computing could decrease the execution time; a parallel execution unit can be defined in Algorithm 3.1 for the loop at line 2 and in Algorithm 3.2 for the loop at line 7.

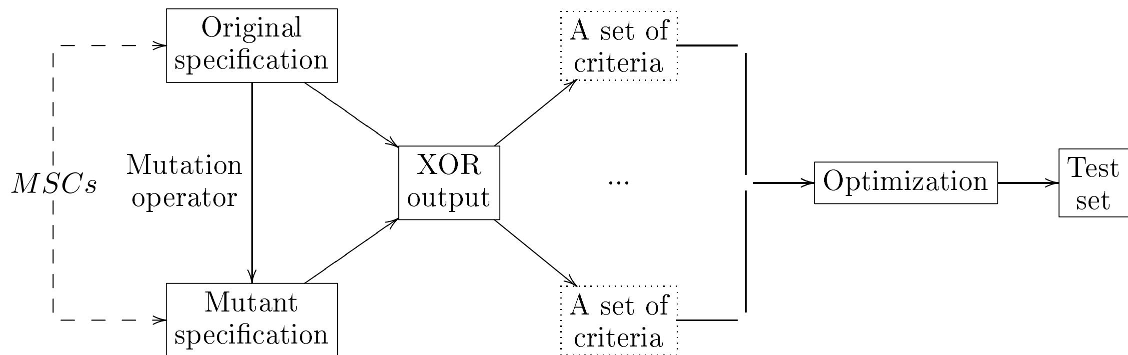


Figure 3.4: Mutation Analysis of SDL specifications Based on an Existing Test Set

Test cases	Faults						
	1	0	1	1	0	0	
	1	0	0	1	0	1	
	1	1	0	0	0	1	
	0	0	0	0	0	0	← Equivalent
	0	0	1	0	0	0	← Can be omitted
	0	0	1	1	0	1	

Weak test case (the 0 in column 5 of the last row)
Important test case (column 2)

Figure 3.5: A sample matrix of criteria

The test set of the second algorithm may contain a huge number of inappropriate cases. Matrices of criteria characterize the test sets. Experience shows that there are permutation of them that are rather block-structured. Certain faults are detected by many test cases, which appears in matrix **C** as a submatrix containing only 1 values. “Similar” test cases detect “similar” faults, that is, most of the true values in the matrix are in blocks. From the point of view of test selection the important criteria can be found in parts that are only rarely filled with true values. It indicates that it is not necessary to fill in the whole matrix. At the cost of a larger resulting test set, intelligent algorithms can reduce the number of executions and therefore the time required.

The efficiency of the proposed approach depends on the size of the input/output alphabet as well. This is a common weakness of all automatic test generation methods. This formal framework builds on EFSM model given in Definition 2.2.3. According to that definition the parameterized SDL signals with different parameter lists are mapped to different elements of the I input or O output sets. So the derive search algorithm must tackle the problem of test data selection.

Algorithm 3.2: Selecting test cases from an existing set

input: $m = (S, V, I, O, T)$, an SDL specification; Σ test set; *stop* condition
output: Test suite $\Sigma = \{\ldots, \sigma, \ldots\}$, where σ is an IOTS.

```
1  data(C boolean matrix);
   /* Initialization */
2  i := 1;
3  C := 0;
4  foreach t_i ∈ T do
5      foreach ω ∈ Ω do
6          Let m' = (S, V, I, O, T'), where t' = ω(t), t' ∈ T', t ∈ T;
7          foreach σ_j ∈ Σ do
8              if out(m, σ_j) ≠ out(m', σ_j) then C_ij = 1;
9          endfch
10     endfch
11 endfch
12 return Σ' := reduce(Σ, select(C));
```

3.2.2 The test generation framework and tool

To implement the automatic test selection procedure, we implemented a Test Selector tool in Java.(The main dialog can be seen in Figure 3.6.) The tool consists of several components indicated by rectangles in Figure 3.7.

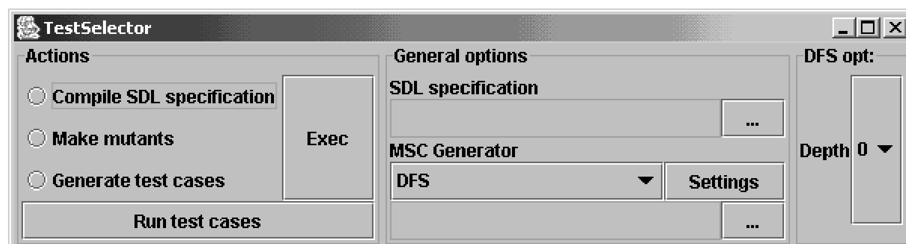


Figure 3.6: The main dialog

The mutant generator component takes the SDL Phrase Representation (SDL/PR) description original specification as an input. It implements the mutation operators defined in Section 3.3.1 and shown in Table 3.1, and creates mutant specifications also in SDL/PR format.

The compiler generates Java code from the SDL/PR description. 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, from the point of view of the presented method, however, they are invariant. It is important to note that we replace

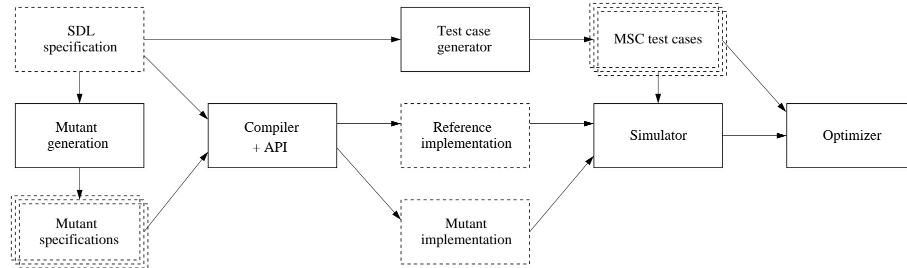


Figure 3.7: Components of the tool

non-boolean predicates with a sequence of boolean predicates, and apply the operators on them.

The MSC generator module generates test cases in MSC form. The input and output signals and the timers and their parameters are extracted from the SDL specification and used for creating an input/output transition system by means of the state space search algorithms of Section 2.4.3. Finally MSC traces are written in textual form.

During test case generation, timeouts in the SDL systems are considered simple inputs, and accordingly the input mutation operator mutates them. To be able to test timeout transitions, timeout events are made controllable from the environment. That is, whenever a test case 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 case explicitly indicates that the tester will have to wait for the duration of the timer. This way, methods for test case generation in the next section (3.2.1) become time independent.

The test environment is the core of the tool (see Figure 3.8). This component executes the test cases against the implementations, sends messages according to the MSC test cases, and evaluates the messages received. The test environment outputs a boolean matrix, which is the subject of optimization. The optimization module that implements the select function (Definition 2.4.3) is built on a bacterial evolutionary algorithm [Vin02].

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 [BA96] and the parser generator Constructor of Useful Parsers (CUP) [Fla96].

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 also unfolds non-boolean decisions. The remaining parts are left untouched. This is followed by the code generation 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 MSC specifications as scripts

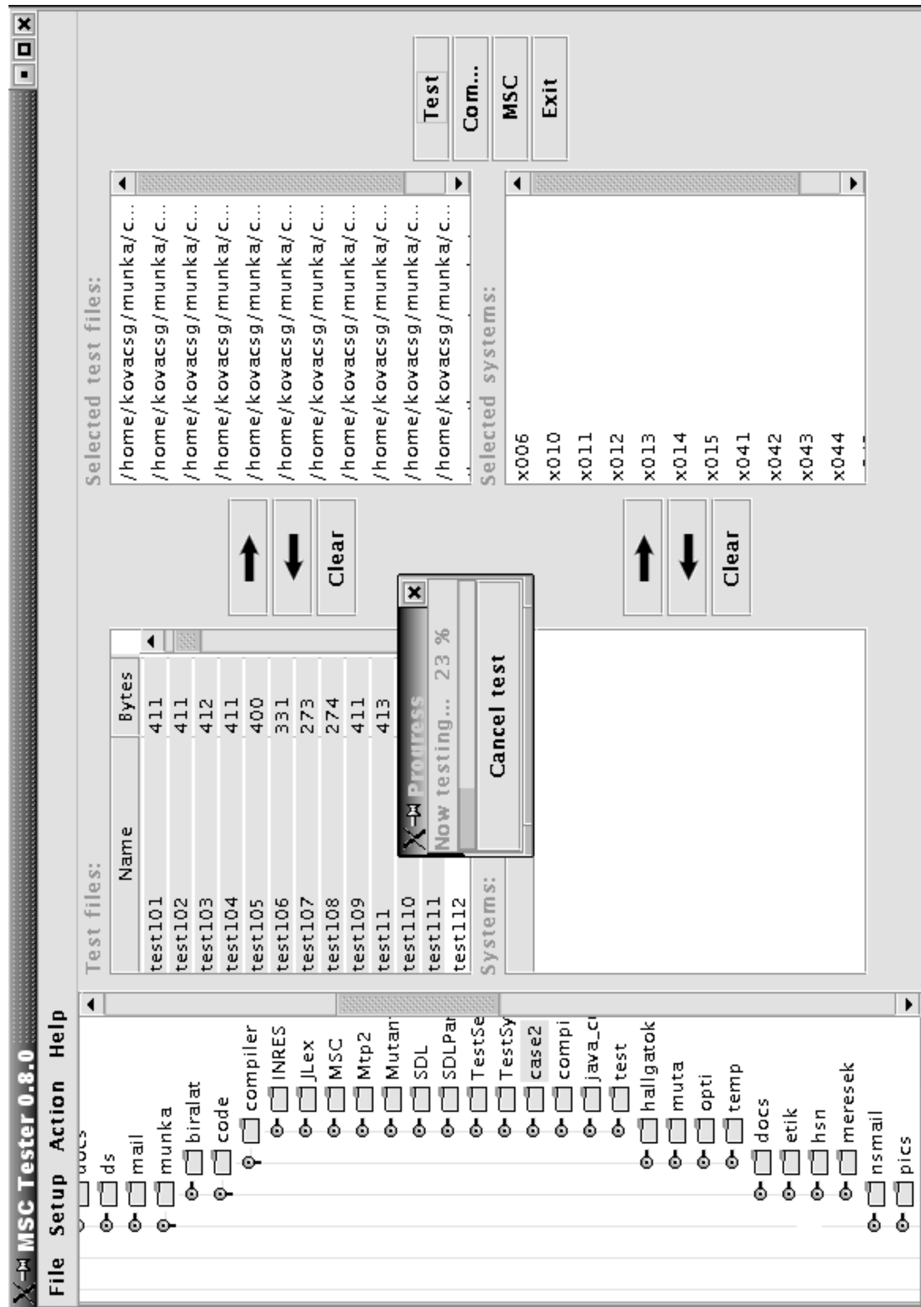


Figure 3.8: The test selection dialog

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 case 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 First-In First-Out (FIFO) channels. By-passing messages and the most important control functions are available as well.

3.3 Empirical analysis

This section shows the results of experiments conducted on sample systems (INRES, Conference Protocol) and real-life protocols (WAP WTP, SS7 MTP level 2). First in Section 3.3.1 a fault model introduced in Zoltán Pap's Ph.D. thesis [Pap06] in presented, which is used in the experiments.

3.3.1 Mutation operators for SDL systems used in the experiments

Much research has been done concerning mutation operator sets for different formalisms. A very important additional consideration for the definition of operators is that they should allow of the automation of the mutation testing process. To automate the process, mutation operators have to be defined in a way that enables their application to any specification regardless of actual realization and data types. It is also essential to generate syntactically correct mutants to ensure that executable mutant systems are created.

Definition 3.3.1 (Syntactic change operator) *Let $m = (S, V, I, O, T)$ and $m' = (S, V, I, O, T')$ be two EFSMs. Let the $\omega : M \rightarrow M$ be the syntactic change operator that changes $m \in M$ to $m' \in M$ with the following properties: $\exists Z \in T : \forall t \in Z : \omega(t) \in T'$, and $|Z| = 1$* [2]*. Let $\Omega : M \times \mathcal{M}$ represent the set of all considered ω changes, where $\mathcal{M}$ is the set of mutant systems.*

According to these considerations, six types of mutation operators for SDL model are used depending on which part of the transition they are applied to, and additionally the semantics of SDL were also taken into account [Pap06]:

- state modification operator,
- input modification operator,
- output modification operator,
- action modification operator,
- predicate modification operator,
- save missing operator.

[2] This means that one fault is considered at a time.

Operator 1 (State) Modifying states. Here only the exchange of inputs should be considered. Mutating the next state in transition $t \in T$: $t = (s, i, P(V), A(V), o, \omega(s'))$. The next state of a transition is replaced, so the system is brought to a wrong state and incorrect operation is induced. The mutation of the initial state is a special case of state mutation, where s_0 is modified: $\Omega(s_0)$.

Operator 2 (Input) The input mutation in the transition $t \in T$: $t = (s, \omega(i), P(V), A(V), o, s')$.

1. *Using $\omega(i) := null$. This mutation transforms a transition for an input at a given state into implicit "null" transition.*
2. *Using $\omega(i) := i_x$, where $i_x \in I_s$, $I_s \subseteq I$ is the set of inputs, that have transitions explicitly defined at the given s state.*
3. *Assigning the transition of input i_i ($i_i \in I$, but $i_i \notin I_s$, where $I' \subseteq I$ is the set of inputs, that have explicit transitions defined at the given s) to the existing transition branch of input $i \in I$. This mutation means that we add a transition for the input i_i at state s, that was implicitly consumed previously. Using this mutation, also the processing of inopportune (valid input arriving at wrong time) inputs can be inspected.*
4. *As mentioned previously, inputs and outputs may have parameters. If $i \in I \times V$ then we can mutate not only the input symbol, but the input parameter leaving the input symbol unchanged. This type of mutation is practically an action mutation, and can be viewed as the mutation of the implicit action assigning the new values. In this case, $\omega(i) = i(\omega(v))$, where $\omega(v) := null$ – according to the action mutation (see below).*

Operator 3 (Output) The mutation of an output event in the transition $t \in T$ is: $t = (s, i, P(V), A(V), \omega(o), s')$. If the output symbol has parameters ($o \in O \times V$) then we have the possibility to modify the parameter: $\omega(o) = o(\omega(v))$, where $\omega(v) := null$.

Operator 4 (Action) It is difficult to define a general mutation operator for actions, because of the presence of abstract data types. As no prior knowledge is available on what operations are defined by a type, it can not be generalized. Only the $\omega(a(V)) := null, a(V) \in A(V)$ operator, that is the deletion of an action is suitable for this case. Missing action operator: $t = (s, i, P(V), \omega(A(V)), o, s')$.

Operator 5 (Predicate) The mutation of boolean predicates has similar effects to the mutation of inputs. (As it can be seen in the graph representation, Figure 2.4, inputs and answers to conditional expression define outgoing arcs from a node.)

1. *Exchanging two answers of a boolean conditional expression in the transition $t \in T$ can be done simply negating the whole expression $t = (s, i, \omega(P(V)), A(V), o, s'$, where $\omega(P(V)) := not(P(V))$.*
2. *Setting the predicate to be stuck-at-true ($\omega(P(V))$:=$true$) or stuck-at-false ($\omega(P(V))$:= $false$) brings on the removal of the other branch.*

Operator 6 (Save) Since this chapter considers SDL semantics to be the specification language used, we have to take its specialties like the save mechanism into account. Note that save is not part of the EFSM model. This operator removes the save statement.

Table 3.1 demonstrates some examples of the application of the mutation operators on the INRES protocol.

Table 3.1: Mutation Operators for SDL

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

3.3.2 SDL systems examined

This section gives an overview of the complexity of the SDL specifications. The systems examined in the experiments are: the INRES[EHS97], the Conference Protocol [BFV+99], the real-life protocol standards: the WAP WTP [For98] and the SS7 MTP level 2 [IT97a].

Table 3.2: Complexity of the systems examined

System	INRES	Conference Protocol	WAP WTP	SS7 MTP2
Block	3	2	2	1
Process	6	2	2	12
Signal	20	8	5	68
Timer	1	0	6	7
Data types	3	4	5	4
State	12	3	11	34
Input	26	7	12	148
Output	35	7	12	224
Variable	17	27	15	67
Action	30	55	52	248
Predicate	8	20	15	69

Table 3.2 summarizes the complexity of the protocols used. It shows the number of

occurrences of keywords of block, process, signal, timer declaration, new data type definition, control state, input, output, variable, action and predicate in the SDL specification. [3]

The INRES and the WAP WTP operate in a client-server architecture and are thus asymmetric. So the client (initiator) and the server (responder) sides of these protocols should be tested separately.

3.3.3 Test generation for the INRES protocol

At first we used the well-known sample telecommunications system INRES [EHS97] to investigate the method presented. We chose a sample protocol that includes all the typical properties of real life protocols, and is built on the Open Systems Interconnection (OSI) concept. INRES is a connection-oriented protocol that operates between two protocol entities Initiator and Responder. These protocol entities communicate over a Medium service.

Algorithm 2.3 and Algorithm 2.1 (employed in existing tools) were used for generating stimuli for the system and formulating an MSC test case. The mutant generator of our software tool created 125 mutants for the INRES system using the fault model in Section 3.3.1.

Table 3.3: Data from the INRES case study

Mutation Operator	Generated Mutants	Detection Ratio	
		Exhaustive (5)	Random
State	30	63%	100%
Input	28	82%	100%
Output	36	77%	100%
Action	22	68%	100%
Predicate	8	63%	100%
Save	1	100%	100%

The result of the experiment is shown in Table 3.3. The first column of the table defines the type of fault injected. The second column contains the number of mutant systems generated with the fault type of the first column. The third and fourth columns show the fault detection ratios for exhaustive (Algorithm 2.3) and random (Algorithm 2.1) test generation strategies.

At depth 5, the exhaustive strategy produced 125 test cases; 23 of those were able to detect 72.8% of the faults. Random search was able to detect all faults with a maximum depth of 15. After the optimization the size of the random set could be reduced to 12 cases.

[3] Note that these values are derived from the EFSM representations of the protocols.

3.3.4 Test generation for 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.

Just as in the previous example, Algorithm 2.3 and Algorithm 2.1 were used for generating stimuli for the system and formulating an MSC test case. The mutant generator of our software tool created 116 mutants for the Conference Protocol system using the fault model in Section 3.3.1.

Table 3.4: Data from the Conference Protocol case study

Mutation Operator	Generated Mutants	Detection Ratio	
		Exhaustive (6)	Random
State	6	100%	100%
Input	28	100%	100%
Output	7	100%	100%
Action	55	85%	100%
Predicate	20	80%	100%

The structure of Table 3.4 is the same as the structure of Table 3.3. The exhaustive set of depth 6 (216 traces) was able to detect 89.6% of the injected faults, after the optimization that set could be reduced to 27 cases. Random search was able to find all faults with the maximum depth of 12, after the optimization of the test set 10 test cases remained.

3.3.5 Test generation for the WAP WTP

The Wireless Transaction Protocol provides services for interactive applications in the Wireless Application Protocol stack. It operates in wireless datagram networks and defines a light weight connectionless transaction oriented service in a client-server architecture. WTP provides reliable service, retransmission and acknowledgement for upper layer messages.

It is implemented in every mobile phone and used for browsing and Multimedia Messaging Service (MMS) messaging.

Algorithm 2.1 was used for generating stimuli for the system and formulating an MSC test case. The exhaustive test generation was impractical in this case because of the large number of possible input event. Although the SDL specification defines only five signals,

each is parameterized with a complex data structure. This results in a large number of possible input/output events.

The mutant generator of our software tool created 418 mutants for the WAP WTP system using the fault model in Section 3.3.1.

Mutation Operator	Generated Mutants	Detection Ratio
State	21	100%
Input	108	59%
Output	50	78%
Action	137	47%
Predicate	102	59%

Table 3.5: Data from the WAP WTP experiment

The results of the experiment are shown in Table 3.5. Random search was able to detect 248 mutants, 59%. The longest test case contained 48 interactions. The generated set of test cases could be reduced to 39 traces.

3.3.6 Test generation for the SS7 MTP2

The SS7 is protocol stack used for exchanging call control and service related information in Public Switches Telephone Networks (PSTNs) in a dedicated signalling network. MTP level 2 [IT97a] is responsible for the reliable, in order and error free transfer of signal units between neighboring nodes of the signalling network.

MTP2 has three Protocol Data Units (PDUs) on the control pane and several messages on the management plane. The control pane PDUs contain two 7-bit counter, so the input/output alphabet for the search algorithms of Section 2.4.3 contains about 200000 different events. To reduce the complexity of search, an improve the efficiency of test generation only a subset of this large input-output event set was used in the experiment. 93 events including the 7 timeout events were selected by a preliminary test data selection, those that are reachable from the initial state with valid sequences containing two input events. A huge part of the state space can not be reached with this set, so many faults injected can not be detected.

Similarly to the WTP experiment, Algorithm 2.1 was used for generating stimuli for the system and formulating an MSC test case. The exhaustive test generation was even more impractical in this case. The mutant generator module produced 720 faulty systems.

The result of the experiment are shown in table 3.6. Random search with the limited input-output alphabet was able uncover 110 faults, 15.2%. The longest MSC trace contained 102 events. The optimization reduced the 720 cases to 53.

Mutation Operator	Generated Mutants	Detection Ratio
State	31	35%
Input	148	16%
Output	224	12%
Action	248	16%
Predicate	69	12%

Table 3.6: Data from the SS7 MTP2 experiment

3.3.7 Performance

Fundamental questions of the empirical analysis are 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 a test generation procedure will take. The experiments above were conducted on a PC with Intel P4 3GHz processor, 2GB RAM and Linux operating system. Table 3.7 shows the running times of the individual modules. Data were acquired with the UNIX command time and given in form of [hh:]mm:ss. The maximum amount of memory assigned to the processes was limited to 1GB (java -XmX1g). The two results in the Testing and Test Generation fields for the INRES and the Conference Protocol apply to the exhaustive and random test generations.

Table 3.7: Execution times ([hh:]mm:ss)

Protocol	Mutant Generation	Code Generation & Compiling	Test Generation	Testing	Optimization
INRES	00:46	08:13	00:12 / 00:35	01:24/06:15	00:03
Conference Protocol	00:40	00:07:03	00:14 / 00:38	01:26/05:51	00:03
WAP WTP	00:01:20	01:20:11	03:18	1:37:01	5:15
SS7 MTP2	00:02:46	02:47:53	07:02	03:04:12	22:18

The generation of mutant systems and the initial test case set does not take much time. The most time was spent compiling the mutant systems to Java, because the javac compiler of the JDK 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 cases in the initial set, and on the complexity of the individual test cases.

3.4 Conclusion

Existing software tools and their test generation theory support exhaustive search and random walk based test generation. Increasing the depth of exhaustive state space exploration depth increases the fault coverage, but exponentially increases the number of test cases generated. Random walk on the other hand provides a test set with uncertain quality.

The fault based method proposed in this chapter can be applied to SDL specifications to generate a reduced set of abstract test cases in MSC form automatically. The number of test cases generated has an upper bound determined by an input fault model. After the optimization the method further reduces the number of test cases generated with random search, and preserves the same level of fault coverage at the same time. Experiments show that the method and the software tool we implemented can handle industrial size protocols.

Though this approach does not unfold the specification to an FSM, it has still a high computation requirement that depends on the size of the input-output alphabet and the size of the fault model. The number of mutant systems is a polynom of the size of the fault model. The input-output events influence the effectiveness of search methods, which is a weakness of this solution, because different input and output parameter vectors are mapped to different abstract events. Distributed computing can help in decreasing the computation demands.

Chapter 4

String edit distance based test set optimization

This chapter proposes a string edit distance based test selection method to make test sets for telecommunications software more compact. Following the results of previous research, a trace in a test set is considered to be redundant if its edit distance from others is less than a given parameter. The algorithm first determines the minimum cardinality of the target test set in accordance with the provided parameter, then it selects the test set with the highest sum of internal edit distances. The selection problem is reduced to an assignment problem in bipartite graphs.

Extensions are given to Vuong's and Feijs' approach that are described in [VAC92] and [FGMT02]. Vuong's original approach compares two traces and tells if those are redundant. This method determines which of the two should be removed from a containing test set. Two selection criteria are introduced to find the most compact test set with the highest possible diversity with regard to the distance based coverage metric. First the minimum cardinality of the test set for a given ε parameter is determined by reducing the distance based selection problem to an assignment problem in bipartite graphs. Then a test set with the highest overall internal edit distance is selected with the help of the k-assignment problem defined by Dell'Amico and Martello in [DAM97]. Finding both the minimum size and the highest overall internal edit distance are shown to be P-space problems.

4.1 Background and related work

4.1.1 Related work

The original idea was proposed in [VAC92] and extended in [FGMT02] . The former paper by Vuong and Alilovic-Curgus introduces a test coverage metric based on the concept of testing distance between traces. This metric is used to approximate differences among patterns of system behavior. Traces are considered to be similar (redundant) if they can be transformed to each other with a cost no more than a given ε parameter, so that the test set is said to

be ε-dense. An important contribution of that paper is a formula to determine the edit cost associated with each symbol of a string such that the edit costs are normed.

In the latter paper Feijs et al. generalize the original idea by introducing a reduction heuristic and a cycling heuristic. They suggest that test cases should note the actual state of the specification as well. The notion of marked traces is proposed to tackle the problem of traces revisiting states of the specification via the same loop at most a given number of times determined by the reduction heuristic. Formulae are given to precisely calculate the normed distance between traces containing symbols and marked traces (loops).

Test suite size reduction has been addressed in several papers like [HGS93] or [CK97] or the one presented in Chapter 3. A common property is that those methods try to find a maximum matching between the sets of test cases and the set of test case requirements or set of test case sub-purposes or set of faults injected. That problem is NP-hard, thus integer programming [WP02] or soft computing solutions have been proposed.

4.1.2 String representation of traces

Traces (see Section 2.2.4) can be represented as strings on an arbitrary alphabet C. A mapping $\mu : \{I \cup O\}^+ \rightarrow C$ defines a set of pairs of an event sequence and a string of alphabet C: $\langle (x_{i1}x_{i2}\ldots x_{im}), c_i \rangle$, where $x_{ik} \in I \cup O, k = 1\ldots m, k \in \mathbb{N}, c_i \in C$. [1] Note that according to this definition more than one successive event (such as a loop around a state) may be mapped to a character of the given alphabet. Such mapping μ is the marked trace notation defined in [FGMT02].

Throughout this chapter it is considered that specification machines have the reliable reset capability, and a finite set of traces is generated from the same same initial state (i.e. starting with reset) by means of random-walk (see Section 2.4.3), but any other test derivation algorithm may be considered as well.

Example Let us use the `ISAP Manager Ini` process of the INRES system [EHS97] as example; its SDL representation can be seen in Figure 4.1. The process has three states $S = \{$`disconnected`, `wait`, `connected`$\}$, the initial state is `disconnected`. The machine is not completely specified: input signals such as the `IDATreq` in state `disconnected` are not shown and considered to be implicitly consumed, that is, they do not change the state of the machine and do not produce output. The input and output signal sets are the following: $I = \{$`ICONreq`, `T`, `ICONF`, `IDATreq`, `IDIS`$\}$ and $O = \{$`ICON`, `IDISind`, `ICONconf`, `IDAT`$\}$. The timeout `T` is considered to be an input signal and according to the specification it is only possible in state `wait`.

Let us assume a simple signal sequence to character mapping μ defined by Table 4.1, which maps each possible transition of the process `ISAP Manager Ini` to a character. The table indicates implicit events as well, dash represents that no output signal is produced. The state transition diagram of the unfolded FSM and the graph with mappings applied are presented in Figure 4.2(b). Note that the cycling heuristic of [FGMT02] is not used in the mapping for the sake of simplicity.

[1] In the context of this chapter, each different i/o pair that appears in the set of transitions of the FSM is denoted with a separate character.

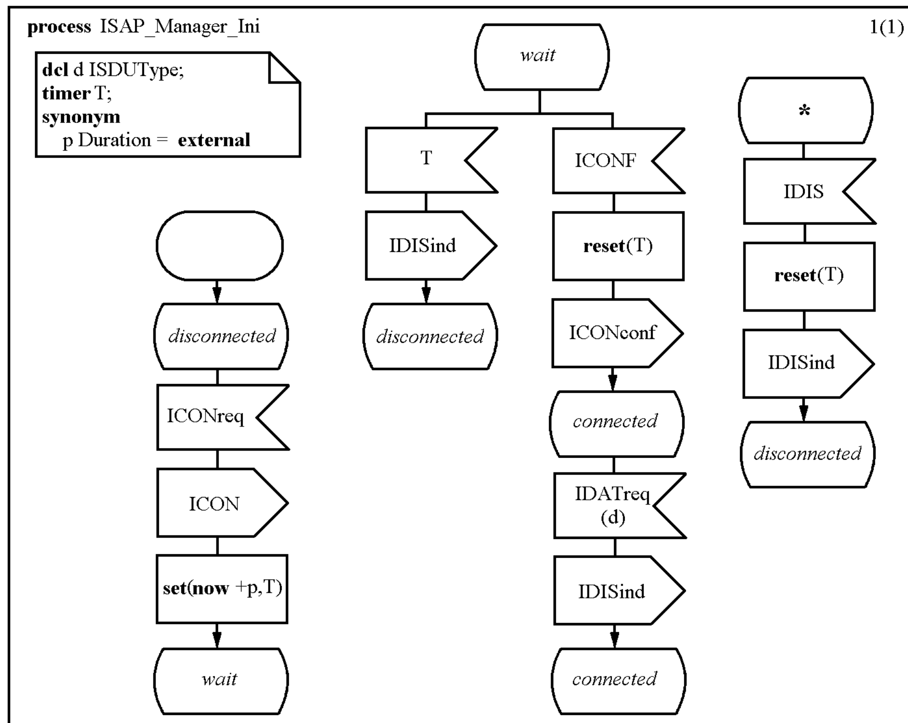
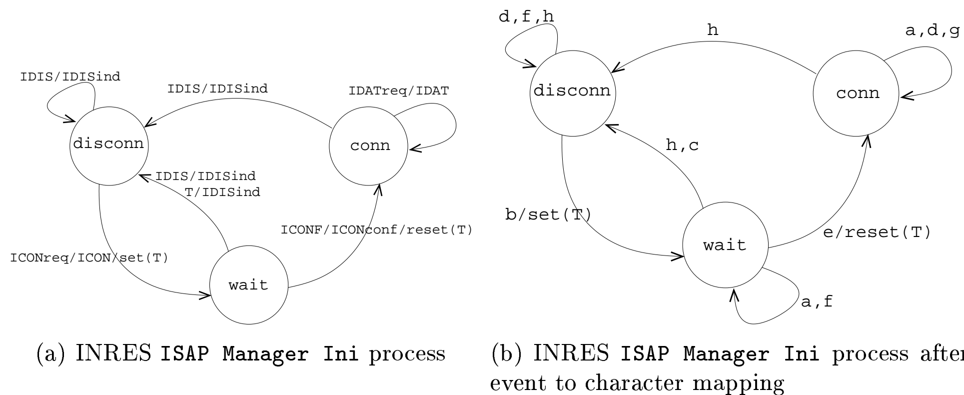


Figure 4.1: INRES ISAP Manager Ini process [EHS97]

Figure 4.2(a) shows the state transition graph of the `ISAP Manager Ini` when that is unfolded into a finite state machine, and Figure 4.2(b) shows the graph after applying the string mapping defined above.



(a) INRES `ISAP Manager Ini` process

(b) INRES `ISAP Manager Ini` process after event to character mapping

Figure 4.2: The state transition diagram of the FSM of the ISAP Manager Ini process from the sample INRES protocol in original form and after event to character mapping

ICONreq/−	a
ICONreq/ICON	b
T/IDISind	c
ICONF	d
ICONF/ICONconf	e
IDATreq/−	f
IDATreq/IDAT	g
IDIS/IDISind	h

Table 4.1: Signal sequence to character mapping for ISAP Manager Ini

4.1.3 Trace distance

The edit distance between strings σ_1 and σ_2 is the minimum number of edit operations (character insert, character delete and character overwrite) needed to transform σ_1 to σ_2. Different edit operations may have distinct costs assigned, but for the sake of simplicity we consider unit edit operator costs in the current paper.

Definition 4.1.1 (String edit distance metric) *Let Σ be a finite set of strings over the alphabet C. $d : \Sigma \times \Sigma \to \mathbb{R}$ is a distance metric if for all $\sigma_1, \sigma_2, \sigma_3 \in S$ the distance of any pairs of strings is non-negative $d(\sigma_1, \sigma_2) \geq 0$, the distance to self is zero, $d(\sigma_1, \sigma_1) = 0$, the relation is symmetric $d(\sigma_1, \sigma_2) = d(\sigma_2, \sigma_1)$ and the triangle inequality holds $d(\sigma_1, \sigma_3) \leq d(\sigma_1, \sigma_2) + d(\sigma_2, \sigma_3)$.*

If the lengths of σ_1 and σ_2 are l_1 and l_2 respectively, the time and space complexity of computing the distance is $O(l_1 l_2)$. The distance metric computation above is according to one defined in [WF74], but other approaches can be considered as well.

Definition 4.1.2 (Distance matrix) *Let $\mathbf{D} = [d_{ij}]$ be a distance matrix of the set of traces Σ. Let $d_{ij} = d(\mu(\sigma_i), \mu(\sigma_j))$, where $1 \leq i, j \leq |\Sigma|, i, j \in \mathbb{N}$.*

Note that $\mathbf{D}$ is symmetric because of the symmetric nature of the distance and the size of the matrix is $|\Sigma| \times |\Sigma|$. Let $\mathbf{D}_U$ and $\mathbf{D}_L$ denote the strictly upper and lower triangular matrices of $\mathbf{D}$ respectively.

The metric space involves the normalization of edit distance values as is done in [VAC92] and [FGMT02]. For the sake of simplicity – and without any loss of generality – in this chapter we dispense with the normalization and consider that the distances are non-negative integer numbers.

Example Let the test set Σ be composed of the MSCs of Figure 4.3 derived with random walk for the `ISAP Manager Ini` process. According to the μ mapping of Table 4.1 $\mu(\Sigma)$=\{*dbafacfd, fhdhbehf, beghbfff, dhbchfdf, ddbcdfff, hdbafchd, bachdfhb,bhbfeggg*\} are the string representations of Σ. Note that reset inputs are omitted in the figure for the sake of perspicuity, and all test sequences have a reset input as prefix, thus starting from initial

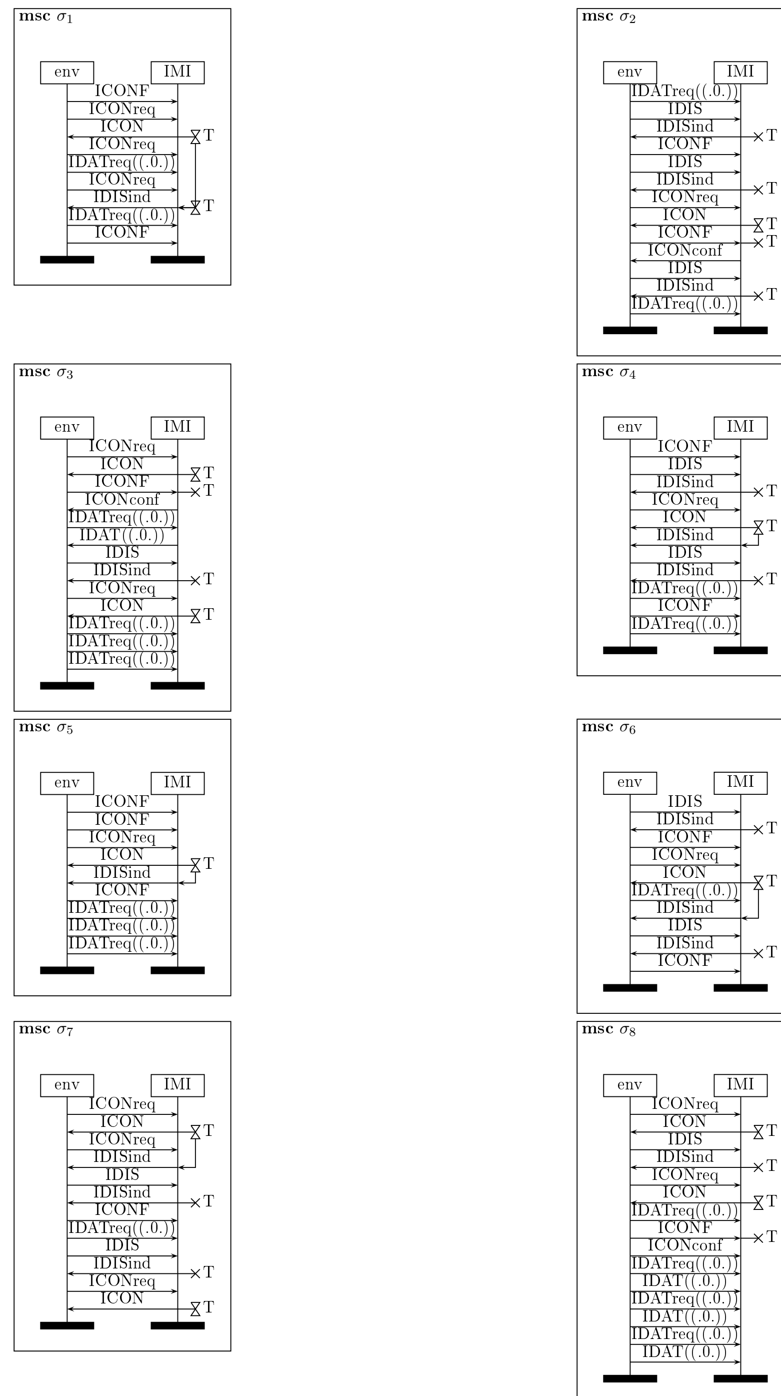


Figure 4.3: The Σ test set of eight MSCs generated with random walk. (IMI abbreviates `ISAP Manager Ini` from the INRES system)

state `disconnected`. These eight strings together traverse 11 of the 13 transitions of the machine in Figure 4.2(b).

The distance matrix of this trace set $\mu(\Sigma)$ with unit cost edit operations is:

$$\mathbf{D} = \begin{bmatrix} 0 & 8 & 7 & 6 & 6 & 3 & 6 & 7 \\ 8 & 0 & 5 & 5 & 7 & 6 & 6 & 7 \\ 7 & 5 & 0 & 6 & 5 & 8 & 5 & 6 \\ 6 & 5 & 6 & 0 & 3 & 6 & 6 & 6 \\ 6 & 7 & 5 & 3 & 0 & 6 & 6 & 7 \\ 3 & 6 & 8 & 6 & 6 & 0 & 6 & 7 \\ 6 & 6 & 5 & 6 & 6 & 6 & 0 & 7 \\ 7 & 7 & 6 & 6 & 7 & 7 & 7 & 0 \end{bmatrix}$$

4.2 Maximizing the diversity in trace sets

The papers of Vuong [VAC92] and Feijs [FGMT02] give a solution to determine if two test cases are redundant. However pairwise redundancy check and the immediate removal of one of the redundant cases from a test set of more than two traces does not necessarily give the best reduction. One trace removed due to redundancy early on may have covered others later. Therefore the pairwise redundancy search must be extended to the whole test set to find the maximum reduction.

This section proposes a method for selecting a smallest test set with the highest possible diversity with regard to the distance based coverage metric. The inputs of the method are the $\mathbf{D}$ distance matrix of the traces, and an ε parameter. The method consists of two stages: first the minimum cardinality of the target test set is calculated assuming the ε parameter, then the test set with the maximum internal distance is selected.

For our discussions we assume the notion of ε-approximation as defined by Feijs et. al. in [FGMT02]:

Definition 4.2.1 ***ε-cover.*** *Σ' is an ε-cover of Σ, where $\Sigma' \subseteq \Sigma, \varepsilon \geq 0 \iff \forall \sigma \in \Sigma : \exists \sigma' \in \Sigma' : d(\mu(\sigma), \mu(\sigma')) \leq \varepsilon$.*

Definition 4.2.1 implies that for each Σ and ε there exists at least one minimal cardinality Σ' ε-cover of Σ, for which $\forall \sigma'_i, \sigma'_j \in \Sigma' : d(\mu(\sigma'_i), \mu(\sigma'_j)) > \varepsilon$. The test set Σ is divided into two disjoint subsets: a Σ'' subset of test cases that can and a $\Sigma'_0 \subseteq \Sigma'$ subset of test cases that can not be ε-covered by other test cases. The test cases from Σ'_0 must be included in Σ', and from Σ'' the minimum number of cases must be selected: $\Sigma' = \Sigma'_0 \cup \mathsf{reduce}(\Sigma'')$. Thus the maximum reduction of Σ'' yields the most compact Σ'.

4.2.1 Finding the cardinality of the optimal subset

Algorithm 4.1 finds the size of the most compact test suite Σ' that can be achieved with the string edit distance based selection in polynomial time for a given distance matrix. The

inputs of the algorithm are the distance matrix of Σ and a ε threshold parameter. For the computation two matrices $\mathbf{A}$ and $\mathbf{A}_\psi$ of boolean values and a bipartite graph G' are used. The algorithm first determines which test cases of Σ ε-cover each other, and if they do, it is marked in $\mathbf{A}$ with 1 value (lines 4-7). Then, the size of the Σ'_0 set is determined in lines 8-10. In lines 11-15, if exactly the same coverage is found for two test cases, then one of them is eliminated. Lines 16-22 construct a bipartite graph G' based on matrix $\mathbf{A}$. The Hopcroft-Karp algorithm [CLRS01] is used for finding a maximum cardinality assignment ψ in G' (line 23). The k minimal size returned in line 28 is the size of Σ'_0 plus the rank of the upper or lower triangular matrix of $\mathbf{A}_\psi$ constructed based on the maximum matching in lines 24-27.

The complexity of the construction of $\mathbf{A}$ (lines 4-7) and G' (lines 16-22) are $O(|\Sigma|^2)$. The worst-case complexity of the Hopcroft-Karp algorithm [CLRS01] applied to the graph $G' = (N', E')$ in line 23 is $O(\sqrt{|N'|}|E'|)$. Since $|N'| \leq 2|\Sigma|$ and $|E'| \leq |\Sigma|^2$ according to lines 16-22, its complexity is $O(|\Sigma|^{5/2})$. Hence the worst case complexity of this algorithm is determined by the search for same rows in $\mathbf{A}$ in lines 11-15, which is $O(\Sigma^3)$ [2].

Example The ε-coverage of matrix $\mathbf{D}$ of the example at the end of Section 4.1.3 with $\varepsilon = 5$ (note that we dispense with normalization) and the bipartite graph G' constructed from $\mathbf{A}$ are in Figure 4.4. In matrix $\mathbf{A}$ on the left the eighth is the only row that contains only 0 values, therefore $\Sigma'_0 = \{\sigma_8\}$. The second and the fifth row are identical, so we may leave either σ_2 or σ_5 from the further processing as redundant case. The bipartite graph G' and the matching problem constructed from Σ'' without the redundant σ_5 can be seen on the right. Edges show that a trace ε-covers an other test case. Bold lines select a maximum cardinality matching: the selected elements of $\mathbf{A}_\psi$ boxed in the matrix $\mathbf{A}$ are $(1,6)$,$(2,4)$,$(3,7)$,$(4,2)$,$(6,1)$,$(7,3)$. The rank of the upper or lower triangular matrices of $\mathbf{A}_\psi$ is 3, therefore beside σ_5, three additional test cases can be removed: $\Sigma''_U = \{\sigma_1, \sigma_2, \sigma_3, \sigma_5\}$ or $\Sigma''_L = \{\sigma_4, \sigma_5, \sigma_6, \sigma_7\}$. The resulting Σ' candidates are $\Sigma'_1 = \{\sigma_1, \sigma_2, \sigma_3, \sigma_8\}$, if σ_5 was considered to be redundant, or $\Sigma'_2 = \{\sigma_1, \sigma_3, \sigma_5, \sigma_8\}$, if σ_2 was considered to be redundant and $\Sigma'_3 = \{\sigma_4, \sigma_6, \sigma_7, \sigma_8\}$. In general more maximum cardinality assignments may exist, but all with the same cardinality.

4.2.2 Optimizing the test suite using the given density and cardinality

According to [VAC92] and [FGMT02] two patterns of behavior are approximated to be less similar if the distance between their string representation is greater. The redundancy among the test cases in a test set is the least if the sum of all pairwise distances is maximal. Hence, if more than one minimal cardinality Σ' solution exists, the one with the maximum internal distance should be preferred.

Selecting the Σ' trace set with minimal cardinality from the trace set Σ, such that $|\Sigma'|$ is an ε-cover of $|\Sigma|$ and the test cases differ from each other as much as possible, can be calculated in a polynomial time of $|\Sigma|$.

[2] The complexity can further be reduced. Searching for identical rows in matrices can be done more effectively with the Rabin-Karp algorithm [CLRS01], which operates on the hash values of patterns. That algorithm can find all matching rows in $O(\Sigma)$. So the resulting complexity of Algorithm 4.1 is $O(|\Sigma|^{5/2})$

Algorithm 4.1: Deriving the string edit distance based test case selection problem to a matching problem in bipartite graphs

```
   input: D distance matrix of Σ test set; ε threshold
   output: k, the maximum number of redundant cases for the given ε
 1 data(A = [a_ij], a_ij ∈ {0,1}; A_ψ = [a_ψij], a_ψij ∈ {0,1};
 2 G' = (N', E') bipartite graph, where N' = N'_R ∪ N'_C)
   /* Initialization */
 3 k := 0, N'_R := ∅, N'_C := ∅, E' := ∅, A := 0, A_ψ := 0;
   /* Computing the A matrix */
 4 foreach i, j, 1 ≤ i, j ≤ |Σ| do
 5     if d_ij < ε then a_ij = 1;
 6     else a_ij = 0;
 7 endfch
   /* Counting the elements of Σ'_0 */
 8 foreach i do
 9     if ∑_j a_ij = 0 then k := k + 1;
10 endfch
   /* Finding test cases with the same coverage */
11 foreach k, l, 1 ≤ k ≤ |Σ| − 1, k < l ≤ |Σ| do
12     if ∑_j (a_kj xor a_lj) = 0 then
13         foreach j, 1 ≤ j ≤ |Σ| do  a_lj := 0, a_jl := 0;
14     endif
15 endfch
   /* Constructing the G' graph */
16 foreach σ_i ∈ Σ do
17     N'_R := N'_R ∪ σ_i;
18     N'_C := N'_C ∪ σ_i;
19 endfch
20 foreach i, j, 1 ≤ i, j ≤ |Σ| do
21     if a_ij > 0 then E' := E' ∪ {(n'_i, n'_j)};
22 endfch
23 Let E'_ψ be the matching selected by the Hopcroft-Karp algorithm;
   /* Marking the pairs of the maximum matching in A_ψ */
24 foreach i,j, 1 ≤ i, j ≤ |Σ| do
25     if (n'_i, n'_j) ∈ E'_ψ then A_ψij = 1;
26     else A_ψij = 0;
27 endfch
28 return k := k + rank(A^L_ψ)
```

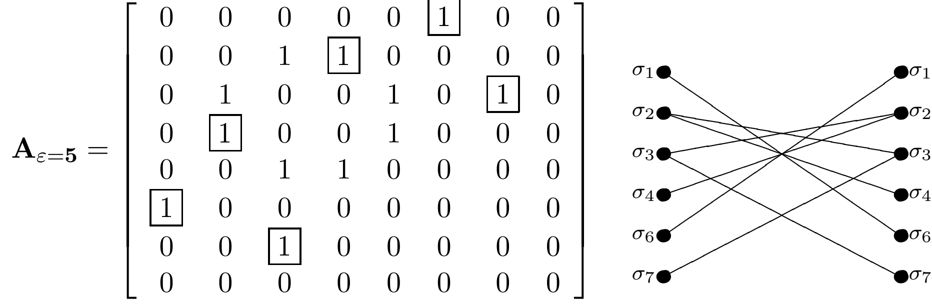


Figure 4.4: The **A** matrix and the bipartite graph G' constructed from it

To prove this statement first we introduce a metric for test suites. Definition 4.2.2 defines a metric space for the Σ superset of all test cases over the given i/o alphabet and the ψ mapping with the dd distance function.

Definition 4.2.2 (String edit distance based test set metric) *The pair $(\mu(\Sigma), dd)$ defines a metric space, where Σ is a superset of test cases over a given $I \cup O$ i/o alphabet, μ is a mapping function, and $dd : \Sigma \times \Sigma \to \mathbb{R}$ such that $\forall \Sigma_1, \Sigma_2 \subseteq \Sigma$:*

$$dd(\Sigma_1, \Sigma_2) = \Big| \sum_{\sigma_i \in \Sigma_1, \sigma_j \in \Sigma_1} d(\mu(\sigma_i), \mu(\sigma_j)) - \sum_{\sigma_i \in \Sigma_2, \sigma_j \in \Sigma_2} d(\mu(\sigma_i), \mu(\sigma_j)) \Big|$$

We use the definition above to set up an optimization problem such that $dd(\Sigma', \emptyset)$ is maximized. This means that the sum of distances between all pairs of the traces of Σ' must be maximal, therefore the optimization problem is:

$$\max \sum_{\sigma_i \in \Sigma', \sigma_j \in \Sigma'} d(\mu(\sigma_i), \mu(\sigma_j)), \tag{4.1}$$

where $\forall i, j : d(\mu(\sigma_i), \mu(\sigma_j)) > \varepsilon$.

A Σ' with the minimum k can be determined with Algorithm 4.1. There may be many Σ's with this k cardinality, and there is at least one with the maximum distance sum.

Let $\mathbf{C} = [c_{ij}]$, where $c_{ij} \in \{0, 1\}$, be a capacity matrix, and let the distance matrix $\mathbf{D} = [d_{ij}]$ of Σ be a cost matrix. This problem is equivalent to the following minimum cost maximum flow problem.

Let $G' = (N', E')$ be a bipartite graph constructed from the distance matrix $\mathbf{D}$ as in Algorithm 4.1. Let $G = (N, E)$ be directed weighted graph extending G' such that $N = \{s, s^*\} \cup N' \cup \{t\}$ and $E = \{(s, s^*), (s^*, n'_i)\} \cup E' \cup \{(n'_j, t)\}$, for all $n'_i \in N'_R, n'_j \in N'_C$ and let all edges between nodes N'_R and N'_C be directed from N'_R to N'_C. Each edge is assigned a capacity value and a cost defined as follows. Let the capacity of all edges $e \in E$ be $c = 1$, except for (s, s^*) that has a capacity of k. Hence, the maximum flow capacity is determined by the $\{(s, s^*)\}$ cut and it equals k. Let the cost of edges (s, s^*),(s^*, n'_i) and (n'_j, t) be 0, and let the cost of edges (n'_i, n'_j), where $n'_i \in N'_R, n'_j \in N'_C$, be $d'_{ij} = \max_{i,j}(d_{ij}) - d_{ij}$ for all i and j.

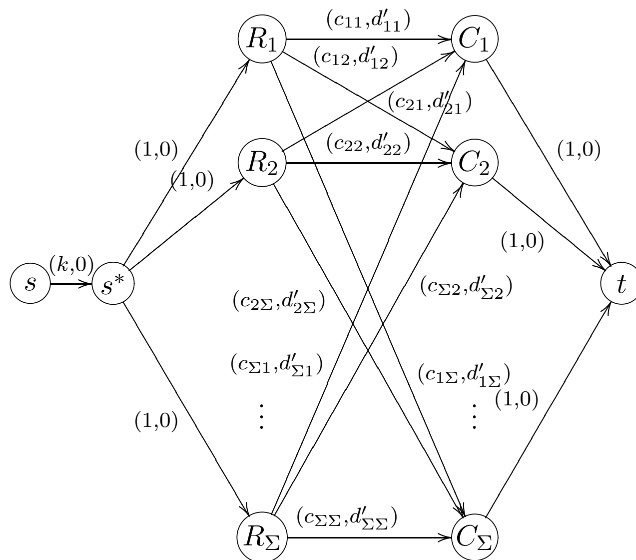


Figure 4.5: The flow problem equivalent to the maximum distance k-cardinality matching

The minimum cost maximum flow can be found by solving the following optimization with linear programming, that is, the problem is in other words:

$$\max \sum_{i=1}^{|\Sigma|} \sum_{j=1}^{|\Sigma|} c_{ij} d_{ij}, \tag{4.2}$$

$$\sum_{j=1}^{|\Sigma|} c_{ij} \leq 1, i = 1, ..., |\Sigma|, \tag{4.3}$$

$$\sum_{i=1}^{|\Sigma|} c_{ij} \leq 1, j = 1, ..., |\Sigma|, \tag{4.4}$$

$$\sum_{i=1}^{|\Sigma|} \sum_{j=1}^{|\Sigma|} c_{ij} = k, \tag{4.5}$$

where $c_{ij} \in \{0, 1\}$.

This problem has been defined as the k-cardinality assignment problem by Dell'Amico and Martello and has been shown to be a P-space problem in [DAM97].

Example It has been shown that $k = 4$ for the $\varepsilon = 5$ case and therefore the "best" solution contains $|\Sigma| - k = 8 - 4 = 4$ test cases. In the example three solutions have been found. The sum of distances between pairs of Σ'_1 is 40, for Σ'_2 and Σ'_3 it is 38, so Σ'_1 is the

best reduced suite.

4.3 Empirical analysis

This section gives simulation based evaluation of the string edit distance based test case selection method. A process from each of two systems, the `ISAP Manager Ini` process from the INRES [EHS97] and the Conference Protocol [HFT00], both represented in SDL, and test sets derived manually and automatically are used in this investigation. The result of the method proposed in this section are compared to the results of two different selection strategies based on pairing of test cases and test case requirements [HGS93]. One of these two methods is the fault coverage based test selection method proposed in [KPC01], where the test case requirements are considered to be faults injected systematically into the system according to the given fault model. The other is a transition coverage based method that regards the checking of a transition of an FSM as a test case requirement. Hence the two SDL systems are unfolded into FSMs.

Name	States	Inputs	Outputs	Non-implicit transitions	Faults injected	Symbols
INRES	3	5	4	7	25	8
Conference	5	11	3	19	78	22

Table 4.2: Properties of the unfolded SDL processes

Table 4.2 characterizes the two unfolded SDL systems. The INRES `ISAP Manager Ini` process' FSM consists of three states, has four inputs, a timer and four outputs. The total number of transitions is 15, the number of different faults that can be injected into that process according to the fault model proposed in [KPC01] is 25. The input and output events are represented with eight symbols. The Conference protocol limited to three users and at most one conference at a time has five states, eleven inputs and three outputs when unfolded into an FSM. 16 of the 55 transitions are non-implicit, and the same fault model yields 78 mutant systems. 22 symbols are used to represent the input and output events of this process.

Three test sets are used to evaluate the selection method proposed in this section. One is an automatically generated set of the eight cases that has already been used in the example in Section 4.1.2. The second is a manually generated one of thirteen cases, the transition checking test generation method was used. These first two sets are executed against the `ISAP Manager Ini` process. A third automatically generated set is run against the Conference Protocol.

The results of simulations can be seen in Table 4.3. The first column defines the method of test generation used to obtain the test set to reduce, the second column gives the system the test cases are derived from, the third column defines the three reduction methods, which are the proposed string edit distance based method (denoted as string), the transition coverage based method (transition) and the fault coverage based method (fault). In the

Test set	System	Method	Number of selected cases	Number of faults detected	Transitions covered
Automatic (8)	INRES	String	4/8	19/25	11/13
		Transition	3/8	19/25	11/13
		Fault	4/8	23/25	11/13
Manual (13)	INRES	String	6/13	19/25	6/13
		Transition	8/13	19/25	10/13
		Fault	4/13	23/25	8/13
Automatic (40)	Conference	String	14/40	59/78	39/55
		Transition	11/40	59/78	40/55
		Fault	6/40	60/78	25/55

Table 4.3: Results of the selection experiments

experiment on the INRES system's `ISAP Manager Ini` process with eight automatically generated test cases, the string edit distance based method and the transition coverage based method provide exactly the same three test cases. These three cases miss four faults that could be detected with the fault coverage method. As automatically generated cases are more likely to check implicit transitions their transition coverage is greater than in case of the manual cases. In the experiment with 13 manually generated cases, the string edit distance based method achieves a bigger reduction (13 to 6 vs. 13 to 8) than the transition coverage based method at the cost of not traversing all transitions. Both provide the same fault coverage, but miss four injected faults that could be found.

In case of the Conference Protocol all three methods provide nearly the same fault coverage. The biggest reduction is achieved by the fault coverage based method, but that traverses the least transitions.

In general, according to the experiments shown above, the smallest reduced test and the highest fault coverage ration is provided by the fault based method. The highest transition coverage can be achieved by the transition coverage based method. The string based method provides results close to the transition coverage based method, but this is automatic and its computational complexity is smaller than the automatic fault based method.

	INRES automatic (8)	INRES manual (13)	Conference automatic (40)
Generation	1s	n/a	1s
String based	1s	1s	1s
Transition based	1s	1s	2s
Fault based	12s	15s	32s
Optimization	3s	4s	42s

Table 4.4: Execution times

Table 4.4 shows the execution times for automatically performed test set reduction activities. The experiments above were conducted on a PC with Intel P4 3GHz processor,

2GB RAM and Linux operating system, the data in the table were acquired with the UNIX command `time`. Columns represent the test sets in the rows of Table 4.3. Rows represent the execution times of different test set reduction and coverage evaluation activities.

The execution times are low because of the small sample systems and test sets used in the experiments. The μ mapping activity is not included in the table, because that was predefined for the experiment for both protocols. It is an input parameter for the string edit distance based method, so the time required for mapping must be taken into account in real life application.

4.4 Conclusion

This chapter proposed an efficient approach to select a subset of a test set in polynomial time of its size by searching for similar patterns of events. The approach builds on previous results of string edit distance based test selection methodology, it can exactly determine which of any two test cases that ε-cover each other should be removed. Two selection criteria are given: one to identify the minimum cardinality of the target test set for a given a ε-cover, and an other to select the test cases that differ from each other as much as possible assuming the string distance based metric.

The method is compared with fault and coverage based test selection techniques using the sample systems INRES and Conference Protocol. The string edit distance based method provides similar fault detection capability as the transition coverage based selection, but does not achieve the same rate of reduction, more test cases remain in the resulting set. Its main advantage is that it requires less computation than the other two methods. It is shown to be a polynomial time approach, while other selection approaches are proven to be NP-hard.

To make the string edit distance based method applicable to industrial size systems, further research is necessary on efficient μ mapping functions.

Generalization of this approach is possible for labeled, rooted trees to support test selection for test cases represented in TTCN-2 (or TTCN-3) form. This requires the calculation of the distance matrix of such trees, which does not impose any fundamental change to the method.

Chapter 5

Iterative test specification

This chapter proposes a method for iterative test specification, which is on the one hand an alternative extension to single staged methods like the ones in Chapter 3 and Chapter 4, and on the other hand an efficient approach to facilitate test set maintenance.

The extensive size of automatically generated test sets has increased significance both for progressive software development and for software maintenance, when test suites need to be periodically modified and extended. The goal of this chapter is hence dual:

- reduce the computation demands made by the size of the automatically generated test set, and
- support the automatic maintenance of test sets of systems developed incrementally in evolutionary processes like the spiral development process [Boe88].

The first goal is important when the proposed method is applied to a reduction method based on matching between a set of test cases and a set of test case properties, which is NP-hard [HGS93]. When a system is developed incrementally this method allows the efficient combination of old test cases and newly generated ones in a test suite.

The large computation demands of the test selection method proposed in Section 3.2 can be reduced. This method is based on test set metrics and the heuristics of partial matching among the set of test cases and a set of test case properties. The optimum is determined for a subset of the test case set, and in each iteration cycle the metrics of the old and new sets are compared.

Since this approach does not specify the time between two iteration cycles, it can be used for maintaining test sets for incrementally developed systems.

This chapter proposes two iterative test generation algorithms in Section 5.2. The first one aids the incremental development of a test set: a test case is added to the test set only if it increases the testing power of the test set. The second one performs an immediate optimization for each newly generated set of test cases. Two different metrics are proposed to decide if a test case should be added or not. The fault based metric of Section 5.3 is based on the solution of Chapter 3. The string edit distance based metric uses the approach of Chapter 4.

5.1 Related work

Recently several research papers [WW06, LI07, KHS08] proposed test data selection solutions with evolutionary methods. These approaches build on a similar idea as the one used in this chapter, and may effectively support the search algorithms (Section 2.4.3) employed in the solution of this chapter. Yen et al. propose a test generation method for object-oriented software that operates on a tree-based representation of method call sequences, and apply a distance-based fitness function that accounts for runtime exceptions. Lefticaru and Ipate [LI07] investigates the use of genetic algorithms in test data generation for the chosen paths in the state machine, so that the input parameters provided to the methods trigger the specified transitions. Kalaji et al. [KHS08] propose an evolutionary approach that searches for input parameters to be applied to a set of functions to be called sequentially.

5.2 Iterative test generation algorithms

5.2.1 Evolutionary algorithms in test generation

An evolutionary algorithm [SDJB$^+$93] follows a computational model of the biological evolution process. A "population" of structures is maintained by the algorithm. These structures evolve in cycles according to the rules given by means of selection, mutation and reproduction. In every cycle, the fitness of each element of the population is measured. In the selection process, the ones with the best fitness are considered. The selected population is mutated to provide a new population.

In this chapter, instead of creating a large test suite and then selecting the adequate cases, iterative test generation algorithms develop adequate test suites iteration by iteration. In each iteration cycle some test cases in the test suite are modified, replaced by new ones, or even new cases are added. Then it is checked whether the new suite is "better" than the old one from the point of view of a metric that can be derived from, for instance, the matrix of Algorithm 3.2 or the metric of Definition 4.2.2.

5.2.2 Iterative test derivation

Definition 5.2.1 describes how a test set optimization can be reduced to a matching problem in bipartite graphs.

Definition 5.2.1 (Test set optimization) *Let Σ denote a set of test cases, and let F denote a set of features. A bipartite graph $G = (\Sigma \cup F, E)$ is constructed from the elements of Σ and F such that there is an edge $e \in E$ between $\sigma \in \Sigma$ and $f \in F$ if the f feature is true for the σ. A matching problem on G is an optimization problem of Σ.*

An iteration cycle can be constructed based on that matching problem and a break condition. For instance, F can be the set of abilities to detect faulty systems derived from the specification m using a specific fault model (see Section 3.3.1), or other matching problems, i.e. selection criteria, can be given. The *stop* is a break condition for the cycle

that can be for example reaching a certain coverage level over F in the matching problem, iteration count ($k = N$), or exceeding a time limit.

In the algorithms below $\mathbf{\Phi} = [\phi_{ij}]$ is a matrix of integer values that represents the matching problem, where row i represents test case $\sigma_i \in \Sigma$ and column j represents feature $f_j \in F$. In the rest of the chapter, let $X[k]$ denote the value of variable X in the kth iteration cycle.

In Algorithm 5.1 selectivity is achieved by not adding test cases considered to be redundant according to a given metric. The inputs are a specification machine, a feature set and a stop condition. The latter is used to break the iteration cycle. The output is a set of test cases. The operation of the algorithm is as follows. The fault based (see Chapter 3) or string edit distance based (see Chapter 4) matching problem is determined in each cycle for the actual suite, and it is expressed with matrix $\mathbf{\Phi}$. The cycle has three steps, which are repeated while the stop condition is false. First in Step 8 either a new test suite is generated based on the one of the previous cycle (see Section 5.3.1) or a new test case is added to the current set. Then the matching problem is evaluated for this new suite (Step 13). Finally in Step 14 the metrics for the actual and new test suite are compared (see Section 5.3). If the new test suite is found “better” according to the given metrics based on the feature set F, it is kept as the actual. If the old test suite is better, the conditions remain the same for the next cycle.

Test selections problems have been shown to be NP-hard, hence it takes significantly less time to run the optimization for small test suites, rather than to optimize a complete test suite. Algorithm 5.2 uses this observation to generate a test suite that has already been optimized according to a given metric, making further test selection unnecessary.

The input parameters are the same as the case of Algorithm 5.1, the output of the procedure is likewise a test suite. After the initialization, the iteration cycle is executed until the stop condition is false. In the beginning of each cycle either a new test case is derived or a subset of the test cases of the previous test set is modified. In the cycle, the optimum is computed not only from the new suite, but also from the union of the old and new.

To drop the redundant test cases, that is, to select an optimal test suite, this chapter uses the bacterial evolutionary algorithm discussed in [Vin02]. This is referred as the optimization and reduction of a test suite according to the $\mathbf{\Phi}$ matrix of the matching problem.

5.3 Iterative test generation with fault based test suite metrics

Figure 5.1 shows the configuration of the fault based iterative test generation procedure. Mutant systems are created from the system specification using a given fault model. The test environment of Section 3.2.2 has been extended with the iteration cycle detailed in the sections below.

The search algorithms in Section 2.4.3 generate traces that are – sooner or later – going to fail against the specification they are derived from. The next function (fails_after) returns the number of events, after which test case σ is observed to fail against the given system m.

Algorithm 5.1: Iterative test derivation

```
   input: m, F, stop
   output: Σ
   /* the actual test suite in cycle k, a matrix of integer values in cycle
      k */
 1 data(Σ, Φ);
   /* Initialization */
 2 k := 0;
 3 Σ[0] := ∅;
 4 Φ[0] := 0
   /* The k^th iteration cycle: */
 5 repeat
 6     switch random do
 7         case modify existing cases
 8             foreach σ ∈ Σ[k − 1] do σ := derive_inc(m, σ);
 9         case generate a new case
10             σ := derive(m);
11             Σ[k] := Σ[k − 1] ∪ {σ};
12     endsw
13     Compute Φ[k] based on the actual Σ[k] and F ∪ {m};
                  (F)
14     if Φ[k]  >  Φ[k − 1]  then
15         Σ := Σ[k];
16     endif
17 until stop = false ;
```

Algorithm 5.2: Iterative test generation with immediate optimization

```
   input: m, F, stop
   output: σ
   /* the actual test suite in cycle k, a matrix of integer values in cycle
      k, a matrix of boolean values in cycle k */
1  data(Σ,Φ, C);
   /* Initialization */
2  k := 0  Σ[0] := ∅;
3  Φ[0] := 0;
4  C[0] := 0;
   /* The k^th iteration cycle:  */
5  repeat
6      switch random do
7          case modify existing cases
8              foreach σ ∈ Σ[k − 1] do σ := derive_inc(m, σ);
9          case generate a new case
10             σ := derive(m);
11             Σ[k] := Σ[k − 1] ∪ {σ};
12     endsw
13     Compute Φ[k] based on the actual Σ[k] and F ∪ {m};
14     Compute C[k] from Φ[k];
15     Let Σ [k] := reduce (append (Σ [k − 1], Σ [k]), select (append(C [k − 1], C [k])));
16     Let Φ[k] := reduce(append(Φ[k − 1], Φ[k]), select(append(C[k − 1], C[k])));
17 until stop = false ;
```

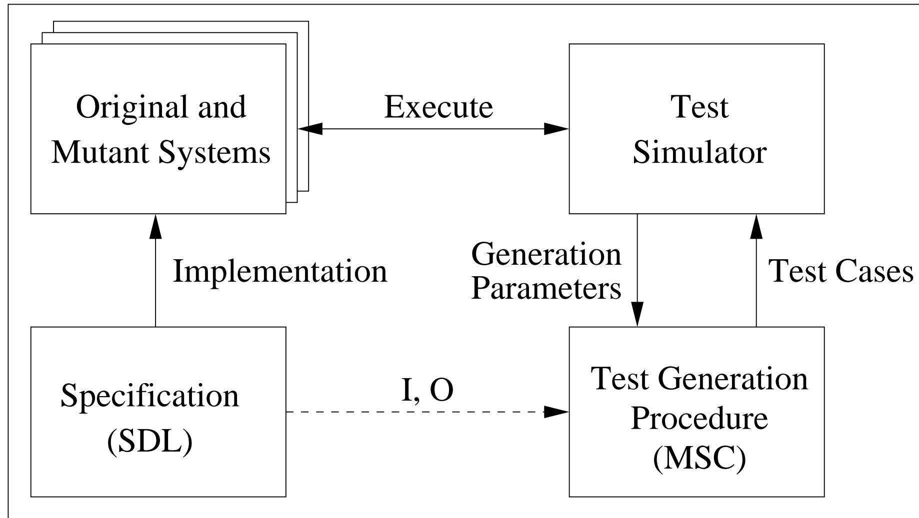


Figure 5.1: Functional architecture for fault based iterative test generation

If σ does not fail, then its length is returned.

Definition 5.3.1 (Fail verdict) *Let* $\mathsf{fails_after} : \Sigma \times M \rightarrow \mathbb{N}$ *be the* $\mathsf{fails_after}$ *function. Let* $\forall \sigma \in \Sigma, m \in M : \mathsf{fails_after}(\sigma, m) = l \iff \mathsf{obs}(\mathsf{substring}(\sigma, l), m) = \mathsf{fail}$, $\mathsf{obs}(\mathsf{substring}(\sigma, l-1), m) = \mathsf{pass}$, *where* $l \leq length(\sigma)$, $l \in \mathbb{N}$. *Of course,* $\mathsf{obs}(\sigma, f) = \mathsf{pass} \iff \mathsf{fails_after}(\sigma, m) = length(\sigma)$.

A test selection metric can be acquired by comparing the return values of the fails_after function for a test case function when that is symbolically executed against the specification and a mutant system. Therefore these return values are stored in a failure matrix (Definition 5.3.2), where rows represent the test cases and columns the test selection criteria, in this case the faults.

Definition 5.3.2 (Failure matrix) *Let* $\mathbf{\Phi}$ *be a matrix of* $|\Sigma|$ *rows and* $|F|+1$ *columns containing integer values, where* $\mathbf{\Phi}_{ij} := \mathsf{fails_after}\,(\sigma_i, f_j), \forall i, j \in \mathbb{N}, 1 \leq i \leq |\Sigma|, 0 \leq j \leq |F|$. *Let the* 0th *column represent the specification* m.

This means that if the test case σ_i fails against the mutant f_j where it is expected to fail according to the specification m, then it does not detect the fault f_j. Otherwise σ_i is an adequate counterexample that indicates the absence of refusal preorder relation between m and the machine with the fault f_j. The feature matrix $\mathbf{C}$ stores this information, and is automatically computed from $\mathbf{\Phi}$.

Definition 5.3.3 (Feature matrix) *Let* $\mathbf{C}$ *be a matrix of* $|\Sigma|$ *rows and* $|F|$ *columns containing boolean values (0 or 1), where* $\forall i, j \in \mathbb{N}, 1 \leq i \leq |\Sigma|, 1 \leq j \leq |F|$:

$$\mathbf{C}_{ij} = \begin{cases} 0 \iff \mathbf{\Phi}_{i0} = \mathbf{\Phi}_{ij} \iff \mathsf{fails_after}(\sigma_i, m) = \mathsf{fails_after}(\sigma_i, f_j). \\ 1 \iff \mathbf{\Phi}_{i0} \neq \mathbf{\Phi}_{ij} \iff \mathsf{fails_after}(\sigma_i, m) \neq \mathsf{fails_after}(\sigma_i, f_j). \end{cases}$$

To compare two test suites fitness functions are evaluated for test suites. Fault based testing provides the heuristic fitness functions applied in this section.

The evaluation criteria are provided as the four-level test suite fitness vector, which is based on a fault set derived from the specification by means of mutation analysis. It is important to note, however, that evaluation criteria may have also different bases according to the decision of the tester.

Definition 5.3.4 (Fault based test set fitness functions) *To evaluate a test suite the fault based test selection matrices* $\mathbf{\Phi}$ *and* $\mathbf{C}$ *are used. The next vector gives fitness functions for evaluating the test suite* Σ*:*

1. *The number of faults detected by the* Σ *test set is a number that can be used to evaluate a test set. Let* $cr_1 : \mathbf{C} \rightarrow \mathbb{N}$ *be a fitness function, where* $\mathbf{C}$ *is a matrix of boolean values:*

$$cr_1(\mathbf{C}_\Sigma) = \sum_{j=1}^{|\Sigma|} \mathsf{sgn}(\sum_{i=1}^{|F|} c_{\Sigma ij}),$$

   where $\mathsf{sgn}(x)$ *is the signum function:* $x < 0 \Rightarrow \mathsf{sgn}(x) = -1, \mathsf{sgn}(0) = 0, x > 0 \Rightarrow \mathsf{sgn}(x) = 1$.

2. *Let the fitness function* $cr_2 : \mathbf{\Sigma} \rightarrow \mathbb{R}$ *be a fitness function that calculates the the average length of the test cases in* Σ*:*

$$cr_2(\mathbf{\Phi}_\Sigma) = \frac{\sum_{i=0}^{|F|} \sum_{j=1}^{|\Sigma|} \phi_{\Sigma ij}}{|\Sigma|}$$

3. *Let* $cr_3 : \mathbf{\Sigma} \rightarrow \mathbb{R}$ *be a fitness function for the test set* Σ *such that:* $cr_3(\Sigma) = |\Sigma|$.

4. *Let* $cr_4 : \mathbf{C}_\Sigma \rightarrow \mathbb{R}$ *be a fitness function defined by the uniformity of the fault detection capability of* Σ*:*

$$cr_4(\mathbf{C}_\Sigma) = \frac{\sum_{j=1}^{|\Sigma|} (\sum_{i=1}^{|F|} \mathbf{c}_{\Sigma ij})^2}{|\Sigma|}$$

Function 1 calculates the number of faults which can be detected by the test suite. It is the number of columns in $\mathbf{C}$ which contain at least one true (1) value. The larger this value is, the better the test suite is.

Function 2 checks the average length of the test cases in a test suite. It is calculated from the average of all values in the matrix $\mathbf{\Phi}$. Longer test cases may exercise the implementation better.

The third level (Function 3) uses the size of the test suites. The execution of a test suite containing fewer test cases takes presumably less time.

Finally, Function 4 checks the distribution of column sums in $\mathbf{C}$. The closer it is to a uniform distribution, the small the calculated value is. So, a smaller value is considered to be better. The rationale of this fitness condition is to preserve the faults already detected for future cycles. If a test case is removed (replaced by a new test case of a future cycle), the test suite is still likely to detect the faults covered by it, if the distribution of the true values in the $\mathbf{C}$ matrix is close to a uniform distribution.

The fitness functions are compared level-by-level with the first level as strongest. If the relation between the two test sets can not be decided at a level, the next level is considered.

Function 1

C1	f1	f2	f3	f4
t1	0	1	0	0
t2	0	0	1	0
t3	0	0	0	1

C2	f1	f2	f3	f4
t1	0	1	0	0
t2	0	0	1	1
t3	1	0	0	0

Function 2

D1	f1	f2	f3	f4
t1	7	13	11	5
t2	7	13	10	5
t3	7	13	8	5
t4	7	16	8	5

D2	f1	f2	f3	f4
t1	9	13	11	5
t2	9	13	10	3
t3	9	13	8	5
t4	9	19	8	5

Function 3

C1	f1	f2	f3	f4
t1	0	1	1	0
t2	0	0	1	1
t3	1	0	0	0

C2	f1	f2	f3	f4
t1	0	0	1	0
t2	0	0	0	1
t3	1	0	0	0
t4	0	1	0	0

Function 4

C1	f1	f2	f3	f4
t1	1	1	0	0
t2	1	0	1	0
t3	1	0	0	1

C2	f1	f2	f3	f4
t1	1	1	0	0
t2	0	1	1	0
t3	1	0	0	1

Figure 5.2: Examples for the test suite fitness evaluation

Figure 5.2 shows examples of comparing two matrices. The rows of the matrices in the figure represent the test cases of the test suite, and the columns represent the faults. In the first example where the $\mathbf{C}$ matrix of Σ_1 and Σ_2 is investigated, the second matrix is considered better, because it indicates that the second test suite detects all faults, while the first only three. In the second example, according to Criterion 2 the $\mathbf{\Phi}$ matrix of the second test suite is considered better as it contains test cases of more events. In the example for

Criterion 3, the first **C** matrix is considered better as it consists of fewer test cases. The distribution of column sums (Criterion 4) in the second **C** matrix of the last example is closer to the uniform distribution, therefore it is considered better.

5.3.1 Updating a test set for the next cycle

In this section, the incremental development of test cases of a test suite is discussed, which can be an implementation of Definition 2.4.2. There are two ways to modify a test suite. One is to add new test cases to it, the other is to replace an existing case with a new one. The modification of a test suite takes three parameters: the maximum number of test cases, the number of test cases modified or added in a cycle, and the test suite to be modified itself. While the size of the test suite is less than the maximum, new cases may be derived and added. Otherwise existing cases are modified.

The test cases to be modified are those that have the smallest values in the failure matrix $\mathbf{\Phi}$, because longer test cases may exercise the system under test better.

Let test case $\sigma \in \Sigma$ be split up into two parts $\sigma = \alpha\beta$ such that $\alpha :=$ substring $(\sigma, \mathsf{random}(0, \mathsf{fails_after}(\sigma, m)))$, where m is the specification machine. Derive a new postfix (β') and let $\sigma' := \alpha\beta'$.

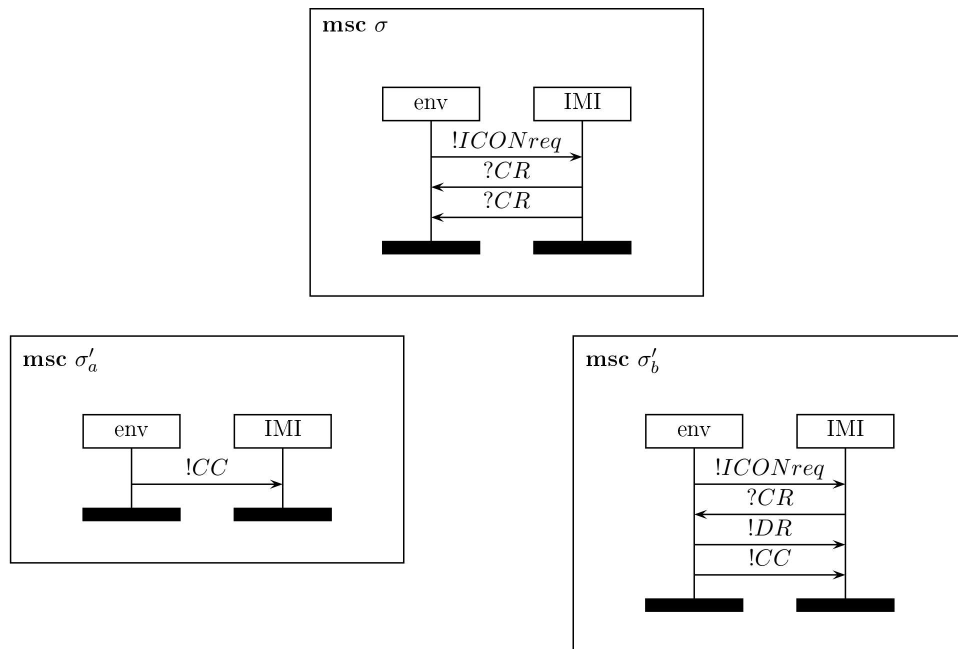


Figure 5.3: Generating a new test case or a new postfix to a test case. Test case σ is the original test case to be modified. Test case σ'_a is a new test case. Test case σ'_b is the modification of σ such that events of σ are removed after the $\mathsf{fails_after}(\sigma, m)$ value (i.e. from the third event), and a new postfix is added.

Figure 5.3 shows two examples of the modification of test case σ represented in an MSC.

In the first case, bottom left MSC in the figure, the postfix of the test case is replaced. A random number of events between 0 and the fails_after(σ, m) value are retained in the test case σ'_a, and a new postfix is appended to it. The value 0 means that a completely new test case is derived – just like in the figure. In the second case the postfix in the test case σ'_b has been replaced from the last successfully received output before which the test case σ was observed to fail. To generate the new postfix Algorithm 2.2 is used.

5.3.2 Comparison of the algorithms

This section presents an experiment analyzing the algorithms described above. The main purpose is to find out how the proposed methods perform – considering different aspects – compared to other test derivation algorithms including the selection procedure. The main aspects of the examination are the execution time, fault detection capability and the number of derived test cases. Since the proposed methods include heuristic elements, none of these key properties can be exactly calculated or formulated, and are thus analyzed empirically. The methods were tested on a couple of specifications of different complexity: the experiment was on both sample and real life protocol specifications.

In the experiment five algorithms were compared: the random trace test derivation algorithm (Algorithm 2.1), random event sequences (Algorithm 2.2), the proposed iterative test generation method (Algorithm 5.1), its enhancement with the immediate test selection (Algorithm 5.2) and its extension with a tree map that stores all postfixes generated so far (referred as iterative with memory) that does not allow the same postfix to be regenerated. In this experiment, all iterative algorithms used the random event sequence method to derive and modify test cases. The generation of all possible sequences up to a given length (Algorithm 2.3) was found unsuitable even in the case of the most simple protocol as it creates an enormous number of test cases using the given length values.

The systems examined in the experiments are: the INRES[EHS97], the Conference Protocol [BFV+99], the real-life protocol standards: the WAP WTP [For98] and the SS7 MTP level 2 [IT97a]. The experiment on the MTP2 system is again a limited one; the same considerations and configurations were applied in this section as in Section 3.3.6.

Slightly modified modules of the mutation analysis based test selector tool [KLVWHC+03] provided the environment for the experiment. For the test suite optimization, the library of a bacterial evolutionary algorithm [Vin02] was used. The results were obtained using a PC with Intel P4 3GHz processor, 2GB RAM and Linux operating system.

As a first step of the experiment, the same fault model was applied to the different protocol specifications to create mutants. The number of mutant specifications generated was highly dependent on the complexity of the systems. The first row of Table 5.1 contains their number. For all the algorithms the maximum length, i.e. the maximum number of events, of the test cases was defined as a parameter for test generation. The second row of the table shows the appropriate length values. For each algorithm the maximum number of test cases to be generated was defined. The third shows the number of test cases derived for the different protocols.

Table 5.1: The number of mutant systems and parameters for test generation

	Conf. Prot.	INRES	WTP	MTP2
Mutants	117	126	418	720
Test case length	10	10	14	14
Max. number of test cases	300	300	600	2000

Note that the natural stop condition of the iterative algorithms is reaching 100% mutation detection. An additional stop condition was applied in the experiment: the total number of test cases generated during the iteration was limited to the same value used for the random trace and random event sequence algorithms. Besides, the maximum length of test cases was limited to the same value as well. This made a fair comparison of the algorithms possible.

The next three tables (Tables 5.2, 5.3 and 5.4) present the key results of the experiment. Note that in the case of the random trace algorithm and the random event sequence, the tables contain the values of the optimized test suite. In the case of the first three protocols, the results of the iterative methods are the mean of five independent runs, and for the method using random event sequences the best result of five runs is shown. The results for the experiment on MTP2 with the iterative method extended with a hash map if not available because the memory dedicated for the process was not sufficient to complete the process

Table 5.2: The number of test cases in the optimized suite

Method	Conf. Prot.	INRES	WTP	MTP2
Random trace	9	12	34	51
Random event sequence	11	12	38	48
Iterative	10	17	46	64
Enhanced Iterative	9	12	34	52
Iterative with memory	9	10	32	n/a

Table 5.2 shows the resulting number of generated test cases after the test selection. As the data in the table indicate the resulting optimized test suites are nearly of the same size for the different test generation techniques.

Table 5.3: Fault detection ratio [%]

Method	Conf. Prot.	INRES	WTP	MTP2
Random trace	100	100	60	15
Random event sequence	80	86	38	11
Iterative	100	92	48	16
Enhanced Iterative	100	100	51	17
Iterative with memory	100	100	52	n/a

The fault detection ratio of the optimized suite is shown in Table 5.3 In the case of small systems, a high fault coverage ratio can be achieved using even the simplest method with the given number of test cases. As the complexity of the specifications grows this ratio decreases. The reason for the low detection ratio in the case of the more complex systems is that the number of test cases derived was not sufficient. These test suites were capable of detecting only a part of the faults. In case of the MTP2 the detection ability was limited with the reduced input-output alphabet as well.

Table 5.4: Execution times in form of hh:mm

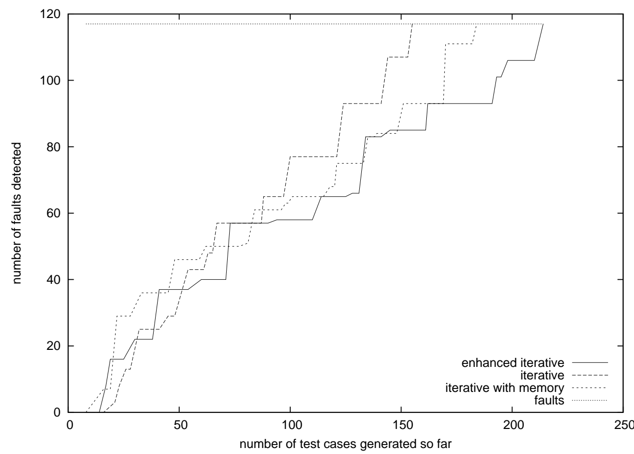
Method	Conf. Prot.	INRES	WTP	MTP
Random trace	00:58	01:02	06:48	14:30
Random event sequence	01:05	00:59	03:40	04:14
Iterative	01:15	01:12	02:49	03:31
Enhanced Iterative	01:24	01:14	03:08	04:01
Iterative with memory	01:38	01:25	03:35	n/a

Table 5.4 shows the execution times of the algorithms including the test derivation and selection in minutes. It is calculated from the execution times of the test derivation and selection processes. Since in these experiments we used the same mutant set for the fault based test selection, the mutant generation time is not taken into account. Significant deviation can be observed between the simple random trace method and the other techniques when they are applied to larger protocols. The reason for this deviation is that the input and

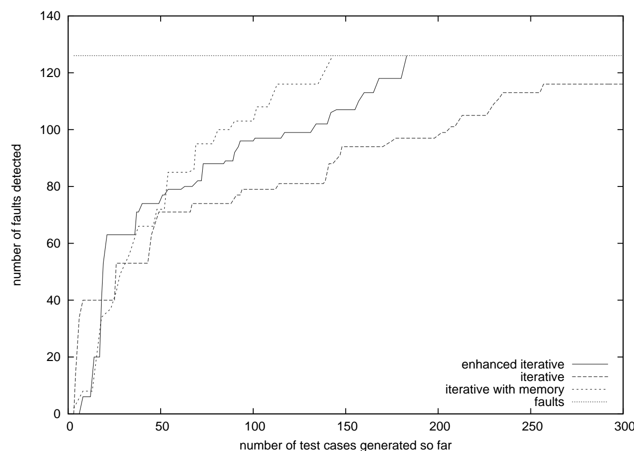
output sets for those protocols are larger. Therefore, the random sequence based methods generate inappropriate test cases more frequently. These tests are likely to fail early and thus provide only a few selection criteria (few true values in the **C** matrix) thus the selection requires less time.

The iterative test generation algorithm produced an optimized test suite nearly as fast as the method using random event sequences. On the other hand, their fault detection capability was observed to be better. This result was in line with our expectations because in the case of iterative algorithms each iteration cycle may improve the previous random sequence. That is, there is correlation among the generated random sequences.

The fault detection ratio of the random trace method was similar to the random sequence based iterative algorithms. Though the iterative algorithms required more time for smaller protocols, they were found less sensitive for the increase in system complexity.

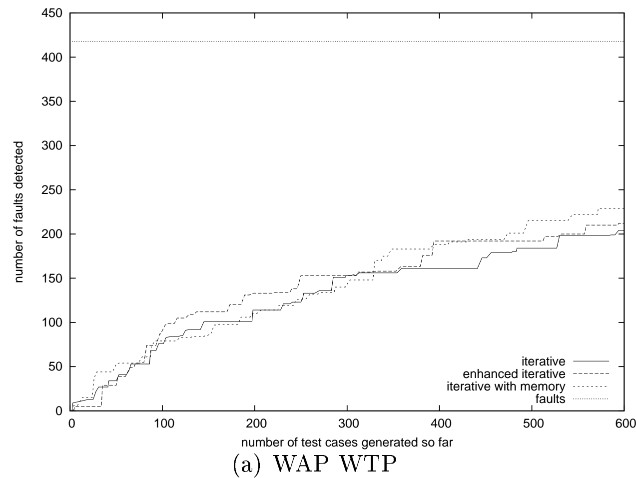


(a) Conference Protocol

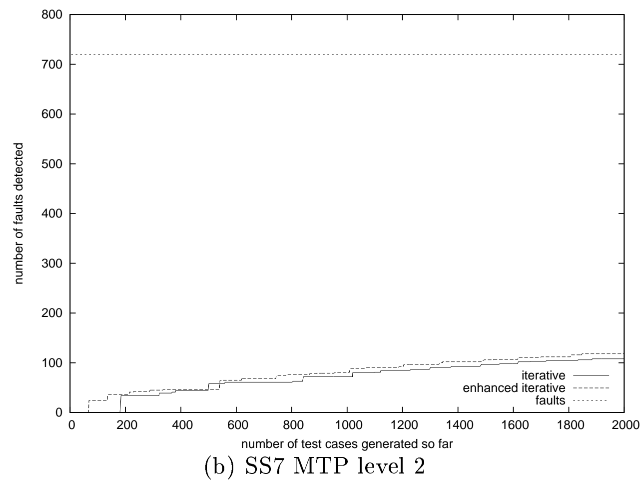


(b) Inres Initiator

Figure 5.4: Iterative fault detection experiments on sample systems



(a) WAP WTP



(b) SS7 MTP level 2

Figure 5.5: Iterative fault detection experiments on real-life systems

Four charts (Figures 5.4(a), 5.4(b), 5.5(a), 5.5(b) show the number of detected faults against the total number of generated test cases in the case of the iterative algorithms for the investigated protocols. (Note that due to their large number, the samples are represented with continuous lines.) The horizontal axis of each figure shows "the number of test cases generated so far", the vertical axis shows "the number of faults detected". Solid, dashed and dotted lines represent the data sets for the methods iterative, enhanced iterative, and the enhanced iterative respectively. The horizontal line shows the number of faults.

The charts show a step function-like fault detection progress. Due to the randomly generated test sequences, jumps in the fault detection ratio and several iteration long steps can be observed. In the case of the enhanced iterative method a more strict iteration condition provided faster convergence to the detection of all the mutants. The reason is that while the iterative method only aims that the new suite detects more faults, in case

of the enhanced iterative method "good" test cases cannot be dropped from the best suite, because it computes the best from the union of the previous and actual suites. Its further extension with memory resulted in a slight improvement at the cost of higher hardware requirements. As the iteration progresses the speed of the convergence is found to get slower. On the other hand a slower convergence can be observed in the case of the more complex systems as well.

5.4 Iterative test generation with string edit distance based metrics

This section concretizes Algorithm 5.2 for the string edit distance based metric such that the ε density of the test set is preserved throughout the iterative process. This method is effective for quick merging of test sets.

We generalize the distance function to evaluate the effect of merging a new trace into an already existing trace set.

Definition 5.4.1 (Generalized distance function) *Let* $d : (\sigma', \Sigma) \rightarrow \mathbb{R}$ *compute the distance between the trace* σ' *and the trace set* Σ *such that* $d(\sigma', \Sigma) = \min_i d(\sigma', \sigma_i)$*, where* $\sigma_i \in \Sigma$*.*

The inputs of the method are a ε selection threshold, a K iteration limit as stop condition, which is also an upper bound for the number of generated test cases. The output is a reduced test set. The algorithm can co-work with any test generation algorithm [SKGH97, TB03, JJ02, HFT00] from Section 2.4.3. In each iteration cycle, a new test case σ' is derived. That test case is added to the test set immediately if its distance from every element of the test set is greater than the given threshold. Otherwise the union of the newly generated test case and the old test set is optimized with the method of 4.2 to maintain the given ε density of the test set.

Example In the example let the iteration limit be $K = 8$, and let MSC traces from Figure 4.3 be generated iteration by iteration. In general the iteration limit, the length and the number of sequences are independent; setting these configuration parameters to 8 suits the simplicity of this example.

Two cases are investigated, the $\varepsilon = 5$ and the $\varepsilon = 6$ case. The string edit operations insert, delete and replace are assumed to have unit cost. The iterative test generation procedure is presented only for the first case. The distance matrix for the test cases of this example can be found in Section 4.1.3, and the coverage matrix can be seen Figure 4.4.

Let us first investigate the $\varepsilon = 5$ case. This $\mathbf{A}_{\varepsilon=\mathbf{5}}$ matrix implies that the resulting set consists of four strings.

Step 1 The first string is $\sigma' = dbafacfd$, so $\Sigma_{\varepsilon=5}[1] = \{dbafacfd\}$. Since this is the only element of Σ in step 1, $\mathbf{D}_{\varepsilon=5}[1] = 0$ and $\mathbf{A}_{\varepsilon=5}[1] = 0$. This sequence traverses 6 of the 13 transitions of the FSM in Figure 4.2(b).

Step 2 In this step the string $\sigma' = fhdhbehf$ is added to $\Sigma_{\varepsilon=5}[1]$, thus $\Sigma_{\varepsilon=5}[2] = \{dbafacfd, fhdhbehf\}$ and

Algorithm 5.3: String edit distance based selective automatic test generation

input: ε; K iteration limit; m specification
output: σ set

1 $\sigma[0] := \emptyset$;
2 $k := 1$;
3 **repeat**
4 $\quad\sigma' :=$ derive(m);
5 $\quad$**if** $d(\sigma', \Sigma) < \varepsilon$ **then** Find the minimum cardinality maximum internal distance ε-cover $\Sigma[k]$ of $\Sigma[k-1] \cup \{\sigma'\}$;
6 $\quad\Sigma[k] :=$ reduce$(\Sigma[k-1] \cup \sigma')$;
7 $\quad$**else** $\Sigma[k] := \Sigma[k-1] \cup \sigma'$ $k := k+1$;
8 **until** $k > K$;
9 **return** $\Sigma[k]$

$$\mathbf{D}_{\varepsilon=5}[2] = \begin{bmatrix} 0 & 8 \\ 8 & 0 \end{bmatrix} \quad \mathbf{A}_{\varepsilon=5}[2] = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

The second sequence traverses also 6 transitions, and the two sequences together traverse 9 different transitions.

Step 3 The new string added to Σ is $\sigma' = beghbfff$. This new string traverses 5 different transitions and provides one new, not yet traversed, transition for $\Sigma_{\varepsilon=5}[2]$. However $d(\sigma_2[2], \sigma') = 5$, that is, these two sequences ε-cover each other, therefore one of them is dropped despite providing a new transition.

$$\mathbf{D}_{\varepsilon=5}[3] = \begin{bmatrix} 0 & 8 & 7 \\ 8 & 0 & 5 \\ 7 & 5 & 0 \end{bmatrix} \quad \mathbf{A}_{\varepsilon=5}[3] = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & \boxed{1} & 0 \end{bmatrix}$$

As the $d(\sigma_1[2], \sigma_2[2]) = 8 > d(\sigma_1[2], \sigma') = 7$, $\sigma_1[2]$ and $\sigma_2[2]$ are kept.

Step 4 The situation is the same as in step 3. The new string $\sigma' = dhbchfdf$ traverses 5 transitions of which none is new for $\Sigma_{\varepsilon=5}[3]$.

$$\mathbf{D}_{\varepsilon=5}[4] = \begin{bmatrix} 0 & 8 & 6 \\ 8 & 0 & 5 \\ 6 & 5 & 0 \end{bmatrix} \quad \mathbf{A}_{\varepsilon=5}[4] = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & \boxed{1} & 0 \end{bmatrix}$$

As the $d(\sigma_1[3], \sigma_2[3]) = 8 > d(\sigma_1[3], \sigma') = 6$, $\sigma_1[3]$ and $\sigma_2[3]$ are kept.

Step 5 In this step $\sigma' = ddbcdfff$ is added to the sequence set that contains $\Sigma_{\varepsilon=5}[5] = \{dbafacfd, fhdhbehf, ddbcdfff\}$. This sequence traverses only 4 different transitions which are already traversed by $\Sigma_{\varepsilon=5}[4] = \Sigma_{\varepsilon=5}[2]$, thus a redundant sequence is added to the set.

Step 6 The sixth string to be added is $\sigma' = hdbafchd$. However $d(\sigma_1[5], \sigma') = 3$, therefore one of them (σ_1 or σ') is considered to be redundant:

$$\mathbf{D}_{\varepsilon=5}[6] = \begin{bmatrix} 0 & 8 & 6 & 3 \\ 8 & 0 & 7 & 6 \\ 6 & 7 & 0 & 6 \\ 3 & 6 & 6 & 0 \end{bmatrix} \quad \mathbf{A}_{\varepsilon=5}[6] = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \boxed{1} & 0 & 0 & 0 \end{bmatrix}$$

Because $d(\sigma_1[5], \sigma_2[5]) + d(\sigma_1[5], \sigma_3[5]) = 14 > d(\sigma', \sigma_2[5]) + d(\sigma', \sigma_3[5]) = 12$, $\sigma_1[5]$ is kept and σ' is dropped.

Step 7 Though the sequence $\sigma' = bachdfhb$ traversing 6 different transitions is completely redundant for $\Sigma_{\varepsilon=5}[6]$, it is added to the set, because $\mathbf{A}_{\varepsilon=5}[7] = 0$. This σ' also does not increase the number of traversed transitions.

Step 8 The last sequence $\sigma' = bhbfeggg$ is also added to $\Sigma_{\varepsilon=5}[8]$ which is now $\Sigma_{\varepsilon=5}[8] = \{dbafacfd, fhdhbehf, ddbcdfff, bachdfhb, bhbfeggg\}$. This last sequence traverses two more transitions so $\Sigma_{\varepsilon=5}[8]$ traverses the 11 of the 13. This resulting set contains five sequences, one more than would have been necessary according to $\mathbf{A}_{\varepsilon=5}$. The final distance matrix is:

$$\mathbf{D}_{\varepsilon=5}[8] = \begin{bmatrix} 0 & 8 & 6 & 6 & 7 \\ 8 & 0 & 7 & 6 & 7 \\ 6 & 7 & 0 & 6 & 7 \\ 6 & 6 & 6 & 0 & 7 \\ 7 & 7 & 7 & 7 & 0 \end{bmatrix}$$

When setting now $\varepsilon = 6$ instead of 5, the resulting set is further reduced. According to the $\mathbf{A}_{\varepsilon=6}[8]$ below, two more traces can be removed: $\sigma_3[8]$ and $\sigma_4[8]$. (Note that in the $\varepsilon = 5$ case these two cases were named previously as redundant.) The remaining three traces $\Sigma_{\varepsilon=6}[8] = \{dbafacfd, fhdhbehf, bhbfeggg\}$ still traverse 11 of the 13 transitions.

$$\mathbf{A}_{\varepsilon=6}[8] = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & \boxed{1} & 0 & 1 & 0 \\ \boxed{1} & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

When using $\varepsilon = 6$ from the beginning of the iteration cycle, the final trace set is $\Sigma_{\varepsilon=6} = \{dbafacfd, fhdhbehf, bhbfeggg\}$. Without the iterative cycle the resulting set would be $\Sigma_{\varepsilon=6} = \{dbafacfd, fhdhbehf\}$ with 9 transition traversed.

Figure 5.6 shows how the transition coverage changes during the iteration in case $\varepsilon = 5$. The transition coverage is the same for both the $\varepsilon = 5$ and the $\varepsilon = 6$ cases. In step 3 one new transition is discovered, but the trace is dropped due to redundancy. Two new transitions are found in step 8 including the one dropped in step 3.

The total number of distance calculations is 20 in the $\varepsilon = 5$ case and 13 in the $\varepsilon = 6$ case. Without the iterative cycle it would be 28, hence matrix $\mathbf{A}$ is provided with less calculations. The gain depends on the relation between trace lengths and the total number of traces generated. The iterative method performs better with longer traces, while the single stage method is better if the number of traces is large.

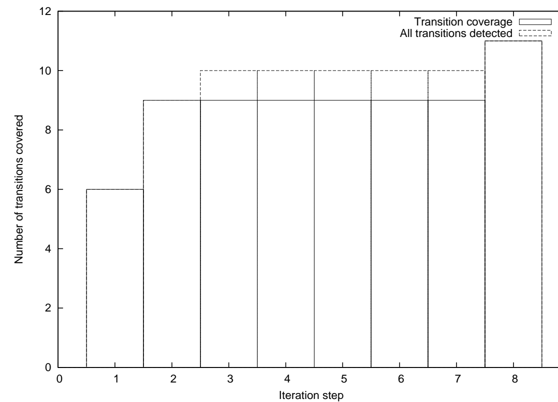


Figure 5.6: Transition coverage during the iterations

5.5 Conclusion

This chapter investigated the automatic generation of test sets from the methodological point of view: test specification was considered to be an iterative process instead of a single stage activity. The aim of this extension is to produce compact test suites automatically with high coverage and reduced generation time by operating with smaller test suites throughout the iteration cycle. It is important, because both evaluating features for test selection and optimizing the test set with that feature set can take significant amounts of time for large test suites.

The two abstract iterative algorithms were concretized with fault based and string edit distance based metrics. Investigations showed that they are effective in reducing the computational complexity of test selection while they provide similar coverage values to non-iterative processes.

Chapter 6

Summary

The aim of the thesis was to give automatic methods to generate compact sets of black box abstract test cases while maintaining their ability to preserve a high level of confidence in conformance with the input specification given in an EFSM model. That is, in the thesis methods were given that keep the size of the test suite within limits during generation time while they provide high level of coverage.

The motivation and background were discussed in Chapter 1. Chapter 2 gave an overview of model based test generation by interpreting the *Formal Framework for Conformance Testing* in the FSM/EFSM world. The chapter also introduced test generation algorithms used in available tools. In Chapter 3 our mutation analysis based test generation and selection approach was presented. Chapter 4 extended the string edit distance based methodology with an algorithm that provides that maximal possible test set compression. Methodological aspects of Chapter 5 extended the test specification to an iterative process to reduce the total computation complexity and to support efficient test set maintenance.

In the following the summary of the main contributions of this dissertation is given.

6.1 Mutation based automatic test generation for SDL specifications

In Chapter 3, we concentrated on the fault-based fully automatic test generation for SDL systems. The main results of the chapter can be summarized as follows:

- We designed a formal framework for automatic test generation for models represented as an EFSM or, more specifically, in SDL. The framework consist of an automatic MSC test generation algorithm, an automatic test selection algorithm, an execution and evaluation model, and a formal fault model for EFSM containing automatically faults. Those can be applied to SDL specifications automatically. The framework injects faults into an SDL specification to create mutant specifications. Then, the evaluation of the refusal preorder implementation relation between the original and the mutant specifications can be used to reduce the size of an abstract conformance

test set. We implemented this approach in a Test Selector Tool and performed studies on sample protocols.

- We presented an automatic test generation algorithm that is based on the fault model and the refusal preorder implementation relation. When using this approach the size of the test set generated automatically by any search strategy is limited to the polynomial function of the size of the set of operators. We studied the fault detection capability of this method on sample protocols and on WAP WTP.

The proposed method method has several advantages over the theoretical solutions implemented in commercial software tools. Those tools support the use of exhaustive test set generation methods which produce test sets with quasi-infinite execution cost, and random based test generation method which lack the capability of exercising the whole state space. My approach has an upper bound on the number of test cases generated automatically. This upper limit reduces the execution cost of the generated test set. Experiments show that my approach has a higher fault detection ability than the random search based solution.

The weakness of my method is that these advantages are achieved at the cost of a higher computation demand. My method does not give a solution to the dependence on size of the input/output alphabet which is a common problem for all automatic test generation approaches.

The software tool we developed can be used in conjunction with commercial tools or separately to generate reduced test suites automatically for an input SDL specification and a fault model.

The results presented in Chapter 3 were published in [C5, C6, J2, C2]. There are four independent [1, 2, 3, 4] and one dependent [5] citations of these papers.

6.2 String edit distance based test set optimization

In Chapter 4, a solution was proposed that improves the string edit distance based test selection method introduced by Vuong et al. and extended with some heuristics by Feijs et al. While these metrics provide an excellent basis to find out if a test set is redundant, the pairwise comparison does not support the optimal test set reduction.

The main contributions of this chapter are:

- We proposed an algorithm that can determine the minimal size of a test set with respect to a density parameter and a given event to string mapping. The Algorithm 4.1 reduces the problem of finding the minimum size, i.e. the maximum compression, to a matching problem in bipartite graphs. We showed the complexity of that algorithm is proportional to the third power of the size of the input test set when the distance matrix has already been computed.

- We introduced a string edit distance based metric that can be used for evaluating test sets.

- We showed that finding the most diverse subset of the input test set for a given density parameter and a computed subset size can be done in polynomial time. We showed that this problem is equivalent to the k-assignment problem.
- We have carried out comprehensive simulation based analysis of different test selection approaches including our fault based and string edit distance based solutions. Our methods can achieve a similar reduction in the size of the test set as the transition coverage method.

The advantage of my extension to the string edit distance based methodology is that we can now compute the maximum reduction of a input test set in polynomial time of the size of that input with regard to a density parameter. Other theoretical test set reduction solutions have been shown to be inherently NP-hard.

A general weakness of the string edit distance based approach is that it may remove an important test case just because that is found similar to another test case (their distance is lower than the density threshold).

This is a theoretical solution that requires further research before it can make its way to practical application. Research in two areas could improve the applicability of this method. An open research field is the test case to string mapping. Vuong and Alilovic use the test event to character mapping, Feijs et al. propose the marked trace notation, and this thesis uses a transition to character mapping. There is no study that aims to find the most efficient way of mapping with special respect to the input parameters. In practice, test cases are usually represented as trees and not as a set of traces. Another research topic is the generalization of edit distance based reduction method to trees.

The results presented in Chapter 4 were published in [J3, C12, C4].

6.3 Iterative test specification

In chapter 5 we proposed a method that investigates automatic test generation from the methodological point of view.

The main result are the following:

- We proposed a methodological extension to formal model-based system development where the test specification process is composed of an iterative cycle containing two activities: automatic test generation and automatic test selection.
- We designed two abstract iterative algorithms built on the concept of evolutionary algorithms that can automatically generate and maintain reduced test sets and maintain with less computation than the waterfall process with separate generation and selection activities. The algorithms provide selectivity by means of test suite metrics.
- We defined fault based metrics, which are used both as iteration condition and selection criteria when the algorithms are concretized. We extended our test generation framework with this iterative approach, carried out simulations on sample protocols,

and compared the iterative selective automatic test specification process with the single stage test generation and optimization process.

- We used our string string edit distance based metrics to construct an iterative automatic test generation process and showed with an example that it can achieve the same coverage as the non-iterative version, but requires less computation.

The strengths of this solution can be summarized as follows. My approach can make use of different theoretical coverage metrics for automatically generating reduced test sets. If this approach is applied to a computation intensive coverage metric like the fault coverage metric, it can reduce the total time required for test case generation.

The weakness of this solution is that the price of the reduction in the test case generation is that the size of the resulting test set may increase which may require more time for the execution.

The results presented in Chapter 5 were published in [C8, J1, C12, C4].

Bibliography

[AB99] P.E. Ammann and P.E. Black. *A Specification-based Coverage Metric to Evaluate Test Sets.* In *Proceedings of Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE 99)*, pages 239–248, 1999.

[ABM98] P.E. Ammann, P.E. Black, and W. Majurski. *Using Model Checking to Generate Tests from Specifications.* In *Second IEEE International Conference on Formal Engineering Methods*, pages 46–54, 1998.

[AU72] A.V. Aho and J.D. Ullman. The Theory of Parsing, Translation, and Compiling, volume 1 of *Automatic Computation.* Prentice-Hall, 1972.

[BA96] E. Berk and C.S. Ananian. *JLex: A Lexical Analyzer Generator for Java.* Princeton University, `http://www.cs.princeton.edu/~appel/modern/java/JLex/`, 1996.

[BCL99] C. Besse, A. Cavalli, and D. Lee. An automatic and optimized test generation technique applying to tcp/ip protocol. In *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*, page 73, Washington, DC, USA, 1999. IEEE Computer Society.

[BDA] C. Bourhfir, R. Dssouli, and E.M. Aboulhamid. *Automatic Test Generation for EFSM-based Systems.* `http://citeseer.nj.nec.com/114451.html`.

[BFS05] A. Belinfante, L. Frantzen, and C. Schallhart. Tools for test case generation. In Broy M., Leuckner M., Jonsson B., Katoen J.-P., and Pretschner A., editors, *Model based testing of reactive systems*, volume 3472 of *LNCS*, pages 391–438. Springer, 2005.

[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 Csopaki G., Dibuz S., and Tarnay K., editors, 12th *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.

[BH89] L. Bromstrup and D. Hogrefe. Tesdl: Experience with generating test cases from sdl specifications. In *Fourth SDL Forum*, pages 267–279, 1989.

[Boe88] B. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

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

[CHK00] Gy. Csopaki, G.A. Horváth, and G. Kovács. Communication protocol implementation in java. In *Interactive Distributed Multimedia Systems and Telecommunication Services, LNCS 1905*, pages 254–265, Enschede, The Netherlands, October 2000.

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

[CK97] T. Csöndes and B. Kotnyek. *A mathematical programming method in test selection. EUROMICRO 97*, pages 8–13, 1997.

[CKS01] T. Csöndes, B. Kotnyek, and J.Z. Szabó. Application of heuristic methods for conformance test selection. *European Journal of Operational Research*, August 2001.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, second edition, 2001.

[DAM97] M. Dell' Amico and S. Martello. The k-cardinality assignment problem. *Discrete Applied Mathematics*, 76:103–121, 1997.

[DMLS78] 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.

[DÜU04] Ali Y. Duale and M. Ümit Uyar. A method enabling feasible conformance test sequence generation for efsm models. *IEEE Transactions on Computers*, 53(5):614–627, 2004.

[EHS97] J. Ellsberger, D. Hogrefe, and A. Sarma. SDL Formal Object-oriented Language for Communicating Systems. Prentice Hall, 1997.

[FGMT02] L.M.G. Feijs, N. Goga, S. Mauw, and J. Tretmans. Test selection, trace distance and heuristics. In *Testing Communication Systems XIV*, pages 267–282, Berlin, Germany, 2002.

[Fla96] F. Flannery. *CUP Parser Generator for Java.* Princeton University, `http://www.cs.princeton.edu/~appel/modern/java/CUP/`, 1996.

[FMDM94] S.C.P.F. Fabbri, J.C. Maldonado, M.E. Delamaro, and P.C. Masiero. *Mutation Analysis Testing for Finite State Machine.* In *Proc. ISSRE'94 - Fifth International Symposium on Software Reliability Engineering*, pages 220–229, California, USA, 1994.

[FMM+95] 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. FORTE'95 - 8th International IFIP Conference on Formal Description Techniques for Distributed Systems and Communications Protocol*, 1995.

[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 *Proc. ISSRE'99 - 10th International Symposium on Software Reliability Engineering*, pages 210–219, Florida, USA, 1999.

[For98] WAP Forum. Wireless application protocol architecture specification. Specification, April 1998.

[FWH97] P.G. Frankl, S.N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, Sept 1997.

[GHN93] J. Grabowski, D. Hogrefe, and R. Nahm. Test Case Generation with Test Purpose Specification by MSCs. North Holland, 1993.

[Hen64] F.C. Hennie. Fault detecting experiments for sequential circuits. In *IEEE 5th Annual Synposium on Switching Circuits Theory and Logical Design*, 1964.

[HFT00] L. Heerink, J. Feenstra, and J. Tretmans. Formal test automation: The conference protocol with phact. In H. Ural, R. L. Probert, and G. von Bochmann, editors, *13th IFIP International Conference on Testing of Communicating Systems (TestCom 2000)*, pages 211–220. Kluwer Academic, 2000.

[HGS93] M.J. Harrold, R. Gupta, and M.L. Soffa. A methodology for controlling the size of a test suite. *Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[HKN01] D. Hogrefe, B. Koch, and H. Neukirchen. Some implications of MSC, SDL and TTCN time extensions for computer-aided test generation. In *SDL Forum*, pages 168–181, 2001.

[HL05] M. Hong and Z. Lu. A framework for testing web services and its supporting tool. In *SOSE 2005: IEEE International Workshop on Service-Oriented System Engineering*, pages 199–206, Beijing, China, October 20-21 2005.

[Hol91] G.J. Holzmann. *Design and validation of computer protocols*. Prentice Hall, 1991.

[ISO89] ISO/IEC. *ISO-880: LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, 1989.

[ISO97] ISO. *ISO-9074: Estelle – A formal description technique based on an extended state transition model*, 1997.

[IT94] ITU-T. *Recommendation X.680: Abstract Syntax Notation One (ASN.1): Specification of basic notation*, 1994.

[IT95] ITU-T. *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications - General concepts*, 1995.

[IT97a] ITU-T. *Recommendation Q.0703: Specifications of Signalling System Nr. 7 – Message transfer part – Signalling link*, 1997.

[IT97b] ITU-T. *Recommendation Z.100 – Supplement 1: SDL+ methodology, Use of MSC and SDL (with ASN.1)*, 1997.

[IT00a] ITU-T. *Recommandation Z.120: Message Sequence Chart*, 2000.

[IT00b] ITU-T. *Recommendation Z.100: Specification and Description Language*, 2000.

[JHS+08] Y. Jiang, S.S. Hou, J.H. Shan, L. Zhang, and B. Xie. An approach to testing black-box components using contract-based mutation. *INTERNATIONAL JOURNAL OF SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING 18: (1) 93-117 (2008)*, 18(1):93–117, 2008.

[JJ02] C. Jard and T. Jeron. TGV: Theory, principles and algorithms. In *6th World Conference on Integrated Design and Process Technology (IDPT 2002)*, June 2002.

[KHS08] A. Kalaji, R.M. Hierons, and S. Swift. Automatic generations of test sequences from EFSM models using evolutionary algorithms. Technical report, Brunel University, `http://hdl.handle.net/2438/2630`, 2008.

[Kim08] T-H. Kim. Test generation for a protocol specified in sdl with complex loops by event-based efsm modeling. *IJCSNS International Journal of Computer Science and Network Security*, 8(3), March 2008.

[KLVWHC+03] G. Kovács, D. Le Viet, A. Wu-Hen-Chang, Z. Pap, and G. Csopaki. *Applying Mutation Analysis to SDL Specifications*. In *SDL-Forum*, Stuttgart, Germany, 2003.

[KPC01] G. Kovács, Z. Pap, and G. Csopaki. *Automatic Test Selection based on CEFSM Specifications.* In *7th Symposium on Programming Languages and Software Tools*, pages 84–97, Szeged, 2001.

[Kuh92] D.R. Kuhn. *A Technique for Analyzing the Effects of Changes in Formal Specifications. The Computer Journal*, 35(6):574–578, 1992.

[Kuh99] D.R. Kuhn. *Fault Classes and Error Detection in Specification Based Testing. ACM Transactions on Software Engineering Methodology*, 8(4), October 1999.

[LI07] R Lefticaru and F. Ipate. Automatic state-based test generation using genetic algorithms. In *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 188–195, Timisoara, 26-29 Sept 2007.

[LPvB94] G. Luo, A. Petrenko, and G. v. Bochmann. *Test Selection based on Communicating Nondeterministic Finite State Machines using a Generalized Wp-Method. IEEE Trans.*, SE-20(2), 1994.

[LY93] D. Lee and M. Yiannakakis. Optimization problems from feature testing of communication protocols. In *1st International Conference on Network Protocols*, pages 66–75, San Francisco, CA, USA, October 19-22 1993.

[LY96] D. Lee and M. Yiannakakis. *Principles and Methods of Testing Finite State Machines – A Survey. Proc. of the IEEE*, 43(3):1090–1123, 1996.

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

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

[Pap06] Z Pap. *Detection, diagnosis and correction of faults in telecommunications software development based on formal methods.* PhD thesis, Budapest University of Technology and Economics, 2006.

[PG91] R.L. Probert and F. Guo. *Mutation Testing of Protocols: Principles and Preliminary Experimental Results.* In *Proc. Protocol Test Systems, III.*, pages 57–76, 1991.

[PP05] A. Pretschner and J. Phillips. Methodological issues in model-based testing. In Broy M., Leuckner M., Jonsson B., Katoen J.-P., and Pretschner A., editors, *Model based testing of reactive systems*, volume 3472 of *LNCS*, pages 281–291. Springer, 2005.

[PvBD93] A. Petrenko, G v. Bochmann, and R. Dssouli. Conformance relations and test derivation. In O. Rafiq, editor, *6th Int. Workshop on Protocol Test Systems*, pages 157–178. North Holland, 1993.

[SDJB+93] W.M. Spears, K.A. De Jong, T. Bäck, D.B. Fogel, and H. de Garis. *An Overview of Evolutionary Computation.* In *Proceedings of the European Conference on Machine Learning (ECML-93)*, volume 667, pages 442–459, Vienna, Austria, 1993. Springer Verlag.

[SKGH97] M. Schmitt, B. Koch, J. Grabowski, and D. Hogrefe. Autolink - a tool for the automatic and semi-automatic test generation. In *Seventh GI/ITG Technical Meeting on Formal Description Techniques for Distributed Systems*, 1997.

[SMFLDS00] S.R.S. Souza, J.C. Maldonado, S.C.P.F. Fabbri, and W. Lopes De Souza. *Mutation Testing Applied to Estelle Specifications. Software Quality Journal*, 8(04), 2000.

[SMW04] T. Sugeta, J.C. Maldonado, and W.E. Wong. Mutation testing applied to validate sdl specifications. In *Proceedings of 16th IFIP Testing of Communicating Systems*, pages 193–208, Oxford, UK, 2004.

[Tau] Telelogic Tau. `http://www.telelogic.com`.

[TB03] G.J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In *First European Conference on Model-Driven Software Engineering*, pages 31–43, Nuremberg, Germany, December 11-12 2003.

[TPvB95] Q.M. Tan, A. Petrenko, and G. v. Bochmann. *Modeling Basic LOTOS by FSMs for Conformance Testing.* In *Protocol Specification, Testing and Verification*, pages 137–152, 1995.

[Tre00] J. Tretmans. *Specification Based Testing with Formal Methods: A Theory.* In A. Fantechi, editor, *FORTE / PSTV 2000 Tutorial Notes*, Pisa, Italy, October 10 2000.

[VAC92] Son T. Vuong and Jadranka Alilovic-Curgus. On test coverage metrics for communication protocols. In *Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems IV*, pages 31–45, 1992.

[vBP94a] G. v. Bochmann and A. Petrenko. *Protocol Tesing: Review of Methods and Relevance for Software Testing.* In *Software testing and analysis*, pages 109–124, 1994.

[vBP94b] G. v. Bochmann and A. Petrenko. *Protocol Testing: Review of Methods and Relevance for Software Testing.* In *ISSTA*, pages 109–124, 1994.

[Vin02] G. Vincze. Test selection using evolution algorithms (in hungarian). Master's thesis, Budapest University of Technology and Economics, 2002.

[WF74] R.A. Wagner and M.J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, January 1974.

[WL93] C.-J. Wang and M. T. Liu. *Generating Test Cases for EFSM with Given Fault Models*. In *INFOCOM 93*, volume 2, pages 774–781, 1993.

[WP02] A.W. Williams and R.L. Probert. Formulation of the interaction test coverage problem as an integer program. In *Testing Communication Systems XIV*, pages 283–298, Berlin, Germany, 2002.

[WRQC08] W. Eric Wong, Andy Restrepo, Yu Qi, and Byoungju Choi. An efsm-based test generation for validation of sdl specifications. In *AST '08: Proceedings of the 3rd international workshop on Automation of software test*, pages 25–32, New York, NY, USA, 2008. ACM.

[WW06] S. Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm. In G.G. Yen, L. Wang, P. Bonissone, and S.M. Lucas, editors, *2006 IEEE Congress on Evolutionary Computation*, pages 3193–3200, Vancouver, 6-21 July 2006. IEEE Press.

Publications

Journal papers

[J1] **G. Kovács**, Z. Pap, G. Csopaki, and K. Tarnay, “Iterative automatic test generation method for telecommunication protocols,” *Computer Standards and Interfaces*, vol. 28, no. 4, pp. 412–427, 2006.

Journal papers appeared in Hungary

[J2] **G. Kovács**, Z. Pap, and G. Csopaki, “*Automatic Test Selection based on CEFSM Specifications*,” *Acta Cybernetica*, vol. 15, pp. 583–599, 2002.

Journal papers in hungarian

[J3] **G. Kovács**, “Teszteset válogatás távolság metrikával,” (in hungarian) *Híradástechnika*, vol. LXI, pp. 41–43, January 2006.

Lecture Notes in Computer Science papers

[C1] G. Csopaki, G. Horváth, and **G. Kovács**, “*Communication Protocol Implementation in Java*,” in *Lecture Notes in Computer Science 1905: Interactive Distributed Multimedia Systems and Telecommunication Services*, (Enschede, The Netherlands), pp. 254–265, October 17-20 2000.

[C2] **G. Kovács**, Z. Pap, D. Le Viet, A. Wu-Hen-Chang, and G. Csopaki, “*Applying Mutation Analysis to SDL Specifications*,” in *11th International SDL Forum: SDL 2003: System Design* (R. Reed and J. Reed, eds.), (Stuttgart, Germany), pp. 269–284, 2003.

[C3] Z. Pap, M. Subramaniam, **G. Kovács**, and G. Á. Németh, “A bounded incremental test generation algorithm for finite state machines,” in *19th IFIP Testing of Communicating Systems (TestCom/FATES)*, (Tallinn, Estonia), pp. 244–259, June 26-29 2007.

[C4] **G. Kovács**, G. Á. Németh, M. Subramaniam, and Z. Pap, “*Optimal string edit distance based test suite reduction for SDL specifications*”, in *LNCS 5719: 14th International SDL Forum: SDL 2009* (R. Reed, A. Bilgic and R. Gotzhein, eds.), (Bochum, Germany), pp. 82-97, September 22-24 2009.

Conference papers

[C4] G. Gordos, G. Csopaki, Z. Werner, G. Horváth, **G. Kovács**, and T. Kerecsen, "*Automatic converion of SDL into modern oop languagues with a Java-based compiler*," in *International Workshop on Intelligent Communication Technologies and Applications*, (Neuchatel, Switzerland), May 5-7 1999.

[C5] **G. Kovács**, Z. Pap, and G. Csopaki, "*Automatic Test Selection based on CEFSM Specifications*," in *7th Symposium on Programing Languages and Software Tools* (T. Gyimothy, ed.), (Szeged, Hungary), pp. 84–97, June 15-16 2001.

[C6] **G. Kovács**, Z. Pap, and G. Csopaki, "*Automatic test selection method applied to WAP*," in *EUNICE 2001*, (Paris, France), September 3-5 2001.

[C7] G. Csopaki, T. Kasza, **G. Kovács**, and M. Szücs, "*Applicability of UML in the protocol development process*," in *Polish-Czech-Hungarian Workshop on Circuit Theory, Signal Processing and Telecommunication Services*, (Budapest), September 14-17 2001.

[C8] **G. Kovács**, D. Le Viet, and A. Wu-Hen-Chang, "*Iterative automatic test generation*," in *EUNICE 2003*, (Balatonfüred), September 8-10 2003.

[C9] C. Lukovszki, L. Kovács, **G. Kovács**, A. Foglar, E. Areizaga, and Z. Ghebretebsaé, "Revolutionary ipv6 optimized access solution," in *11th European Conf. on Networks and Optical Communications (NOC)*, (Berlin, Germany), July 11-13 2006.

[C10] H. Mickelsson, A. van Neerbos, P. Nooren, M. Prins, K. Oberle, D. Jocha, B. Radier, **G. Kovács**, M. Thakur, I. Pinilla, E. Areizaga, A. Gamelas, and A. Sitek, "Nomadism/fmc use cases and aaa impact," in *BroadBand Europe*, (Geneva, Switzerland), December 12-14 2006.

[C11] Z. Kardkovács, E. Lejtovicz, and **G. Kovács**, "Context identification: A relational database approach," in *The 3rd Language & Technology Conference* (Z. Vetulani, ed.), (Poznan, Poland), pp. 211–215, October 1-5 2007.

[C12] **G. Kovács**, G. Á. Németh, Z. Pap, and M. Subramaniam, "Deriving compact test suites for telecommunication software using distance metrics," in *Software, Telecommunications and Computer Networks (SoftCOM)*, (Split-Dubrovnik, Croatia), September 25-27 2008.

Other publications

[O1] D. Jocha and **G. Kovács**, "Introduction to the MUSE FMC architecture," in *MUSE Summer School*, (Budapest), July 5 2007.

Citations

[1] T. Sugeta, J. Maldonado, and W. Wong, "Mutation testing applied to validate sdl specifications," in *Testing of Communicating Systems. 2004.03.17-2004.03.19 193-208. 2004.*, (Oxford, England), pp. 193–208, March 17-19 2004.

[2] M. Hong and Z. Lu, "A framework for testing web services and its supporting tool," in *SOSE 2005: IEEE International Workshop on Service-Oriented System Engineering*, (Beijing, China), pp. 199–206, October 20-21 2005.

[3] Y. Jiang, S. Hou, J. Shan, Z. L., and B. Xie, "Contract-based mutation for testing components," in *ICSM 2005: 21st IEEE International Conference on Software Maintenance*, (Budapest, Hungary), pp. 483–492, September 26-29 2005.

[4] Y. Jiang, S. Hou, J. Shan, L. Zhang, and B. Xie, "An approach to testing black-box components using contract-based mutation," *INTERNATIONAL JOURNAL OF SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING 18: (1) 93-117 (2008)*, vol. 18, no. 1, pp. 93–117, 2008.

[5] Z. Andriska, G. Bátori, D. Le Viet, A. Wu-Hen-Chang, and G. Csopaki, "Tool for automatic test selection based on formal specification," *Híradástechnika*, vol. LVII, pp. 37–43, December 2002.

[6] K. Oberle, S. Wahl, and A. Sitek, "Enhanced methods for SIP based session mobility in a converged network," in *16th Mobile and Wireless Communications Summit*, (Budapest, Hungary), pp. 1–5, July 1-5 2007.