Search-Based Software Engineering: A Search-Based Approach for Testing from Extended Finite State Machine (EFSM) Models

AbdulSalam Kalaji

A thesis submitted for the degree of Doctor of Philosophy

Brunel University

Department of Information Systems and Computing

July, 2010

Abstract

The extended finite state machine (EFSM) is a powerful modelling approach that has been applied to represent a wide range of systems. Despite its popularity, testing from an EFSM is a substantial problem for two main reasons: path feasibility and path test case generation. The path feasibility problem concerns generating transition paths through an EFSM that are feasible and satisfy a given test criterion. In an EFSM, guards and assignments in a path's transitions may cause some selected paths to be infeasible. The problem of path test case generation is to find a sequence of inputs that can exercise the transitions in a given feasible path. However, the transitions' guards and assignments in a given path can impose difficulties when producing such data making the range of acceptable inputs narrowed down to a possibly tiny range. While search-based approaches have proven efficient in automating aspects of testing, these have received little attention when testing from EFSMs. This thesis proposes an integrated search-based approach to automatically test from an EFSM. The proposed approach generates paths through an EFSM that are potentially feasible and satisfy a test criterion. Then, it generates test cases that can exercise the generated feasible paths. The approach is evaluated by being used to test from five EFSM cases studies. The achieved experimental results demonstrate the value of the proposed approach.

Table of Contents

Abstract ii
Table of Contents iii
List of Tables vii
List of Figures ix
Acknowledgements xiv
Related Publications xvi

Chapter 1: Introduction 1

1.1 OVERVIEW 1
1.2 CONFORMANCE TESTING 3
1.3 SEARCH-BASED TESTING 4
1.4 THE PROBLEM AREA 6
1.5 THE THESIS'S AIMS AND OBJECTIVES 8
1.6 SUMMARY OF THE CONTRIBUTIONS 8
1.7 THE THESIS'S STRUCTURE 9
1.7.1 Chapter 2: Literature Review 9
1.7.2 Chapter 3: Generating Feasible Transition Paths (FTPs) for Testing from EFSM Models 10
1.7.3 Chapter 4: Automatic Test Cases Generation to Exercise Feasible Transition Paths 10
1.7.4 Chapter 5: Generating Feasible Transition Paths for Testing from EFSMs with Counter Problem 11
1.7.5 Chapter 6: Conclusions and Future Work 11

Chapter 2: Literature Review 12

2.1 INTRODUCTION 12
2.2 WHY AUTOMATE TESTING? 13
2.3 SOFTWARE TESTING PREAMBLE 13
2.4 TESTING APPROACHES 14
2.4.1 Black-Box Testing 15
2.4.2 White-Box Testing 16
2.5 METHODS USED TO AUTOMATE TESTING 18
2.5.1 Symbolic Execution 18

2.5.2 Constraint Satisfaction 20
2.5.3 Search-Based Software Testing (SBST) 21
2.5.3.1 Random Search 25
2.5.3.2 Hill Climbing 25
2.5.3.3 Simulated Annealing 26
2.5.3.4 Evolutionary Algorithms (EAs) 28
2.5.3.5 Evolutionary Testing 31
2.5.3.6 Model Checkers 34
2.6 METHODS USED TO SUPPORT TESTING 35
2.6.1 Slicing 36
2.6.2 Testability Transformation 37
2.7 MODEL BASED TESTING 38
2.7.1 Finite State Machine (FSM) 39
2.7.1.1 Conformance Testing 41
2.7.1.2 Testing From an FSM 41
2.7.2 Extended Finite State Machine (EFSM) 43
2.8 TESTING FROM EFSMS 45
2.8.1 Related Work of Testing from EFSMs 48
2.8.2 Motivation for Automatic Testing from an EFSM 50
2.9 CONCLUSION 51

Chapter 3: Generating Feasible Transition Paths (FTPs) for Testing from EFSM Models 53
3.1 INTRODUCTION 53
3.2 CASE STUDIES 54
3.3 PROBLEM AREA 62
3.4 THE PROPOSED APPROACH 66
3.4.1 Dependencies Representation and Penalties 69
3.4.2 The Fitness Metric 73
3.4.3 The GA Encoding 78
3.4.4 FTP Verification Method 79
3.5 EXPERIMENT 79
3.5.1 Experimental Design 80
3.5.2 Experimental Results 82

3.5.2.1 Results of the Lift EFSM 83
3.5.2.2 Results of the In-Flight EFSM 85
3.5.2.3 Results of the ATM EFSM 88
3.5.2.4 Results of The Inres Initiator EFSM 91
3.5.2.5 Results of The Class 2 EFSM 94
3.5.2.6 Summary of Results 96
3.6 CONCLUSION 97

Chapter 4: Automatic Test Cases Generation to Exercise Feasible Transition Paths 99

4.1 INTRODUCTION 99
4.2 PROBLEM AREA 100
4.3 THE PROPOSED APPROACH 102
4.3.1 The Fitness Function 102
4.3.2 GA Encoding 107
4.3.3 Using Constraint Satisfaction to Trigger an FTP 107
4.4 EXPERIMENT 108
4.4.1 Design of the First Experiment 109
4.4.2 Experimental Results for the Three Search-Based FTPs Test Cases Generators 110
4.4.2.1 Results Derived from the Lift EFSM 111
4.4.2.2 Results Derived from the In-Flight EFSM 113
4.4.2.3 Results Derived from the ATM EFSM 116
4.4.2.4 Results Derived from the Inres Initiator EFSM 119
4.4.2.5 Results Derived from the Class 2 EFSM 121
4.4.2.6 Summary of the Results 124
4.4.3 Design of the Second Experiment 126
4.4.4 Results of the Second Experiment 128
4.4.4.1 CBT Performance on Both Groups of FTPs 128
4.4.4.2 Correlation Study (Without FTP Clustering) 129
4.4.4.3 Regression Analysis (Without FTP Clustering) 135
4.4.4.4 Correlation Study (Clustered FTPs) 138
4.4.4.5 Regression Analysis (Clustered FTPs) 140
4.5 CONCLUSION 143

Chapter 5: Generating Feasible Transition Paths for Testing from EFSMs with Counter Problem 145
5.1 INTRODUCTION 145
5.2 PROBLEM AREA 146
5.3 CASE STUDIES 148
5.4 THE PROPOSED APPROACH 149
5.4.1 Dependencies Representation 153
5.4.2 Finding the Required Sequence of Transitions 154
5.4.3 Determining the Length of TPs 161
5.5 EXPERIMENT 166
5.5.1 Experimental Design 167
5.5.2 Experimental Results 169
5.5.2.1 Results of the Inres Initiator EFSM 169
5.5.2.2 Results of the ATM EFSM 174
5.5.2.3 Summary of Results 180
5.6 CONCLUSION 181

Chapter 6: Conclusions and Future Work 183
6.1 CONCLUSION OF ACHIEVEMENTS 183
6.1.1 Problems Associated with Testing from EFSMs 183
6.1.2 Generating Feasible Transition Paths (FTPs) 185
6.1.3 Generating Test Cases to Trigger FTPs 186
6.1.4 By-Passing The Counter Problem When Generating FTPs 188
6.2 POINTS FOR FUTURE WORK 189
6.2.1 The Fitness Metric 190
6.2.2 FTPs Test Cases Generations 191
6.2.3 The Normalisation Function During FTPs Test Cases Generation 192
6.2.4 Complex FTPs' Test Cases Generation 192
6.2.5 Calibrating the TP Fitness Metric 193
6.2.6 An Iterative Approach 194
6.3 GENERAL CONCLUSION 195

References 196

List of Tables

Table 2.1: Tracey et al. fitness calculations for different types of guards. The constant *k*, *k* > 0, is added when the guard is not satisfied.................................. 32
Table 3.1: The transitions description of the In-Flight safety system 55
Table 3.2: The transitions description of the class 2 transport protocol.............. 57
Table 3.3: The transitions description of the Lift system.................................... 59
Table 3.4: The suggested penalty values where *INF* is a large positive integer to indicate that a given dependency represents an infeasible case. 71
Table 3.5: Assignment's types representation.. 72
Table 3.6: Lift EFSM GA & RA generated TPs 85
Table 3.7: The In-Flight EFSM GA & RA generated TPs................................. 87
Table 3.8: ATM EFSM GA & RA generated TPs 89
Table 3.9: Inres initiator EFSM GA & RA generated TPs 93
Table 3.10: The Class 2 EFSM GA & RA generated TPs 96
Table 3.11: Summary of the results achieved by GA and random searches on generating FTPs from five EFSM case studies. 97
Table 4.1: Tracey et al. fitness calculations for different types of guards. The constant *k*, *k* > 0, is added when the guard is not satisfied................................ 103
Table 4.2: The performance of three test case generation methods on two groups of subject TPs derived from the Lift EFSM... 113
Table 4.3: The performance of three test case generation methods on two groups of subject TPs derived from the In-Flight EFSM. 116
Table 4.4: The performance of three test case generation methods on two groups of subject TPs derived from the ATM EFSM.. 118
Table 4.5: The performance of three test case generation methods on two groups of subject TPs derived from the Inres initiator EFSM...................................... 121
Table 4.6: The performance of three test case generation methods on two groups of subject TPs derived from the Class 2 EFSM.. 124
Table 4.7: The performance of the three test case generators on the five EFSM case studies.. 125
Table 4.8: GA group of FTPs (no clustering) - Correlation among FTPs' fitness metric, GA average generations, GA average time and CBT average time. 130

Table 4.9: Random group of FTPs (no clustering) - Correlation among FTP fitness metric, GA average generations, GA average time and CBT average time. 131
Table 4.10: Both groups of FTPs (no clustering) - Correlation among FTP fitness metric, GA average generations, GA average time and CBT average time. 132
Table 4.11: FTPs (no clustering) - Correlation among FTP fitness metric, GA average generations, GA average time, CBT average time, number of inequality constraints and number of equality constraints. 133
Table 4.12: GA group of FTPs (clustered) - Correlation among FTPs' fitness metric, GA average generations, GA average time and CBT average time. 139
Table 4.13: Random group of FTPs (clustered) - Correlation among FTP fitness metric, GA average generations, GA average time and CBT average time. 139
Table 4.14: Both groups of FTPs (clustered) - Correlation among FTP fitness metric, GA average generations, GA average time and CBT average time. 140
Table 5.1: Guards representation as integers... 153
Table 5.2: Operations representation as integers... 153
Table 5.3: Calculating the TP length for the Inres initiator and the ATM EFSMs. 166
Table 5.4: The Inres initiator EFSM generated TPs by the proposed approach (GA-1) and by the previously defined TP fitness metric approach (GA-2)........ 172
Table 5.5: The ATM EFSM generated TPs by the proposed approach (GA-1) and by the previously defined TP fitness metric approach (GA-2). 177
Table 5.6: Summary of the results achieved by the proposed approach (GA-1) and the previously defined TP fitness metric approach (GA-2) on generating FTPs from two EFSMs that suffer from the counter problem.................................... 180

List of Figures

Figure 1.1: A fire alarm system represented as extended finite state machine, each transition is given in the form of: [guards/operations]. 4
Figure 1.2: An example of local and global optima 5
Figure 2.1: A fire alarm system represented as a directed graph 15
Figure 2.2: The greatest common divisor and its CFG 16
Figure 2.3: Examples of two fitness functions landscapes 23
Figure 2.4: Hill climbing algorithm for a maximisation optimisation problem with an input domain *D* 26
Figure 2.5: Simulated annealing algorithm for a minimisation optimisation problem with an input domain *D* 27
Figure 2.6: High level description of a basic genetic algorithm 30
Figure 2.7: An example of branch distance calculation by using Wegener et al. approach. A Critical node has a false branch (shown in a dashed arrow) that diverts the execution flow 33
Figure 2.8: An example program slicing. The original program was used to produce a slice for variable *x* at node 4. 36
Figure 2.9: An FSM of a traffic light control system represented as a directed graph. 38
Figure 2.10: Conformance testing approach 41
Figure 2.11: The Inres Initiator EFSM model 43
Figure 3.1: The EFSM model of In-Flight safety system 54
Figure 3.2: The EFSM model of core transitions of class 2 transport protocol model 56
Figure 3.3: The EFSM model of the Lift system 58
Figure 3.4: The EFSM model of Inres Initiator 60
Figure 3.5: The EFSM model of the ATM System 61
Figure 3.6: An example of a tuple representation of the dependencies between an affecting and affected-by transitions 73
Figure 3.7: The algorithm that calculates TP fitness metric 74
Figure 3.8: The recursive subroutine *Check* which traces back a transition's dependencies. 75

Figure 3.9: Lift EFSM TPs. The sets *a* & *b* have a TP length = 9, sets *c* & *d* have a TP length = 12 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are GA generated by using the TP fitness metric guidance. Sets *b*, *d* & *f* are the alternative randomly generated sets plotted by using logarithmic scale................................ 84
Figure 3.10: In-Flight EFSM TPs. The sets *a* & *b* have a TP length = 9, sets *c* & *d* have a TP length = 12 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are GA generated by using the TP fitness metric guidance. Sets *b*, *d* & *f* are the alternative randomly generated sets plotted by using logarithmic scale................................ 86
Figure 3.11: ATM EFSM TPs. The sets *a* & *b* have a TP length = 9, sets *c* & *d* have a TP length = 12 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are GA generated by using the TP fitness metric guidance. Sets *b*, *d* & *f* are the alternative randomly generated sets plotted by using logarithmic scale................................ 90
Figure 3.12: Inres Initiator EFSM TPs. The sets *a* & *b* have a TP length = 9, sets *c* & *d* have a TP length = 12 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are GA generated by using the TP fitness metric guidance. Sets *b*, *d* & *f* are the alternative randomly generated sets plotted by using logarithmic scale.............. 92
Figure 3.13: Class 2 EFSM TPs. The sets *a* & *b* have a TP length = 9, sets *c* & *d* have a TP length = 12 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are GA generated by using the TP fitness metric guidance. Sets *b*, *d* & *f* are the alternative randomly generated sets plotted by using logarithmic scale................................ 95
Figure 4.1: The SDL representation of transitions t_5, t_8 and t_{15} of Inres initiator EFSM where transition's guards are sequenced as nested IF statements 101
Figure 4.2: An example of fitness calculation by using Wegener et al. (2001) approach. 105
Figure 4.3: An example of a path fitness calculation by using the proposed approach. 106
Figure 4.4: Lift EFSM GA and random groups of subject TPs. Plots *a* & *b* show the performance of the GA-1 approach on TPs in GA group and TPs in the random group respectively. Similarly, Plots *c* & *d* and Plots *e* & *f* show the performance of GA-2 and Rand approaches respectively. The fitness metric of subject TPs in random group is plotted by using logarithmic scale................... 112

Figure 4.5: In-Flight EFSM GA and random groups of subject TPs. Plots *a* & *b* show the performance of the GA-1 approach on TPs in GA group and TPs in the random group respectively. Similarly, Plots *c* & *d* and Plots *e* & *f* show the performance of GA-2 and Rand approaches respectively. The fitness metric of subject TPs in random group is plotted by using logarithmic scale.................... 114

Figure 4.6: ATM EFSM GA and random groups of subject TPs. Plots *a* & *b* show the performance of the GA-1 approach on TPs in GA group and TPs in the random group respectively. Similarly, Plots *c* & *d* and Plots *e* & *f* show the performance of GA-2 and Rand approaches respectively. The fitness metric of subject TPs in random group is plotted by using logarithmic scale.................... 117

Figure 4.7: Inres EFSM GA and random groups of subject TPs. Plots *a* & *b* show the performance of the GA-1 approach on TPs in GA group and TPs in the random group respectively. Similarly, Plots *c* & *d* and Plots *e* & *f* show the performance of GA-2 and Rand approaches respectively. The fitness metric of subject TPs in random group is plotted by using logarithmic scale.................... 120

Figure 4.8: Class II EFSM GA and random groups of subject TPs. Plots *a* & *b* show the performance of the GA-1 approach on TPs in GA group and TPs in the random group respectively. Similarly, Plots *c* & *d* and Plots *e* & *f* show the performance of GA-2 and Rand approaches respectively. The fitness metric of subject TPs in random group is plotted by using logarithmic scale.................... 122

Figure 4.9: The CBT performance on both groups of FTPs. Plot *a* shows the performance on the GA group of FTPs while Plot *b* shows the performance on the random group of FTPs. 129

Figure 4.10: Fitness line fit plots of regression analysis (without clustering). Plots *a*, *c* & *e* are the prediction of GA time in seconds from FTPs in GA group, random group and both groups respectively. Similarly, Plots *b*, *d* & *f* are the prediction of CBT time in seconds from the same groups respectively. 136

Figure 4.11: Fitness line fit plots of regression analysis. Plots *a*, *c* & *e* are the prediction of GA time in seconds from clustered FTPs in GA group, random group and both groups respectively. Similarly, Plots *b*, *d* & *f* are the prediction of CBT time in seconds from the same groups respectively. 141

Figure 5.1: Inres Initiator EFSM. Initialisers, updaters and target transitions are coloured green, blue and red respectively. Transition t_{12} represents an 'escape' transition and is represented by a dashed arrow. 149
Figure 5.2: The EFSM Model of the ATM System. Initialisers, updater and target transitions are coloured green, blue and red respectively. Transition *t'* represents an 'escape' transition and is represented by a dashed arrow. 150
Figure 5.3: Example of affecting and affected-by (a counter) matrices for Inres initiator EFSM.. 154
Figure 5.4: The algorithm which finds sequence of transitions 155
Figure 5.5: The Guard_Check routine.. 155
Figure 5.6: The algorithm which validates a given triple 157
Figure 5.7: The routine which verifies a tripe existence in a TP 159
Figure 5.8: The initial and first steps of Dijkstra's algorithm between states S_d & S_s in Inres Initiator EFSM 163
Figure 5.9: The second and last steps of Dijkstra's algorithm between states S_d & S_s in Inres Initiator EFSM 164
Figure 5.10: Inres EFSM TPs. The sets *a* & *b* have a TP length = 13, sets *c* & *d* have a TP length = 14 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are generated by using the proposed approach (GA-1). Sets *b*, *d* & *f* are the alternative sets that are generated by using the previously defined TP fitness metric approach (GA-2). 170
Figure 5.11: Inres initiator EFSM TPs. Each figure shows a TP fitness metric value vs. the average number of generations in ten tries to trigger this TP. Figure *a* plots all the generated TPs by using the proposed approach (GA-1). Figure *b* plots all the generated TPs by using the previously defined TP fitness metric approach (GA-2). For clarity, the horizontal axis is a logarithmic scale. 174
Figure 5.12: ATM EFSM TPs. The sets *a* & *b* have a TP length = 9, sets *c* & *d* have a TP length = 12 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are generated by using the proposed approach (GA-1). Sets *b*, *d* & *f* are the alternative sets that are generated by using the previously defined TP fitness metric approach (GA-2). 175

Figure 5.13: The prevalence of the ‘escape’ transition *t'* in the three sets of subject TPs derived from the ATM EFSM by using the proposed approach................. 178
Figure 5.14: ATM EFSM TPs. Each figure shows a TP fitness metric value vs. the average number of generations in ten tries to trigger this TP. Figure *a* plots all the generated TPs by using the proposed approach (GA-1). Figure *b* plots all the generated TPs by using the previously defined TP fitness metric approach (GA-2). For clarity, the horizontal axis is a logarithmic scale... 179
Figure 6.1: Two sets of nested guards that are assigned the same TP fitness metric value. However, the second set is harder to be satisfied.................................... 190
Figure 6.2: A counter problem and a TP fitness metric mechanism................. 190
Figure 6.3: Calculating a path fitness by using only Wegener et al. approach .. 191
Figure 6.4: Calculating a path fitness by using ‘Program Stretching’ approach (Ghani and Clark, 2009b).. 193
Figure 6.5: An iterative approach to calibrate the TP fitness metric.................. 194
Figure 6.6: An iterative search-based approach to test from an EFSM 195

Acknowledgements

First and foremost, my words of thanks must go to my supervisor Prof. Robert Mark Hierons who has been of great help to this work. Prof. Hierons has been a constant source of support, guidance and encouragement throughout the periods of my Master of Research and my PhD courses in Brunel University. I am sincerely grateful for the professional supervision I have received together with the insightful comments and advices that made the work on this thesis possible.

I would like also to thank Dr. Stephen Swift, my second supervisor during my study in Brunel University, for the help he has provided and for reading my thesis. Dr. Swift has provided important comments and advices which have been of much help during the work on this thesis.

I am thankful to the staff in the Department of Information Systems and Computing for the lovely research environment that they all have provided.

My thanks go also to my family for their endless care and support and to my friends for being around me.

My sincere gratitude goes also to Aleppo University- Syria for sponsoring my study in Brunel University.

Dedication

There are no words that can sufficiently describe the great man who constantly worked hard to provide me with every possible support. This thesis is dedicated to my father who passed away couple of months ago.

Related Publications

- Abdul Salam Kalaji, Robert Mark Hierons and Stephen Swift, "Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM)," In *ICST09*: *Proceeding of the 2nd IEEE International Conference on Software Testing Verification and Validation*, pp.230-239, Denver, USA, IEEE Press, April 2009.

- Abdul Salam Kalaji, Robert Mark Hierons, Stephen Swift, "A Testability Transformation Approach for State-Based Programs," In *SSBSE* 09: *Proceeding of the 1st International Symposium on Search Based Software Engineering*, pp.85-88, Windsor, UK, IEEE Press, May 2009.

- Abdul Salam Kalaji, Robert Mark Hierons, Stephen Swift, "A Search-Based Approach for Automatic Test Generation from Extended Finite State Machine (EFSM)," In *TAIC-PART 09*: *Proceeding of the Testing: Academic and Industrial Conference - Practice and Research Techniques*, pp.131-132, Windsor, UK, IEEE Press, September 2009.

- Abdul Salam Kalaji, Robert Mark Hierons and Stephen Swift, "Generating Feasible Transition Paths for Testing from an Extended Finite State Machine with the Counter Problem," In *ICSTW10*: *Proceeding of the 3rd IEEE International Conference on Software Testing Verification and Validation workshops*, pp. 232-235, Paris, France, IEEE Press, April 2010.

Chapter 1: Introduction

1.1 Overview

Software testing is a vital practice that is included in the software development process to provide confidence in the software quality. Software testing can be seen as a method that can detects errors by mimicking the real interactions with the software and then monitoring how the software responds. This can be achieved by deriving a set of test scenarios and then applying them to the software. Since there can be a large number of possible test scenarios that can be derived for a given system, a test criterion is required. Test criteria can be considered as properties that the testing process should satisfy in order to be considered adequate.

Although software testing does not add any extra functionality to the software, it is widely known that testing can cost up to 50% of the overall software cost (Boehm, 1981, Hierons et al., 2009). Therefore, it is natural for the software engineering community to try to develop methods to reduce the cost associated with testing while enhancing the testing process. One important point to be realised is the fact that manual testing is a poor method to achieve this goal. Unfortunately, manual testing is known to be expensive, time consuming and vulnerable to errors. Since such aspects are significant contributors to the cost and validity of testing, automation is desirable.

In order for testing to be automated, there are various steps to be achieved such as the automatic derivation of the test scenarios and the automatic verification of the system responses to the applied test scenarios. Importantly, the automatic derivation of test scenarios is a substantial task (Harman and McMinn,

2009). There are various approaches that can be used, including search-based testing techniques.

To conduct testing, there are two main approaches that can be used: white-box testing and black-box testing. White box testing can be applied when a tester has access to the internal software structure such as the code and algorithms. Such access allows the tester to use the knowledge about the software to design the test scenarios. In contrast, black-box testing does not use information about the internal system structure and only two aspects are available to the tester: the inputs and the outputs. Therefore, when designing the test scenarios, the tester uses the system specification (usually represented as a model) to derive the test scenarios. Then these scenarios are applied to the implementation under test (IUT) and the resultant outputs are monitored. By comparing the produced outputs to those stated in the specification, the tester can determine whether the IUT conforms to the specification on that test case. Therefore, such testing is commonly referred to as conformance testing. In order to apply conformance testing, it is necessary to have a system specification, usually represented in terms of a model.

Two common modelling approaches that can be used for this purpose are: finite state machine (FSM) and extended finite state machine (EFSM) (Petrenko et al., 2004). An FSM basically comprises a finite set of states and transitions among the states. Each transition has a start state and an end state. Also, a transition requires an input to be fired and if so it produces an output. The FSM is used to model a system that has a control part, for example the basic telephone device. However, if the system has, in addition to the control part, a data part, then an extended FSM can be used. An EFSM can model both control and data parts since it extends the FSM structure with a set of variables (memory). Therefore, in an EFSM, a transition can have guards (preconditions) over the machine's variables and also can have operations (assignments) to these variables.

Testing from an EFSM is conformance testing (black-box). Because of the EFSM's structure (the combination of control and data parts) conformance testing from an EFSM is generally a difficult task. However, search-based approaches

can potentially ease such a process. In the subsequent sections, conformance testing from an EFSM together with the search-based testing are highlighted.

1.2 Conformance Testing

In conformance testing, there are two aspects: the system specification and the actual system or the implementation under test (IUT). The tester has to check whether the implementation conforms to the specification. In order to achieve this, the system specification is usually represented in terms of a model (such as an FSM or an EFSM). Because systems can be complex, a model can enhance the understanding about such systems and allow better reasoning (Beizer, 1990). The model is then used to derive test scenarios according to a test criterion. The test scenarios are applied to the IUT and the responses of the IUT are compared to the outputs stated in the specification to determine whether there has been a failure.

When representing system specification, a modelling approach such as an FSM or an EFSM can be used. The EFSM is one such modelling approach that is capable of representing a wide range of systems.

As aforementioned, when conducting testing, there can be vast number of possible test scenarios and so a test criterion is required. Testing from an EFSM can be based on coverage test criteria such as the transition coverage or the state coverage (Tahat et al., 2001). Transition coverage, for example, requires test scenarios that exercise each transition in the system at least once. Nevertheless, a major task when testing from an EFSM is the derivation of the test scenarios according to a given test criterion.

Consider for example Figure 1.1 which shows the EFSM representation of a fire alarm system. In this EFSM, there are two states: Safe and Warning, six transitions and two variables: *updated* and *smokeLevel*. A transition t_1 first checks that a new reading of the smoke level has not been performed (guard: *updated* = false) and if so it reads the current smoke level in the atmosphere through the parameter *SmokeInput*. Then, the operations associated with t_1 assign this value to the *smokeLevel* variable and set the variable *updated* to true

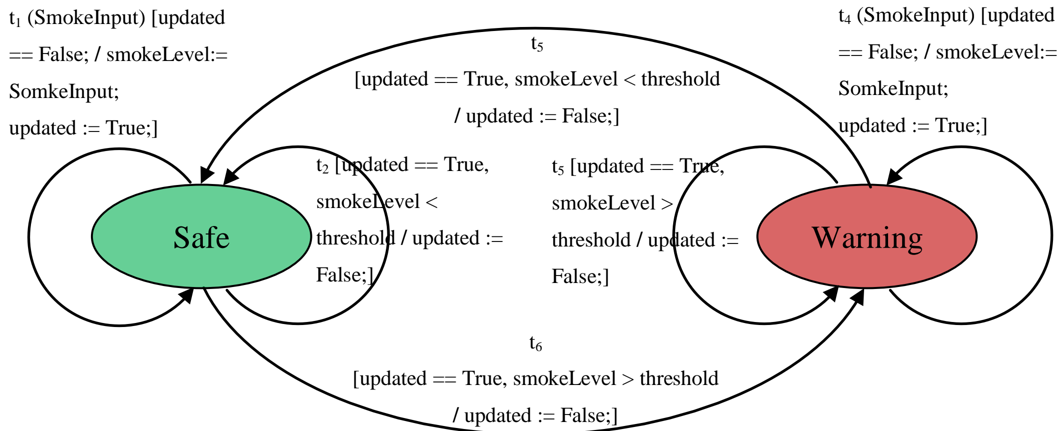


Figure 1.1: A fire alarm system represented as extended finite state machine, each transition is given in the form of: [guards/operations].

(indicating a new reading was performed). Now, another transition such as t_2 checks that a new reading has been performed (guard: *updated* = true) and if so it checks whether the current value of *smokeLevel* is less than a constant (guard: *smokeLevel* < threshold). If this is true, t_2 is fired, the associated operation (*updated* := false) is executed and the machine state is Safe.

For the transition coverage test criterion, test scenarios can consist of two parts: transition paths and the paths test cases. A transition path specifies a sequence of transitions such as t_1t_2 whereas a path test cases is given the inputs that are required to execute the path (for the path t_1t_2, a test case can be a suitable value of *SmokeInput* i.e. *SmokeInput* < threshold).

Therefore, testing from an EFSM requires the generation of a set of transition paths through the EFSM together with suitable inputs that can exercise these paths.

1.3 Search-Based Testing

Search-based testing refers to the use of optimisation techniques to automatically derive tests. Optimisation techniques such as genetic algorithms, simulated annealing and hill climbing are frameworks that can be adapted to find solutions to problems of interest. In white-box testing, a problem can be, for example,

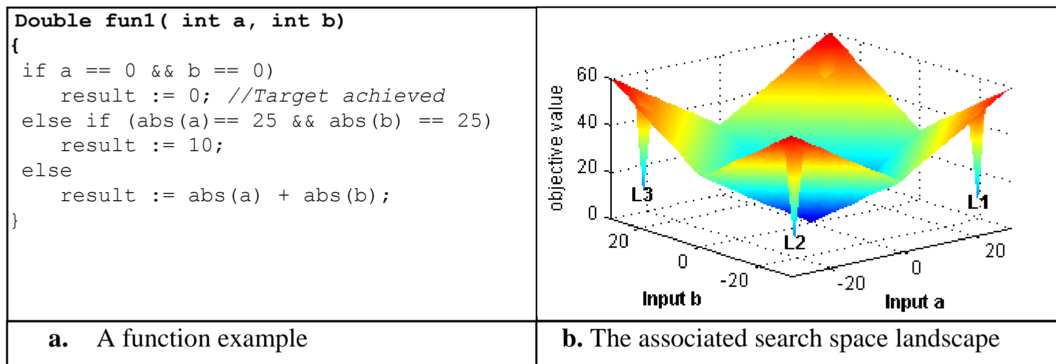


Figure 1.2: An example of local and global optima

finding a set of inputs that can exercise a set of conditions. Usually, the search space of such input values is large. However, the conditions may potentially narrow down the range of the acceptable values to be just a small range. Therefore, finding such input values can be difficult. Nevertheless, optimisation techniques can efficiently explore the search space for solutions that can be considered acceptable. In order to differentiate between potential solutions, optimisation techniques use a fitness function and this is problem dependant. A fitness function rewards every candidate solution with a value that states how good this solution is in terms of the considered problem. Generally the quality of potential solutions can range between global minima / maxima or local minima / maxima to unacceptable solutions. A candidate solution in the search space is said to be a global solution or global optimum if its fitness is the highest (for maximisation) or lowest (for minimisation) of any candidate solution in the search space. Often there is some notion of a neighbourhood and when this is the case a candidate solution in the search space is a local solution or local optimum if none of its neighbours have a better fitness.

Consider for example the set of conditions shown in Figure 1.2a. The function requires two inputs of type integer. The test target is to find values of input *a* and input *b* so that the result is 0 (minimisation). If the input values are given in the range [-30..30], then the search space can have the landscape shown in Figure 1.1b. There is one global optimum (minimum) that touches the zero surface (when both *a* and *b* are zero) and four local minima labelled L1, L2, L3 and L4.

When applying optimisation techniques to a given problem, the accepted solutions are not necessarily global minima / maxima. There can also be solutions that can be considered good or excellent. Consider for example the problem of minimising the cost associated with manufacturing a mobile phone while increasing the number of the preferred services (identified by users). For such a problem, there can be a range of solutions that can be considered excellent or adequate.

In order to apply optimisation techniques, the key step is to derive a fitness function which can differentiate best, good and worse solutions and so acts as the search guide. Consider for example the problem of maximising the following function:

$$\boldsymbol{Max}(\boldsymbol{x}^2);\ \textit{where}\ \boldsymbol{x} \in [-1000..1000]$$

For such a problem, the fitness function can be derived from the function that states the problem i.e. $fitness(x) = |\ x\ |$ and so the greater x, the better the fitness. The information returned by the fitness function informs the search to progress towards its goal.

The other important step in applying an optimisation technique is the solution's representation which can also be based on the problem description. For example, a possible solution representation of the problem above is to use the integer representation.

For software testing, the application of optimisation techniques is referred to by search-based testing. Search-based testing has proven efficient and shown considerable success in automating the process of white-box testing. Examples of such works are reported in (Korel, 1990, Jones et al., 1998, Michael et al., 2001, McMinn, 2004, Harman and McMinn, 2009). However, for black-box approach, search-based testing has received little attention.

1.4 The Problem Area

An EFSM is a powerful modelling approach and has been widely used for the purpose of modelling and testing (Dssouli et al., 1999, Duale et al., 1999, Ural et al., 2000, Hierons et al., 2001, Petrenko et al., 2004, Keum et al., 2006, Sinha et

al., 2007, Lorenzoli et al., 2008, Wong et al., 2009). While there are many testing approaches to test from FSM (Lee and Yannakakis, 1996), testing from an EFSM is more complex since an EFSM combines both control and data flows.

The combination between the control flow and data flow imposes difficulties when testing from EFSMs. These difficulties concern path feasibility and path test cases generation. The path feasibility problem arises when no sequence of inputs can lead to the path being followed. Therefore, when a path is selected to cover certain aspects (i.e. provides part of the transition coverage), this path can be infeasible. If the path is infeasible, it cannot be executed and so the intended test cannot be applied.

For example, if the path $t_1t_1t_5$ in the EFSM shown in Figure 1.1 is selected to cover transition t_5, this transition is not covered because the path is infeasible. Since the first transition t_1 sets the variable *updated* to true while the next transition t_1 requires *updated* to be false, the assignment of the first t_1 falsifies the guard of the next t_1.

The problem of the path feasibility in an EFSM is generally undecidable (Dssouli et al., 1999, Hierons et al., 2009). Furthermore, the development of good methods to overcome this issue is an open research problem (Duale et al., 1999, Duale and Uyar, 2004).

The problem of path test case generation concerns finding a sequence of inputs that can exercise a given feasible path. For example, consider again the machine in Figure 1.1. If a path $t_1t_2t_1t_6$ is selected, then two suitable inputs are required to exercise such a path. Since transitions in a path can have guards and assignments, the inputs should be selected to satisfy these guards so that the path can be fired. Generally, finding a suitable set of test cases that can trigger a given path is a substantial task (Ural and Yang, 1991). This is because the input domain is usually large while the suitable values constitute just a small subset of the input domain.

There are many studies that have applied search-based approaches to test systems using the white-box approach. These studies have demonstrated the capability of search-based testing to significantly ease the testing process. Examples of such studies are reported in (McMinn, 2004). However, search-based

approaches have received little attention when testing from an EFSM. Therefore, the subject of this thesis is to investigate the application of search-based approaches to test from EFSMs by considering the two main problems associated with testing from EFSMs. The aim is then to develop a framework based on search-based testing which can significantly enhance testing from EFSMs.

Such a framework can allow other EFSM testing approaches (Chanson and Zhu, 1993, Derderian et al., 2005, Duale and Uyar, 2004, Duale et al., 1999, Hierons et al., 2004, Koh and Liu, 1994, Lee and Yannakakis, 1996, Petrenko et al., 2004, Ramalingom et al., 2003, Sarikaya et al., 1987, Wang and Liu, 1993) to benefit from the flexibility and the efficiency of search based testing.

1.5 The Thesis's Aims and Objectives

The thesis aims and objectives are the following:

1. To identify challenges associated with testing from extended finite state machine (EFSM) models. Furthermore, highlighting limitations associated with current EFSM testing approaches.
2. To determine barriers to the application of search-based approaches to testing from EFSM models.
3. To propose an integrated search-based approach to automate the process of testing from EFSM models.

1.6 Summary of the Contributions

The main contributions of this thesis are:

1. The description of a novel method to represent the dependencies among an EFSM's transitions.
2. The definition of a new fitness metric that can be easily utilised by heuristic search techniques such as a genetic algorithm (GA) to facilitate

the automatic generation of feasible transition paths (FTPs) through EFSMs for the purpose of testing.

3. The description of a search-based approach to generate test cases that can trigger given FTPs through an EFSM model
4. The proposal of a fitness function that is suitable to guide the search for test cases in the presence of function calls.
5. The investigation of the relationship between an FTP's fitness metric value and the effort, in terms of time, required by a test cases generation approach to trigger the FTP. Furthermore, the investigation of the TP fitness metric capability to predict the effort required by test cases generators to exercise FTPs.
6. The proposal of a method to bypass the counter problem by automatically determining whether a transition's guard references a counter, which other transitions are involved and how many times they have to be called.
7. The proposal of a method that suggests a suitable length of test cases to be generated in the presence of the counter problem.
8. This thesis is the first to propose an integrated search-based approach to automatically testing from EFSM models.
9. The empirical evaluation of the proposed search-based approach on five EFSM cases studies.

1.7 The Thesis's Structure

The next subsections provide overviews of the five subsequent chapters of this thesis.

1.7.1 Chapter 2: Literature Review

This chapter highlights the importance of testing in general and particularly the importance of automatic testing. The chapter also describes the main testing

approaches: black-box testing and white box testing. Then, details are provided about methods used in testing and methods used to support testing. Search-based testing is also introduced together with the commonly used optimisation algorithms that are utilised. Furthermore, the chapter describes the importance of model-based testing and the two widely applied modelling approaches: finite state machines (FSMs) and extended finite state machines (EFSMs). Finally, the chapter reviews the approaches that are used to test from EFSMs, highlights the main problems associated with testing from EFSMs and motivates the use of search-based approaches to test from EFSMs.

1.7.2 Chapter 3: Generating Feasible Transition Paths (FTPs) for Testing from EFSM Models

This chapter highlights the problem of generating feasible transition paths (FTPs) and discuses the main limitations of the current EFSM testing approaches. Then, the chapter progresses by describing a search-based approach that can be used to generate FTPs from a given EFSM. The proposed approach uses new fitness metric to guide a search towards transition paths that satisfy a given test criterion and are *likely* to be feasible

1.7.3 Chapter 4: Automatic Test Cases Generation to Exercise Feasible Transition Paths

The problem of generating test cases (test data) to follow given FTPs in an EFSM is described. Then, the chapter presents a search-based approach to automatically generating test cases that can trigger given FTPs. The chapter also presents a statistical analysis of the relationship between the fitness metric of an FTP and the time that is required to trigger the FTP. Finally, it investigates the possibility of using the fitness metric to predict the time required by two test cases generators to exercise an FTP.

1.7.4 Chapter 5: Generating Feasible Transition Paths for Testing from EFSMs with Counter Problem

In this chapter, the causes of path infeasibility problem are highlighted again with the focus being on the counter problem, which can result in a transition path being infeasible. An approach to detect the counter problem in a given TP is then described together with the algorithm that achieves this. The proposed approach is then used together with that proposed in Chapter 3 to solve the counter problem.

1.7.5 Chapter 6: Conclusions and Future Work

This chapter summarises the main achievements by highlighting the work presented in each chapter. Then, the potential points for future work are proposed and discussed.

Chapter 2: Literature Review

2.1 Introduction

The advances in information technologies impact many aspects of life and the use of software systems becomes necessary to ensure the provision of better services if not the services themselves. The involvement of software systems varies from performing ordinary tasks such as text editing to managing critical tasks such as auto-piloting an aircraft. Since the reliance on software to perform some functions is increased, it is important to verify, as much as possible, that a piece of software correctly performs its intended tasks.

Testing is one method for verifying whether a given system is capable of correctly performing its defined tasks. However, testing is generally an expensive manual process. According to the National Institute of Standards and Technology, the estimated cost of software bugs and defects to the US economy is approximately $59.5 billion annually. If an improved testing infrastructure was used to try to remove software faults, this estimated cost could be reduced by at least one third (NIST, 2002, Harman and McMinn, 2009). Although the practice of software testing does not add extra functionalities to the final product, it constitutes up to 50% of the total software cost (Boehm, 1981, Hierons et al., 2009).

2.2 Why Automate Testing?

Because errors in systems can have severe consequences, software testing plays a vital role during the software development process. The testing phase is important in order to deliver software with an acceptable level of reliability. Although the process of testing cannot provide full confidence that the software is error-free (Dijkstra, 1970), it is still essential to ensure that software can meet a certain level of correctness.

Testing can be seen as a method to exercise the functions that the system is designed to carry out. Generally, this can be manually conducted by a tester with a set of scenarios that are designed according to some test criteria. This can give the tester the chance to detect defects since these may arise when the test cases are applied.

However, as software systems become more complex, manual testing becomes increasingly a poor option. It seems that there is a general agreement in the software engineering community that manual testing is imprecise, can be very expensive and error prone. Thus the aim is to minimise the human intervention in the testing phase in order to reduce the cost and enhance reliability (Korel, 1990, Elfriede et al., 1999, Tracey et al., 1998a, Derderian et al., 2006, Hierons et al., 2009).

Automating the practice of software testing is desirable and clearly this has been a subject of interest to many researchers along the past two decades to develop methods that can replace the conventional manual approach (Korel, 1990, Tracey et al., 1998a, Michael et al., 2001, Myers, 2004, Bertolino, 2007, Harman and McMinn, 2009, Ghani and Clark, 2009a).

2.3 Software Testing Preamble

In order to test a given system, there is a need to derive a set of test cases to exercise the system's functionalities. A test case can be defined as a set of inputs

that, if applied to the system under test, produces a set of outputs. The resultant outputs can then be used to verify whether the system performs as intended.

A test case can be seen as a possible scenario which a system is known to carry out. However, it is important to note that the number of possible scenarios that a system may perform is generally very large and can be considered, in most cases, to be infinite. For example, consider deriving test cases to exercise all the possible scenarios of a program that adds two one-digit binary numbers. This test can be achieved in practice since there are only four test cases to cover the entire set of possible scenarios. Unfortunately, this is not generally the case with the vast majority of systems. If the same program is performing, instead, the addition of two double numbers, then the number of possible scenarios is prohibitively large.

This motivates the use of a test adequacy criterion, which is a property that the test needs to satisfy in order to be considered adequate. Thus, given a system and a test adequacy criterion, a central task of testing is to derive a finite set of test cases that aims to satisfy the given test criterion. Such a finite set of test cases is referred to as a test suite. An example of a test adequacy criterion is statement coverage which requires that each statement of the program, to be tested, is exercised at least once (Rapps and Weyuker, 1985). If the statement coverage test criterion is considered for the program that adds two double numbers, then one test case may be adequate to satisfy this test criterion.

2.4 Testing Approaches

Deriving test cases to exercise a given system with the intention to reveal errors cannot be conducted arbitrarily. This is because usually there are a vast number of possible test cases. As a result, it is important to decide about a test strategy which can reduce the time and the cost associated with testing while providing confidence about software correctness (Myers, 2004). For this purpose, there are common approaches that are widely applied to conduct testing such as: black-box testing and white-box-testing. These two approaches are described in the next subsections.

2.4.1 Black-Box Testing

Black-box testing (also called functional testing) is an approach in which the tester has no knowledge about the internal system structure (the tester acts as an

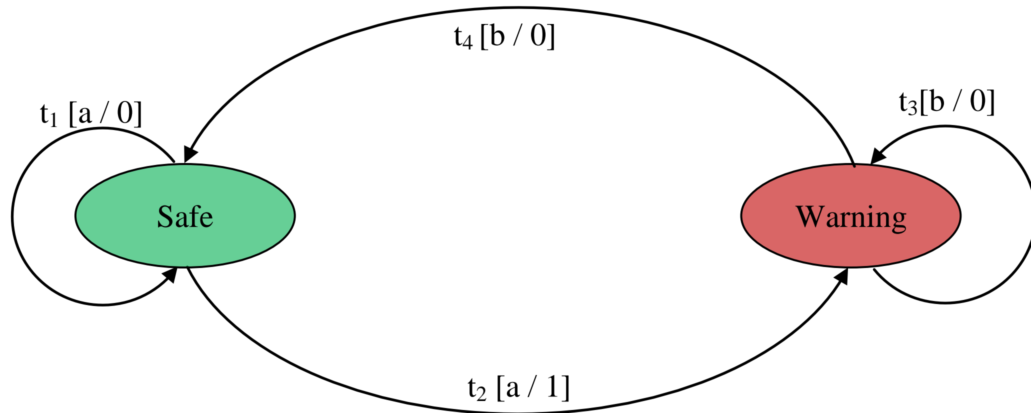


Figure 2.1: A fire alarm system represented as a directed graph

external observer) and the system to be tested is seen as a "black-box". In this approach, the system specification is used as a reference to derive test cases (usually by representing the specification in terms of a model) and to verify the performance of the implementation under test (IUT). When conducting black-box testing, there are only two aspects available to the tester: an input or a sequence of inputs (which can be a valid or invalid input(s) given as a test case) and the output(s) that the tester observes as the IUT responds to the input. The output is then used to verify whether the IUT performed as intended by referring to the IUT specification. Deriving test cases to conduct black-box testing of an IUT can be based on many test adequacy criteria. Such test criteria, for example state coverage, aim to cover aspects of the IUT. The state coverage criterion requires every state in the specification to be visited at least once (Tahat et al., 2001). For example, Figure 2.1 shows the specification of a fire alarm system where the system has two states: Safe and Warning and four transitions among these two states where each transition has a specific [input/ output]. Given an IUT of a fire alarm system, to conduct the black-box testing with the state coverage test criterion, a test case comprising the inputs ‹ *aa* › fires the transition sequence t_1t_2 that visits the system states. Thus, such a test case could be a candidate input to satisfy the test criterion.

The black-box testing approach is effective in testing the behaviours from the specification, however, black-box testing is poor at detecting extra behaviours that are included in the IUT but not in the specification. Such behaviours can be detected by using a white-box testing approach.

2.4.2 White-Box Testing

White-box testing (also called structural testing) is an approach in which the internal system structure (such as the algorithms and code) is available to the tester in order to derive test cases from the system. Particularly, a tester employs

```
function gcd (a,b : integer): integer;
var g: integer;
s. begin
1.   while (a ≠ 0) do
       begin
2.       g = a;
3.       a = b mod g;
4.       b = g;
     end;
5.   result = b;
e. end;
```

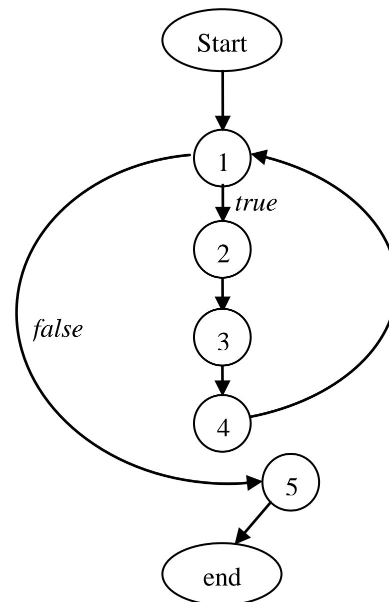


Figure 2.2: The greatest common divisor and its CFG

this approach to inspect a specific part of the system, using the knowledge regarding its internal structure to design test cases. White-box testing is thus an effective approach to test the behaviours that are implemented in the IUT. Nevertheless, the approach is poor at determining whether some behaviours have been missed since it does not consider the specification. White-box testing is usually based on the notions of program data flow and control flow.

A program control flow graph (CFG) is the 4-tuple (N, E, s, e) directed graph where:

- N is a finite set of nodes
- E is a finite set of edges
- s is the graph's entry node,
- e is the graph's exit node.

Each statement in the program P is represented as a node in the CFG. Some statements such as IF, WHILE and FOR have one entry and two exits called true and false branches. These nodes are referred to as branching statements since they have a control decision through their true or false exits. A node which is located on one of these two exits is control dependant on that branching node (Ferrante et al., 1987). An edge between two nodes n_1 and n_2 represents a control flow from n_1 and n_2. For example, Figure 2.2 shows the code of a Greatest Common Divisor GCD program for two positive integers together with the corresponding CFG. In this example, node 1 is a branching (While) node and thus nodes 2, and 5 are control dependant on node 1.

Given a program and a variable x within this program, a statement at which x appears can be an assignment to x or a use of x (or both). An assignment statement defines or updates the value of x and so x is said to be *defined* at such a statement. A use of x occurs when x is referenced in a predicate (a predicate use/p-use) or x is referenced in a computation that either updates the value of a variable or is produced as output (a computation use/c-use). Given a program path between two statements n_1 and n_2, if x is not defined after n_1 and before n_2 then the path from n_1 to n_2 is a definition clear path for x. If, in addition, n_1 is a definition of x and n_2 is a use of x, then statements n_1 and n_2 form a definition-use (*du*) pair for x and there is data flow dependence between n_1 and n_2 (Tai, 1984). For example, in Figure 2.2, node 2 is a definition node for variable g while node 3 is a c-use of variable g. Therefore, nodes 2 and 3 are a *du* pair for variable g.

In white-box testing, a test case is derived to cause a specific path of the program to be executed. Naturally, the input values included in the test case are a subset of values that the program can accept and thus belong to the program input domain D (D is usually a set of subsets where each subset represents the input domain for a given input). When paths are selected through a program, the

selection of these paths has to serve the testing purpose and so test cases are derived in accordance with this purpose. For example, when using statement coverage (each statement must be executed at least once by some test case), test cases should be derived so that specific paths are executed and the test criterion is satisfied.

Deriving test cases for a given program can be based on the control flow aspects such as branch coverage where all the branches have to be exercised at least once. Also, it can be based on data flow information such as all *du* paths where the paths that fall between a definition and a use of each variable have to be executed at least once.

As mentioned earlier, the manual derivation of a set of test cases is an undesirable practice. Instead, automating the test activities is generally preferred while a fully automated testing remains the optimal goal for the testing community (Bertolino, 2007).

2.5 Methods Used to Automate Testing

Automating some testing aspects can be performed by the means of using algorithms that are designed to produce test cases so that the human intervention during this phase can be either eliminated or kept minimal. The automation can also be assisted by employing some techniques that aim to simplify the test subject and consequently ease the testing process. The next subsections highlight the commonly used algorithms and techniques that can serve or support the purpose of automatic testing.

2.5.1 Symbolic Execution

Symbolic execution is an analysis approach which allows a program to be executed using a set of symbolic inputs (King, 1976). The execution here is not different from the normal program execution. However, the inputs are given in

terms of symbols, and thus the outputs of the program are symbols and expressions over these symbols. This is particularly useful to understand the relation between a given input and its associated output. Compared to the normal execution, the symbolic execution offers the possibility to trace back how each output was formed (which and how inputs are involved in this output). Nevertheless, the normal execution produces an output as a final value i.e. a numerical value and does not keep information about how the related inputs contribute to this output.

When generating program test cases using the symbolic execution, the problem of test cases generation can be reformulated to the problem of solving a set of algebraic expressions. These expressions are a result of symbolically executing a set of selected paths. For example, consider the path <1, 2, 3, 4, 1, 5> of the GCD program shown in Figure. 2.2. In order to execute this path symbolically, two symbolic inputs (a_0, b_0) can be used and so (a = a_0) and (b = b_0). Since this path requires running the statement at Node 1 two times, at the first instance, the condition at Node 1 should be true and thus ($a_0 \neq 0$). At Nodes 2, 3 and 4 the assignment statements will assign (g = a_0), (a = b_0 mod a_0) and (b = a_0) respectively. The condition at Node 1 should now be false so that the Node 5 can be executed, thus ((b_0 mod a_0) == 0) and finally, at Node 5 the assignment statement will be (result = a_0). Executing this path symbolically by using the inputs (a_0, b_0) results in two algebraic expressions:

$$a_0 \neq 0 \qquad (1)$$

$$(b_0 \bmod a_0) == 0 \qquad (2)$$

If these expressions can be solved, an actual test case that can exercise the path <1, 2, 3, 4, 1, 5> is generated. For this example, a possible solution can be ($a_0 = 2$, $b_0 = 4$). However, if the resultant algebraic expressions have no solution, then the path is infeasible (i.e. there are no test cases that can execute it).

There are many approaches that employ the symbolic execution technique to generate test cases for the purpose of program testing; examples of such work are reported in (Clarke, 1976, Darringer and King, 1978, DeMillo and Offutt, 1991). However, symbolic execution has some known limitations that can restrict its applicability in practice. One of these limitation concerns some code constructs

such as pointers and arrays. For such code constructs, applying symbolic execution is difficult. Although it is possible to use other code constructs to replace pointers and arrays, it is generally impractical to restrict such coding options. Furthermore, if a path contains a loop that is bounded by a variable instead of a constant, symbolic execution may iterate infinitely and so the execution time in this case is unknown (Michael et al., 2001). Even if the loop iteration is bounded by a number, the resultant algebraic expressions can be sometimes too complex to be handled.

2.5.2 Constraint Satisfaction

The constraint satisfaction problem (CSP) is the problem of finding suitable values from a given input domain to a finite set of variables where the values that these variables can accept, at the same time, are restricted by a set of constraints (Tsang, 1993). Formally, the CSP is defined (Dechter and Pearl, 1989) by three sets (V, D, C) where:

- V: is a finite set of variables
- D: is a finite set of input domains that represents the domain value of each element of V
- C: is a finite set of constraints on V or on a subset of V and C can be empty

The set of variables $V_1..V_n \in V$ is defined by their input domain $D_{1..}D_n \in D$ while a constraint $C_i \in C$ over a variable V_{i_1}, .., $V_{i_j} \in V$ is given as a subset of the Cartesian product $D_{i_1} \times .. \times D_{i_j}$ which determines the values which the variables can accept simultaneously. The CSP can have no, one or many solutions where any solution is simply the values that should be assigned to V so that the constraints hold.

Employing CSP for the purpose of program testing is called constraint-based testing (DeMillo and Offutt, 1991). In constraint-based testing (CBT), paths are selected (according to the test criterion i.e. the statement coverage) through the program under test, then these paths are executed symbolically and a set of constraints are derived for each path. Solving the set of constraints for a given path results in the required test case that can exercise this path.

There are techniques to solve the CSP such as the Constraint Logic Programming (CLP) (Jaffar and Maher, 1994), however, applying constraint-based testing requires applying symbolic execution to derive the constraints. Therefore the applicability of CBT techniques inherits the same limitations of symbolic execution applicability. Even if symbolic execution can be applied, the resultant constraints can sometimes be non-linear and therefore difficult to solve. For example, solving a set of non-linear constraints over integer variables is generally an undecidable problem (Zhang, 2008).

2.5.3 Search-Based Software Testing (SBST)

In the field of software engineering, there are problems that have a complex nature (such as testing) and conventional techniques such as linear and dynamic programming can be inefficient (Harman and Jones, 2001). Reformulating software engineering as a search problem (search-based software engineering) is an approach which tries to represent software engineering problems as optimisation problems (minimisation / maximisation). Then metaheuristic search techniques such as hill climbing, simulated annealing and evolutionary algorithms can be applied in order to find solutions that can be acceptable for the considered problem (Clark et al., 2003).

Search-based software testing (SBST) is therefore an approach of testing which reformulates software testing problems as optimisation problems. This reformulation serves the purpose of automatically deriving test cases that can satisfy a given test criterion. Reformulating the problem of testing as an optimisation problem is particularly useful for many reasons (Clark et al., 2003):

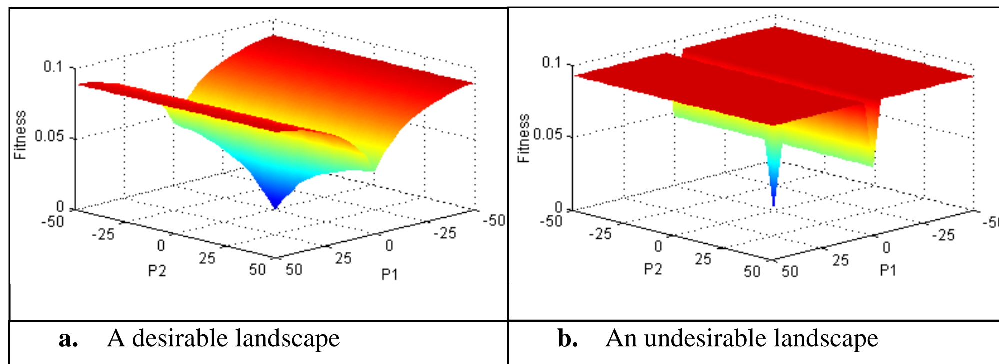
1. Exhaustive testing is generally not always possible due to the complex nature of the systems under test so that an alternative is required.
2. The input domain associated with testing is usually large and thus it is desirable to have a method that can effectively explore the input domain.
3. There are many problems that do not have a known complete solution and therefore approaching the problem by using search can provide new insights.

4. Generating candidate solutions can be made cheaper by using search. Furthermore, evaluating candidates solutions may be straightforward since there can be many standard metrics to be utilised to evaluate the potential solutions.

In order to apply a search technique to test from a particular system, a representation of the candidate solutions is required. Furthermore, it is essential to have a method, a fitness function, which can evaluate the candidate solutions.

The solution representation allows a search technique to manipulate candidate solutions. Generally, there are many ways to represent candidate solutions. If the candidate solutions are numbers, then it is possible to use binary encoding. In binary encoding, each number is represented by its equivalent binary value. For example, two inputs (7, 8) are represented in the form of binary as (0111, 1000). However, some neighbour values, which are close in the decimal form, are far in the binary form (e.g., 7 and 8). In order to avoid this problem, gray encoding can be used where each two successors differ only in one bit (Whitley, 1999). For example (7, 8) are represented by gray encoding as (0100, 1100). Other forms of encoding can also be used depending on the input domain of the problem. For example, if the inputs are integers then integer valued representation is possible. Similarly, real valued encoding is an option when inputs are double numbers.

In order to compare candidate solutions, a fitness function is required. The fitness function is a property that measures how good each candidate solution is in terms of the considered problem. The fitness function simply rewards each candidate solution by a positive numerical value which specifies haw far it is from being an acceptable solution. If the optimisation problem is a minimisation one, then candidate solutions which receive lower fitness values are better and acceptable solutions usually have a fitness value equal to zero. Similarly, for maximisation optimisation problems, candidate solutions that have larger fitness values are better.



a. A desirable landscape | **b.** An undesirable landscape

Figure 2.3: Examples of two fitness functions landscapes

Deriving a fitness function for a given problem is a central task when applying a search technique. For some problems, a fitness function can be derived directly. For example, consider minimising the function $\boldsymbol{f(x) = x^2}$:

$$\boldsymbol{Min(x^2);} \qquad \textit{where } \boldsymbol{x} \in [0..1000]$$

For such a problem, a fitness function can be similar to the function which describes the problem (i.e. $fitness(x) = x^2$) or with a slight difference (i.e. $fitness(x) = x$). However for some other problems, a fitness function can be derived from metrics that are based on some information such as statistical or source code information (Harman and Clark, 2004). For example, consider the metric that measures the statement coverage of source code. This metric can be utilised as a fitness function to measure the number of statements that are covered by a given test case (candidate solution).

Apart from its role in evaluating candidate solutions, the fitness function can also reveal information about how easy it is for a search to progress towards its goal. This is because the fitness function essentially acts as the search guidance. The guidance of a given fitness function can be viewed by plotting the fitness function landscape. If the landscape of a given fitness function is smooth and sloped towards the target, then the search is expected to receive adequate guidance and progress easily. However, fitness functions landscapes that have plateaux or flat surfaces may not provide the search with the adequate information to progress. Consequently the search may fail or become stuck in local optima.

For example, Figure 2.3 shows two arbitrary fitness landscapes (2.3.a and 2.3.b). The first landscape has a smooth sloped surface which allows search to progress easily towards its goal. However, the second landscape is mostly

dominated by a flat surface. In this case, many candidate solutions receive the same fitness value and so the search cannot distinguish these candidate solutions. In such a case, the search performance is likely to be similar to that of random search (McMinn, 2004, Kalaji et al., 2009c).

There are many works that study applying SBST technique to generate test cases from programs by considering the fitness function landscape. The main idea is to develop fitness calculations that are associated with desirable landscapes so that the search can receive adequate guidance. Examples of such works are reported in (Harman et al., 2004, McMinn et al., 2006, Kalaji et al., 2009c).

Unfortunately, the fitness function landscape cannot be easily plotted when the fitness function is related to more than two variables. This is because 3D plotting is limited to two variables and their fitness value is the third dimension. A function of more than two variables, n-dimensions, can be 3D-plotted by using techniques such as principal component analysis. The problem is related to mapping from n-dimensional space to 2D space by reducing the number of the variables. Principal component analysis reduces the number of correlated variables by transforming them to principle components that are uncorrelated variables (Jolliffe, 2002). Nevertheless, in the absence of an easy plotting technique when there are more than two variables, empirical evaluation can be used. In empirical evaluation, more than one fitness function can be derived for a given problem. Then, a search technique is applied to optimise the problem by using each available fitness function. The performance exhibited by the search technique on each fitness function can then be used to assess which fitness function provides better guidance.

When candidate solutions are represented and the fitness function is defined, a search technique can be applied in order to optimise the problem. There are many search techniques that can be used such as hill climbing, simulated annealing and evolutionary algorithms. These search techniques are explained in the next subsections.

2.5.3.1 Random Search

The random search is the simplest search technique and it randomly generates an input or a set of inputs from the specified input domain of the program. Then, these inputs are evaluated to determine whether they satisfy the test criterion. Generally, the random search is not affected by the fitness function landscape since it is not a guided search. This is because random search does not 'develop' candidate solutions but merely generates inputs that may be, by chance, acceptable solutions. Nevertheless, random search has been widely applied since it is considered as a baseline to compare the performance of other search techniques (Michael et al., 2001).

2.5.3.2 Hill Climbing

Hill climbing is a simple local search technique which works at one candidate solution at a time. The algorithm starts from a randomly generated point (a candidate solution). Then neighbours of this point are evaluated by using the fitness function. The fitness function determines whether one of these neighbours is better than the current point. If a better neighbour is found, then this becomes the current point and the algorithm iterates to search the neighbours of this current point. However, if no better neighbour can be found then the algorithm stops and returns the current point. Figure 2.4 shows a high level description of the hill climbing algorithm.

Hill climbing algorithm accepts only a neighbour which improves the fitness. Thus, this algorithm behaves as a hill climber (when searching the neighbours) on the fitness landscape. Because of this, hill climbing has a known limitation of becoming stuck in a local optimum rather than finding a global optimum. For example, consider a maximisation optimisation for a given problem with fitness function landscape that has two peaks: small and large. If the starting point was at the bottom of the small peak, the algorithm easily climbs the small peak and reports the top of this as the best candidate solution. However, it will not check the next peak because going downwards (from the current peak) will

1. **Generate a random point $x \in$ input domain D**
2. **Repeat**
3. **From x neighbours, $N(x)$, select $x' \in N(x)$ where $fitness(x') \geq fitness(x)$**
4. **$x = x'$**
5. **Until (a solution is found) OR (some stopping conditions are satisfied)**

Figure 2.4: Hill climbing algorithm for a maximisation optimisation problem with an input domain D

worsen the fitness of the current candidate solution and consequently the search is stuck.

Hill climbing, however, is simple, easy to implement and yields quick results for some optimisation problems that are associated with smooth and sloped fitness landscapes (Clark et al., 2003). Nevertheless, for fitness landscapes that include flat surface, the algorithm cannot effectively explore the input domain and is likely to report local optima.

2.5.3.3 Simulated Annealing

Simulated annealing (Kirkpatrick et al., 1983) is another well known local search technique which searches an input domain for candidate solutions that improve the fitness. However, it also has a feature that allows it to accept candidate solutions that may worsen the fitness. This particularly allows the algorithm to avoid being stuck in local optima (the problem found in hill climbing). The term *annealing* refers to the process of rendering a material to a desired structure or surface by heating it to a high temperature and then cooling it slowly. Similarly, when searching candidate solutions, at the beginning, the temperature T is initially high and almost any candidate solution can be accepted. This allows the search to have relatively free movement in the input domain. Then, the temperature is gradually decreased according to a cooling rate. At each temperature level a certain number of moves are allowed. While the temperature decreases, the search becomes more focused on accepting candidate solutions that improve the fitness. However, the search may also accept a number of candidate solutions that worsen

```
1.  Initialise  the temperature T = T0, the initial solution x = x0 ∈ input domain D
2.  Repeat
3.     Repeat
4.         From x neighbours, N(x), select x' ∈ N(x)
5.         Δ = fitness(x') - fitness(x)
6.         If (Δ < 0) Then
7.             x = x'
           Else
8.             r = random(0,1)
9.             If (r < e^(-Δ/T)) Then
10.                x = x'
               EndIf
         EndIf
11.    Until (the allowed number of moves at this temperature is reached)
12.    Reduce the temperature T = αT'  where α ∈ [0,1] is the cooling rate
13. Until (a solution is found) OR (some stopping conditions are satisfied)
```

Figure 2.5: Simulated annealing algorithm for a minimisation optimisation problem with an input domain *D*

the fitness (inferior solutions). The probability of accepting inferiors at a certain temperature *T* is controlled by the parameter P_{accept}:

$$P_{accept} = e^{-\Delta/T}$$

Where *T* is the temperature parameter; Δ is the difference between the fitness value of the current candidate solution and the fitness value of the considered inferior.

The lower the temperature, the less likely an inferior can be accepted and at freezing point, the algorithm behaves like a hill climbing and accepts only better candidate solutions. The idea of simulated annealing is to accept at the early search stages the candidates that worsen the fitness in a hope that these can lead to better neighbours in the final search stages. Figure 2.5 shows a high level description of the simulated annealing algorithm.

When simulated annealing is applied, it is important to properly select the initial temperature and the cooling procedure. The initial temperature should be high so that the search can explore many points of the input domain. Furthermore,

the cooling procedure should allow a slow temperature reduction so that inferiors can still be considered (Clark et al., 2003). If the temperature is reduced quickly, the search may not adequately consider the inferiors and mainly focuses on better candidates. This may lead the search to perform like the hill climbing search and so is likely to be trapped by local optima.

Simulated annealing has been successfully applied to automatically generate test cases for the purpose of structural testing. The aim was to overcome some of the obstacles that are related to the local search. The work of Tracey et al. (Tracey et al., 1998b, Tracey et al., 1998c) describes the approach of utilising simulated annealing for the purpose of automatic test cases generation.

2.5.3.4 Evolutionary Algorithms (EAs)

Evolutionary algorithms are optimisation techniques that adapt the evolution notion as a search mechanism. Genetic Algorithms (GAs) (Holland, 1975) are a class of EA, and probably the most well known form of EAs, inspired by natural selection principles and have been found to be powerful, simple, and sturdy.

Similar to other search techniques, applying a GA to an optimisation problem requires a solution representation (encoding). When solutions are encoded, each is called a *chromosome* and consists of components that are called *genes*. For example, let the initial set of solutions be integer values such as <7, 6, 8>. If binary encoding is performed, then <0111, 0110, 1000> represent the chromosomes. Any bit of a chromosome represents a gene with a value of either 0 or 1.

The GA cycle starts from an initial set of candidate solutions (referred to as a population) that are randomly generated from the problem's input domain. This allows sampling more points of the input domain and thus working on many candidate solutions simultaneously. Then the fitness of each individual within the population is evaluated to understand how 'fit' this individual is. The fitness value assigned to each individual influences its survival chance by being selected as a parent.

Then *selection* based on fitness is made to perform 'breeding'. There are many selection methods that can be applied such as roulette wheel and ranking

(Sadiq and Habib, 1999). Roulette wheel selection was originally used by (Holland, 1975) and it mimics the natural selection principle in that it provides better chance for fitter chromosomes to be selected as parents. In this method, each individual is allocated a portion of the wheel where the portion size is related to the fitness value of this individual. Then the wheel is spun and an individual x_i is selected according to the probability:

$$P_{selection} = fitness(x_i) \,/\, \sum_{j=1}^{m} fitness(x_j)$$

Where m is the population size.

Roulette wheel selection prioritises individuals which have higher fitness and so may cause the search to narrow down too quickly to particular solutions (super individuals). This, in turns, affects the population diversity and blocks the algorithm from investigating different points in the search space. Consequently, the search progress may be negatively impacted. Some studies argue that there should be a constant selection pressure on all population individuals in order to allow individuals with lower fitness to be selected. Therefore, the population diversity can be maintained and the search can still explore new points in the search space (Whitley, 1989). Linear ranking is a selection method that suits this purpose. The method functions by sorting individuals in a rank depending on their fitness values. In this rank, the least fit chromosome comes in the first position of the rank (an index $i = 1$) while the fittest chromosome is placed at the last position of the rank (an index $i = m$). Then, depending on a selection pressure SP value, where $1 \leq SP \leq 2$, the selection can be biased and an individual chance of being selected is given by the probability:

$$P_{selection} = (2\text{-}SP)/m + 2i(SP\text{-}1)/m(m\text{-}1)$$

For example, if $SP = 2$, then fitter individuals are associated with better chance to be selected. Similarly, if $SP = 1$, then all individuals have equal chance of being selected.

After selection is made, breeding is performed between the selected individuals to produce new individuals. This is accomplished by applying a *crossover* operator that acts on two individuals to produce two new individuals. There are several approaches to crossover (Sadiq and Habib, 1999) including one-

1. **Initialise population size $m = m_0$**
2. **Initialise population pool P and offspring pool F**
3. **Generate randomly m individuals $\in$ input domain D and insert them into P**
4. **Repeat**
5. **For i = 1 to m do**
6. **Calculate the fitness of individual $P(i)$**

   endFor
7. **While ($size(F)<m$) do**
8. **Select two individuals ind_1, ind_2 from P**
9. **Crossover(ind_1, ind_2) to produce two offspring ind_1', ind_2'**
10. **Insert the two offspring ind_1', ind_2' in the offspring pool F**

    endWhile
11. **Individuals from P and F are used to construct new pool P'**
12. **For i = 1 to m do**
13. **Mutate the pool P'**

    endFor
14. **$P = P'$**
15. **Reset(F)**
16. **Until (a solution is found) OR (some stopping conditions are satisfied)**

Figure 2.6: High level description of a basic genetic algorithm

point crossover. One-point crossover operates by choosing a random position on the chromosome's bit string, and then the substrings before that position are kept while the tails are swapped (Srinivas and Patnaik, 1994). For example, if the two parents' chromosomes are P_1 and P_2 with crossover point at position 4, then C_1 and C_2 are the offspring chromosomes.

P_1: 011|00 ⟹ C_1: 011|11

P_2: 101|11 ⟹ C_2: 101|00

In order to maintain population diversity, new characteristics are occasionally injected by applying *mutation*. Mutation acts on one chromosome at a time where it randomly changes the values of some of the chromosome's genes (Srinivas and Patnaik, 1994). For example, the chromosomes C_1 above might become C_1' after mutating the bits on positions 1 and 5.

C_1: 01111 ⟹ C_1': 11110

The GA cycle (evaluation, selection, breeding and mutation) yields new individuals (offspring) and selection is used to obtain a new generation (population) from the previous population and the offspring. The population undergoes a number of updates until satisfying one of the stopping criteria such as finding the solution or reaching a maximum number of generations. Figure 2.6 shows a high level description of a basic genetic algorithm.

2.5.3.5 Evolutionary Testing

Evolutionary testing (ET) is a technique that employs EA to automatically generate test cases. Test case generation is represented as a minimisation problem where the lower the fitness of a solution the better it is and the optimal solution(s) will have a fitness equal to zero.

When applying ET to generate test cases for a test target i.e. a program, a fitness function is required that corresponds to the test adequacy criterion (a property that a test must satisfy in order to be considered sufficient). Many test adequacy criteria require that a set of structures in the code or model is covered in testing. For example, consider a test purpose that requires all of the statements in the code to be exercised (covered) in testing (statement coverage). If the test purpose considers the branch coverage criterion, then all the branches in the subject program need to be taken (covered). Test cases generation can involve a sequence of phases where in each phase a single branch is considered. In this case, a fitness function that depends on branch distance can be used in order to evaluate a test case.

Branch distance measures how close a particular input was to executing the target branch that is missed. The work of (Korel, 1990) stated a set of rules to measure the branch distance for each possible branch type. A guard (predicate) between two inputs A and B can be generally expressed as (A *gop* B) where *gop* is a guard operator such as $\{>, <, =\}$. For example, the approach of (Korel, 1990) calculates the branch distance for the guard ($A < B$) as $A - B$. If ($A - B < 0$), then the guard is satisfied. In Korel's method, it is necessary to check whether the branch distance value is positive, negative and / or equal to zero according to a list of predefined relations to understand whether the guard is satisfied. The work of

Guard	Fitness calculation
Boolean	if *TRUE* then 0 else k
$a = b$	if $abs(a - b) = 0$ then 0 else $abs(a - b) + k$
$a \neq b$	if $abs(a - b) \neq 0$ then 0 else k
$a < b$	if $a - b < 0$ then 0 else $(a - b) + k$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + k$
$a > b$	if $b - a < 0$ then 0 else $(b - a) + k$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + k$
$\neg a$	Negation is moved inwards and propagated over a

Table 2.1: **Tracey et al. fitness calculations for different types of guards. The constant k, $k > 0$, is added when the guard is not satisfied.**

(Tracey et al., 1998a, Tracey et al., 1998b) provides a remedy for this by proposing a set of rules to calculate the branch distance. In Tracey et al. approach, the branch distance is either positive when the guard is false or zero when the guard is satisfied. This is achieved by adding a constant value $k > 0$ to the branch distance whenever the guard is evaluated to false. For example, the branch distance for the guard ($A < B$) is 0 if $A - B < 0$ otherwise it is $(A - B) + k$. In this way, the lower the branch distance is the closer A is to B and the closer the test is to taking the branch. The full list, proposed by (Tracey et al., 1998a, Tracey et al., 1998b), for different types of guards and their branch distance computations is provided in Table 2.1.

Naturally, programs can have nested guards, for example an IF statement could be contained in a loop. In this case, a fitness function which only employs branch distance is not sufficient to detect which branch was not satisfied. Consequently, the search does not receive adequate information to progress. In order to identify which branch has caused the execution flow to divert, the fitness function should include extra information. This is given in terms of approximation level or approach level (Wegener et al., 2001).

Approach level measures how close an input was to executing the structure under test. A central notion to approach level calculation is a *critical node* which is a branching node at which the execution flow may divert. Approach level is calculated by subtracting 1 from the number of critical nodes away from the target node at which the computation diverges (Equation 2.2).

```
int foo (int x, int y, int z )
1. {  If (x > 10) Then
2.      If ( y < x ) Then
3.        If (z = x+ y) Then
4.           Return 0; // Target
   }
```

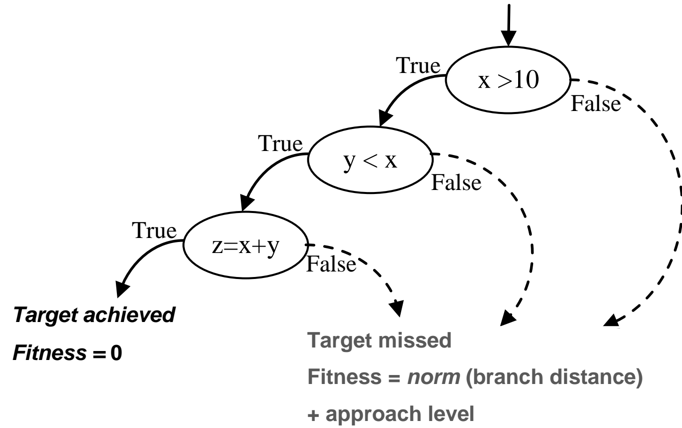


Figure 2.7: An example of branch distance calculation by using Wegener et al. approach. A Critical node has a false branch (represented by a dashed arrow) that diverts the execution flow.

$$\textbf{\textit{norm}}\ (branch_distance) = 1 - 1.05^{-(branch_distance)} \quad (2.1)$$

$$\textbf{\textit{approach level}} = NumOfCriticalNodesAwayFromTarget - 1 \quad (2.2)$$

$$\textbf{\textit{fitness}} = approach\ level + norm\ (branch_distance) \quad (2.3)$$

Since it is necessary that the branch distance of the upper IF statement is always greater (worse) than the ones in a lower level, the branch distance of each IF statement is normalised, using the *norm* function, to a value in the range of [0..1] (Equation 2.1). The normalised branch distance is then added to the approach level of that branch to form the fitness value of the test case (Equation 2.3). As a result, a test case that achieves more guards (longer path) will have a better (lower) fitness than a test case that achieves fewer guards. For example, Figure 2.7 shows a function with three nested guards together with the CFG. In order to test this function, three inputs are required. By using the Wegener et al. approach, the fitness calculation should be performed at each critical node. The approach level at the first critical node is 2, at the second node is 1 and at the third node is 0.

There are many studies that employ SBST approaches to conduct testing. These studies have demonstrated that SBST approaches are robust and effective in test automation. Examples of these studies are reported in a survey by (McMinn, 2004) which focused on search-based test data generation.

2.5.3.6 Model Checkers

A model checker is a formal verification approach that can automatically verify whether a system requirement conforms to the specification (Clarke, 2008). A system requirements or a design of the system is used to describe a model of the system. Then, a property that the system is known to perform is derived from the system specification. Both the model and the property are the inputs to the model checker which verifies whether the system has this property. If the property is violated, the model checker provides a counterexample that illustrates why the system does not satisfy this property. The counterexample is useful since it shows how to detect and correct the faults which caused the property to be violated.

The system requirements that are fed to the model checkers are the state transition graph (Kripke structure) whereas the property to be verified is represented as a temporal logic formula. A Kripke structure (Clarke, 2008) is a representation of the system behaviour in terms of states and transitions among these states. The temporal logic formula is to express the property in terms of a scenario that can be true or false depending on the time. A model checker verifies whether the Kripke structure has a path so that the temporal logic formula, the property, is true. A path in Kripke structure represents a possible scenario of the system. For example, all the values that a variable may have in a given system are represented as states in Kripke structure.

The application of model checkers to software testing can be seen by the model checkers ability of providing counterexamples. Counterexamples describe the fault cause and how it can be corrected and so they can be treated as test cases. The main idea is to generate a counterexample that can be related to the intended correct behaviour. This can be achieved by model-checking the system against a negated temporal logic property that the system is known to satisfy (Hierons et al., 2009). It is possible to express test adequacy criteria such as statement coverage in terms of a temporal logic formula. Since the property is negated, the model checker will produce a counterexample that shows where the error is. These counterexamples are the desired test cases. However, these test cases have to be mapped down to the actual system since they are derived at the model level (Hierons et al., 2009).

Although model checkers can effectively serve the testing purpose by producing counterexamples, the problem is how these counterexamples can generally be produced in a systematic way so that they are adequate to satisfy the test criterion (Fraser et al., 2009). Furthermore, search-based testing may outperform the performance of model checkers, for example, a study by (Nilsson et al., 2006) found that when testing from dynamic systems, the search problem become more difficult, however, the performance of a GA search outperformed the model checker on the considered problem. Also, a recent study by (Wenzel et al., 2009) states that using model checkers to generate test cases is expensive and it is more expensive than using heuristic techniques. Moreover, since the underlying model, which is an input to the model checker, is the state transition diagram, the problem of state explosion cannot be ruled out. The state explosion problem concerns the final number of states in the considered model. For a given program, the number of states grows exponentially, being of $O(V^N)$, where N is the number of the variables and V is the number of possible values that each variable can have. For example, a program that has 10 integer variables in the range [-1000..1000] results in 2000^{10} states. Actually, state explosion problem is one important obstacle in the domain of model checker applications (Clarke, 2008). A recent survey about the applications of model checkers to the domain of software testing is provided by (Fraser et al., 2009).

2.6 Methods Used to Support Testing

Although the techniques highlighted in the previous sections are very useful to automate many aspects of the testing process, test subjects can be complex. This can impose difficulties when test generation techniques are applied in order to derive test cases. For this purpose, it is desirable to simplify the test subject in order to ease the testing process. For example, in order to apply symbolic execution to a test subject that includes arrays, a transformation can be applied so that the program is partially re-coded to replace arrays with other code constructs (testability transformation). Similarly, if the test is focused on a particular variable

or a path within a program, then other parts of the program that have no effect can be removed so that testing considers a particular part of the program (slicing).

Slicing and testability transformation are two commonly used approaches to aid the testing process. These two approaches are described in the next subsections.

1. Read (x, y , z);	**1. Read (x, y , z);**
2. z = x * y;	**2.**
3. while (x < 100)	**3. while (x < 100)**
{	**{**
4. x = x-1;	**4. x = x-1;**
5. y = y + x;	**5.**
6. z := z * x	**6.**
}	**}**
Original program	**A Slice of variable *x* at node 4**

Figure 2.8: An example program slicing. The original program was used to produce a slice for variable *x* at node 4.

2.6.1 Slicing

Program slicing is a technique to extract certain parts of a program which affect the value of a set of variables *V* of interest at a specific program point *n* while preserving the program syntax (Weiser, 1981). Any statement that does not affect the value of variables in *V* at *n* can be deleted making the resultant program, the slice, smaller.

Slicing therefore simplifies the test subject by producing parts of the test subject that are related to the test. For example, if a test considers the value of variable *x* at the program node 4 in the program shown in Figure 2.8, then slicing can be applied to remove other statements that do not affect the value of variable *x*. Consequently, the test can focus on a slice of the program which is usually smaller and thus easier to handle than the complete program.

Slicing can be static or dynamic (Harman and Hierons, 2001). Also, amorphous slicing (Harman and Danicic, 1997) is a method of slicing that does not preserve a program syntax but it allows many forms of transformations, in

addition to statement deletion, in order to produce much simpler and smaller slices. A survey of program slicing and applications is provided by (Xu et al., 2005).

2.6.2 Testability Transformation

A testability transformation (TeTra) (Harman et al., 2004, Harman et al., 2002) is a source-to-source code transformation that intends to improve the efficiency of a test case generation approach. The transformed test subject does not replace the original test subject; it is only used to derive test cases which then can be applied to the original test subject. In this way, the transformed test subject can be discarded once the required test data are produced.

TeTra thus does not change the original program code which developers have produced and maintained. The transformed program code is merely produced by testers for the purpose of enhancing testing. The importance of the transformed test subject is to help producing test cases that are difficult to be produced directly from the original test subject.

TeTra has been studied to enhance testing for both structured and unstructured programs (Harman et al., 2004, Hierons et al., 2005). The applications of TeTra are, particularly, the transformation of test subjects so that the associated fitness function calculation is enhanced. This was mainly shown by comparing the fitness function landscapes before and after TeTra was applied to a given test subject. Examples of such works are (Baresel et al., 2004, McMinn et al., 2009, Kalaji et al., 2009c). However, the limitation of TeTra is that it is difficult to decide on a suitable set of steps to be applied. For example, TeTra can involve a variety of steps to transform a test subject such as recoding, pushing statements from many functions into one function and a decision on which steps to be applied to a particular problem can be difficult.

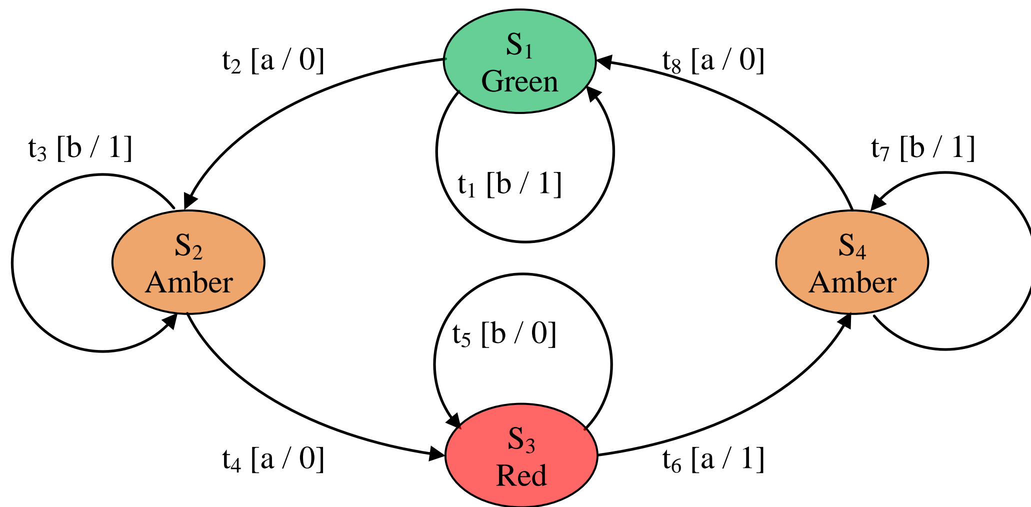


Figure 2.9: An FSM of a traffic light control system represented as a directed graph.

2.7 Model Based Testing

A model of a system can be viewed as an approach to capture the system aspects which are of interest. Generally, systems tend to be too complex to be understood and thus a model can help to simplify these aspects by providing some level of abstraction by mathematically representing these aspects of a system (Beizer, 1990).

For software, a model can be seen as a representation of the system behaviours such as the input / output, data flow and control flow behaviours. Model based testing is an approach that allows test cases to be derived from a model that is generally constructed from a reference specification. Model based testing has proven to enhance and simplify the testing process and thus reduce the cost associated with testing (Apfelbaum and Doyle, 1997, Dalal et al., 1999, Hierons et al., 2009).

A system's behaviours can be modelled by many approaches (El-Far and Whittaker, 2001) such as statecharts (David and Amnon, 1996), Unified

Modelling Language (UML) (OMG, 2002), Specification and Description Language (SDL) (ITU-T, 1994), Estelle (Turner, 1993), Labelled Transition System (Tretmans, 2008), Finite State Machine (FSM) and Extended Finite State Machine (EFSM). Among these approaches, the finite state machine and extended finite state machine are two modelling approaches that are commonly used for the purpose of test cases derivation (Petrenko et al., 2004).

2.7.1 Finite State Machine (FSM)

A finite state machine (FSM) is a Mealy machine (or transducer such as Moore machine) which has a finite set of states, finite set of inputs, and a finite set of outputs. An output is produced upon state transition and this occurs when an input is applied to the machine (Lee and Yannakakis, 1996). Formally, the finite state machine is defined as a 6-tuple ($S, s_0, I, O, \lambda, \delta$) where:

- S is a finite set of the states that the machine can be in
- s_0 is the machine initial state where $s_0 \in S$
- I is a finite set of inputs
- O is a finite set of outputs
- λ is the output function
- δ is the transition function among the machine's states

When the machine is at a given state $s \in S$, upon receiving an input $i \in I$, a transition t is fired, the machine moves to the state s' by using the transition function $\delta(s, i) = s'$ and outputs $o \in O$ by using the output function $\lambda(s, i) = o$. A transition t is defined by the 3-tuple ($s_s, s_e, i/o$) where:

- $s_s \in S$ is the transition start state
- $s_e \in S$ is the transition end state
- i/o where $i \in I$ is the input to be applied to the machine in order to fire the transition and $o \in O$ is the output that is produced upon the state transition

An FSM can be represented by a directed graph G (V, E) where the graph vertices V represent the machine states and the graph directed edges E represent the state transitions (Aho et al., 1991). Each directed edge has a label that specifies the name of the transition and the input / output associated with this transition.

For example, Figure 2.9 shows an FSM state transition diagram for a traffic light control system. In this figure, there are four nodes that represent the machine states. Also, there are eight edges that represent the transitions among these states. Each transition has a label that shows the input / output of this transition. If, for example, the machine current state is s_1 and it receives an input *a* then transition t_2 is fired and the machine outputs ‹0› and moves to state s_2.

For a given FSM, two states s_1 and s_2 are *distinguishable* when there is an input sequence α that if applied, while the machine is at state s_1, the output will be different from that produced when α is applied while the machine is at state s_2. This can be formally stated as these exists α such that $\lambda^*(s_1, \alpha) \neq \lambda^*(s_2, \alpha)$. However, when the outputs are the same for all α, then these two states are said to be equivalent. Therefore, two FSMs M_1 and M_2 are said to be equivalent if their initial states are equivalent.

An FSM is said to be:

- *deterministic*: if for any group of transitions leaving the same state, applying an input $i \in I$ can fire only one transition of this group
- *non-deterministic*: if it is possible that more than one transition could be fired by the same input at a given state
- *completely specified*: if for any state $s \in S$, applying any input $i \in I$ to the machine will fire a specific transition
- *partially specified*: if for any state $s \in S$, not every input $i \in I$ can fire a transition
- *initially connected*: if it is possible to visit any state $s \in S$ by starting from the initial state s_0 through applying an input sequence α
- *strongly connected*: if there is an input sequence α that can move the machine between any two states $s_1, s_2 \in S$.
- *minimal*: if there is no equivalent FSM with fewer states

An FSM model can successfully capture the control behaviour of a system such as a traffic light system and so the FSM has been widely used to model the control part of systems such as sequential circuits, communication protocols and web applications (Browne et al., 1986, Holzmann, 1991, Aho et al., 1991, Sidhu and Leung, 1989, Lee and Yannakakis, 1996, Andrews et al., 2005).

2.7.1.1 Conformance Testing

Conformance testing is an approach that aims to verify that the behaviours of an implementation under test (IUT) conform to its reference specification. Conformance testing considers an IUT as a black-box where the internal system

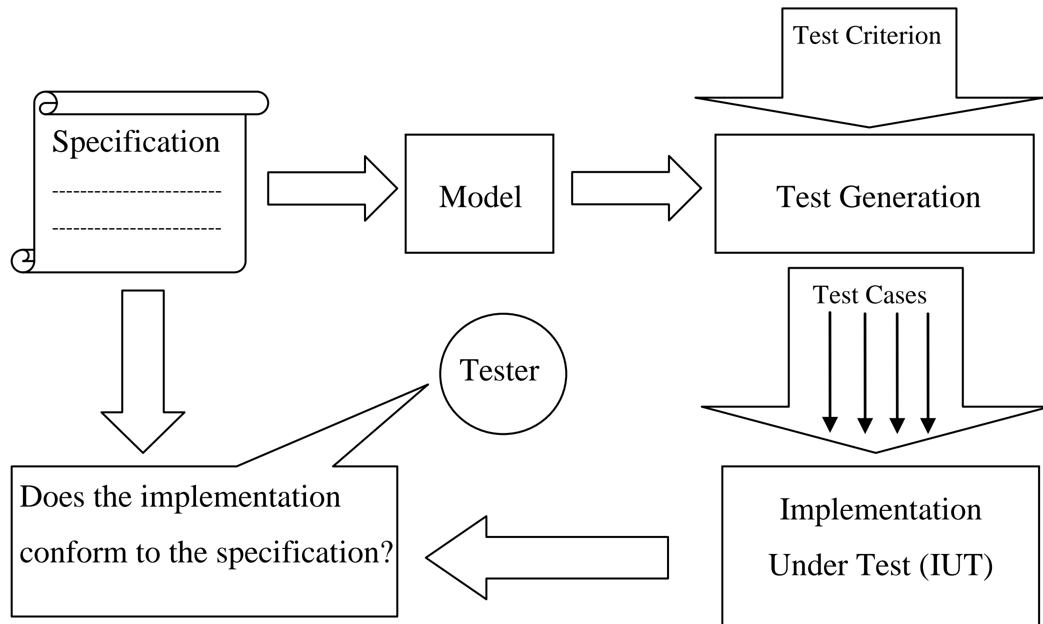


Figure 2.10: Conformance testing approach

structure is unknown to the tester and only the input / output signals are observable (see Figure 2.10).

Therefore, when performing conformance testing, there is a need for a reference specification of the IUT against which the IUT's behaviours can be verified. Usually, this specification is represented in terms of a model. For example, if the test concerns the control aspects of a system, then the reference specification can be represented by an FSM model.

2.7.1.2 Testing From an FSM

Here the IUT is a black-box and the tester has a reference specification of the IUT as an FSM. In conformance testing, the tester might exercise a specific transition t (s_s, s_e, i/o) of the implementation under test (IUT) and thus verify that the behaviour of this transition conforms to the reference specification modelled as FSM. This can be performed through three main steps:

1. *reaching the start state:* so that the desired transition *t* to be tested can be fired
2. *firing a transition*: by applying the specific input *i* and thus observing that the expected output *o* is produced otherwise there is a fault
3. *verifying the end state:* so that the transition did not move the machine to a different state (a wrong state) otherwise there is a fault.

For the first step, it is always possible to derive an input sequence which can move the machine from the initial state to the desired one (Kohavi, 1978, Rivest and Schapire, 1993). In order to make the testing process easier, it is useful to have a reliable *reset* function that can bring the machine to its initial state every time it is applied. Such a function can be simply seen as switching the machine off and on (Hierons, 2004). This is particularly useful when applying conformance testing since the process is naturally iterative.

When the transition's start state is reached, the desired transition can be fired by applying its specific input. If there is a fault, then the expected output *o* will be different from the one stated in the reference specification. This kind of fault is easy to detect since it is based on the observation. Finally, when the transition is fired, it is important to verify that the transition's end state is actually the correct one. This is known as the *state identification* which verifies that the machine is at a specific state. State identification has been addressed by three well known approaches:

1. Distinguishing Sequence (DS) (Gonenc, 1970)
2. W- method or the characterisation sequence (Chow, 1978)
3. Unique Input Output (UIO) (Sarikaya and Bochmann, 1984)

The distinguishing sequence *DS* is an input sequence that if applied to an FSM while the machine is at a given state, the output will be different from all other outputs produced by applying *DS* to the machine while it is at any other possible state. Thus, this method can identify each state of an FSM, however, there may be no *DS* for a given FSM.

The W-method is a set (W-set) of inputs that can identify all the states of an FSM by producing a different set of outputs at each state. Thus, W-method

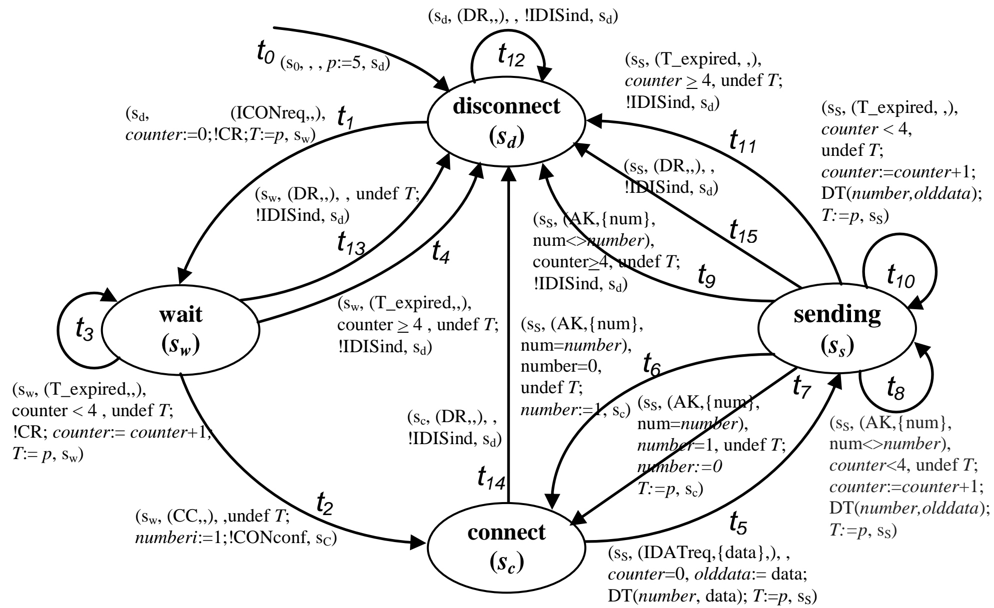


Figure 2.11: The Inres Initiator EFSM model

requires firing the same transition for every input sequence from W-set and this may increase the test execution time.

The Unique Input Output (UIO) method is a set of input sequences where each input sequence can identify only one state of the machine by producing a unique output. Thus, a UIO of a given state must produce an output that is different from all other outputs that are produced when the same UIO is applied to machine while it is at any other possible state.

When testing from an FSM, the focus is on testing the control behaviours of the system that is modelled by the FSM. However, when the subject system has a data part in addition to the control part, the FSM model cannot represent the data part, such behaviours can be represented by the Extended Finite State Machine (EFSM) model.

2.7.2 Extended Finite State Machine (EFSM)

The Extended Finite State Machine (EFSM) can be considered as a solution to the limitations found in the FSM model (FSM can capture only the control behaviours of a system). An EFSM model can represent both the data and control parts of a

system since it has an internal store (memory) to represent the data aspects. In this way, the EFSM machine is a Mealy machine extended with internal variables, predicates, and operations.

An EFSM can be defined as a 6-tuple (Ural and Yang, 1991) (S, s_0, V, I, O, T) where:

- S is the finite set of logical states
- $s_0 \in S$ is the initial state
- V is the finite set of internal variables
- I is the set of input declarations
- O is the set of output declarations
- T is the finite set of transitions

The transition $t \in T$ is represented by the 5-tuple (s_s, i, g, op, s_e) in which:

- s_s is the start state of t
- i is the input where $i \in I \cup \text{Nil}$
- g is the guard and is either Nil or is represented as a set of logical expressions given in terms of variables in V' where $\phi \neq V' \subseteq V$
- op is the sequential operation which consists of simple statements such as output statements and assignment statements
- s_e is the end state of t

In an EFSM model, there is a set of variables. One variable in particular (this can be a tuple of values) is used to represent the machine state and is called *state* or *major state* in order to differentiate it from the other variables which are called *context variables*. The state variable is used to represent the logical state, such as Idle, Connect, Wait for connection and so on, whereas other machine data such as port number, acknowledgement number and sequencing numbers are stored in context variables.

In the EFSM model, in order for a transition to be fired, there may be some required input values and associated conditions on context variables to be satisfied. According to this, EFSM transitions can be classified into two types: spontaneous and non-spontaneous transitions. Spontaneous transitions do not require input values in order to be taken; however, non-spontaneous transitions depend on some input value(s) in order to be initiated. Both spontaneous and non-

spontaneous transitions can be partitioned to conditional and unconditional transitions depending on whether or not there exists one or more associated guards to be satisfied before a state transition can occur.

A state transition occurs when one of the machine's transitions is fired. If the machine is at state s_s then transition $t = (s_s, i, g, op, s_e)$ can be fired if the input i is received and the guard g is satisfied. If this happens, the operations in *op* are executed and the logical state becomes s_e. Both g and *op* can refer to input parameters and context variables.

An EFSM is a deterministic machine if for any group of transitions with the same input that leave a state, it is not possible to satisfy the guards of more than one transition in this group at the same time otherwise the machine is *non-deterministic* (Shih et al., 2005). Figure 2.11 shows an EFSM example of Inres initiator protocol. The work on this thesis considers EFSMs that are deterministic without spontaneous transitions.

The EFSM is a powerful generic modelling approach that can capture almost all the aspects of a system and has been commonly used for the purpose of modelling and testing (Petrenko et al., 2004, Lorenzoli et al., 2008). It has been used to testing UML Use Cases (Sinha et al., 2007), UML state-charts (Kim et al., 1999, Shuhao et al., 2004), communication protocols (Dssouli et al., 1999), SDL specification (Ural et al., 2000, Wong et al., 2009), Statecharts and Z specifications (Hierons et al., 2001) and web services (Keum et al., 2006).

2.8 Testing From EFSMs

Generally, a test suite consists of either one or many test cases. In the first case, a single test case is applied to the initial state of an IUT. However, in the second case, a test suite contains more than one test case and the IUT is brought back to its initial state every time a test sequence is applied (Petrenko et al., 1996). Deriving a test suite for testing from an EFSM can be based on a model-based testing criterion. There are four commonly used model-based testing criteria that can be used when testing from an EFSM (Tahat et al., 2001):

1. *State coverage*: A test suite contains test cases that allow visiting each state in the model at least once.
2. *Transition coverage*: A test suite contains test cases that exercise each transition in the model at least once.
3. *Path coverage*: a test suite contains test cases that exercise all the possible paths in the given model at least once. Generally, this test criterion is limited to the models that do not exhibits cyclic behaviours otherwise there can be infinite number of paths which cannot be handled in practice.
4. *Constrained path coverage*: This criterion is a special case of the path coverage criterion where each path in the model should be exercised at least once. However, each transition in the model should not be fired more than *n* times. The constraint *n* imposed over the number of allowed executions of each transition avoids the problem of infinite number of paths found on the path coverage criterion.

For any test criterion, a test suite that tests from an EFSM comprises a set of test cases that is, in essence, a set of transition paths (TPs) through the considered EFSM together with the transition inputs. Therefore, the problem of deriving a test suite to test from an EFSM can be seen as the problem of finding a set of TPs that achieves the test criterion.

Because the EFSM model combines both the control and data aspects of a system, the problem of deriving a set of TPs to achieve the test purpose is a challenging task. Mainly there are two problems that make the process of generating a set of TPs difficult.

The first problem is related to the feasibility of a given TP. An EFSM transitions have generally guards and actions over a set of context variables. The actions of a given transition within a TP may assign values to some context variables. A later transition within the same TP may have guards that require different values of the previously set variables. Here a conflict may occur between the assignments of the earlier transition and the guards of the later transition. This conflict results in this TP being infeasible. An infeasible TP is not desired in a test suite since it simply cannot be exercised and consequently the test does not achieve its intended purpose. Nevertheless, determining if a given TP is feasible

in advance is generally an undecidable problem (Dssouli et al., 1999, Hierons et al., 2009). Furthermore, developing of good methods to approach the infeasibility problem is an open research problem (Duale et al., 1999, Duale and Uyar, 2004).

The second problem is related to exercising a given set of feasible TPs (FTPs) within a test suite. An FTP can require a sequence of inputs that are related to the interaction parameters of each transition included in this FTP. Furthermore, there may be guards over the inputs which must be satisfied so that the FTP can be exercised. Therefore, a suitable set of test data is required to cause each FTP to be taken. Generally, finding a suitable set of test data that can trigger a given FTP is a substantial task (Ural and Yang, 1991) since the input domain is usually large and the suitable values constitute just a small subset of the input domain.

These two problems associated with EFSM testing can be eliminated if it is possible to test an EFSM from an FSM point of view. Since testing from an FSM model has been well studied with many approaches that are available, it is useful if these techniques can be applied to EFSM as well. Generally, applying FSM-based approaches to an EFSM requires converting an EFSM to an FSM. There are two techniques to convert an EFSM to an FSM. The first technique abstracts out the data from an EFSM so the resultant is an FSM. The second approach expands an EFSM to become an FSM.

If an EFSM's data part is abstracted (Hierons et al., 2001) then the resultant is an FSM model. Clearly, the approach then considers only the control aspect of an EFSM. In this way, a set of TPs can easily be derived from the corresponding FSM since there is no feasibility problem in the FSM model. However, when these TPs are mapped down to the original EFSM, the feasibility problem may occur. Consequently, not all the TPs that are derived in this way are feasible.

When the data part of an EFSM is expanded (Hierons and Harman, 2004), the EFSM is converted to an FSM. In this method all the TPs that are derived from the corresponding FSM are feasible in the original EFSM. However, the problem associated with this approach is the large number of states in the resultant FSM which may easily lead to the state explosion problem.

2.8.1 Related Work of Testing from EFSMs

Many test generation approaches for systems modelled as EFSMs appear in the literature (Chanson and Zhu, 1993, Cheng and Krishnakumar, 1996, Dahbura et al., 1990, Duale and Uyar, 2004, Duale et al., 1999, Hierons et al., 2004, Lefticaru and Ipate, 2008, Petrenko et al., 1996, Ramalingom et al., 1996, Ramalingom et al., 2003, Sarikaya et al., 1987, Ural and Yang, 1991, Derderian et al., 2005, Bourhfir et al., 1996, Chanson and Jinsong, 1994, Derderian et al., 2010).

An approach to generate a unified test sequence (UTS) for EFSM models is presented in (Chanson and Zhu, 1993) based on two techniques: one to test the control part (FSM) and the other to test the data part by using data flow analysis technique. The resultant UTS is then checked for executability by using a constraint satisfaction method. After the UTS was generated and verified feasible, a later approach (Chanson and Jinsong, 1994) utilises symbolic execution together with constraint satisfaction method to generate test data that can trigger the considered UTS. However, the proposed approach imposes some assumptions about the EFSM model (i.e. the existence of self-loop influencing: a loop that modifies a global predicate variable) which restrict its applicability. Furthermore, since the approach depends on symbolic execution, its applicability is restricted by the same limitations found on symbolic execution applicability. Employing symbolic execution and constraint satisfaction methods for testing from an EFSM is also reported in (Koh and Liu, 1994, Bourhfir et al., 1996, Zhang et al., 2004).

Generating test sequence from EFSMs by employing functional program testing was studied in (Sarikaya et al., 1987). The approach converts the specification written in Estelle (Budkowski and Dembinski, 1987) into a simpler form in order to construct control and data flow graphs to be used in test cases derivation. However, the approach does not consider the path feasibility problem. Also, the approach operates in a manual fashion.

Other methods that test from an EFSM using FSM-based test techniques (Lee and Yannakakis, 1994, Cheng and Krishnakumar, 1996, Dahbura et al., 1990, Petrenko et al., 1996) require an EFSM to be converted into an FSM. As discussed earlier, there are two main approaches, the first being to expand the data

in the EFSM. However, the number of states in the resultant FSM can easily become prohibitively large (Hierons and Harman, 2004). The alternative is to abstract the data from the EFSM to produce an FSM but paths taken from the produced FSM to the original EFSM may not be necessarily feasible (Hierons et al., 2001).

A technique for generating unique state identification sequences for EFSM models is presented in (Ramalingom et al., 1996, Ramalingom et al., 2003). The technique is based on computing a new type of state identification for each state called *context independent unique sequence (CIUS)*. This requires that all the paths that start from any state be context independent. That is, all the guards included in any path can be interpreted symbolically and each state must have a CIUS. This requirement appears to limit the applicability of the approach. Furthermore, the study did not consider the problem of generating test inputs to trigger the generated paths.

An approach which employs software data flow testing to derive a test sequence from EFSM models is presented in (Ural and Yang, 1991). The selection of each test case depends on identifying all the associations between each output and all the inputs that affect that output. However, as stated in (Bourhfir et al., 1996), the approach might not always provide the intended coverage. Furthermore, the approach did not investigate the path feasibility problem.

Approaches that study the path feasibility problem are introduced in (Duale and Uyar, 2004, Hierons et al., 2004). In (Duale and Uyar, 2004), a method is given to convert EFSMs into other EFSMs in which all paths are feasible but this requires guards and operations to be linear. In (Hierons et al., 2004), the infeasible path problem was overcome through two steps. First, the *SDL* (*Specification and Description Language*) model is rewritten in order to derive a *normal form-EFSM* (*NF-EFSM*). Second, the resultant *NF-EFSM* is extended to *Expanded-EFSM* (*EEFSM*) in order to aid testability. As a result, all the paths present in the output *EEFSM* are feasible. However, these two approaches (Duale and Uyar, 2004, Hierons et al., 2004) did not tackle the problem of generating test inputs that trigger the resultant paths.

Approaches that utilise search algorithms to test from EFSMs are introduced in (Derderian et al., 2005, Derderian et al., 2010, Lefticaru and Ipate, 2008). The approach proposed in (Lefticaru and Ipate, 2008) describes a fitness calculation method to find a test sequence for a path. The considered fitness function applies the Tracey et al. (Tracey et al., 1998c) technique to each transition in a path. The path fitness is defined by considering each function in the path as a critical node. The limitation of this study is the assumption that each function does not have an internal path i.e. nested IF statements and thus the approach may not always provide sufficient guidance as argued in (McMinn, 2004). Furthermore, the work did not consider the problem of choosing a path that is likely to be feasible. In (Derderian et al., 2005, Derderian et al., 2010), a GA approach to generate FTPs from EFSM model was presented. This is the only previous work that utilises a GA to generate FTPs from a given EFSM. The approach evaluated the feasibility of a given TP according to the number and the types of guards found in that TP. However, the dependences between transitions in a path were not considered. Furthermore, the approach did not consider generating test inputs to trigger the generated paths.

2.8.2 Motivation for Automatic Testing from an EFSM

Although the approaches that are reviewed in the previous subsection have made considerable progress towards EFSM testing, these approaches have limitations that may restrict their applicability and therefore automatic test generation from an EFSM remains a substantial and challenging problem. Much of the previous work can be categorised to three main categories:

1. Rewriting an EFSM to construct another form of EFSM which does not suffer from the path infeasibility problem.
2. Converting an EFSM to an FSM so that FSM-based testing techniques can be applied
3. Using symbolic execution and constraint satisfaction methods to check path feasibility and to generate path test data.

The first category imposes restrictions on the EFSMs that can be considered (Duale and Uyar, 2004). Furthermore, automating the approaches that are based on this notion is a difficult task (Hierons et al., 2004). Approaches based on the second category can either lead to the state explosion problem or still suffer from the path feasibility problem (Hierons and Harman, 2004). Finally, approaches based on the last category are affected by the applicability limitations of both symbolic execution and constraint satisfaction techniques (Michael et al., 2001, Zhang, 2008).

There has been relatively little work towards investigating the application of SBST approaches to the domain of EFSM testing. SBST approaches have proven efficient and effective in automating the process of testing. Further, SBST approaches are very promising when the considered problem is software testing (Clark et al., 2003). Since easy and efficient testing approaches remain a requirement, applying SBST approaches to test from an EFSM can potentially provide new insights to this area.

Many approaches that test from an EFSM require the generation of transition paths through the considered models that are related to the test criterion. Therefore, generating a set of feasible transition paths according to a given test criterion is a basic requirement for these approaches. Furthermore, deriving test inputs that enable traversing the feasible set of transition paths is the other basic requirement to test from an EFSM. Since SBST approaches have been successfully applied to white-box testing, they fit well in the scope of automating the test generation from EFSM models.

2.9 Conclusion

Software testing is an important activity that aims to provide confidence in a system's performance. Although testing can be conducted manually, automation is desirable. Therefore, there have been many studies that aim to automate the practice of testing in order to cut down the cost and enhance reliability. There are several approaches to automating some aspects of testing. Among these

approaches, search based software testing techniques are very promising and have been shown to address many problems in the software testing domain.

Model based testing is a robust approach that aims to ease the testing process when testing from reference specifications. Models such as FSM and EFSM that represent systems to allow an efficient testing process are widely used. EFSM models, in particular, are powerful modelling approach that can capture almost all aspects of a system. However, testing from an EFSM is hindered by two main problems: path feasibility and path test cases generation.

Although they are efficient, SBST approaches have received little attention when testing from EFSM models. The application of SBST approaches to the domain of model-based testing is the main topic of this thesis.

Chapter 3: Generating Feasible Transition Paths (FTPs) for Testing from EFSM Models

3.1 Introduction

This chapter proposes a search-based approach for generating a set of feasible transition paths (FTPs) for testing from EFSM models. A classification of an EFSM's transitions in terms of guards and actions is proposed. The classification facilitates a novel representation of the dependencies among an EFSM's transitions. The representation simplifies the process of detecting any possible relation among the transitions of a given path and thus eases formulating the problem as a search-based problem. A new fitness metric that is based on transitions dependencies is then derived to act as a fitness function. The fitness metric is then utilised by a GA search to facilitate FTP generation. The approach is validated empirically on five EFSM case studies. A GA search that implemented the proposed fitness metric is used to generate sets of transition paths (test suites) from each considered EFSM to satisfy the transition coverage test criterion. The results obtained from a GA search are compared to the results of a random search.

The chapter starts by describing five EFSM case studies in Section 3.2. The presented case studies are used throughout the thesis to aid the description of the proposed EFSM testing approaches. Then in Section 3.3, the problem of generating FTPs for testing from an EFSM is described. The proposed approach is explained in Section 3.4. The Subsections 3.4.1 and 3.4.2 describe the transitions dependencies representation and the fitness metric together with the algorithm that

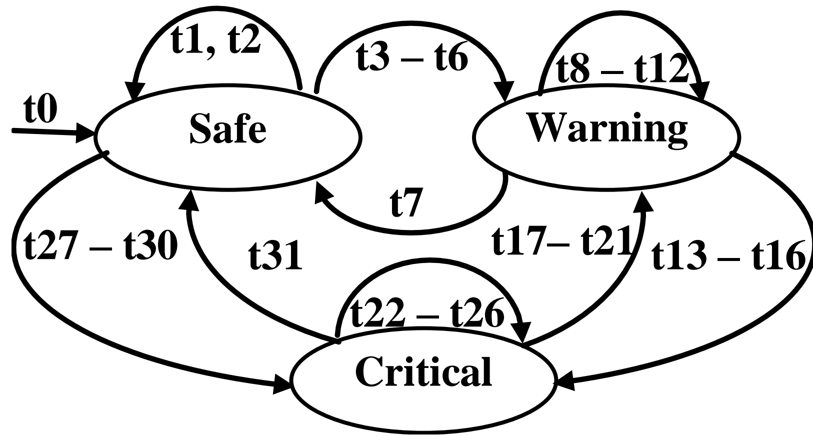


Figure 3.1: The EFSM model of In-Flight safety system

calculates a TP fitness metric. In Subsection 3.4.3, a TP encoding method as a sequence of integers is explained. The experiment is presented in Section 3.5 while concluding remarks are in Section 3.6.

3.2 Case Studies

This section introduces five EFSM case studies that are used throughout the thesis as test subjects. Three of these EFSMs were chosen because they were previously used in the literature to validate other EFSM testing approaches. These EFSMs are: the Inres initiator; the core of the class 2 transport protocol; and an automated teller machine (ATM). The other two case studies describe an in-flight safety system and a lift system. The two synthesised EFSMs have many of their transitions guards consisting of nested conditions and thus it is relatively difficult to generate tests from these EFSMs. The size of the considered case studies in terms of the number of states and transitions allows them to be used in experiment with prototype tools and yet for test generation to be non-trivial. The following describes each of these EFSMs:

1- In-Flight Safety System: A synthesised system that functions as a monitor of the craft's cabin while the craft is in-flight. The system monitors the cabin safety in terms of four factors: vibration, pressure, temperature and smoke. The system

Trans.	Input	Guards	Actions
t_0 $s_0 \rightarrow s_1$	reset	Nil	VarsRead= False; !SetWarningLights(all, off); !SwitchSounds(all,off);
t_1 $s_1 \rightarrow s_1$ t_8 $s_2 \rightarrow s_2$ t_{22} $s_3 \rightarrow s_3$	?Read(Pvb, Ppr, Psm, Ptm)	VarsRead == False	Vb = Pvb; Pr = Ppr; Sm= Psm; Tm = Ptm; VarsRead = True;
t_2 $s_1 \rightarrow s_1$ t_7 $s_2 \rightarrow s_1$ t_{31} $s_3 \rightarrow s_1$	?MainCheck1 ()	VarsRead == True **&** Vb ≥ 0 **&** Vb ≤10 Pr ≥ 86 **&** Pr ≤ 100 **&** Sm ≥0 **&** Sm ≤ 10 Tm ≥ 11 **&** Tm ≤ 35	VarsRead= False; !SetWarningLights(all, off); !SwitchSounds(all,off);
t_3 $s_1 \rightarrow s_2$ t_9 $s_2 \rightarrow s_2$	?CheckVb1()	VarsRead == True **&** Vb ≥ 11 **&** Vb ≤25	VarsRead= False; !SetLight(Seatbelt, on);
t_4 $s_1 \rightarrow s_2$ t_{10} $s_2 \rightarrow s_2$	?CheckPr1()	VarsRead == True **&** Pr ≥ 50 **&** Pr ≤ 85	VarsRead= False; Release(masks); !SetLight(Seatbelt, on);
t_5 $s_1 \rightarrow s_2$ t_{11} $s_2 \rightarrow s_2$	?CheckSm1()	VarsRead == True **&** Sm ≥ 11 **&** Sm ≤ 25	VarsRead= False; !SetSound(Sm, off);
t_6 $s_1 \rightarrow s_2$ t_{12} $s_2 \rightarrow s_2$	?CheckTm1()	VarsRead== True **&** (Tm ≥ 36 **&** Tm ≤ 46) **V** (Tm ≥ 3 **&** Tm ≤ 10)	VarsRead= False; !SetLight(Tm, on);
t_{13} $s_2 \rightarrow s_3$ t_{23} $s_3 \rightarrow s_3$ t_{27} $s_1 \rightarrow s_3$	?CheckVb2()	VarsRead == True **&** Vb >25	VarsRead= False; !SetLight(Seatbelt, on);
t_{14} $s_2 \rightarrow s_3$ t_{24} $s_3 \rightarrow s_3$ t_{28} $s_1 \rightarrow s_3$	?CheckPr2()	VarsRead == True **&** Pr ≥ 0 **&** Pr ≤ 49	VarsRead= False; !Release(masks); !SetLight(Seatbelt,on); !SetSound(Pr, off);
t_{15} $s_2 \rightarrow s_3$ t_{25} $s_3 \rightarrow s_3$ t_{29} $s_1 \rightarrow s_3$	?CheckSm2()	VarsRead == True **&** Sm > 25	VarsRead= False !SetSound(Sm, off);
t_{16} $s_2 \rightarrow s_3$ t_{26} $s_3 \rightarrow s_3$ t_{30} $s_1 \rightarrow s_3$	?CheckTm2()	VarsRead= True **&** (Tm >46) **V** (Tm ≤2)	VarsRead= False !SetLight(Tm, on); !SelLight(AC, on);
t_{17} $s_3 \rightarrow s_2$	?MainCheck2()	VarsRead == True **&** Vb ≥ 11 **&** Vb ≤25 **&** Pr ≥ 50 **&** Pr ≤ 85 **&** Sm ≥ 11 **&** Sm ≤ 25 **&** (Tm ≥ 36 **&** Tm ≤ 46) **V** (Tm ≥ 3 **&** Tm ≤ 10)	VarsRead= False !SetWarningLights(all, on); !SetWarningSounds (all, off); !Release(masks);
t_{18} $s_3 \rightarrow s_2$	?MainCheck2()	VarsRead == True **&** Vb ≥ 11 **&** Vb ≤25 **&** Pr ≥ 86 **&** Pr ≤ 100 **&** Sm ≥0 **&** Sm ≤ 10 **&** Tm ≥ 11 **&**Tm ≤ 35	VarsRead= False; !SetLight(Seatbelt, on);
t_{19} s3$\rightarrow s_2$	?MainCheck2()	VarsRead == True **&** Vb ≥ 0 **&** Vb ≤10 **&** Pr ≥ 50 **&** Pr ≤ 85 **&** Sm ≥0 **&** Sm ≤ 10 **&** Tm ≥ 11 **&**Tm ≤ 35	VarsRead= False; !Release(masks);!SetLight(Seatbelt, on); !SetSound(Pr, off);
t_{20} $s_3 \rightarrow s_2$	?MainCheck2()	VarsRead == True **&** Vb ≥ 0 **&** Vb ≤10 **&** Pr ≥ 86 **&** Pr ≤ 100 **&** Sm ≥ 11 **&** Sm ≤ 25 **&** Tm ≥ 11 **&**Tm ≤ 35	VarsRead= False; !SetSound(Sm, off);
t_{21} $s_3 \rightarrow s_2$	?MainCheck2()	VarsRead == True **&** Vb ≥ 0 **&** Vb ≤10 **&** Pr ≥ 86 **&** Pr ≤ 100 **&** Sm ≥0 **&** Sm ≤ 10 **&** (Tm ≥ 36 **&** Tm ≤ 46) **V** (Tm ≥ 3 **&** Tm ≤ 10)	VarsRead= False !SetLight(Tm, on); !SelLight(AC, on);

Table 3.1: **The transitions description of the In-Flight safety system**

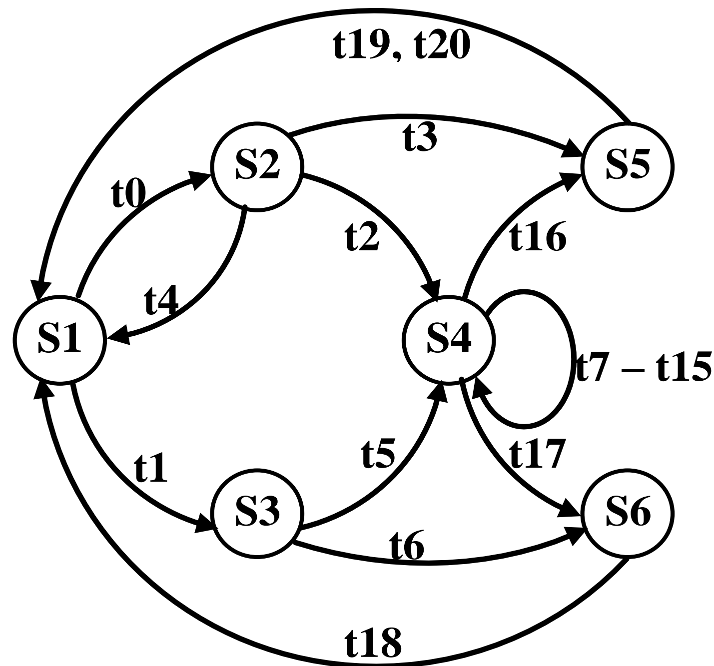


Figure 3.2: The EFSM model of core transitions of class 2 transport protocol model

comprises three states: Safe, Warning and Critical. At each state a new reading of the four factors is performed and stored in four context variables. For each new reading, a check on these four factors must be followed. This is controlled by using a Boolean variable, *VarsRead*, which must be false before any new reading can be performed and then is set to true after each new reading. Any checking process requires this Boolean variable to be true and then resets it to false after the check is performed. Therefore, the system does not allow two or more new readings in a sequence without a check or two or more checks in a sequence without a reading. The system remains in the *Safe* state whilst the values of these four factors are within a set of pre-defined ranges. However, when the value of one or more factors is within another set of pre-defined ranges, the system is in the *Warning* state. Here the pilot should take one or more actions according to a pre-defined list. Also, the system can respond with some necessary actions i.e. when the air pressure is low, oxygen masks are released automatically. The system moves to the *Critical* state when the value of one or more factors is in a critical range. In this state, the pilot has to directly intervene. For example, if the pressure cannot be brought back to normal, an emergency landing might take place. The EFSM that represents the specifications has three states *S* = {Safe,

Trans.	Input	Guards	Actions
t_0 $s_1 \rightarrow s_2$	U?TCONreq (dst_add, prop_opt)	Nil	opt = prop_opt; R_credit =0; N!TrCR
t_1 $s_1 \rightarrow s_3$	N?TrCR (peer_add, opt_ind, cr)	Nil	opt= opt_ind; S_credit=cr; R_credit=0; U!TCONind
t_2 $s_2 \rightarrow s_4$	N?TrCC (opt_ind, cr)	opt_ind $\leq$ opt	TRsq=0; TSsq=0; opt=opt_ind; S_credit=cr; U!TCONconf
t_3 $s_2 \rightarrow s_5$	N?TrCC (opt_ind, cr)	opt_ind > opt	U!TDISind; N!TrDR
t_4 $s_2 \rightarrow s_1$	N?TrDR (disc_reason, switch)	Nil	U!TDISind; N!terminated
t_5 $s_3 \rightarrow s_4$	U?TCONresp(accpt_opt)	accpt_opt $\leq$ opt	opt= accpt_opt; TRsq=0; TSsq=0; N!TrCC
t_6 $s_3 \rightarrow s_6$	U?TDISreq ()	Nil	N!TrDR
t_7 $s_4 \rightarrow s_4$	U?TDATAreq (Udata, E0SDU)	S_credit > 0	S_credit= S_credit -1; TSsq = (TSsq +1)mod128; N!TrDT
t_8 $s_4 \rightarrow s_4$	N?TrDT (Send_sq,Ndata, E0TSDU)	R_credit $\neq$ 0 **&** Send_sq= TRsq	TRsq=(TRsq+1) mod128; R_credit=R_credit -1; U!DATAind; N!TrAK
t_9 $s_4 \rightarrow s_4$	N?TrDT (Send_sq, Ndata, E0TSDU)	R_credit = 0 **V** Send_sq $\neq$ TRsq	U!error; N!error
t_{10} $s_4 \rightarrow s_4$	U?U READY (cr)	Nil	R_credit= R_credit + cr; N!TrAK
t_{11} $s_4 \rightarrow s_4$	N?TrAK (XpSsq, cr)	TSsq $\geq$ XpSsq **&** cr + XpSsq – TSsq $\geq$ 0 **&** cr +XpSsq – TSsq $\leq$ 15	S_credit = cr + XpSsq – TSsq
t_{12} $s_4 \rightarrow s_4$	N?TrAK (XpSsq, cr)	TSsq $\geq$ XpSsq **&** (cr + XpSsq – TSsq < 0 **V** cr +XpSsq – TSsq >0)	U!error; N!error
t_{13} $s_4 \rightarrow s_4$	N?TrAK (XpSsq, cr)	TSsq < XpSsq **&** cr + XpSsq – TSsq – 128 $\geq$ 0 **&** cr + XpSsq – TSsq – 128 $\leq$ 15	S_credit= cr+ XpSsq –TSsq – 128
t_{14} $s_4 \rightarrow s_4$	N?TrAK (XpSsq, cr)	TSsq < XpSsq **&** (cr + XpSsq – TSsq – 128 < 0 **V** cr + XpSsq – TSsq – 128 > 15)	U!error; N!error
t_{15} $s_4 \rightarrow s_4$	N?Ready	S_credit > 0	U!Ready
t_{16} $s_4 \rightarrow s_5$	U?TDISreq	Nil	N!TrDR
t_{17} $s_4 \rightarrow s_6$	N?TrDR (disc_reason, switch)	Nil	U!TDISind; N!TrDC
t_{18} $s_6 \rightarrow s_1$	N?terminated	Nil	U!TDISconf
t_{19} $s_5 \rightarrow s_1$	N?TrDC	Nil	N!terminated; U!TDISconf
t_{20} $s_5 \rightarrow s_1$	N?TrDR (disc_reason, switch)	Nil	N!terminated

Table 3.2: **The transitions description of the class 2 transport protocol**

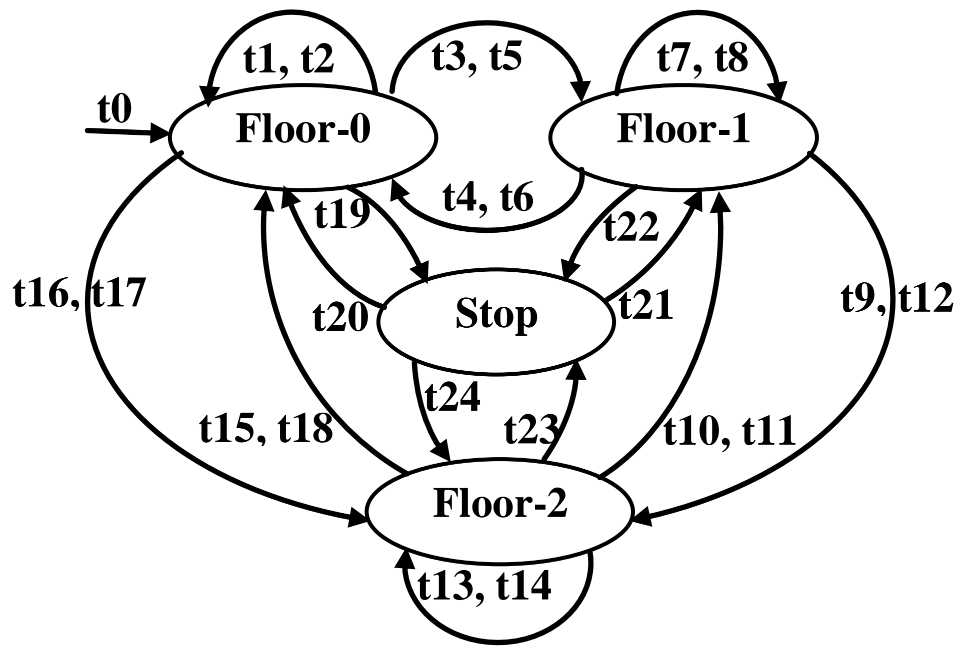


Figure 3.3: The EFSM model of the Lift system

Warning, Critical}, five context variables V = {VarsRead, Vb, Pr, Sm, Tm} and 31 transitions. Figure 3.1 shows the EFSM state transition diagram and Table 3.1 describes the transitions.

2- Class 2 Transport Protocol: This EFSM is a major model based on the access point module (*AP-module*) of the simplified version of a class 2 transport protocol. The EFSM model represents the core protocol transitions as described in (Ramalingom et al., 2003) and (Bochmann, 1990). The system has two interaction points: U for connecting to transport service access point and N for connecting to a mapping module. The system is involved in connection establishment, data transfer, end-to-end flow control and segmentation. The EFSM that represents the specifications has seven states $S = \{s_0, .., s_6\}$, five context variables V = {opt, R-credit, S-credit, TRsq , TSsq} and 21 transitions. The state transition diagram of this EFSM is shown in Figure 3.2 and the transitions are described in Table 3.2.

3- Lift System: A synthesised lift system for a building with three floors. The system consists of four states: Floor-0, Floor-1, Floor-2 and Stop. The lift provides five operations: Open, Close, Request, Service and Stop. The operations Open and Close are used to handle the cabin's door when the lift is situated in the

Trans.	Input	Guards	Actions
$t_0 \rightarrow s_0$	reset	Nil	Floor = 0; DrSt = 0; w = 0;
$t_1\ s_0 \rightarrow s_0$	?DrOp(Pos)	DrSt == 0 **&** Pos ≥ 0 **&** Pos ≤15	DrSt = 1;
$t_2\ s_0 \rightarrow s_0$	?DrCl(Pos, Pw)	DrSt == 1 **&** Pos ≥ 0 **&** Pos ≤15	DrSt = 0; w = Pw
$t_3\ s_0 \rightarrow s_1$	?Srv(Pf, Ph, Ps)	DrSt == 0 **&** Pf == 1 **&** w ≥15 **&** w ≤ 250 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 1; !Display(Floor);
$t_4\ s_1 \rightarrow s_0$	?Srv(Pf, Ph, Ps)	DrSt == 0 **&** Pf == 0 **&** w ≥15 **&** w ≤ 250 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 0; !Display(Floor);
$t_5\ s_0 \rightarrow s_1$	?Req (Pf, Ph, Ps)	DrSt == 0 **&** Pf == 1 **&** w =0 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 1; !Display(Floor);
$t_6\ s_1 \rightarrow s_0$	?Req (Pf, Ph, Ps)	DrSt == 0 **&** Pf == 0 **&** w =0 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 0; !Display(Floor);
$t_7\ s_1 \rightarrow s_1$	?DrOp(Pos)	DrSt == 0 **&** Pos ≥ 0 **&** Pos ≤15	DrSt = 1;
$t_8\ s_1 \rightarrow s_1$	?DrCl(Pos, Pw)	DrSt == 1 **&** Pos ≥ 0 **&** Pos ≤15	DrSt = 0; w = Pw
$t_9\ s_1 \rightarrow s_2$	?Srv(Pf, Ph, Ps)	DrSt == 0 **&** Pf == 2 **&** w ≥15 **&** w ≤ 250 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 2; !Display(Floor);
$t_{10}\ s_2 \rightarrow s_1$	?Srv(Pf, Ph, Ps)	DrSt == 0 **&** Pf == 1 **&** w ≥15 **&** w ≤ 250 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 1; !Display(Floor);
$t_{11}\ s_2 \rightarrow s_1$	?Req (Pf, Ph, Ps)	DrSt == 0 **&** Pf ==1 **&** w =0 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 1; !Display(Floor);
$t_{12}\ s_1 \rightarrow s_2$	?Req (Pf, Ph, Ps)	DrSt == 0 **&** Pf == 2 **&** w =0 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 2; !Display(Floor);
$t_{13}\ s_2 \rightarrow s_2$	?DrOp(Pos)	DrSt == 0 **&** Pos ≥ 0 **&** Pos ≤15	DrSt = 1;
$t_{14}\ s_2 \rightarrow s_2$	?DrCl(Pos, Pw)	DrSt == 1 **&** Pos ≥ 0 **&** Pos ≤15	DrSt = 0; w = Pw
$t_{15}\ s_2 \rightarrow s_0$	?Srv(Pf, Ph, Ps)	DrSt == 0 **&** Pf == 0 **&** w ≥15 **&** w ≤ 250 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 0; !Display(Floor);
$t_{16}\ s_0 \rightarrow s_2$	?Srv(Pf, Ph, Ps)	DrSt == 0 **&** Pf == 2 **&** w ≥15 **&** w ≤ 250 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 2; !Display(Floor);
$t_{17}\ s_0 \rightarrow s_2$	?Req (Pf, Ph, Ps)	DrSt == 0 **&** Pf ==2 **&** w =0 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 2; !Display(Floor);
$t_{18}\ s_2 \rightarrow s_0$	?Req (Pf, Ph, Ps)	DrSt == 0 **&** Pf == 0 **&** w =0 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 0; !Display(Floor);
$t_{19}\ s_0 \rightarrow s_s$	?Stp (Pf, Ph, Ps)	DrSt == 0 **&** Pf == 100 **&** w ≥15 **&** w ≤ 250 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 100; !Display(Floor);
$t_{20}\ s_s \rightarrow s_0$	?Srv(Pf)	DrSt == 0 **&** Pf == 0	Floor = 0; !Display(Floor);
$t_{21}\ s_s \rightarrow s_1$	?Srv(Pf)	DrSt == 0 **&** Pf == 1	Floor = 1; !Display(Floor);
$t_{22}\ s_1 \rightarrow s_s$	?Stp (Pf, Ph, Ps)	DrSt == 0 **&** Pf == 100 **&** w ≥15 **&** w ≤ 250 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 100; !Display(Floor);
$t_{23}\ s_2 \rightarrow s_s$	?Stp (Pf, Ph, Ps)	DrSt == 0 **&** Pf == 100 **&** w ≥15 **&** w ≤ 250 **&** Ph ≥ 10 **&** Ph ≤ 35 **&** Ps ≥ 0 **&** Ps ≤25	Floor = 100; !Display(Floor);
$t_{24}\ s_s \rightarrow s_2$	?Srv(Pf)	DrSt == 0 **&** Pf == 2	Floor = 2; !Display(Floor);

Table 3.3: **The transitions description of the Lift system**

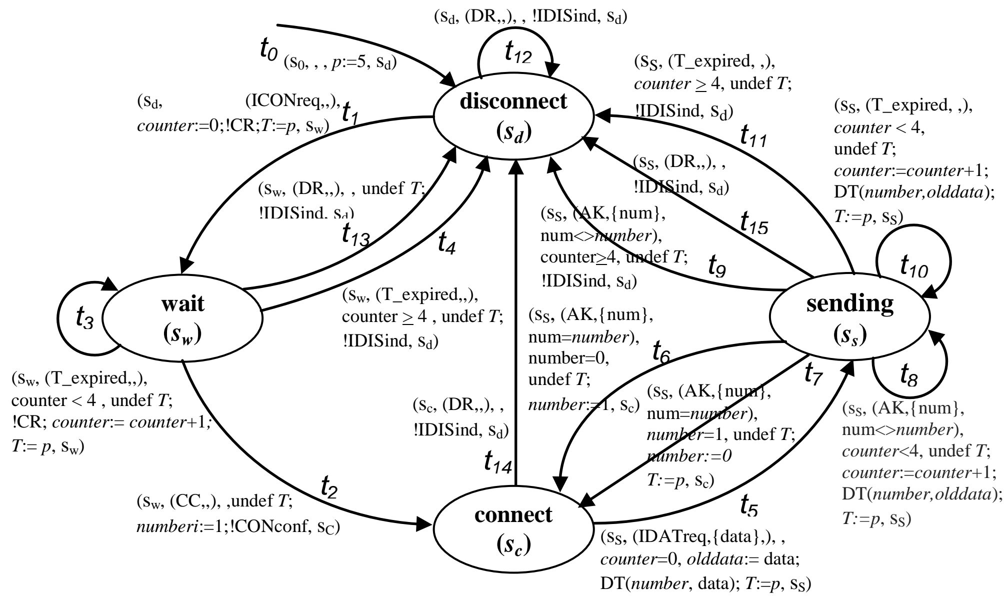


Figure 3.4: The EFSM model of Inres Initiator

specified position at each given floor. The door cannot be opened if it is already opened and similarly for close. This is controlled through a Boolean variable that is either true or false depending on the door situation. The Request operation allows a user to order the lift from a specific floor whereas the Service operation moves the lift from one floor to another floor. The stop operation facilitates an unusual case when a user wants to immediately halt the current service and so the lift will be situated at any given position. When the lift is in the Stop state, the operations Open, Close and Request are not allowed. If the cabin door is requested to be opened or closed, the system specification requires that the lift is situated at the specified place, the door frame, within a margin that does not exceed 15%. When the cabin door is closed, the cabin load's weight is read and stored in a context variable. In order for the cabin to move (Service or Request) the door should be closed. Furthermore, for safety purposes, the temperature and smoke levels inside the cabin should be within pre-defined ranges. Also, the lift does not provide a service if the weight of the cabin's load is less than or equal to 15 KG and thus a small child cannot operate the lift alone. The lift EFSM has four states S= {$Floor_0$, $Floor_1$, $Floor_2$, Stop}, three context variables V= {Drst, w, Floor} and 24 transitions. The EFSM state transition diagram is shown in Figure. 3.3 whereas the transitions are described in Table 3.3.

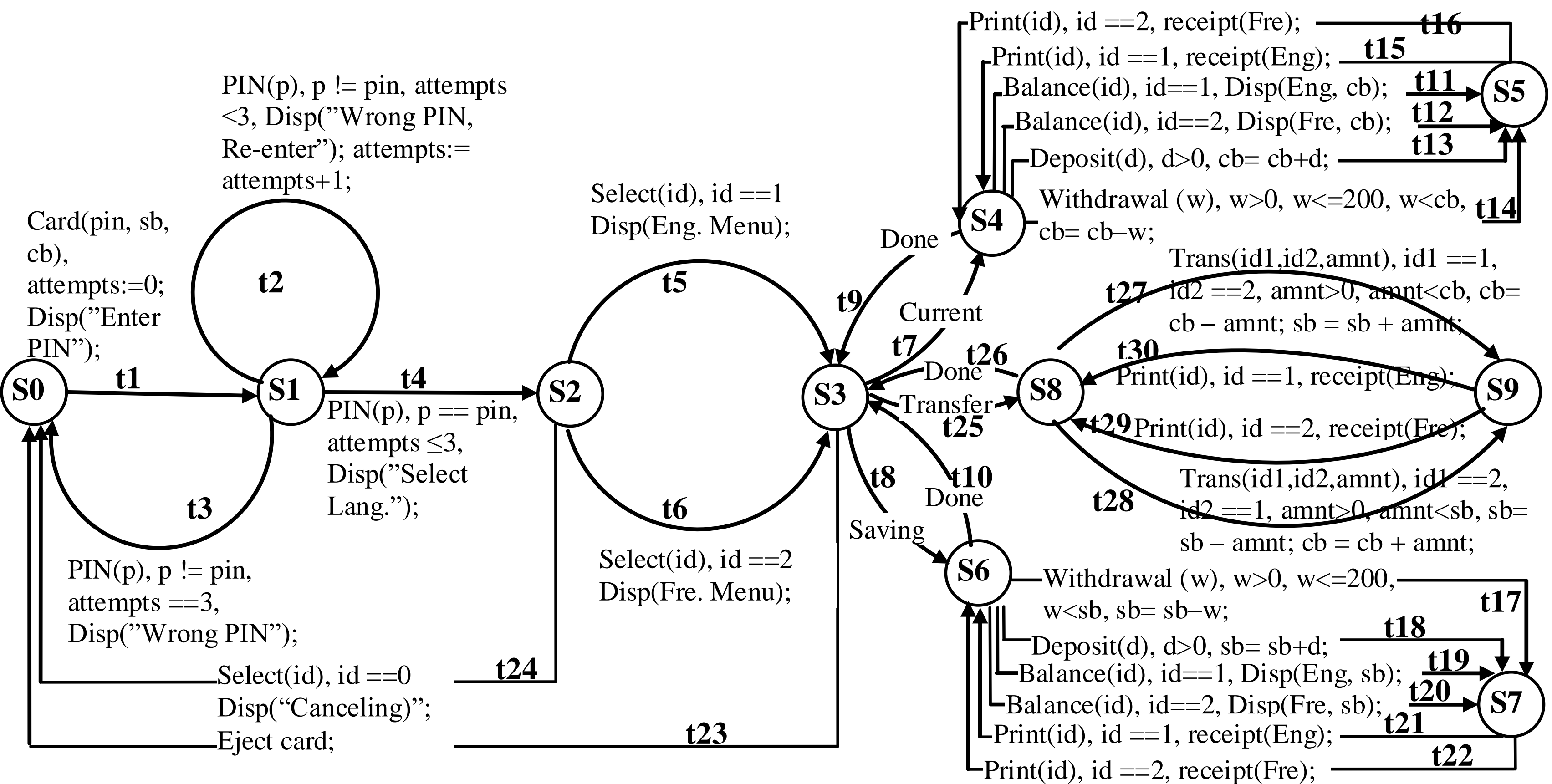


Figure 3.5: The EFSM model of the ATM System

4- Inres Initiator: The Inres (Hogrefe, 1991) protocol is connection-oriented and comprises the initiator, which establishes a connection and sends data, and the responder which receives data and terminates connections. The Inres protocol is asymmetrical and offers many concepts from the OSI (Open Systems Interconnection) standard. The protocol was designed to be similar to real protocols and yet small enough to allow experiments to be conducted for research purposes.

The Inres initiator has five states S = {s_0, disconnect, wait, connect, sending}, four context variables V = {counter, number, T, p} and 15 transitions. Figure 3.4 shows the Inres initiator EFSM together with the transitions description.

5- Automated Teller Machine (ATM): This represents an extension of the machine described in (Korel et al., 2002). The machine offers the option of English or French menu and provides three services: Deposit, Withdrawal and Transfer between two accounts (Current and Saving). In order for any transaction to occur, a user must provide a valid PIN within three tries otherwise the machine will cancel the operation. A user may perform a Withdrawal operation if the requested amount is less than or equal to £200 and the amount is available in the account from which the fund will be deducted (Current or Saving). Similarly, a user may transfer funds between the two accounts if the transferred amount is available in the source account. The ATM EFSM that represents the specification has ten states S= {s_0, .. , s_9}, four context variables V= {PIN, cb, sb, attempts} and 30 transitions. Figure 3.5 shows the EFSM state transition diagram together with the transitions description.

3.3 Problem Area

The FSM model can only represent systems that have control aspect. Naturally many systems have control and data aspects and thus the FSM cannot be applied. An FSM extended with memory is an EFSM. By including both data and control aspects, the EFSM has become a powerful modelling approach that can represent

a wide range of systems. While this extension allows more capabilities, it brings a new challenge that is not found in the FSM.

The challenge is mainly related to conduct testing from EFSM models. One part of an EFSM testing problem can be expressed in terms of finding suitable transition paths (TPs) through the EFSM that satisfy a given test criterion. However, since the EFSM combines the control and data aspects of a system, not every TP through a given EFSM is feasible. A transition in an EFSM can have guards and actions over the machine context variables. The guards must be satisfied in order for this transition to be fired. If this happens, the associated actions are executed and thus the values of some or all the context variables are updated. When considering a path through an EFSM, more than one transition needs to be fired in a sequence. If an earlier transition in the path has an action that sets the value of a context variable v whereas a later transition in the path has a guard that requires the same context variable v to have a different value, then there is an *opposition* between the actions and the guards and the considered path is infeasible.

Consider, for example, the EFSM model of the In-Flight system shown in Figure 3.1. If the TP $t_0t_2t_1$ is selected to cover the transition t_2, then this transition is not exercised since this TP is infeasible. This is because the earlier transition t_0 has an action that sets the value of the context variable *VarsRead* to be false (see Table 3.1). However, the next transition t_2 has a guard that requires *VarsRead* to be true. In this case, the selected TP is infeasible and so cannot be triggered. Similarly, any generated TP that contains this subsequence of transition (t_0t_2) is also infeasible.

This demonstrates that testing from EFSMs is generally complicated by the presence of infeasible paths (Hierons et al., 2004). Infeasible TPs included in a test suite may be expensive since any attempt to trigger these TPs will be unsuccessful. Generally, an infeasible TP is not desired in a test suite since it simply cannot be exercised and consequently the test suite may not satisfy the associated test criterion. Nevertheless, determining in advance whether a given TP is feasible is generally an undecidable problem (Dssouli et al., 1999, Hierons et al., 2009). Furthermore, developing good methods to approach the infeasibility

problem is an open research problem (Duale et al., 1999, Duale and Uyar, 2004).

Previous studies in this area have mainly approached the path feasibility problem by:

1- Rewriting an EFSM to construct another form of EFSM which does not suffer from the path infeasibility problem (Duale and Uyar, 2004, Hierons et al., 2004). However, these approaches are either not automated or can only be applied to certain types of EFSM.
2- Converting an EFSM to an FSM so that FSM-based testing techniques can be applied (Lee and Yannakakis, 1994, Cheng and Krishnakumar, 1996, Dahbura et al., 1990, Petrenko et al., 1996). Such approaches may lead to the state explosion problem.
3- Using symbolic execution and constraint satisfaction methods to check path feasibility after TPs were generated (Chanson and Jinsong, 1994, Bourhfir et al., 1996). However, symbolic execution and constraint satisfaction methods are not always applicable i.e. it is difficult to apply a symbolic execution to test subjects with arrays. Furthermore, for some types of constraints, the problem of solving them may be an undecidable problem (Zhang, 2008).

Although a precise assessment of TP feasibility is not always possible, it may include an attempt to execute the TP in order to determine whether it is feasible. However, executing a TP to understand its feasibility may encounter two problems. The first problem is related to finding a suitable test case to trigger a given TP. If the selected TP is not triggered, then another TP should be selected to be tried out. However, there can be a large number of alternative TPs that can be formed depending on the test criterion. Thus, the other problem is the possibility of trying to execute a large number of alternative TPs until one of them is triggered. The problem of trying to choose one of these alternative TPs based on the execution possibility can be an expensive approach.

This shows that deriving a set of TPs to satisfy a test criterion by merely considering the test criterion is insufficient and extra information is required to try to help choose TPs that satisfy the test criterion and also likely to be feasible. The argument here is that the problem of generating FTPs from an EFSM can be

approached by having a metric that estimates TPs feasibility. Importantly, this can be included in the TP generation. This aim fits well in the scope of search-based testing because:

1- The problem of generating FTPs from an EFSM has generally an undecidable nature and using search-based approaches can provide new insights.
2- There is a large number of TPs that can be formed from a given EFSM and a fitness metric can potentially direct search towards TPs that satisfy the test criterion and are likely to be feasible.

One motivation for the work described in this chapter is the observation that there has been relatively little research related to applying search-based testing to the problem of testing from EFSM models. Furthermore, when generating a set of paths to satisfy a test criterion there are many alternative choices and it is desirable to produce a set that contains paths that are feasible. Thus, the approach presented here aims to produce a fitness metric that can be computed quickly and that can be used as part of an overall fitness function. In particular, if it is possible to have a fitness function that directs search towards paths that satisfy a current test objective (part of a test criterion) then the problem of producing an appropriate path can be seen as a multi-objective search problem. Ultimately, the approach has the potential to be incorporated into the search when using any available testing technique that require the generation of a set of feasible paths through an EFSM model to satisfy a particular test criterion (Chanson and Zhu, 1993, Derderian et al., 2005, Duale and Uyar, 2004, Duale et al., 1999, Hierons et al., 2004, Koh and Liu, 1994, Petrenko et al., 2004, Ramalingom et al., 2003, Sarikaya et al., 1987, Wang and Liu, 1993, Ural and Yang, 1991). The approach presented in this chapter aims to form part of the solution to the following problem:

Given: an EFSM model and a test adequacy criterion

Problem: generate a set of TPs that are feasible and satisfy the test criterion by using a search-based approach. The primary contributions of this chapter are the following:

1. The chapter describes a novel method to represent the dependencies found among an EFSM transitions.
2. The chapter defines a new fitness metric that can be easily utilised by heuristic search techniques such as a GA to facilitate the automatic generation of (FTPs) through EFSMs for the purpose of testing.
3. The chapter empirically validates the efficiency of the proposed approach by using it with five EFSM case studies.

3.4 The Proposed Approach

This Section describes the proposed search-based FTPs generation approach. Before providing a detailed description, the following definitions are introduced:

Definition 3.4.1: A *transition path* (*TP*) of length n through an EFSM is a sequence of n consecutive transitions $t_1, t_2, .., t_n$.

Definition 3.4.2: A TP is feasible (an *FTP*) if it is possible to trigger each transition t_i, where $1 \leq i \leq n$, in the order that it appears in this TP.

Any path from the initial state of an EFSM defines a TP but only some of these paths may be FTPs. For example, for the In-Flight EFSM shown in Figure 3.1, the TP $t_0t_1t_2$ is an FTP but the TP $t_0t_1t_1$ is not since t_1 sets the value of the context variable *VarsRead* to true and then the next t_1 requires *VarsRead* to be false.

Based on the definition of an EFSM transition that is given in Chapter 2, any transition can generally have guards and operations. A transition's guard has the form of (e gop e') where e and e' are expressions and $gop \in \{>, <, \geq, \leq, =, \neq\}$ is the guard operator.

Given an expression e, let *Ref*(e) denote the set of variables that appear in e. According to e and e' a transition's guard can be classified into the following types:

1. $\boldsymbol{g^{pv}}$: a comparison involving a parameter and one or more context variables where $Ref(e) \cup Ref(e')$ contains a parameter and also context variables. An example is the transition t_2 in the ATM, shown in Figure 3.5, since it inputs a PIN *p* and then compares this with the correct PIN.
2. $\boldsymbol{g^{vv}}$: a comparison among context variables' values where every element of $Ref(e) \cup Ref(e')$ is a context variable and both *e* and *e′* are not constant.
3. $\boldsymbol{g^{vc}}$: a comparison between a constant and an expression involving context variables; all elements of $Ref(e) \cup Ref(e')$ are context variables and either *e* or *e′* is a constant. An example is the transition t_3 in the Inres initiator (Figure 3.4) since its guard references a context variable *counter*, compares it to a constant and does not reference an input parameter.
4. $\boldsymbol{g^{pc}}$: a comparison between a constant and an expression involving a parameter; there exists a parameter $p \in Ref(e) \cup Ref(e')$ and either *e* or *e′* is a constant. Transition t_5 in the ATM (Figure 3.5) is an example since it compares the input *id* to a constant.
5. $\boldsymbol{g^{pp}}$: a comparison between expressions involving parameters; there exists a parameter $p \in Ref(e) \cup Ref(e')$ and both *e* and *e′* are not constant.

An assignment that occurs in a transition *t* has the form of $v = e$, where *v* is a context variable and *e* is an expression. An assignment to a context variable *v* can be classified as one of the following types:

1. $\boldsymbol{op^{vp}}$: it assigns to *v* a value that depends on the parameter and so there is a parameter $p \in Ref(e)$. An example is the transition t_2 in the Lift system (Figure 3.3) since it inputs the weight of a cabin's load *pw* and updates the value of the context variable *w* on the basis of this.
2. $\boldsymbol{op^{vv}}$: it assigns to *v* a value that depends on the context variable(s) and so some the elements of $Ref(e)$ are not parameters. An example is the transition t_2 in the ATM (Figure 3.5) since it updates the value of the context variable *attempts* by using the value of the context variable *attempts*.
3. $\boldsymbol{op^{vc}}$: it assigns to *v* a constant value and so *e* is a constant. An example is the transition t_1 in the Inres initiator (Figure 3.4) since it defines the value of the context variable *counter* to be a constant.

Based on the classifications of guards and assignments, two types of transitions can be distinguished: affecting and affected-by transitions.

Definition 3.4.3: In a TP t_1, t_2, .., t_n, t_i is an *affecting* transition if t_i has an assignment $op \in \{op^{vp}, op^{vc}, op^{vv}\}$ to v and there exists a guarded transition $t_j \in$ TP, where $1 \leq i < j \leq n$, t_j has a guard $g \in \{ g^{pv}, g^{vv}, g^{vc}\}$ over v and a path from t_i to t_j is definition clear for v. t_j is also said to be an *affected-by* transition.

For example, in the In-Flight EFSM (Figure 3.1), the transition t_1 assigns a value to the context variable *VarsRead* and the guard of t_2 references this variable. Furthermore, there is a definition clear path, t_1t_2, from t_1 to t_2 and so for the subsequence t_1t_2 the transition t_1 is an affecting one whereas t_2 is an affected-by transition.

Definition 3.4.4: For variable v, assignment op of type op^{vc} is *opposed* to guard g of type g^{vc} when the path from op to g is definition clear for v and either the constants that appear in op^{vc} and g^{vc} are the same and $gop \in \{<, >, \neq\}$ or are different and $gop \in \{=\}$.

Consider again the In-Flight EFSM, the assignment to *VarsRead* in transition t_2 is opposed to the guard in t_3 since t_2 sets *VarsRead* to false and t_3 requires *VarsRead* to be true. As a result, any path that contains the subsequence t_2t_3 must be infeasible.

Definition 3.4.5: For variable v, guards g_1 and g_2 of type g^{vc} are *opposed* when the path from g_1 to g_2 is definition clear for v and one of the following hold:

1. The constants that appear in g_1^{vc} and g_2^{vc} are the same and (one $gop \in \{\neq, >, <\}$ and the other $gop \in \{=\}$ or one $gop \in \{>, \geq\}$ and the other $gop \in \{<\}$ or one $gop \in \{<, \leq\}$ and the other $gop \in \{>\}$).
2. The constants are different and both $gops \in \{=\}$.

For example, in the Lift system (Figure 3.3), the guard of transition t_{16} requires the weight of the cabin's load, w, to be equal to or greater than 15 whereas transition t_{18} requires w to be 0. A subsequence $t_{16}t_{18}$ is a definition clear

path for w and therefore a TP that includes this subsequence is infeasible. By Definitions 3.4.3, 3.4.4 and 3.4.5, two cases can be defined where a TP is clearly infeasible:

Definition 3.4.6: A TP $t_1, t_2, .., t_n$ with length $n > 1$ is *definitely infeasible* if one of the following hold:

1. There exists a variable v and a pair of transitions (t_i, t_j) where $1 \leq i < j \leq n$, t_i is an affecting transition of type op^{vc}, t_j is an affected-by transition of type g^{vc} and op^{vc} opposes g^{vc}.
2. There exists a variable v and a pair of transitions (t_i, t_j) where g_i and g_j are of type g^{vc} and g_i opposes g_j.

An Example of the first case is the transition sequence t_1t_4 in the Inres initiator (Figure 3.4). Since t_1 has an operation that assigns 0 to the *counter* while t_4 has a guard that requires *counter* > 4. An example of the second case is the transition subsequence t_4t_5 in the Lift system (Figure 3.3). Transition t_4 requires the value of the variable w to be in [15.. 250] while t_5 requires the value of w to be 0 and also the path t_4t_5 is definition clear for w. Thus any path that contains the subsequence t_4t_5 must be infeasible.

3.4.1 Dependencies Representation and Penalties

This subsection describes the TP fitness metric which aims to estimate the ‘feasibility’ of a given TP without executing it. In order to estimate the feasibility of a TP, all dependencies among the affecting and affected-by transitions in this TP are found. The aim is to have a fitness metric that can be used in search and so there is a need to have the fitness metric to be computationally simple. The TP fitness metric is therefore based on a set of approximate penalty values (Kalaji et al., 2009b) that are determined in advance.

The penalty value is a numerical estimation of how easily a given guard can be satisfied. Since a guard can be affected by a previous operation, there are three factors that have to be considered when assigning a penalty value to a pair of

(affecting, affected-by). The first factor is related to the guard type. For example, a guard of type g^{vc} can be classified as the hardest since the option of selecting the values of either c or v is not available. In contrast, a guard of the type g^{pv} is typically easier to satisfy since it is possible to choose the value of the parameter. The second factor concerns the guard operator. For example, the operator = is normally the most difficult to satisfy and ≠ is the easiest. Finally, the third factor is related to the operation type of an affecting transition. For example, an operation of type op^{vp} is potentially useful since the parameter provides an opportunity to try to select a suitable value for v while op^{vc} is the worst since it is not possible to select the value of c. In addition to the penalty between a pair of (affecting, affected-by), it is possible to have a guard that is not affected by any operation (e.g. g^{pc}) and for such a case, only the first two factors are considered when assigning a penalty value.

Table 3.4 shows the suggested penalty values for all possible combinations among affecting and affected-by transitions. For cases where there are no affecting transitions, the symbol '–' is used to indicate that the choices op^{vp}, op^{vv} and op^{vc} are irrelevant. In the case where a TP is definitely infeasible, INF[1] represents a large positive integer to help the search to avoid TPs with such dependencies.

A guard can be given using nested IFs or predicates linked by AND and OR. For guards that are represented as nested IFs or linked by AND, the sum of penalties is applied, however, the minimum penalty is considered when an OR operator is present (Tracey et al., 1998c).

The dependency between affecting and affected-by transitions can occur on the basis of one or more context variables and an affected-by transition can be affected by one or more transitions in a given TP. Therefore, each dependency between a pair of (affecting, affected-by) transitions is recorded together with the

---

[1]INF represents a large positive integer to indicate that a given path is infeasible. In all experiments INF was set to be 1×10^4 since the penalty values associated with transitions dependencies (see Table 3.4) cannot lead to a given TP being assigned a penalty value $\geq 10^4$ unless this TP is infeasible. However, other large positive integers can also be used.

Guard & Operator	Assignments			
	(nop)	(op^{vp})	(op^{vv})	(op^{vc})
$g^{pv}(=)$	4	8	16	24
$g^{pv}(<, >)$	3	6	12	18
$g^{pv}(\leq, \geq)$	2	4	8	12
$g^{pv}(\neq)$	1	2	4	6
$g^{vv}(=)$	16	20	40	60
$g^{vv}(<, >)$	12	16	32	48
$g^{vv}(\leq, \geq)$	8	12	24	36
$g^{vv}(\neq)$	4	8	16	24
$g^{vc}(=)$	40	30	60	INF if False and 0 otherwise
$g^{vc}(<, >)$	32	24	48	INF if False and 0 otherwise
$g^{vc}(\leq, \geq)$	24	18	36	INF if False and 0 otherwise
$g^{vc}(\neq)$	16	12	24	INF if False and 0 otherwise
$g^{pc}(=)$	12	-	-	-
$g^{pc}(<, >)$	8	-	-	-
$g^{pc}(\leq, \geq)$	4	-	-	-
$g^{pc}(\neq)$	1	-	-	-
$g^{pp}(=)$	6	-	-	-
$g^{pp}(<, >)$	4	-	-	-
$g^{pp}(\leq, \geq)$	2	-	-	-
$g^{pp}(\neq)$	1	-	-	-
g_i opposes g_j	INF	-	-	-

Table 3.4: **The suggested penalty values where *INF* is a large positive integer to indicate that a given dependency represents an infeasible case.**

context variable at which the dependency occurs. There are three types of assignments and each type is represented by an integer. The integers -2 and -1 mean an assignment of a constant value (op^{vc}) and an assignment of a parameter value (op^{vp}) respectively. However, an assignment that references a context variable (op^{vv}) is represented by a positive integer in [1.. m] (m context variables). A number in [1..m] represents the corresponding context variable appearing on the right-hand side of the assignment. If an assignment of type (op^{vv}) references more than one context variable, the calculation is simplified by using only one of these. The observation here is that if it is possible to easily set (choose) the value of one of these context variables then it may be less important whether it is possible to set the values of the others. Consider, for example, the problem of

op	Representation	Meaning
op^{vp}	-1	An assignment to *v* that references a parameter and no context variables
op^{vc}	-2	An assignment of a constant to *v*.
op^{vv}	$v_{1..}v_n$	An assignment to *v* that references context variables
nop	0	There is no assignment and so no dependency or open ended dependency

Table 3.5: **Assignment's types representation**

satisfying a guard $v=v'$ for context variables *v* and *v'*. If the value of *v* can easily be set by using a parameter *p*, then it may be possible to choose values for the other parameters, note the value of *v'* and then decide the value of *p*. As a result, the referenced variable v_j is chosen by considering its assignment in the previous transition as the following preference: (1) the assignment (to v_j) references a parameter, (2) the assignment references a constant and (3) the assignment references context variables. Having chosen the v_j, the value is computed as shown in Table 3.4. It is possible that there is no assignment (*nop)* and so no dependency between the transitions, or there is an open-ended dependency (a variable references another variable which is not defined). Such cases are represented by 0. Table 3.5 lists the dependency types and their integer representation.

For example, the In-Flight EFSM (Figure 3.1) has five context variables VarsRead, Vb, Pr, Sm, Tm which will be referred to by v_1, v_2, v_3, v_4, v_5 respectively. Consider transitions t_1 and t_2, from Table 3.1 transition t_2 is an affected-by of type g^{vc} and t_1 is an affecting transition of type op^{vc} at v_1 and of type op^{vp} at v_2, v_3, v_4 and v_5. From Table 3.4, the penalty value for each dependency is 0, 18, 18, 18 and 18 respectively. Dependencies between t_1 and t_2 occur at all context variables so the dependencies can be represented as a tuple with seven fields as shown in Figure 3.6. The first five fields record the dependency and penalty which occur at each context variable and the sixth, *gp*, records the sum of penalties of guards that do not involve context variables. The last field is a Boolean and used to record whether there is a penalty between the two considered transitions. The first five fields have two parts: the dependency type and the associated penalty value. The information in the mentioned tuple (Figure 3.6) can be read by the help of Table 3.4 and Table 3.5 as: there is a

Assignment type | Penalty

	t_1					**gp: $g^{pc\&pp}$**	**Dependency?**
t_2	v_1= -2 | 0	v_2 = -1 | 18	v_3 = -1 | 18	v_4 = -1 | 18	v_5 = -1 | 18	0	True

Figure 3.6: An example of a tuple representation of the dependencies between an affecting and affected-by transitions

dependency between transitions t_1 and t_2 at v_1 where the dependency is an assignment of a constant value and the associated penalty is 0. Similarly there are dependencies at v_2, v_3, v_4 and v_5 that end (when working backwards) with an assignment that references parameter values and the penalty is 18 points for each. Also, all guards of t_2 involve context variables and so the *gp* field has the value of 0.

The tuples of information are stored in a matrix, a *relation matrix*, to represent the dependencies and penalties among all the transitions in a given EFSM. The matrix has size *n* x *n* where *n* is the number of transitions in the considered EFSM. Affected-by transitions are rows whereas columns represent affecting transitions. Each cell in this matrix has a similar form of the tuple shown in Figure 3.6

3.4.2 The Fitness Metric

Figure 3.7 and Figure 3.8 show a high-level description of the algorithm that calculates the TP fitness metric. The inputs are the transition relation matrix and a TP with length $n > 1$. The algorithm first considers the penalty of any guards that do not involve context variables (Line 10). It then treats the last transition as a potential affected-by transition and determines which previous transitions are affecting (Line 13). If the current pair of transitions (t_{n-1}, t_n) forms a pair of (affecting, affected-by) then a loop is entered (Line 16) to decide at which context variables there is a dependency or a penalty (to be incurred). There are two cases: (1) The dependency type is in [-2..0], the related variable is set to be checked (Line 20) and if the corresponding penalty is greater than 0 (Line 21), this is accumulated. (2) The dependency type is greater than 0 which means that the dependency may continue by an assignment referencing context variables, the

A TP Fitness Metric

1. **input: TP of length n, EFSM relation matrix**
2. **output: non negative integer value**
3. **goal: evaluate a TP complexity**
4. **initialise: result := 0; bool array [1..v_k]** *// k is the number of context Vars.*
5. **begin**
6. **for i := n downto first_transition** *// start from the last Trans in the TP*
7. **begin**
8. **bool array [1..v_k]:= false;** *// reset the array so there is currently no // recorded dependency at any Var.*
9. **j:= i;**
10. **result := result + [t_i,t_j].gp;** *// get the penalty of guards that do not have context Vars.*
11. **while (j > first_transition) do**
12. **begin**
13. **j := j -1;** *// get the id of the previous transition*
14. **if [t_i,t_j].dependency == true then** *// if there is a dependency between the pair*
15. **begin**
16. **for v_s := v_1 to v_k do** *// scan all the context Vars. to check*
17. **begin** *// at which Var. the dependency occurs*
18. **if ([t_i,t_j].v_s(type) $\leq$ 0) and (not bool[v_s]) then** *// the dependency may end by a*
19. **begin** *// Param., Const., or no dependency*
20. **bool[v_s] := true;** *//don't check at this Var next time*
21. **If [t_i,t_j].v_s(penalty) > 0 Then** *//there is a dependency at Var. v_s*
22. **result := result + [t_i,t_j].v_s(penalty)** *//collect the penalty*
23. **end;**
24. **if ([t_i,t_j].v_s(type) > 0) and (not bool[v_s]) then** *// the dependency may continue by*
25. **begin** *// referencing a context Var.*
26. **bool[v_s] := true;** *// don't check at this Var next time*
27. **If [t_i,t_j].v_s(penalty) > 0 Then** *// the dependency continues at Var. v_s*
28. **result := result + [t_i,t_j].v_s(penalty) + check(t_i,t_j,v_s);** *// call Check function // to trace back the dependencies that propagated at the Var. v_s*
29. **end;**
30. **end;**
31. **end;**
32. **end;**
33. **end;**
34. **return result;**
35. **end.**

Figure 3.7: The algorithm that calculates TP fitness metric

Function *check* all of a transition dependencies

```
A1.  input: TP, ti,tj,vs
A2.  output: non negative integer value
A3.  goal: trace back a flow dependence on variable vs
A4.  initialise: result := 0; found := false;
A5.  begin
A6.    p := j + 1;
A7.    while (p > first_transtion) and (not found) do
A8.    begin
A9.       p := p – 1;
A10.      if [ti,tp].vs(type) ≠ 0 then          // check if there is a dependency
A11.      begin
A12         case [ti,tp].vs(type) of           // check the type of dependency
A13.          -2  : result := result + 60;     // Assignment to a constant
A14.          -1  : result := result + 20;     // Assignment to a Param.
A15.          1..k : result := result + 40 + check(tp, tp-1, v1..k); // Assignment to
                                      // a context Var. recall check function to trace back
                                      // the dependency propagated at this context Var.
A16.        end;
A17.        found := true;                    // a dependency is found, break the loop
A18.      end;
A19.   end;
A20.   if found then
A21.        return result
A22.   else return result + 60;               // the dependency is left open ended
A23. end.
```

Figure 3.8: The recursive subroutine *Check* which traces back a transition's dependencies.

related variable is set to be checked (Line 26), and if the corresponding penalty is greater than 0, then the dependency continues. Thus, the penalty is accumulated and a call is made to a subroutine *check* to detect all the previous assignments that are propagated to the current context variable (Line 28).

The recursive *check* subroutine performs data dependency analysis by starting from both the context variable and affecting transition which are passed to the call and then working backwards to find all previous transitions that may affect the value of the context variable (Line A10). If an earlier transition t_p is found to affect the context variable, then the subroutine finds the type of the

assignment (Line A12). If the assignment type is found to be less than 0 then the context variable is assigned either a constant or a parameter value. Then the subroutine penalises referencing to a constant with 60 points and to a parameter with 20 points and stops (no earlier assignments affect this assignment). If the assignment type is greater than 0, the assignment references a context variable v'. Here, the subroutine penalises this referencing by 40 points and repeats the process by calling *check* with t_p and v' (Line A15). If the dependency is open ended (depends on an undefined initial value of a variable) then 60 points are added (Line A22). When the subroutine stops (Line A21 or A22) it returns the sum of penalties. After the current pair of transitions (t_{n-1}, t_n) is scanned, another cycle starts to detect any possible relation and penalty between the next pair (t_{n-2}, t_n) (Line 13) and so forth.

The proposed algorithm that calculates the TP fitness metric has a polynomial running time $T(n) = O(n^5 \times s)$ where n is length of the TP and s is the number of context variables. It seems likely that this complexity can be significantly reduced but it was not found to be problematic in the experiments.

Let's consider for example the fitness metric calculation of a TP that consists of four arbitrary transitions $t_1t_2t_3t_4$ where the transitions details are:

- t_1 (*guards*: [$p_1 == 1$, $p_2 > p_1$], *operations*: [$v_1 := p_1$; $v_2 := p_2$])
- t_2 (*guards*: [*no guards*], *operations*: [$v_3 := 10$])
- t_3 (*guards*: [$v_3 > 0$], *operations*: [$v_1 := v_2 + v_3$; $v_3 := v_2$])
- t_4 (*guards*: [$v_1 > v_2$], *operations*: [*nop*])

For the considered path, the following part of the relation matrix is required:

Pairs (*aff-by*, *aff*)	**Dependency at v_1**		**Dependency at v_2**		**Dependency at v_3**		***gp***	**Dependency ?**
	Type	**Penalty**	**Type**	**Penalty**	**Type**	**Penalty**		
t_1 affected-by t_1	-1	0	-1	0	0	0	16	False
t_2 affected-by t_1	-1	0	-1	0	0	0	0	False
t_3 affected-by t_1	-1	0	-1	0	0	0	0	False
t_4 affected-by t_1	-1	16	-1	16	0	0	0	True
t_3 affected-by t_2	0	0	0	0	-2	0	0	True
t_4 affected-by t_2	0	0	0	0	-2	0	0	False
t_4 affected-by t_3	2	32	0	0	2	0	0	True

The TP fitness metric algorithm starts from transition t_4 and checks whether it has guards that do not involve variables (*gp* field see Figure 3.6). However, t_4 does not have such guards, so the algorithm tests whether t_3 affects t_4. Since t_4 has a guard that references v_1, and t_3 has an assignment to v_1, there is a dependency between (t_3 (op^{vv}), t_4 (g^{vv})) at v_1. Since the dependency type is 2, the dependency continues through v_2. Here, the algorithm collects the penalty (32) (see Row 6, Column 4 in Table 3.4) and calls the function *check*(t_4,t_3,v_1) to detect earlier transitions that affect the value of v_1 through v_2. The function *check* penalises this by 40 points and computes *check*(t_3,t_2,v_2) to determine earlier transitions that affect v_2. From the relations matrix, t_2 does not affect the value of v_2, thus the function considers a possible earlier assignment and so it performs *check*(t_3,t_1,v_2). From the relation matrix, t_1 assigns a parameter value to v_2 (assignment type = -1). Thus the function *check* penalises this by 20 points and returns the total penalty to the main algorithm. The main algorithm continues to determine whether the pair (t_4, t_3) has dependencies on the remaining context variables v_2 and v_3. Since there are no such dependencies, the algorithm proceeds to the next pair (t_4, t_2) in the given path. Since t_2 does not affect t_4, the next pair of transition is checked (t_4, t_1). For this pair, there are two dependencies at v_1 and v_2 where both dependencies end by an assignment of a parameter value. However, the dependency at v_1 was previously detected, thus only the dependency at v_2 is considered and the penalty (16) is collected (see Row 6, Column 3 in Table 3.4). Since t_1 is the first transition, the algorithm has completed testing all the relations between t_4 and earlier transitions. Now, the algorithm proceeds to determine the dependencies between t_3 and the earlier transitions.

From the relation matrix, only t_2 affects t_3 at v_3, and the dependency ends by assigning a constant to v_3. The algorithm collects the penalty (0) (see Row 10, Column 5 in Table 3.4) and continues to test (t_3, t_1). Again, t_1 is the first transition and the algorithm has completed checking all the relations between t_3 and earlier transitions. Now, the algorithm proceeds to determine the dependencies between t_2 and the earlier transitions. Since t_2 does not have guard, this is not affected by any transition and so no penalty is incurred.

Finally, when the algorithm reaches t_1 to determine its relationships with earlier transitions, it detects that t_1 has guards that involve only parameters and constants, thus the value of *gp* field (16) is added. Since there is no earlier transition, the total penalty (124) of the path $t_1t_2t_3t_4$ is reported.

3.4.3 The GA Encoding

The proposed FTPs generation approach uses the encoding technique from (Derderian et al., 2005, Derderian et al., 2010) in which a TP is represented by a sequence of integers where each number defines a transition. Given an EFSM with k states, let $n_1, n_2.. n_k$ be the number of transitions leaving each state. Then, the method calculates the lowest common multiplier *LCM* of $n_1, n_2.. n_k$. The last step is to define the ranges $r_1, r_2.. r_k$ for each state as $r_i = LCM / n_i$. A chromosome is a sequence of integers $i_1, i_2..i_n$, each in the range [1..*LCM*]. Each number i_i is divided by the corresponding r_j to determine the transition it defines. By using this method of encoding, every sequence defines a TP.

For example, the In-Flight EFSM shown in Figure 3.1 has k = 3 states, n_1 = 10, n_2 = 10 and n_3 = 11. Thus *LCM* = 110 and r_1 = 11, r_2 = 11 and r_3 = 10. If a sequence of integer is generated in the range [1..110] i.e. <5, 55 , 99> then by starting from the first state, the first number represents t_1. Since t_1 ends at the same state, then r_1 has to be used and so the second integer represents t_5. Similarly, t_5 ends at the second state and so by using r_2 the last number represents t_{15}. The final TP is therefore: $t_1t_5t_{15}$.

The alternative way is to directly use the transition label number to map down a sequence of integers to a possible TP. For example, if an EFSM has 15 transitions, then a sequence of integers can be generated in the range [1..15] to define a potential TP. However, this approach has the problem that not every sequence of integers defines a TP which is syntactically correct. Therefore, there can be a large number of generated TPs which are redundant, limiting the search capability from exploring other correct TPs that may potentially achieve the search goal.

3.4.4 FTP Verification Method

In order to decide whether a given TP is an FTP, a method is required to follow the generated TP. However, a TP may require a set of input parameters to be applied to its interaction parameter fields so that its guards are satisfied. Therefore, a method that generates test cases is required during the verification process.

A random test cases generator is one method that can be used to check whether a TP is an FTP. Naturally, a random test cases generator may not be successful in triggering all the generated FTPs. Nevertheless, a random test cases generator can estimate how easily a given FTP can be triggered and thus can be used to explain the relation between the fitness metric values and how easy the associated FTPs can be triggered. The random test cases generator is used in the next chapter in which the problem of test cases generation to follow given FTPs is studied.

Since a random test cases generator cannot always be effective, another method is required to determine whether a TP is an FTP. This was the main motivation for the work described in Chapter 4 where a search-based approach that facilitates the automatic test cases generation to follow a given FTP was proposed. The approach proposed in Chapter 4 is used in the experiment as a method to verify whether a TP is an FTP. This method is referred to as a GA test cases generator.

3.5 Experiment

This section provides an empirical evaluation of the proposed fitness metric on five EFSM case studies. Also, it describes how the experiment was designed and reports the experimental results that were achieved by a GA search that implemented the proposed fitness metric and a random search for the FTPs generation problem.

3.5.1 Experimental Design

In designing the experiment, the aim was to evaluate the effectiveness of the proposed TP fitness metric in guiding a GA search towards TPs that are likely to be feasible. In order to achieve this, there are three factors to be considered.

The first factor is related to the length of TPs used (the number of transitions included in each TP). Naturally, a short TP is likely to have a low fitness metric value and be easy to trigger since it has few transitions and thus few guards and operations. Therefore, the experiment considered TPs that are relatively long. That is, since the EFSM case studies consist of [15..31] transitions, TP lengths of 9, 12 and 15 transitions were considered sufficient to avoid the impact of the length factor on the results.

The second factor is related to each EFSM's structure, it may happen that a given EFSM is simple (i.e. its transitions have no guards or very few guards and operations) and so arbitrary generated TPs derived from such an EFSM can be simple and feasible. As a result, for each EFSM, three sets of subject TPs were generated by using a GA search that implemented the proposed TP fitness metric. Each set consisted of a number of TPs equal to the number of transitions in the considered EFSM. A set of TPs was generated through a sequence of search iterations where each iteration produced one TP that covered a particular transition and thus each set of TPs provided a transition coverage test suite for the considered EFSM. The first set had TPs of length 9 whereas the second and third sets had TPs of length 12 and 15 respectively. Furthermore, to understand whether a given machine is simple, three alternative sets of subject TPs were randomly generated. Each randomly generated set of subject TPs is similar to the one that was generated by a GA search in terms of the length of subject TPs and of providing a transition coverage test suite for the considered EFSM. For the purpose of comparison, the fitness metric values were measured for all subject TPs.

The third factor relates to whether a generated TP is an FTP. To verify this, a GA test cases generator, described in the next chapter, was used to verify whether a given subject TP is an FTP. Furthermore, if a TP was not triggered, this

was manually inspected to check whether it was an FTP (determining if there is a dynamic opposition among the transitions).

All search techniques were implemented in the publicly available Genetic and Evolutionary Algorithm Toolbox GEATbx (Pohlheim, 1994-2010). A detailed description of each of the GEATbx parameters is provided at the GEATbx website (Pohlheim, 1994-2010) and the values of these parameter that were used are recorded here to allow the experiment to be replicated.

An integer valued encoding was used to represent individuals which will form TPs through each considered machine. The population size was 100 individuals. The selection method was linear-ranking with selective pressure set to 1.8. Discrete recombination was used to recombine individuals whereas mutate integer method was used for mutation. GEATbx allows the use of standard random approach by setting the recombination and mutation methods to 'recnone' and 'mutrandint' respectively.

For each EFSM, three sets of subject TPs were generated by using a GA search which implemented the proposed fitness metric and another three alternative sets of subject TPs were randomly generated. Both searches (GA and random) were first applied to each machine to derive two sets of subject TPs with length of 9 and thus individuals were consisted of 9 integers. Similarly, individuals consisted of 12 and 15 variables when deriving subject TPs with length of 12 and 15 respectively.

The range of values allowed for each variable varied according to each subject EFSM as described in Subsection 3.4.3. For In-Flight EFSM, variables within each individual had values in the range [1..110]. For Class 2 EFSM, individuals' variables were allowed to take values in the range [1..66]. Values in the range [1..21] were allowed for individuals' variables associated with Lift EFSM. Individuals' variables for Inres initiator EFSM were associated with values in the range [1..28] whereas individuals' variables for ATM EFSM were allowed values in the range [1..60]. Both searches (GA and random) were given 1000 generations before search was terminated.

For each generated subject TP, a GA test cases generator was applied ten times to try to trigger this TP. Any TP is an FTP if it was successfully triggered at

least once. If a TP was associated with a TP fitness metric value that is less than INF and was not triggered, a close examination was performed to check whether this was an FTP.

3.5.2 Experimental Results

A GA search that implemented the proposed TP fitness metric was applied to each EFSM case study to generate three sets of subject TPs of length 9, 12 and 15. Each generated set of subject TPs provided a set of paths that include every transition for the considered EFSM. Similarly, a random path generator was applied to each EFSM case study to generate three alternative sets of subject TPs of length 9, 12 and 15 for comparison.

In the experiment, the penalty value INF was selected to be equal to 1×10^4 and so any subject TP with instances of infeasible TP cases was penalised with 10^4 points for each such instance. Any TP that is associated with fitness metric value $\geq 10^4$ is definitely an infeasible TP, since (for all the considered TP lengths) the penalty values associated with transitions dependencies (see Table 3.4) cannot lead to a given TP being assigned a penalty value $\geq 10^4$. Based on this, the results can be categorised into two cases:

1. Case 1: TPs that are potentially feasible. This category includes any TP that is associated with a fitness metric value less than 10^4. Such TPs were determined feasible by trying to generate test cases to trigger them. When necessary a manual inspection is used to check whether a TP is an FTP.
2. Case 2: TPs that are infeasible. This category includes any TP that is associated with fitness metric value $\geq 10^4$.

The next subsections describe the results for each EFSM and for each derived set of subject TPs. The sets that were generated by GA search are denoted by (a) for TP length = 9, (c) for TP length = 12 and (e) for TP length = 15. Similarly, randomly generated sets are denoted by (b) for TP length = 9, (d) for TP length = 12 and (f) for TP length = 15.

3.5.2.1 Results of the Lift EFSM

For this case study, a GA search generated three sets of subject TPs with length 9, 12 and 15 transitions. Each set had 24 TPs that provided the transition coverage. Similarly three sets were generated by a random search. Each part of Figure 3.9 plots one set of subject TPs in terms of a TP ID[2] against its fitness metric value.

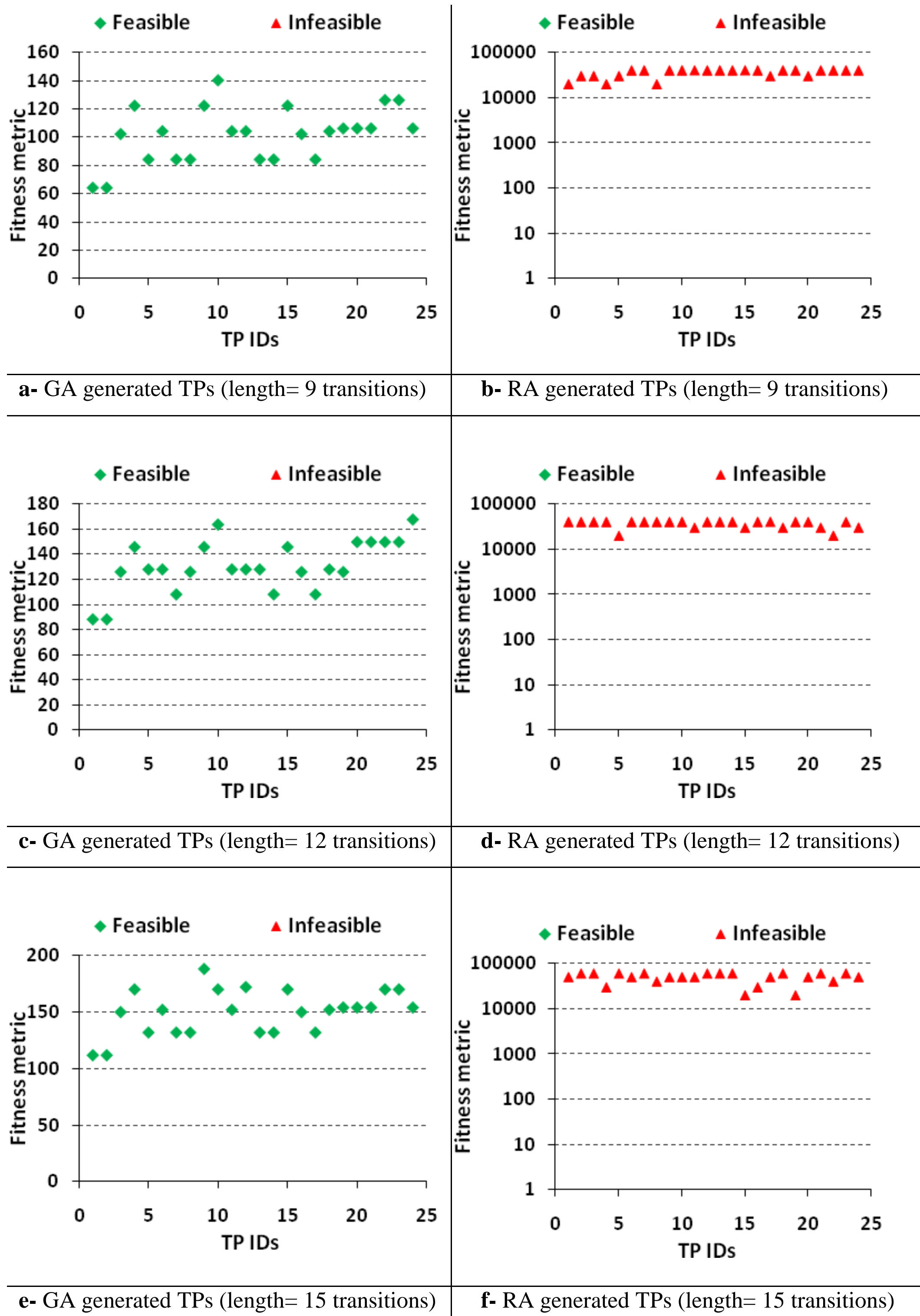
For TPs of length 9, Figure 3.9a shows the TPs that were generated by a GA search where all feasible. However, the alternative randomly generated TPs shown in Figure 3.9b were infeasible. Furthermore, for set (a) the best achieved fitness metric value (the lowest) is 64 and the worst fitness metric value is 140. For the randomly generated set, (b), fitness metric values were generally $\geq 10^4$.

For TPs of length 12, Figure 3.9c shows that the GA generated TPs were all feasible. However, the alternative randomly generated TPs were all infeasible. The best achieved fitness metric value in set (c) is 88 and this value is greater than that observed in set (a) which is 64. Furthermore, the worst fitness metric value in set (c) is 168 and this value is also greater than that observed in set (a) which is 140. This indicates that in this EFSM, as expected, longer TPs incur worse fitness metric values and thus they can be more complex. This trend is also exhibited by the random search, though TPs were infeasible, longer TPs were associated with higher fitness metric values.

Figure 3.9e and Figure 3.9f shows that GA generated TPs, set (e), were all feasible while the alternative randomly generated TPs, set (f), were all infeasible. From Figure 3.9e, the best achieved fitness metric value is 112 and this value is greater than previous values observed in sets (a) and (c). Similarly, the largest fitness metric value observed in set (e) is 188 and this is also larger than the previous values found in sets (a) and (c). The same observation is also shown by the randomly generated set (f) where the fitness metric values were larger than that observed in the previous randomly generated sets (b) and (d). This emphasises that for the considered machine, shorter TPs are associated with better (lower) fitness metric values and thus less complex than longer TPs.

---

[2] An ID of a TP corresponds to the number of the transition which this TP covers.



a- GA generated TPs (length= 9 transitions)

b- RA generated TPs (length= 9 transitions)

c- GA generated TPs (length= 12 transitions)

d- RA generated TPs (length= 12 transitions)

e- GA generated TPs (length= 15 transitions)

f- RA generated TPs (length= 15 transitions)

Figure 3.9: Lift EFSM TPs. The sets *a* & *b* have a TP length = 9, sets *c* & *d* have a TP length = 12 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are GA generated by using the TP fitness metric guidance. Sets *b*, *d* & *f* are the alternative randomly generated sets plotted by using logarithmic scale.

Method	TP Length	Feasible	Infeasible	Avg. Fitness ≈
GA	9	24	0	101
RA		0	24	35466
GA	12	24	0	131
RA		0	24	36341
GA	15	24	0	150
RA		0	24	49256
Total GA	9, 12, 15	72	0	127
Total RA	9, 12, 15	0	72	40355

Table 3.6: **Lift EFSM GA & RA generated TPs**

Table 3.6 provides a summary of the results achieved from the Lift system EFSM. From this Table, the GA search that implemented the proposed fitness metric was successful in generating TPs that are FTPs and had a success rate of 100%. In contrast, the random search did not generate any FTP and so had a success rate of 0%. This indicates that the proposed fitness metric was effective in guiding a GA search towards TPs that are feasible. Furthermore, the average fitness metric value for each set of TPs, according to TP's length, increased when TP's length increased. This trend, which is observed for both GA and random search, shows that for the considered machine, a longer TP is more complex than a shorter one. By referring to Table 3.3, which includes the transitions' details, an extra transition in a TP will increase the total number of guards and therefore will incur additional penalty. Similarly, the Boolean variables (i.e. door opened and door closed) included in this machine seem to increase the instances of infeasible TP cases when TPs are longer and the search is random. For example, if a transition that opens a door is followed by itself, then the TP penalty is increased by 10^4. It seems that the chance of such conflicts is increased when the search is random with longer TP length.

3.5.2.2 Results of the In-Flight EFSM

The GA search was applied to derive three sets of TPs of length 9, 12 and 15 where each set contained 31 TPs that provided transition coverage for this EFSM. A random search was also applied to derive three alternative sets of subject TPs.

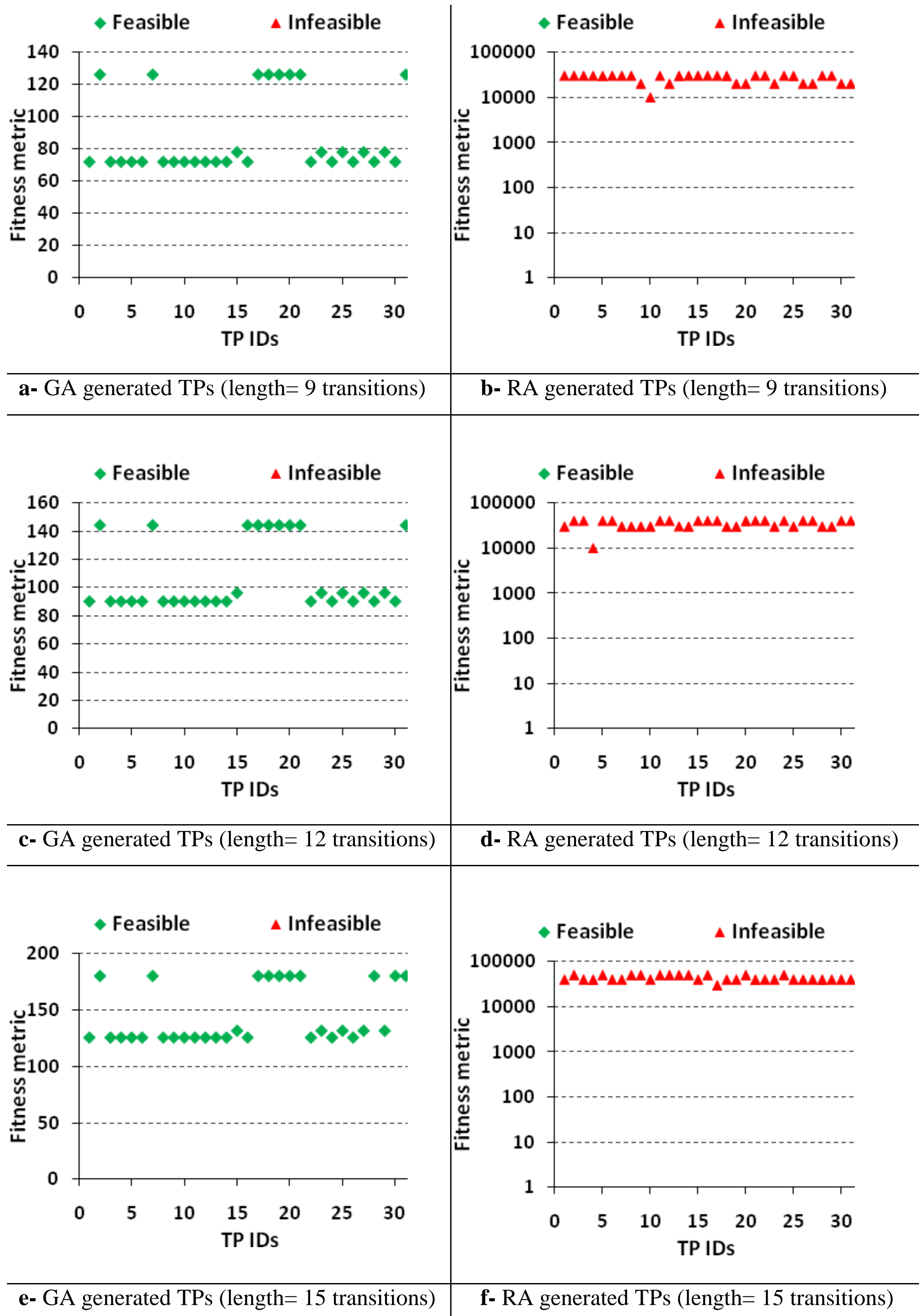


Figure 3.10: In-Flight EFSM TPs. The sets *a* & *b* have a TP length = 9, sets *c* & *d* have a TP length = 12 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are GA generated by using the TP fitness metric guidance. Sets *b*, *d* & *f* are the alternative randomly generated sets plotted by using logarithmic scale.

Method	TP Length	Feasible	Infeasible	Avg. Fitness ≈
GA	9	31	0	87
RA		0	31	26602
GA	12	31	0	107
RA		0	31	35051
GA	15	31	0	144
RA		0	31	43507
Total GA	9, 12, 15	93	0	113
Total RA	9, 12, 15	0	93	35053

Table 3.7: **The In-Flight EFSM GA & RA generated TPs**

Figure 3.10a shows that the first set of GA generated TPs was entirely feasible, however, the alternative randomly generated TPs shown in Figure 3.10b were infeasible. The best achieved fitness metric value in set (a) was 72 while the worst fitness metric value was 126. The randomly generated TPs, set (b), are generally associated with large penalty due to being infeasible.

For TPs of length 12, the GA generated set (c), shown in Figure 3.10c, was entirely feasible. Nevertheless, the alternative randomly generated set (d), shown in Figure 3.10d, was infeasible. The best and worst achieved fitness metric values by GA search were 90 and 144 respectively. Compared to the previously GA generated set (a), these values are larger than that obtained from set (a). This shows that for this machine, longer TPs seem to be associated with larger fitness metric values. For the randomly generated TPs in set (d), the same trend can also be observed where TPs are also associated with greater fitness metric values than that observed in set (b).

For the last set of TPs of length 15, similar results are also observed. The TPs generated by a GA search were all FTPs while those that were randomly generated were all infeasible. Figure 3.10e shows that the best and worst achieved fitness metric values (126 and 180 respectively) produced using the GA search were greater than that observed in sets (a) and (c). This highlights that for the considered EFSM, longer TPs are generally associated with greater fitness metric values and thus can be more complex. This tendency can also be noticed with random search.

Table 3.7 summaries the results derived from this EFSM. The GA search was generally effective and successfully produced three sets of FTPs and thus it

had a success rate of 100%. However, a random search did not produce any FTP and had a success rate of 0%. Furthermore, the fitness metric values increased when the TP length increased. This tendency was observed for both GA and random searches. Since all of the machine's transitions are guarded (see Table 3.1), any extra transition in a TP is likely to increase the TP penalty and so the GA search on longer TPs produced greater fitness metric values than that on shorter TPs. Although the random search did not yield any FTP, longer TPs seem to increase the TP chance to include more instances of infeasible TP cases. Particularly, the Boolean variables (i.e. any transition which performs a reading must be followed by a transition that checks the read values) require specific sequence of transitions otherwise an infeasible TP case occurs.

3.5.2.3 Results of the ATM EFSM

For the ATM EFSM, each GA generated set contained 30 TPs which provided transition coverage for the ATM EFSM. Similarly, three sets of subject TPs where randomly generated.

Figure 3.11a shows the GA generated set of TPs with length 9 and Figure 3.11b shows the alternative randomly generated set. From set (a), 29 TPs were FTPs and one TP was infeasible. From set (b), 20 TPs were FTPs and 10 TPs were infeasible. The best fitness metric value in set (a) was 36 while the worst one was 208. For the randomly generated set (b), the best fitness metric value was 48 and the worst one was greater than 10^4.

For TP with length 12, all but one of the GA generated TPs, plotted in Figure 3.11c, were feasible. Importantly, the infeasible TP in this set had the same ID of the infeasible TP in the previous set (a) and associated with the same fitness metric value (208). The alternative randomly generated set of TPs, plotted in Figure 3.11d, had 15 FTPs and 15 infeasible TPs. This shows a decrease in the number of randomly generated FTPs when compared to set (b). The best and worst fitness metric values observed in set (c) were 36 and 208 respectively. Interestingly, these values were the same as that noticed in the previous set (a). For set (d), the best and worst fitness metric values were 96 and $>10^4$ respectively. Compared to the previous set (b), the best fitness metric value in set (d) was

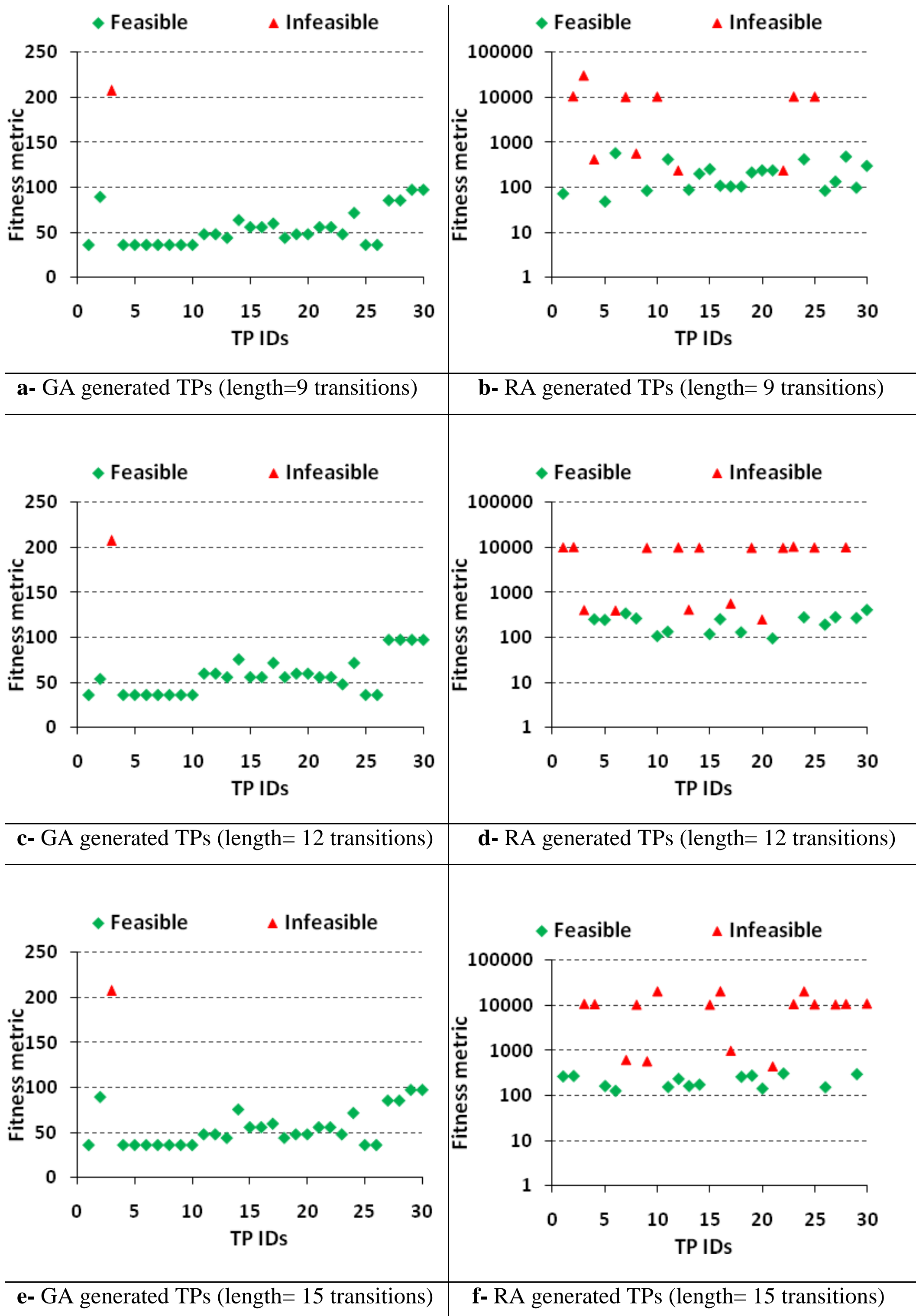
Method	TP Length	Feasible	Infeasible	Avg. Fitness ≈
GA	9	29	1	59
RA		20	10	2894
GA	12	29	1	62
RA		15	15	3603
GA	15	29	1	60
RA		14	16	5323
Total GA	9, 12, 15	87	3	60
Total RA	9, 12, 15	49	41	3940

Table 3.8: **ATM EFSM GA & RA generated TPs**

greater than that in set (b) which is 48. This may suggest that for the considered machine, longer randomly generated TPs may be associated with greater fitness metric values.

For the last set of TPs of length 15, similar results can also be observed. All but one of the GA generated TPs, shown in Figure 3.11e, were all FTPs. The infeasible TP generated by the GA search had also the same ID which previously observed on sets (a) and (c) and the same fitness metric value (208). The randomly generated set of TPs, shown in Figure 3.11f, had 14 FTPs and 16 infeasible TPs. For GA search, the best and worst fitness metric values were 36 and 208 respectively. These values are the same as the ones achieved by the GA search on the previous TP lengths. For a random search, the best and worst achieved fitness metric values were 128 and $>10^4$ respectively. The best fitness metric value in set (f) was also greater than that observed in the previous randomly generated sets (b) and (d). This can support the claim that for this machine, longer randomly generated TPs associated with higher fitness metric values.

Table 3.8 shows that the GA search had a success rate of 96.6% whereas the random search had a success rate of 54.4%. Furthermore, the GA search performed relatively similarly regardless of the TP length. In each GA generated set, there was one TP that was infeasible and this was associated with the same fitness metric value (208). The average fitness metric values achieved from the three GA generated sets were relatively very similar. This indicates that the length factor was not important for the considered machine when the search was a GA search.



a- GA generated TPs (length=9 transitions)

b- RA generated TPs (length= 9 transitions)

c- GA generated TPs (length= 12 transitions)

d- RA generated TPs (length= 12 transitions)

e- GA generated TPs (length= 15 transitions)

f- RA generated TPs (length= 15 transitions)

Figure 3.11: ATM EFSM TPs. The sets *a* & *b* have a TP length = 9, sets *c* & *d* have a TP length = 12 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are GA generated by using the TP fitness metric guidance. Sets *b*, *d* & *f* are the alternative randomly generated sets plotted by using logarithmic scale.

The results achieved from ATM EFSM raised two questions. The first is related to the infeasible TP reported in each GA generated set (the fitness metric value (208) suggests that the TP is an FTP but it is not). The second concerns the similarity of the average fitness metric values that are exhibited by GA search on the three sets of subject TPs although the length was different on each set.

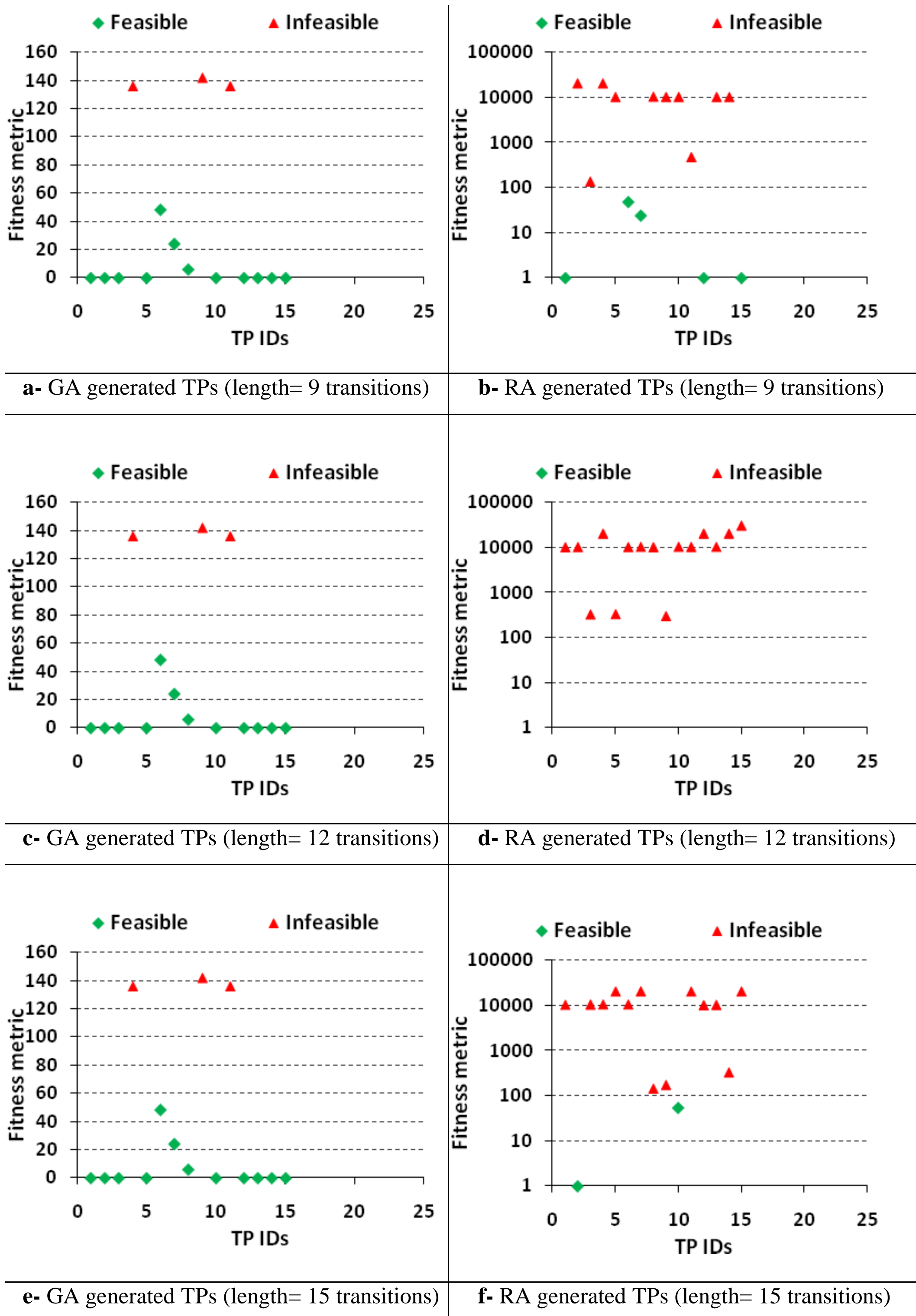
By referring to Figure 3.5 which details the ATM's transitions, there is one transition, t_3, whose guard references the variable (*attempt* =3) and this guard cannot be satisfied unless transition t_2 has previously occurred three times. Such dynamic behaviour cannot be easily estimated (by penalty values) and so any TP that included t_3 is likely to be infeasible. The GA search reported one TP in each set which was infeasible and these TPs included t_3 without sufficient occurrences of t_2. By examining all the randomly generated infeasible TPs, all had at least one instance of t_3. This explains why there is an infeasible TP in each GA generated set with the same ID.

For the second point, by considering again the ATM's transitions details, there are some transitions such as t_7, t_8, t_9, t_{10} that can be called 'escape' transitions since they do not have guards and operations. Any subsequence formed from such transitions (for example t_7, t_9, t_7, t_9, t_7, t_9), and regardless of its length, has a penalty value of zero. Thus, when TPs are longer, GA search targeted such subsequences since it does not worsen the fitness metric value. This explains why longer TPs were not penalised more than the shorter ones. Thus, when an EFSM contains 'escape' transitions, the length factor may not play an important role in worsening the penalty of longer TPs generated by a GA search.

3.5.2.4 Results of The Inres Initiator EFSM

For the Inres Initiator EFSM, the GA search produced three sets of subject TPs where each set consists of 15 TPs providing transition coverage. Also, a random search was applied to generate three alternative sets of subject TPs.

Figures 3.12a, 3.12c and 3.12e show that the GA search performed similarly on different TP lengths. Each of the three GA generated sets (a), (c) and (e) consisted of 12 FTPs and 3 infeasible TPs. Furthermore, for all of these sets, the best and worst achieved fitness metric values were 0 and 142 respectively.



a- GA generated TPs (length= 9 transitions)

b- RA generated TPs (length= 9 transitions)

c- GA generated TPs (length= 12 transitions)

d- RA generated TPs (length= 12 transitions)

e- GA generated TPs (length= 15 transitions)

f- RA generated TPs (length= 15 transitions)

Figure 3.12: Inres Initiator EFSM TPs. The sets *a* & *b* have a TP length = 9, sets *c* & *d* have a TP length = 12 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are GA generated by using the TP fitness metric guidance. Sets *b*, *d* & *f* are the alternative randomly generated sets plotted by using logarithmic

Method	TP Length	Feasible	Infeasible	Avg. Fitness ≈
GA	9	12	3	33
RA		5	10	6724
GA	12	12	3	33
RA		0	15	11445
GA	15	12	3	33
RA		2	13	9472
Total GA	9, 12, 15	36	9	33
Total RA	9, 12, 15	7	38	9214

Table 3.9: **Inres initiator EFSM GA & RA generated TPs**

This shows that for this EFSM, the factor of TP length did not affect the fitness metric values of the generated TPs.

For the random search, the number of FTPs was different for each different length. The first randomly generated set, shown in Figure 3.12b, consisted of 5 FTPs and 10 infeasible TPs. The second randomly generated set (Figure 3.12d) consisted of 15 infeasible TPs whereas the third randomly generated set (Figure 3.12f) consisted of 2 FTPs and 13 infeasible TPs.

Table 3.9 shows that the average fitness metric values achieved by GA search for different TP lengths were the same. However, this was not the case for the random search where the largest average fitness metric value was observed on TPs of length 12 transitions. From Table 3.9, the GA search generated FTPs with a success rate of 80%. However, the random search generated FTPs with a success rate of 15.5%.

The results achieved by GA search on the Inres initiator EFSM raised the same questions that were raised by the results derived from ATM EFSM. The first question related to the fact that for each GA generated set there were three TPs that are associated with fitness metric values that imply these TPs are FTPs, however, they were infeasible TPs. The second question related to the fact that for the GA search longer TPs did not incur greater fitness metric values.

By considering the Inres initiator details shown in Figure 3.4, there are three transitions t_4, t_9 and t_{12} which have guards ($counter \geq 4$) that reference a counter variable. Any subject TP that includes one of these transitions requires other transitions to previously occur and for a certain number of times so that the guard is satisfied. For example, transition t_3 must occur exactly four times before

the transition t_4 can be called. Similar to the ATM case study, TPs that included one of transitions t_4, t_9 or t_{12} were infeasible. This explains why there were three TPs in each GA generated set that were infeasible.

Similar to the ATM EFSM, the Inres initiator includes 'escape' transitions such as t_{12}, t_{13}, t_{14} and t_{15} which do not incur any penalty. Therefore, the GA search targeted these transitions to 'complete' the longer TPs. Thus, longer TPs did not receive larger fitness metric values.

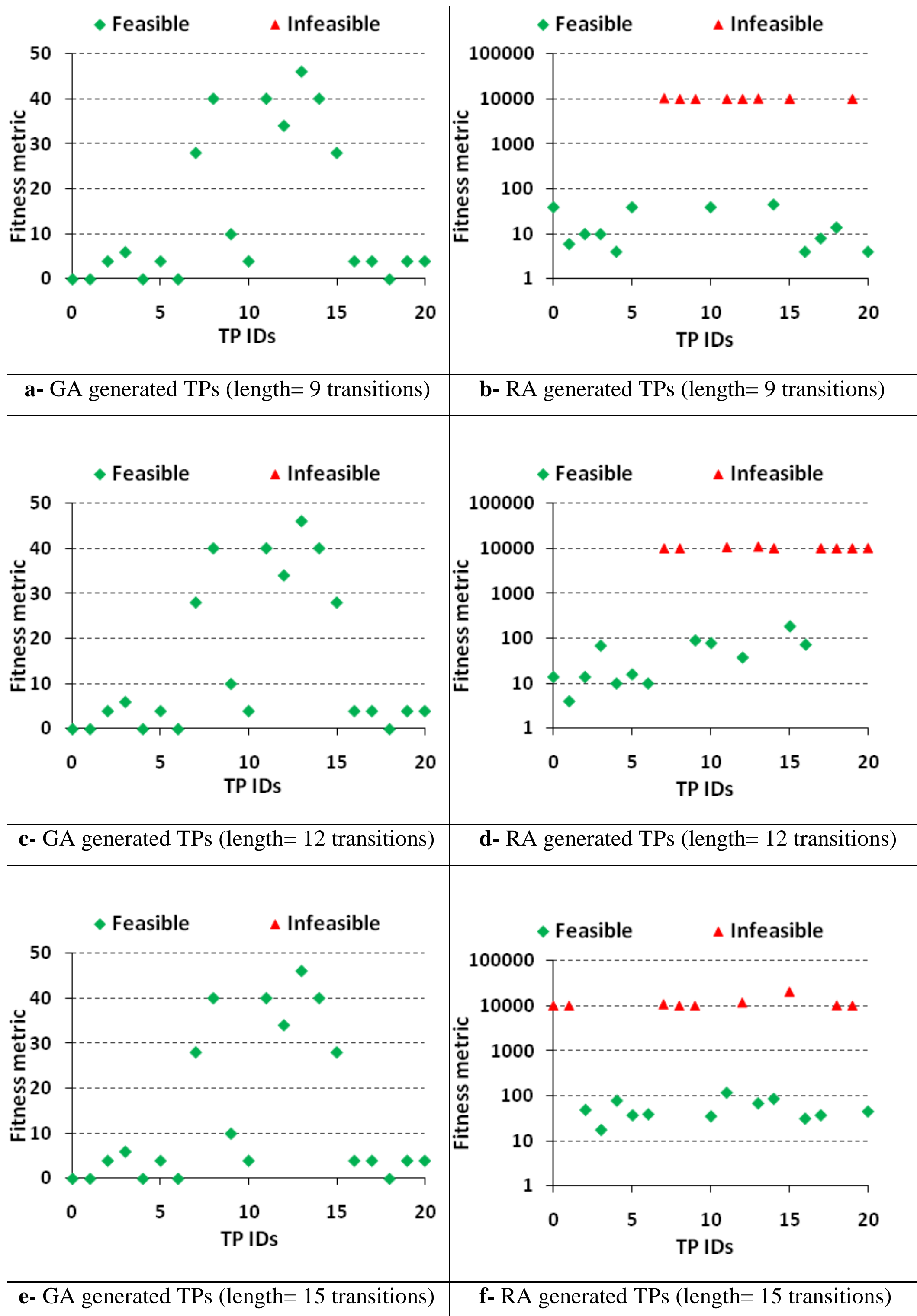
3.5.2.5 Results of The Class 2 EFSM

For this EFSM, each set of subject TPs comprised 21 TPs where three sets were GA generated and three alternative sets were randomly generated.

For all the considered TP lengths, Figures 3.13a, 3.13c and 3.13e show that all the GA generated sets of subject TPs were feasible. Furthermore, in all three sets, the best and worst achieved fitness metric values were 0 and 46 respectively. Similar to the previous case study, longer TP length did not play an important role in worsening the fitness metric values.

For all the considered lengths, the random search performed relatively similar and the number of randomly generated FTPs was 13 FTPs on the first set (Figure 3.13b) and 12 FTPs on the second and third sets (Figures 3.13d and 3.13f). The best achieved fitness metric values were 0, 4 and 18 respectively whereas the worst achieved fitness metric values were $> 10^4$. This shows that for the considered EFSM, the random search was capable of producing FTPs with low fitness metric values.

Table 3.10 shows the summary of the results derived from the Class 2 EFSM. From this Table, the GA search was effective and all the generated TPs were FTPs. This gave the GA search a success rate of 100%. However, for the random search, not all the produced TPs were FTPs and the success rate was 58.7%. Again longer TPs did not have greater fitness metric values when using a GA search. By examining the transitions details, this EFSM consists also from some 'escape' transitions (i.e. t_{18}, t_{19}, t_{20}) that do not incur penalty and are repeated when TPs were longer.



a- GA generated TPs (length= 9 transitions)

b- RA generated TPs (length= 9 transitions)

c- GA generated TPs (length= 12 transitions)

d- RA generated TPs (length= 12 transitions)

e- GA generated TPs (length= 15 transitions)

f- RA generated TPs (length= 15 transitions)

Figure 3.13: Class 2 EFSM TPs. The sets *a* & *b* have a TP length = 9, sets *c* & *d* have a TP length = 12 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are GA generated by using the TP fitness metric guidance. Sets *b*, *d* & *f* are the alternative randomly generated sets plotted by using logarithmic scale.

Method	TP Length	Feasible	Infeasible	Avg. Fitness ≈
GA	9	21	0	14
RA		13	8	3877
GA	12	21	0	14
RA		12	9	4428
GA	15	21	0	14
RA		12	9	5006
Total GA	9, 12, 15	63	0	14
Total RA	9, 12, 15	37	26	4437

Table 3.10: **The Class 2 EFSM GA & RA generated TPs**

3.5.2.6 Summary of Results

Table 3.11 shows the summary of the results achieved using a GA and random search for the five EFSM case studies. For the GA search, the Lift EFSM is associated with the largest average fitness metric value while the Class 2 EFSM is associated with the smallest average fitness metric value. This observation might explain the performance of the random search on these two EFSMs where the random search was the best on the Class 2 EFSM (success rate = 58.7%) while it was the worst on the Lift EFSM (success rate = 0%).

However, the previous observation seemed to be valid when the GA search had a success rate of 100%. For example, this observation does not apply to ATM and Inres initiator EFSMs. The average fitness metric value achieved by GA search on Inres initiator EFSM was 33 while it was 60 on ATM EFSM. However, the random search performance was poorer on Inres than that on ATM. By considering the GA search on these two EFSMs, it is also clear that the GA search performance was poorer on Inres than that observed on ATM. This can be explained by considering these EFSMs structures. The ATM EFSM has only one transition, t_3, whose guard references a counter variable (*attempt*) and so there was always one infeasible TP in each GA generated set of TPs (a TP that intends to cover t_3). However, for Inres initiator EFSM, there are three transitions (t_4, t_9 and t_{11}) that reference a counter variable (*counter*) and so there were always three infeasible TPs in each GA generated set of TPs. This shows that the results achieved from these two EFSMs were affected by the counter problem.

EFSM	Method	Total TPs	FTPs	Avg. Fitness ≈	Success Rate
Lift	GA	72	72	127	100%
	RA		0	40355	0%
Flight	GA	93	93	113	100%
	RA		0	35053	0%
ATM	GA	90	87	60	96.6%
	RA		49	3940	54.4%
Inres	GA	45	36	33	80%
	RA		7	9214	15.5%
Class 2	GA	63	63	14	100%
	RA		37	4437	58.7%
All	GA	363	351	Not applicable	96.6%
	RA		93	Not applicable	25.6%

Table 3.11: **Summary of the results achieved by GA and random searches on generating FTPs from five EFSM case studies.**

The GA search that implemented the proposed fitness metric was generally effective and produced TPs that were entirely FTPs when the considered EFSMs did not suffer from a counter problem. However, the GA search performed relatively worse on EFSMs with the counter problem. This shows that the counter behaviour is an important problem which reduces the efficiency of the proposed fitness metric approach. These results were the motivation for further investigation into the counter problem and so the study described in Chapter 5 where a novel approach is proposed to bypass the counter problem.

The overall results of the random search show that the problem of generating FTPs from the considered case studies is not an easy task. The success rate associated with random search was relatively small (25.6%). Nevertheless the GA search which implemented the proposed fitness metric performed effectively with a success rate of 96.6%.

3.6 Conclusion

Generating feasible transition paths for testing from an EFSM is a challenging task. In order to estimate TP feasibility, a classification of guards and operations

for an EFSM transition was proposed. The classification helped to statically identify all dependencies among a TP's transitions. Each dependency was given a weight (penalty) that estimates its complexity. These weights can be then used by a fitness metric algorithm to form an overall TP fitness metric value.

The proposed approach therefore formulated the FTPs generation as a search-based problem. In order to validate the approach, an experiment was conducted on five EFSM case studies. From each EFSM, three sets of subject TPs were generated by using a GA search that implemented the proposed fitness metric. Furthermore, a random search was also applied to generate similar alternative sets of subject TPs for comparison purposes.

Experimental results showed that generating FTPs from these EFSMs by the means of random search was not effective and therefore the considered task is not easy. Nevertheless, the proposed TP fitness metric effectively guided the search towards TPs that were feasible and associated with low fitness metric values. The overall success rate of FTPs generation was 96.9% for GA search while it was 25.6% for random search.

The results achieved from the experiment highlighted the importance of the counter problem for which the proposed fitness metric could not by-pass. Furthermore, the results suggest that the used penalty values were effective in estimating the TP feasibility. However, these values are by no means definite and future work could calibrate them further.

Chapter 4: Automatic Test Cases Generation to Exercise Feasible Transition Paths

4.1 Introduction

The extended finite state machine is a powerful modelling approach that has been widely applied to represent various systems. Testing from an EFSM can be performed by generating a set of transition paths (TPs) through the considered machine to satisfy a given test criterion. The next step is to generate test cases to fire the generated paths. However, achieving these two steps is a challenging task for two reasons. First, when generating TPs from an EFSM, these should be chosen to be feasible (FTPs) so that they can be triggered. Nevertheless, generating FTPs from an EFSM is a substantial problem. Second, when TPs are generated, there is a need to fire them so that the intended test is applied. However, a given FTP can require a sequence of inputs so that it can be fired. Automated test cases generation through search based testing is a topic of interest to the software engineering community. While there are many search-based techniques for automatically generating test cases for structural testing, the problem of generating test cases from an extended finite state machine (EFSMs) has received little attention. This chapter describes a novel approach that addresses the problem of generating a test case that triggers a given FTP by employing search-based testing. The proposed approach expresses the problem as a search for input parameters to be applied to functions to be called in a sequence. In order to apply a search-based technique, a new fitness function is introduced and the approach is evaluated empirically by using a GA search with the proposed

fitness function. The FTPs used were previously generated from five EFSM case studies by using the TP fitness metric approach described in the previous chapter.

The chapter starts by outlining the problem area. Then Section 4.3 defines the fitness function that can guide a GA search towards test cases that can trigger a given FTP. Furthermore, the section describes the GA encoding method used. Then, a constraint-based testing approach for FTPs test cases generation is described. Section 4.4 describes the experimental design and reports the experimental results. Concluding remarks and future work are given in Section 4.5.

4.2 Problem Area

An EFSM transition can have guards and actions over a set of the machine's context variables. Also, a transition can require some input values to be applied before the transition can be fired. Some of these inputs may update the value of some of the machine's context variables which, in turn, can be referenced by the guards of other machine's transitions.

Consider, for example, Figure 4.1 which shows the SDL specifications (ITU-T, 1994) for transition t_8 of the Inres initiator EFSM shown in Figure 3.4. In order for this transition to be triggered, a suitable value of the input parameter *num* must be provided so that the first guard ($num \neq Number$) is satisfied. Also, the next nested guard ($counter < 4$) over the machine's context variable, *Number*, must then hold so that the transition is fired and the associated operations are executed. However, for a given FTP, there is more than one transition and each transition can require a suitable set of input values. Since an EFSM's transitions share the same set of context variables, the problem of executing the transitions in a path cannot always be handled separately (i.e. triggering each transition separately). Instead, the problem of triggering a given FTP requires executing the FTP's transitions in a sequence. Consider for example two transitions t and t' where t requires an input ($p_1 > 0$) to update the value of context variable x as ($x := p_1$) and t' requires an input ($p_2 > x$) in order to update x as ($x := 0$). If a given FTP

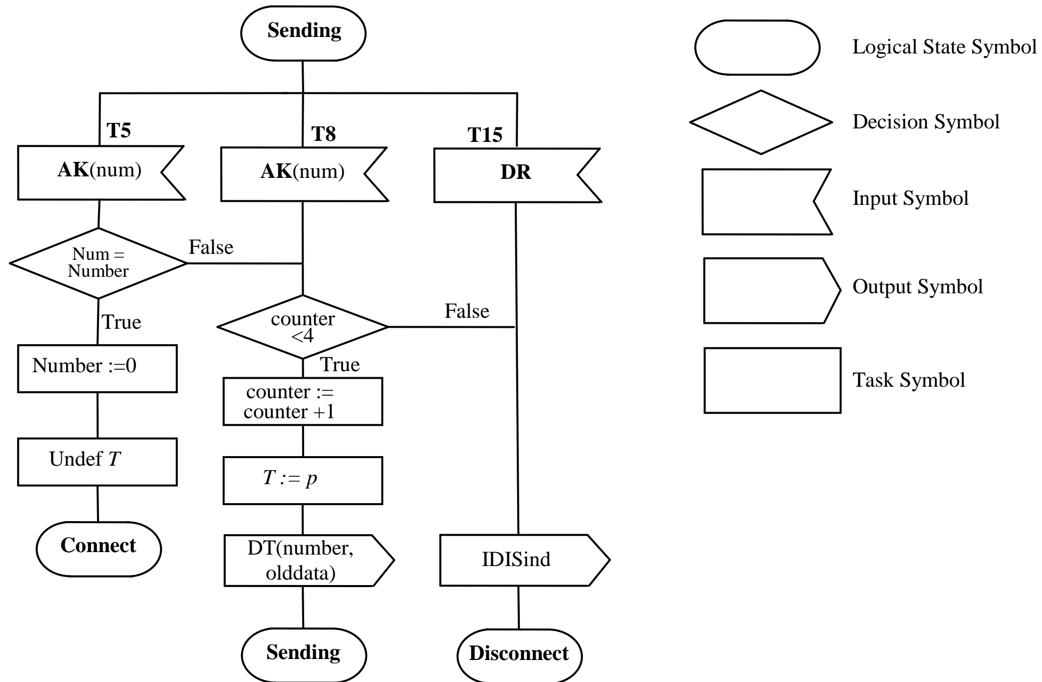


Figure 4.1: The SDL representation of transitions t_5, t_8 and t_{15} of Inres initiator EFSM where transition's guards are sequenced as nested IF statements

starts with transition *t* followed by transition *t'*, there is a need to consider transition *t* before attempting to find input to trigger transition *t'* since a suitable value of p_2 cannot be determined before the value of *x* is determined (through calling transition *t*).

The motivation of the approach presented in this chapter is the observation that there are many EFSM testing techniques that function by producing a set of paths through EFSM models, for example (Chanson and Zhu, 1993, Duale and Uyar, 2004, Hierons et al., 2004, Kalaji et al., 2009a, Derderian et al., 2010). Thus, an approach to trigger the generated paths can be potentially incorporated with such techniques.

Since the approach presented in the previous chapter considers the FTPs generation problem by using search, a search-based approach to trigger the generated FTPs can form an integrated search-based approach to testing from EFSM models. To this end, the approach presented in this chapter aims to address the problem described as:

Given: a feasible path in an EFSM model

Problem: find test cases that can cause this path to be traversed.

The main contributions of this chapter are the following:

1. It proposes a search-based approach to generate test cases that can trigger given FTPs through an EFSM model
2. It proposes a fitness function that is suitable to guide the search for test cases in the presence of function calls
3. The chapter empirically validates the efficiency of the proposed approach by applying it to a set of FTPs derived from five EFSM case studies
4. The chapter also studies the relationship between an FTP's fitness metric value and the effort, in terms of time, required by the proposed GA test cases generator to trigger the FTP. Furthermore, an investigation of the TP fitness metric capability to predict the effort required by test cases generators is also studied.

4.3 The Proposed Approach

The proposed approach utilises a GA search to find test cases to trigger a given FTP in an EFSM model. The basic part of the proposed approach is the definition of a fitness function that can guide a GA search towards suitable test cases.

4.3.1 The Fitness Function

In order to use an optimisation technique to generate a test case that executes the test target, a fitness function is required to guide the search. If a given path, within a program, consists merely of assignment statements, there is no need to derive a test case for such a path. This is because these assignments form a single path from which the execution flow cannot divert. Problems arise when a program's path contains conditional statements such as (IF, FOR and WHILE) for which the execution flow may divert away from the test target. The work of (Tracey et al., 1998a, Tracey et al., 1998b) proposed a fitness calculation method in the presence

Guard	Fitness calculation
Boolean	if *TRUE* then 0 else k
$a = b$	if $abs(a - b) = 0$ then 0 else abs(a − b) + k
$a \neq b$	if $abs(a - b) \neq 0$ then 0 else k
$a < b$	if $a - b < 0$ then 0 else $(a - b) + k$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + k$
$a > b$	if $b - a < 0$ then 0 else $(b - a) + k$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + k$
$\neg a$	Negation is moved inwards and propagated over a

Table 4.1: **Tracey et al. fitness calculations for different types of guards. The constant k, $k > 0$, is added when the guard is not satisfied.**

of conditional statements (see Table 4.1). This method is widely applied when generating test cases to satisfy a given condition (predicate) in a program's path. Consider for example a predicate ($x < y$), for which the search should locate suitable values for both x and y. By referring to Table 4.1, the fitness value (also called a branch distance) is 0 when ($x - y < 0$) which states that the current values of x and y are suitable to satisfy the given predicate. However, if the branch distance is not zero, it reflects how close were the selected values to achieving the predicate (branch distance = $x - y$ + k). Thus, the smaller the branch distance is the closer were the selected values to achieving the predicate.

Naturally programs can have nested predicates. For such a case, the fitness function should reward a test case that achieves more predicates with a fitness value that is less than that associated with a test case that achieves fewer predicates. For such a case, using only the branch distance to guide the search can be insufficient and extra information is required to guide the search. This is given in terms of approach level or approximation level proposed by (Wegener et al., 2001). The approach of (Wegener et al., 2001) defined the critical node as a conditional statement at which the execution flow may divert. Then, the approach level measures how close a given test case was to executing the target statement by subtracting one from the number of critical nodes away from the target (Equation 4.2). Since more achieved predicates should result in a smaller fitness value, the branch distance is normalised to a value in the range [0..1] (Equation 4.1).

$$\textbf{\textit{norm}}\ (branch_distance)\ = 1 - 1.05^{-(branch_distance)} \quad (4.1)$$

$$\textbf{\textit{approach level}} = NumOfCriticalNodesAwayFromTarget - 1 \quad (4.2)$$

$$\textbf{\textit{fitness}} = approach\ level + \ norm\ (branch_distance) \quad (4.3)$$

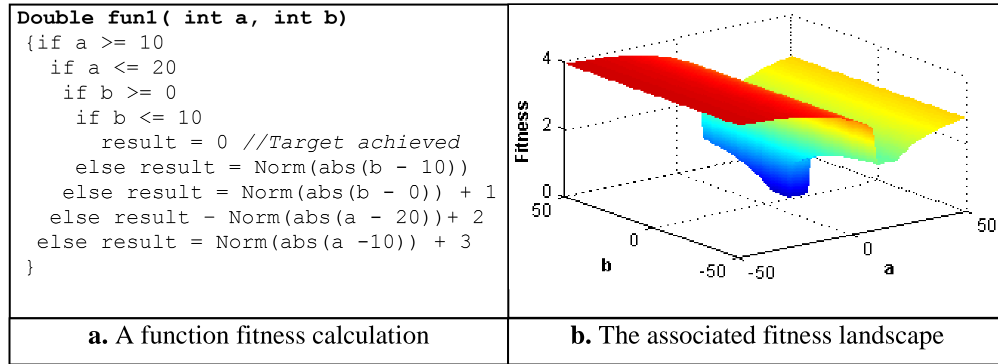
Consider for example, a function, fun_1, shown in Figure 4.2a which requires two integer inputs to satisfy four nested predicates. By applying the fitness calculation method proposed by (Wegener et al., 2001), the associated fitness function landscape (shown in Figure 4.2b) has a smooth sloped surface. Such a landscape provides the search with an adequate guidance to progress towards its goal.

The fitness calculation method proposed by (Wegener et al., 2001) is effective in structural testing where the test target is represented as a single node in the main body of the function or the program. However, this technique is not designed to cope with the case when the test subject involves a sequence of calls to transitions. In this case, the main test target comprises of a set of sub-targets (each transition in an FTP) that have to be achieved in order to attain the main test target i.e. triggering the last transition in a path.

For example, in functional testing it is necessary to trigger a path in order to reach a specific state in the machine. In this scenario, the first transition in the path must be triggered first in order to try to trigger the next transition and so forth. Since a transition in an EFSM can be considered to be a function with input parameters and conditions (Kalaji et al., 2009b), the problem of generating test cases to trigger a given FTP can be seen as finding suitable input parameter values to be applied to each transition (function) in that FTP and in a sequential order.

In order to describe the proposed fitness calculation method, consider again the function, fun_1, shown in Figure 4.2a which requires two suitable input values to achieve a set of four nested IF statements. For a given path comprising the transition sequence $fun_1(a,b) \rightarrow fun_1(c,d)$, the search should first locate suitable input values (a,b) that successfully trigger the first transition and then progress to find the next suitable input values (c,d) that trigger the next transition in the path.

The manipulation of a path in this way is similar to the structure of nested IF statements where each IF statement compares the associated function's return



a. A function fitness calculation	**b.** The associated fitness landscape

Figure 4.2: An example of fitness calculation by using Wegener et al. (2001) approach.

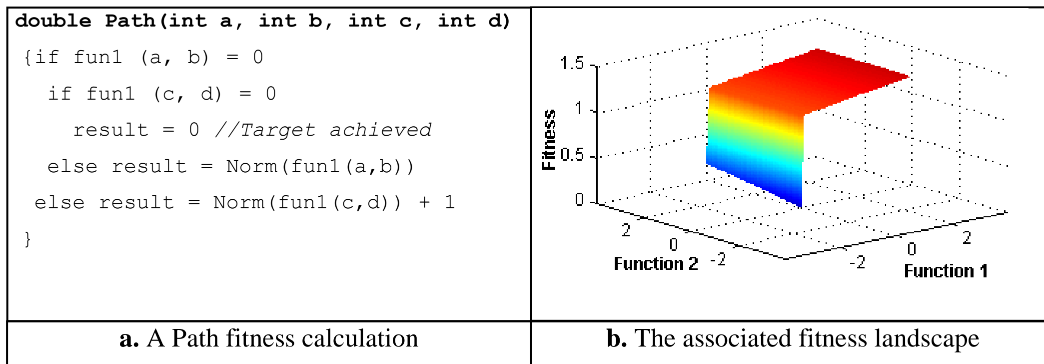
value with 0. By applying (Wegener et al., 2001) fitness calculation to each function in a path, if a function is successfully triggered then its return value is 0 otherwise the return value reflects the fitness of the input values in respect only to this particular function. Let's refer to the return value of a function by a *function distance*. In this way, the first transition in the path can be considered as the upper IF statement and then functions which come next are treated as nested IF statements. Therefore, the fitness function for a given path can be derived in a similar way to the fitness calculation method proposed by (Wegener et al., 2001) for a set of nested predicates. That is, given an FTP that consists of a sequence of transitions, the *function distance* is calculated for each guarded transition by applying the Wegener et al. approach (Equation 4.4). Then, any transition which has guard(s) is considered a critical transition and so the *function approach level* is derived by subtracting 1 from the number of critical transitions away from the target transition (Equation 4.5). Finally, the path fitness is the sum of the *function approach level* and the normalised value of *function distance* at the transition where the execution flow was diverted (Equation 4.6).

Let's consider the path $fun_1(a,b) \rightarrow fun_1(c,d)$ shown in Figure 4.3a. By applying the proposed fitness calculation, the associated fitness function landscape (Figure 4.3b) appears to have a smooth and sloped surface which can provide a search with a sufficient guidance towards its goal.

$$\textbf{\textit{function distance}} = norm\ (branch\ distance) + approach\ level \qquad (4.4)$$

$$\textbf{\textit{transition approach level}} = NumOfCrticalTransAwayFromTarget - 1 \qquad (4.5)$$

$$\textbf{\textit{path fitness}} = norm\ (function\ distance) + transition\ approach\ level \qquad (4.6)$$



a. A Path fitness calculation

b. The associated fitness landscape

Figure 4.3: An example of a path fitness calculation by using the proposed approach.

In an EFSM, transitions' guards can be sequenced as nested IF statements (as shown in Figure 1.4) or linked by logical operators AND and OR. In order to apply the proposed fitness metric, guards linked by AND operators are represented as nested IF statements. Generally, it is always possible to represent guards that are linked by AND as nested IF statements, however, the reverse is not always valid. Thus, one advantage of the proposed fitness calculation is that it considers the general case.

If guards are linked by OR, a transition is split into a number of transitions equal to the number of OR operators + 1. One benefit of doing so is that the test considers satisfying each predicate/condition in a guard. However, the alternative would be to use the minimum fitness value for a set of conditions linked by OR operator as proposed by (Tracey et al., 1998c).

A similar notion of manipulating a path to calculate the fitness is introduced in (Lefticaru and Ipate, 2008). However, the study does not consider the problem when the path includes transitions that have nested guards and so the study considers only the branch distance when calculating the *function distance.* As argued in (McMinn, 2004), in the presence of nested guards, the branch distance cannot always provide a sufficient guidance. Therefore, the proposed approach in this chapter can provide a search with better guidance for the considered FTP's test case generation problem (Kalaji et al., 2009b). Later in this chapter, an experiment is used to compare the performance of the proposed approach with that described in (Lefticaru and Ipate, 2008).

4.3.2 GA Encoding

When using a GA search to generate FTP's test cases, an encoding is required and this can be selected on the basis of the machine input parameter types. It is possible to use binary or integer encoding when all of the considered machine input parameters are of integer data type. However, if some of the input parameters are of double data type, then real valued encoding can be used. A candidate solution that represents a test case consists of components where each component represents one input parameter. For example, a possible solution encoding of the path shown in Figure 4.3a consists of four components of type integer $<C_0, C_1, C_2, C_3>$.

Naturally, FTPs can require different numbers of input parameters. A possible way to cope with this problem is to have an individual that consists of relatively a large number of components. Such a number can be determined from the maximum number of inputs required by a given FTP in a considered EFSM. In this way, when an FTP requires fewer inputs, the extra inputs included in the generated test case are simply ignored.

4.3.3 Using Constraint Satisfaction to Trigger an FTP

The constraint satisfaction method expresses the problem of test case generation in terms of solving a set of constraints. These constraints are derived by symbolically executing a given path. In symbolic executions, the inputs are represented by symbols, and thus the outputs of the program are symbols and expressions over these symbols. This leads to a general representation of the relation between a given input and its associated output. The aim of applying symbolic execution is to express the values of all the variables in a given path in terms of input parameters and / or constants. If a path is symbolically executed, the resultant constraints can be of two types: equality constraints and inequality constraints. Let e and e' be expressions:

1. An equality constraint can be given as $e = e'$ where both e and e' are constants, or e contains input parameters and e' is a constant.

2. An inequality constraint can be given as $e \leq 0$ where either e is a constant or e contains input parameters.

Given a set of equality and inequality constraints, a solver can be applied to try to find values of the parameters for which all the constraints hold.

As mentioned previously, an FTP can be seen as functions to be called in a sequence. Given this description, a set of constraints from a given path can be derived through the following steps:

1. For all the transitions in the path, rename the input parameters for each transition so that all the input parameters have unique names.
2. By starting from the first transition, for each transition, then for each assignment statement, replace the context variable by the expression that is assigned to it using the input parameters and current values of context variables. If the transition contains guards, then for each guard that involves a context variable, replace this variable by its current value in terms of parameters and constants.
3. If there is a transition that still has guards that reference context variables, the given path cannot be executed since the values of these variables are not yet defined. Otherwise, the resultant list of guards is a set of constraints that reference only input parameters and constants.

The set of constraints then can be fed to a solver in order to try to find suitable values for the input parameters included in the constraints so that all the constraints hold. If the set of constraints is solved, the values returned by the solver comprise a test case that can exercise the considered FTP.

4.4 Experiment

This section describes two experiments and reports their results. The first experiment applied three search-based test cases generators to sets of subject TPs derived from five EFSM case studies. The second experiment studied the relationship between the FTP's fitness metric value and the effort, in terms of time, that is required by a test cases generator to trigger the FTP. It also

investigated the capability of the TP fitness metric to predict the required effort by a test cases generator to trigger an FTP.

4.4.1 Design of the First Experiment

In designing the first experiment, the aim was to evaluate the efficiency of the proposed fitness function in guiding a GA search for test cases that can trigger two groups of subject TPs that were derived from the five EFSM case studies. To achieve this, there are two factors to be considered.

The first factor is related to the method by which a subject TP was generated. That is, subject TPs that were generated by using the TP fitness metric approach were potentially associated with the least possible fitness metric values. This may lead to such TPs being relatively easy to trigger. However, subject TPs that were randomly generated can have any possible TP fitness metric value. Thus, the performance of the proposed test cases generation approach on these two types of subject TPs might reveal whether the proposed approach is generally applicable or limited to a certain type of path. Therefore, for each considered EFSM, the proposed approach is applied to two groups of subject TPs. The first group was generated by using the TP fitness metric approach whereas the second group was randomly generated. Each group consists of three sets of subject TPs where each set has a different TP length. These sets were previously generated and reported on Chapter 3.

The second factor is related to comparing the performance of the proposed test cases generation approach with an alternative approach that is reported on the literature (Lefticaru and Ipate, 2008). This can show which approach is capable to trigger more FTPs and thus is more effective than the other.

The third factor is to understand how the proposed approach performs compared to a random test cases generator. Furthermore, the use of a random test cases generator can help to determine how easy it is to trigger a generated TP. That is, if it is possible to quickly randomly find a test case that can trigger a given FTP, then it is possible to state that such an FTP is easy to trigger.

Therefore, in the experiment, three test cases generators were applied to each subject TP. The first one is the proposed approach and will be denoted by (GA-1). The second one is the alternative approach taken from the literature (described at the end of Subsection 4.3.1) and will be denoted by (GA-2). Finally the third approach is a random test cases generator and is denoted by (Rand).

The three test cases generators were implemented by using the publicly available Genetic and Evolutionary Algorithm Toolbox GEATbx (Pohlheim, 1994-2010). A detailed description of each of the GEATbx parameters is provided at the tool's website and the values used in the experiment are recorded here to allow a replication of the experiment. An integer valued encoding was use to represent the input parameters. The population size was 100 individuals where each individual consists of 25 integer variables which represent the maximum number of input parameters required by any considered subject TP. The range of values allowed for each variable was [0..1000]. The selection method was linear-ranking with a selective pressure set to 1.8. Discrete recombination was used to recombine individuals whereas the mutate integer method was used for mutation. GEATbx allows the use of a standard random approach by setting the recombination and mutation methods to 'recnone' and 'mutrandint' respectively. The three test cases generators were given 1000 generations before search was terminated. Finally, the search was repeated with each technique 10 times for each subject TP.

4.4.2 Experimental Results for the Three Search-Based FTPs Test Cases Generators

For each EFSM case study, three sets of subject TPs were generated by using the TP fitness metric approach (described in Chapter 3). Each set potentially provides a transition coverage test suite for the considered EFSM. However, each set has a different TP length which is 9 transitions in the first set, 12 transitions in the second set and 15 transitions in the third set. Also, three similar alternative sets were randomly generated. All of these sets were previously reported in Chapter 3. The three sets that were generated by the TP fitness metric approach were grouped

together and are referred to by a *GA group*. Similarly, the three sets of alternative randomly generated TPs were grouped together and are referred to by *random group*. Then each of the three test cases generators (GA-1, GA-2, Rand) was applied ten times to each subject TP in each group.

4.4.2.1 Results Derived from the Lift EFSM

For this EFSM, each group of subject TPs (*GA* or *random* group) contains 72 TPs. Figure 4.4 comprises of three rows, the first row shows two figures (4.4a and 4.4b) which plot the performance of GA-1 approach on the GA group of TPs and the random group of TPs respectively. Similarly, the second and third rows show the performances of GA-2 and Rand approaches respectively. Each plot shows the fitness metric value of each subject TP plotted against the average number of generations required in ten tries to trigger this TP.

From Figure 4.4a, the proposed test cases generator, GA-1, was successful in triggering the entire subject TPs included in the GA group. This states that the entire subject TPs in the GA group are feasible. Furthermore, GA-1 approach always required more than 600 generations in order to trigger subject TPs in the GA group. However, for the subject TPs in the random group, Figure 4.4b shows that none of these TPs was successfully triggered by GA-1. This is because all of the randomly generated TPs are infeasible where each of them includes at least one instance of the infeasible TP cases (guards opposition or guard and assignment opposition).

Figure 4.4c shows that the GA-2 approach was capable to trigger just a subset of the subject TPs in the GA group. However, the GA-2 method was able to perform faster than the GA-1 on some subject TPs. Furthermore, all the TPs that were triggered by GA-2 are associated with TP fitness metric values that did not exceed 112. However, there are TPs that are associated with lower TP fitness metric values but were not triggered. This shows that for this EFSM, the proposed approach, GA-1, outperformed the alternative approach, GA-2. For the subject TPs that are included in the random group, Figure 4.4d shows that GA-2 did not trigger any of them since all of these TPs are infeasible.

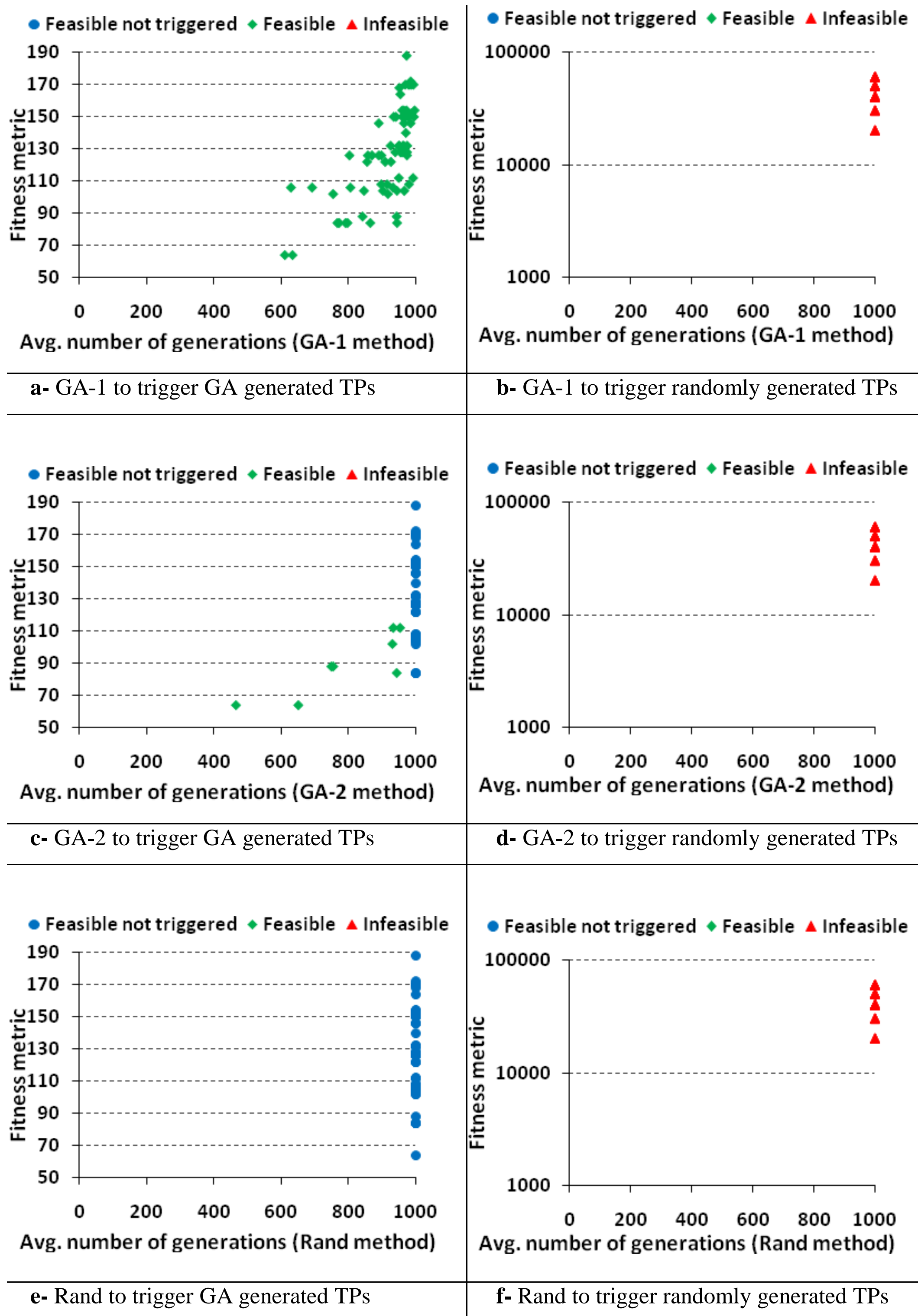


Figure 4.4: Lift EFSM GA and random groups of subject TPs. Plots *a* & *b* show the performance of the GA-1 approach on TPs in GA group and TPs in the random group respectively. Similarly, Plots *c* & *d* and Plots *e* & *f* show the performance of GA-2 and Rand approaches respectively. The fitness metric of subject TPs in random group is plotted by using logarithmic scale.

Triggering Method	TP Generation Method	Total TPs Count	Feasible TPs	Triggered	Success rate
GA-1	GA generated TPs (GA group)	72	72	72	= 100%
GA-2				8	≈ 11%
Rand				0	≈ 0%
GA-1	Randomly generated TPs (Random group)	72	0	0	= 0%
GA-2				0	= 0%
Rand				0	= 0%
GA-1	Both methods	144	72	72	= 100%
GA-2				8	≈ 11%
Rand				0	≈ 0%

Table 4.2: **The performance of three test case generation methods on two groups of subject TPs derived from the Lift EFSM.**

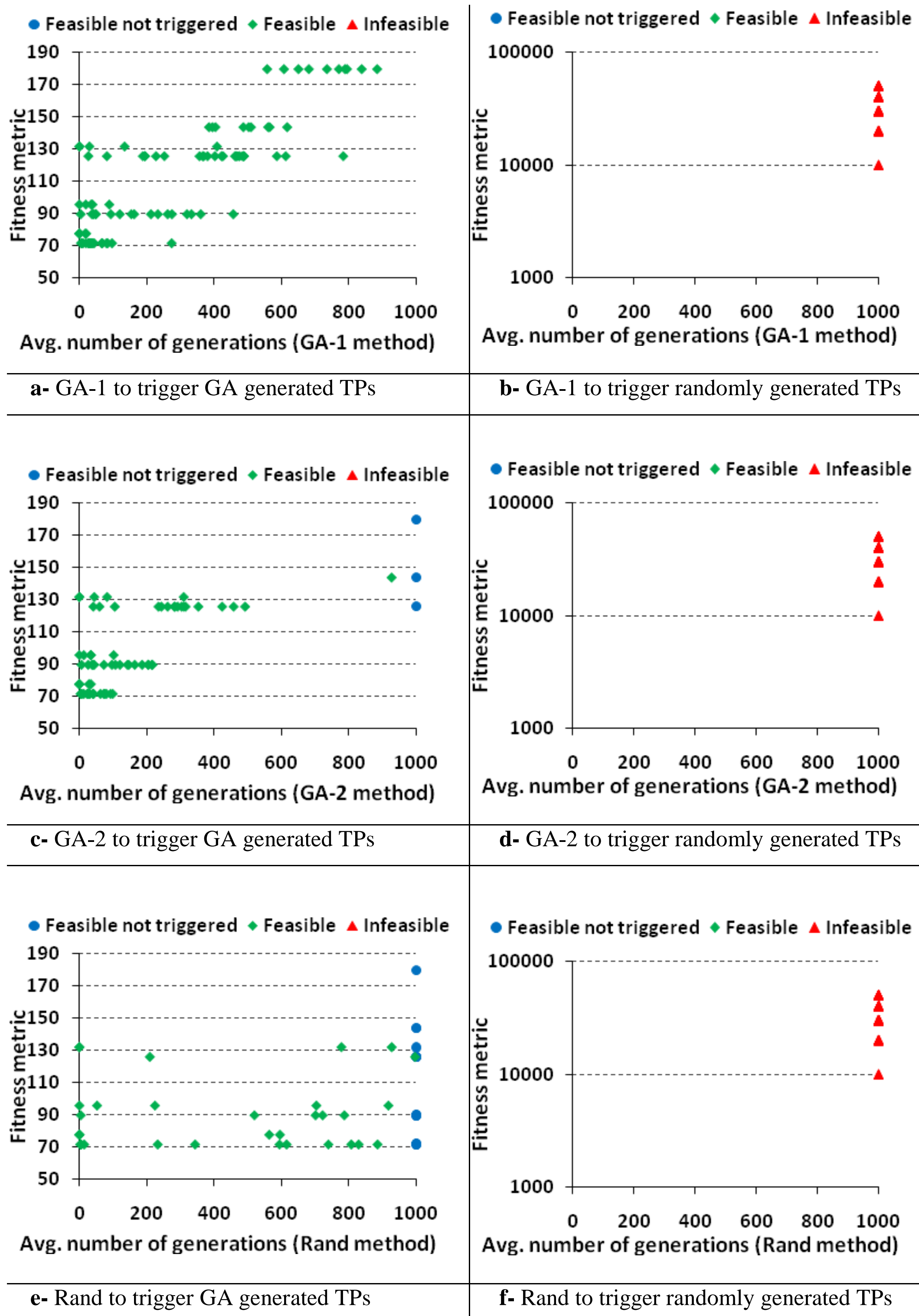
Rand method could not trigger any subject TP of the GA group (Figure 4.4e). This can lead to a conclusion that none of these subject TPs can be classified as being easy to trigger. Since the random group of TPs did not contain any feasible TP (Figure 4.4f), it was not surprising that the Rand performance on these TPs is the same as the performance of the other two approaches (GA-1 and GA-2).

Table 4.2 reports the summary of the results achieved by the three test cases generators on Lift system EFSM. From this table, the proposed approach, GA-1, triggered all the subject TPs in the GA group (success rate = 100%). However, the alternative approach, GA-2 was outperformed by the proposed approach since it was able to trigger only 8 TPs (success rate ≈ 8%). The Rand approach exhibited the worst performance on the GA group and did not trigger any TP (success rate = 0%). Since none of the subject TPs included in the random group is feasible, the considered test cases generation methods did not trigger any of them.

4.4.2.2 Results Derived from the In-Flight EFSM

For this EFSM, the GA group contains 93 subject TPs and so does the random group of subject TPs.

Figure 4.5a shows that the proposed approach, GA-1, was successful in generating test cases that triggered the entire subject TPs included in the GA group. This shows that all the subject TPs in the GA group are feasible.



a- GA-1 to trigger GA generated TPs

b- GA-1 to trigger randomly generated TPs

c- GA-2 to trigger GA generated TPs

d- GA-2 to trigger randomly generated TPs

e- Rand to trigger GA generated TPs

f- Rand to trigger randomly generated TPs

Figure 4.5: In-Flight EFSM GA and random groups of subject TPs. Plots *a* & *b* show the performance of the GA-1 approach on TPs in GA group and TPs in the random group respectively. Similarly, Plots *c* & *d* and Plots *e* & *f* show the performance of GA-2 and Rand approaches respectively. The fitness metric of subject TPs in random group is plotted by using logarithmic scale.

Furthermore, it is clear that the GA-1 approach could trigger some subject TPs as early as one generation. For the subject TPs in the random group, these are associated with a TP fitness metric values which indicate that these TPs are infeasible. Therefore, The GA-1 approach could not trigger any of them. A close examination of these TPs showed that each TP included at least one instance of the infeasible TP cases and so such TPs cannot be triggered. Figure 4.5b shows that the GA-1 reached the maximum number of generations and none of these TPs was triggered.

Figure 4.5c demonstrates that GA-2 approach could trigger many subject TPs of the GA group. However, there are still TPs that are feasible but not triggered. Furthermore, The GA-2 approach seems to perform only when the TP fitness metric values did not exceed 144 but there are still some TPs that have less than 144 TP fitness metric values and were not triggered. This result states that for this EFSM, the proposed approach, GA-1, also outperformed the alternative approach (GA-2). Since the random group contains only infeasible TPs, GA-2 approach could not trigger any of them as shown in Figure 4.5d.

Rand approach successfully triggered some of the subject TPs in the GA group as shown in Figure 4.5e. Furthermore, all the successfully triggered TPs are associated with TP fitness metric value that did not exceed 132. But some subject TPs are associated with a lower TP fitness but were not triggered. Moreover, the Rand approach exhibited different performance on TPs that have the same TP fitness metric values. When the TP fitness metric value is 72, the average number of the required generations to trigger TPs varied between 1 generation and 830 generations. Nevertheless, the results exhibited by Rand on this EFSM shows that the TP fitness metric approach generated some TPs that are easy to trigger. For the random group of subject TPs, the Rand approach could not trigger any TP since the entire group contains only infeasible TPs (see Figure 4.5f).

Table 4.3 summarises the results achieved by the three test cases generation techniques on the In-Flight EFSM. The proposed approach, GA-1, exhibited the best performance on the GA group of subject TPs where it triggered all the 93 subject TPs (success rate = 100%). The alternative approach, GA-2, exhibited better performance than that observed on the Lift EFSM and had a

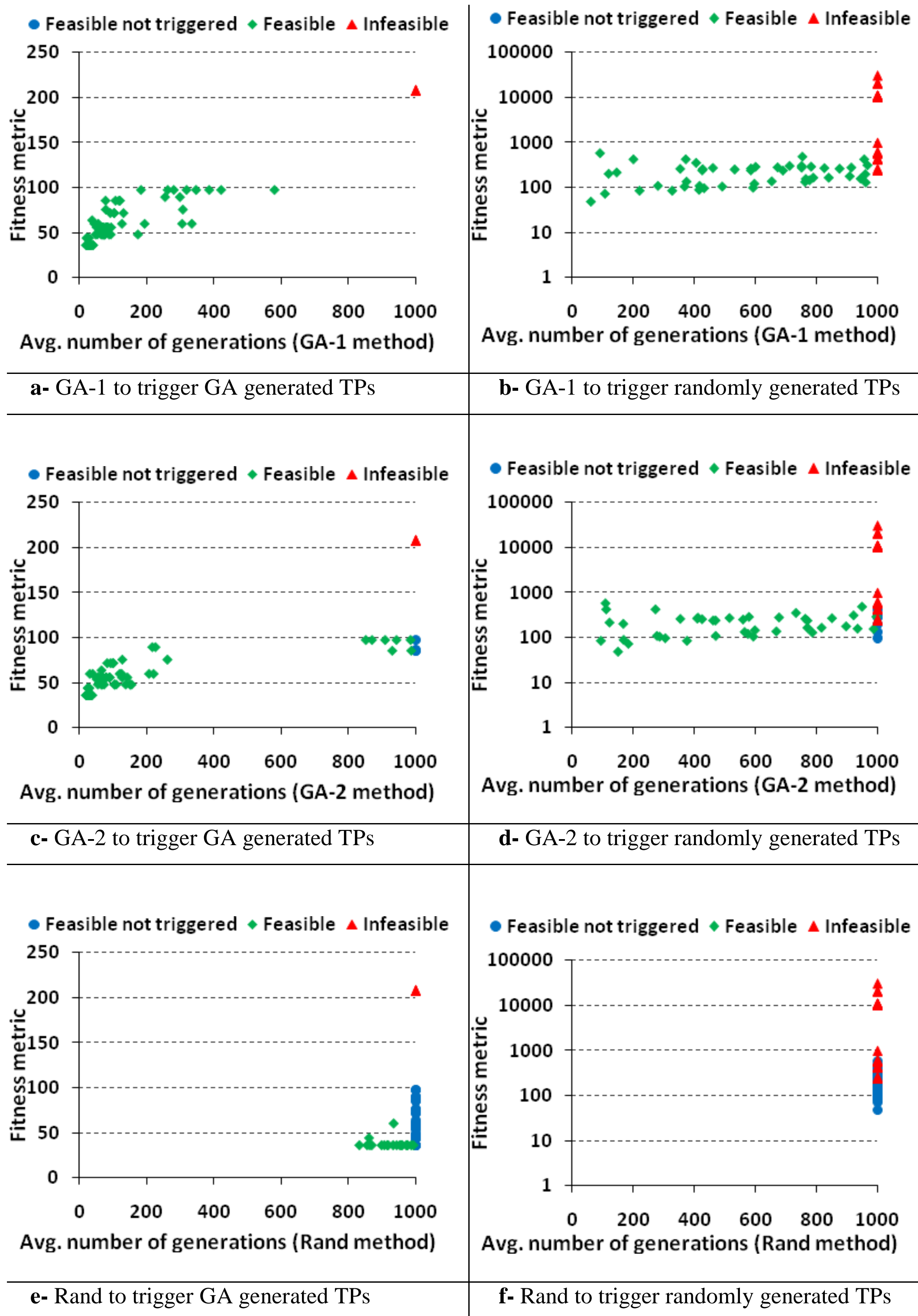
Triggering Method	TP Generation Method	Total TPs Count	Feasible TPs	Triggered	Success rate
GA-1	GA generated TPs (GA group)	93	93	93	= 100%
GA-2				67	≈ 72%
Rand				33	≈ 35.5%
GA-1	Randomly generated TPs (Random group)	93	0	0	= 0%
GA-2				0	= 0%
Rand				0	= 0%
GA-1	Both methods	186	93	93	= 100%
GA-2				67	≈ 72%
Rand				33	≈ 35.5%

Table 4.3: **The performance of three test case generation methods on two groups of subject TPs derived from the In-Flight EFSM.**

success rate approximately 72%. However, it was outperformed by the GA-1 approach. The Rand technique exhibited the worst performance where it triggered only 33 subject TPs (success rate ≈ 35.5%). Nevertheless, the Rand performance on this EFSM was much better than that on the Lift EFSM. This leads to a conclusion that, for the considered EFSM, the TP fitness metric approach generated some subject TPs that are easy to trigger.

4.4.2.3 Results Derived from the ATM EFSM

For this EFSM, each group (GA or random group) contains 90 subject TPs. From Figure 4.6a, the proposed GA-1 approach triggered almost all the subject TPs in the GA group. However, there are three TPs that were associated with the greatest TP fitness metric values (among other TPs) and were not triggered. As reported in the experimental results of Chapter 3, these TPs suffer from the counter problem and are infeasible. By considering only the FTPs in the GA group, the proposed approach, GA-1, did not require, on average, more than 600 generations. This states that for this EFSM, the TP fitness metric approach generated subject TPs (GA group) that are relatively easier to trigger than those produced for the previous two EFSMs. The random group includes subject TPs with greater TP fitness metric values than that seen in the GA group. However, the GA-1 approach triggered all of the FTPs in the random group (Figure 4.6b). All of the subject TPs that belong to the random group and were not triggered, were infeasible because



a- GA-1 to trigger GA generated TPs

b- GA-1 to trigger randomly generated TPs

c- GA-2 to trigger GA generated TPs

d- GA-2 to trigger randomly generated TPs

e- Rand to trigger GA generated TPs

f- Rand to trigger randomly generated TPs

Figure 4.6: ATM EFSM GA and random groups of subject TPs. Plots *a* & *b* show the performance of the GA-1 approach on TPs in GA group and TPs in the random group respectively. Similarly, Plots *c* & *d* and Plots *e* & *f* show the performance of GA-2 and Rand approaches respectively. The fitness metric of subject TPs in random group is plotted by using logarithmic scale.

Triggering Method	TP Generation Method	Total TPs Count	Feasible TPs	Triggered	Success rate
GA-1	GA generated TPs (GA group)	90	87	87	= 100%
GA-2				82	≈ 94.1%
Rand				20	≈ 23%
GA-1	Randomly generated TPs (Random group)	90	49	49	= 100%
GA-2				42	≈ 85.7%
Rand				0	= 0%
GA-1	Both methods	180	136	136	= 100%
GA-2				124	≈ 91.2%
Rand				20	≈ 14.7%

Table 4.4: **The performance of three test case generation methods on two groups of subject TPs derived from the ATM EFSM.**

either these included instance(s) of the infeasible TP cases (TP fitness metric ≥ 10^4) and / or they suffered from the counter problem.

The alternative approach, GA-2, triggered almost all of the FTPs included in the GA group (Figure 4.6c). However, GA-2 required almost the maximum number of generations to perform on FTPs that are associated with the greatest TP fitness metric values (among other FTPs in GA group). Furthermore, the performance of GA-2 on the subject TPs in the GA group is also better than that observed in the previous two EFSMs. From Figure 4.6d, it is also clear that the GA-2 approach performed well on the FTPs included in the random group. However, it failed on some of the FTPs in the random group.

The Rand approach appears to perform poorly on the FTPs included in the GA group (Figure 4.6e) where there are many FTPs that were not successfully triggered. Furthermore, the Rand approach required more than 800 generations for the FTPs that were triggered. From Figure 4.6f, the Rand performance on the random group was the worst and it could not trigger any of the FTPs.

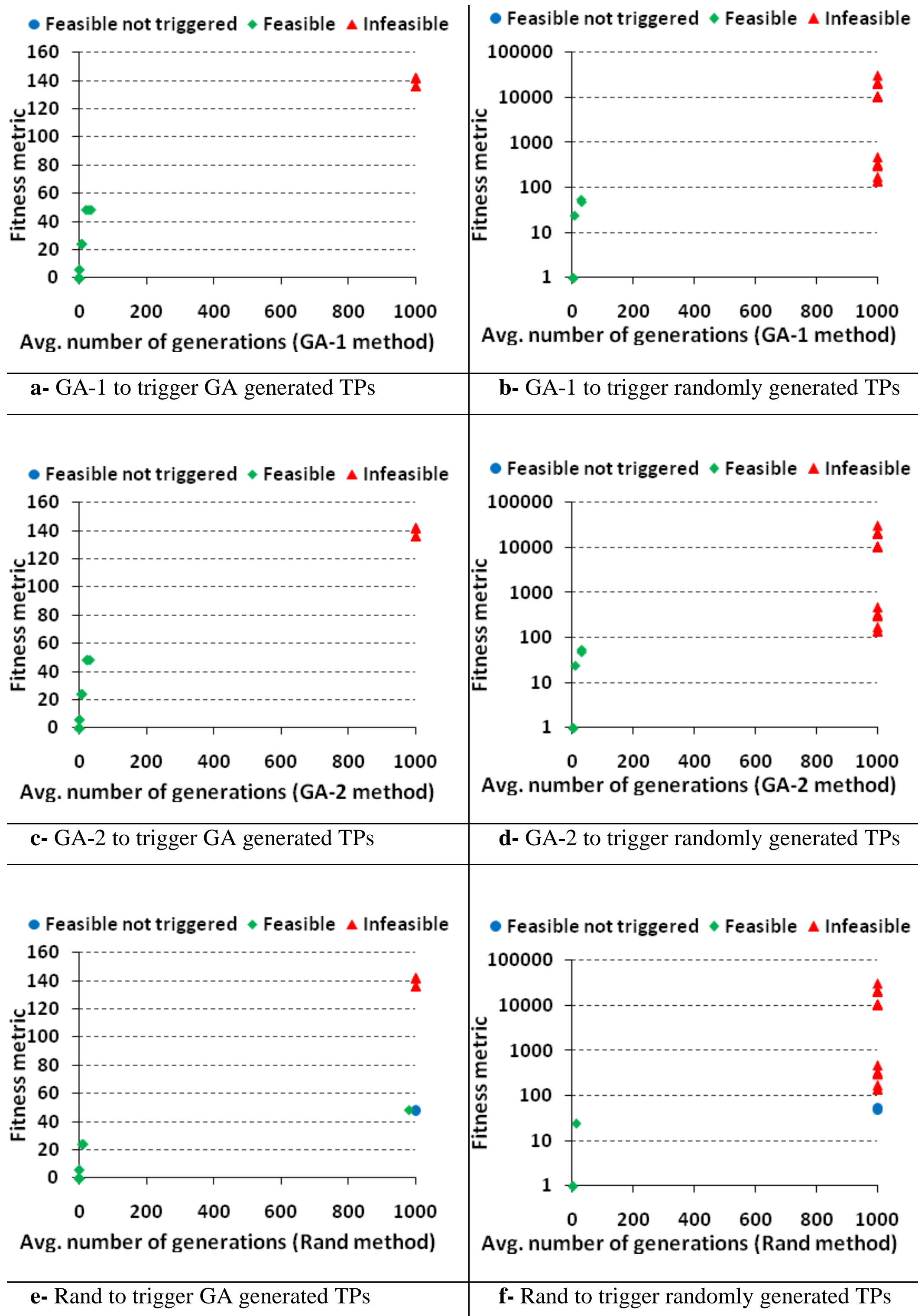
Table 4.4 reports the summary of the results derived from ATM EFSM by the three test cases generators. The best performance was exhibited by the proposed approach. For all the FTPs in both groups (GA and random), the proposed approach, GA-1, had a success rate of 100%. The alternative approach, GA-2, performed relatively similar to the GA-1 approach when subject TPs belonged to the GA group. However it exhibited a slightly worse performance than that of the GA-1 approach when subject TPs were randomly generated. The

overall success rate associated with GA-2 was approximately 91.2%. Finally, the Rand approach exhibited the worst performance with an overall success rate of approximately 14.7%. However, the performance of the Rand approach was much better on GA group of subject TPs than that observed on the random group of subject TPs. This can also support the previous observation that the TP fitness metric approach can help in generating FTPs that are relatively easy to trigger.

4.4.2.4 Results Derived from the Inres Initiator EFSM

For the Inres initiator EFSM, each group of subject TPs contains 45 paths. As shown in Figure 4.7a, the proposed approach, GA-1, triggered almost all the subject TPs included in the GA group. The remaining untaken TPs were infeasible; they suffered from the counter problem. Therefore, for all the FTPs included in the GA group, the GA-1 approach was successful. Furthermore, the average maximum number of generations required by GA-1 to perform did not exceed 36 generations. This can be explained by the fact that the FTPs included in the GA group were associated with low fitness metric values. The performance of GA-1 on the random group of subject TPs is plotted in Figure 4.7b. For all feasible TPs, the GA-1 approach was also successful. Other TPs in the random group that were not triggered were infeasible since they either included an instance(s) of the infeasible TP cases and / or suffered from the counter problem.

Figure 4.7c shows the performance of GA-2 approach on the GA group of subject TPs. The performance exhibited by GA-2 approach on GA group is almost the same as that exhibited by the GA-1 approach. This indicates that when FTPs are associated with relatively low TP fitness metric values (< 55), both approaches GA-1 and GA-2 are likely to perform similarly. This observation is also supported by the GA-2 performance on the random group of subject TPs. Figure 4.7d shows that GA-2 approach performed similarly to the GA-1 approach on the random group of subject TPs where the maximum TP fitness metric value in this group is 54.



a- GA-1 to trigger GA generated TPs | **b-** GA-1 to trigger randomly generated TPs

c- GA-2 to trigger GA generated TPs | **d-** GA-2 to trigger randomly generated TPs

e- Rand to trigger GA generated TPs | **f-** Rand to trigger randomly generated TPs

Figure 4.7: Inres EFSM GA and random groups of subject TPs. Plots *a* & *b* show the performance of the GA-1 approach on TPs in GA group and TPs in the random group respectively. Similarly, Plots *c* & *d* and Plots *e* & *f* show the performance of GA-2 and Rand approaches respectively. The fitness metric of subject TPs in random group is plotted by using logarithmic scale.

Triggering Method	TP Generation Method	Total TPs Count	Feasible TPs	Triggered	Success rate
GA-1	GA generated TPs (all lengths)	45	36	36	= 100%
GA-2				36	= 100%
Rand				34	≈ 94.4%
GA-1	Randomly generated TPs (all lengths)	45	7	7	= 100%
GA-2				7	= 100%
Rand				5	≈ 71.4%
GA-1	Both methods	90	43	43	= 100%
GA-2				43	= 100%
Rand				39	≈ 90.7%

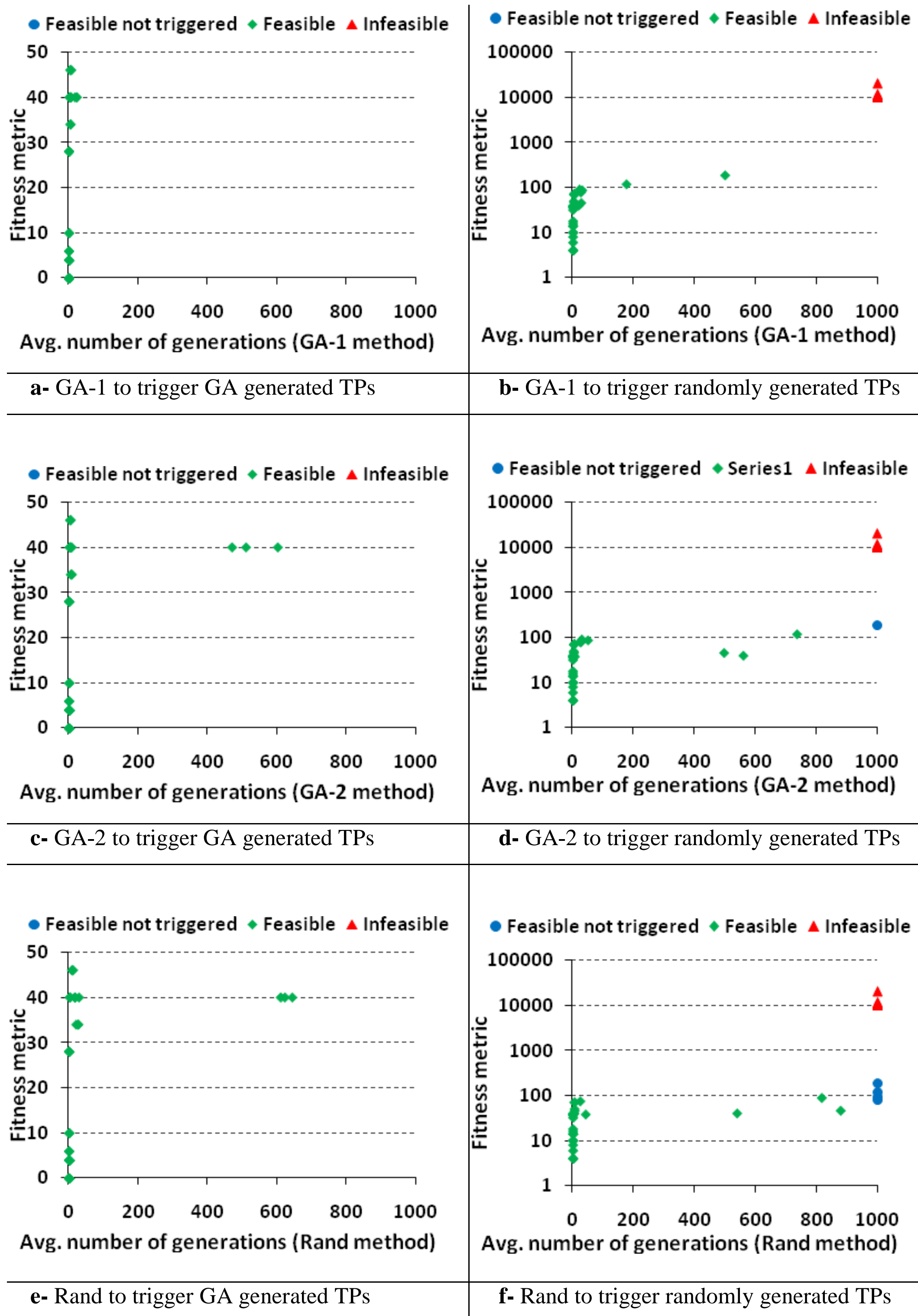
Table 4.5: **The performance of three test case generation methods on two groups of subject TPs derived from the Inres initiator EFSM.**

Rand approach exhibited a better performance than that observed on the previous three EFSMs. For the GA group of subject TPs, Figure 4.7e shows that the Rand approach could trigger almost all of the FTPs (only 2 FTPs were left untaken). Similarly, the Rand approach triggered almost all of the FTPs included in the random group of subject TPs (see Figure 4.7f). The results achieved from the Rand approach support the claim that when the FTPs are associated with low TP fitness metric values, then they are likely to be easy to trigger.

Table 4.5 shows that both the proposed and the alternative approaches could trigger all the subject FTPs and so they have the same overall success rate (100%). The Rand approach performed relatively worse than the GA approaches and the overall success rate was approximately 90.7%. The result achieved from Inres EFSM states that when subject FTPs are associated with low TP fitness metric values (<55), the GA approaches are likely to perform similarly. Furthermore, such FTPs are likely to be easy to trigger by using a random test cases generator.

4.4.2.5 Results Derived from the Class 2 EFSM

The total subject TPs for Class 2 EFSM is 63 in each group (GA or random). For the proposed approach, GA-1, this was able to trigger the entire subject TPs in the GA group relatively quickly (see Figure 4.8a). The maximum average number of



a- GA-1 to trigger GA generated TPs

b- GA-1 to trigger randomly generated TPs

c- GA-2 to trigger GA generated TPs

d- GA-2 to trigger randomly generated TPs

e- Rand to trigger GA generated TPs

f- Rand to trigger randomly generated TPs

Figure 4.8: Class II EFSM GA and random groups of subject TPs. Plots *a* & *b* show the performance of the GA-1 approach on TPs in GA group and TPs in the random group respectively. Similarly, Plots *c* & *d* and Plots *e* & *f* show the performance of GA-2 and Rand approaches respectively. The fitness metric of subject TPs in random group is plotted by using logarithmic scale.

generations required by GA-1 did not exceed 25. However, it is worth noting here that the maximum TP fitness metric value in the GA group is 46. For the random group (Figure 4.8b), the GA-1approach was also successful in triggering all the FTPs. However, the GA-1 required more generations to trigger certain FTPs that were associated with greater TP fitness metric values. The subject TPs that were left untaken were infeasible since they included instances of the infeasible TP cases.

From Figure 4.8c, the GA-2 approach exhibited a similar performance to that of GA-1 on the GA group where it successfully triggered all the subject TPs. Furthermore, GA-2 performance on the random group was similar to that of GA-1 where it triggered all the FTPs included in this group but one FTP. The FTP that was left untaken has the greatest TP fitness metric value in this group (190). The performance of the GA-2 approach on both groups leads to the same conclusion derived from Inres EFSM. That is, when FTPs are associated with TP fitness metric values less than 55, it is likely that the performances of both GA-1 and GA-2, in terms of the triggering capability, are alike.

The Rand approach performance on the GA group is plotted in Figure 4.8e. This approach could also trigger all of the subject TPs included in this group. This can be also related to the observation that these FTPs are associated with low TP fitness metric values. The performance of the Rand approach on the random group is shown in Figure 4.8f. Although the Rand performance was similar to that of GA-2, there are some FTPs that were left untaken. The results of the Rand approach on this EFSM also support the observation that for certain EFSMs, the TP fitness metric approach can produce FTPs that are easy to trigger. Such EFSMs can possibly be determined from the average TP fitness metric value of the derived FTPs (see Table 3.11, pp. 97). As shown in Figure 4.8, the maximum TP fitness metric value did not exceed 50 when TPs were generated by using the TP fitness metric approach.

Table 4.6 shows that the three test cases generation approaches performed similarly on the GA group of subject TPs and triggered all of them. Therefore, these three approaches have a success rate of 100%. For the random group of subject TPs, the GA-1 approach exhibited the best performance and triggered all

Triggering Method	TP Generation Method	Total TPs Count	Feasible TPs	Triggered	Success rate
GA-1	GA generated TPs (all lengths)	63	63	63	= 100%
GA-2				63	= 100%
Rand				63	= 100%
GA-1	Randomly generated TPs (all lengths)	63	37	37	= 100%
GA-2				36	≈ 97.3%
Rand				32	≈ 86.5%
GA-1	Both methods	126	100	100	= 100%
GA-2				99	= 99%
Rand				95	≈ 95%

Table 4.6: **The performance of three test case generation methods on two groups of subject TPs derived from the Class 2 EFSM.**

the FTPs. The performance of GA-2 approach was a little worse than that of GA-1 whereas the Rand approach exhibited the worst performance. The overall success rate of GA-1 approach was 100%, GA-2 approach was 99% and the Rand approach was approximately 95%

4.4.2.6 Summary of the Results

Table 4.7 summarises the results achieved by the three test cases generators on the five EFSM cases studies. The total number of subject TPs that were generated by the TP fitness metric approach (GA group) was 363 TPs. From these TPs, there were 351 FTPs and 12 infeasible TPs (3 TPs belong to ATM and 9 belong to Inres initiator). The proposed approach, GA-1, triggered all of the FTPs in the GA group (success rate = 100%). However, the alternative approach, GA-2, failed on some of these FTPs (success rate ≈ 72.9%). The worst performance was exhibited by the Rand approach where it was able to trigger only 150 FTPs (success rate ≈ 42.7%).

For TPs included in the random group, there were only 93 FTPs out of 363 subject TPs. For all the FTPs included in this group, the proposed approach was also successful and triggered all of them (success rate = 100%). The performance of the alternative approach, GA-2, on the random group was much better than that observed on the GA group where the success rate increased to approximately

Triggering Method	TP Generation Method	Total TPs Count	Feasible TPs	Triggered	Success rate
GA-1	GA generated TPs (all lengths)	363	351	351	= 100%
GA-2				256	≈ 72.9%
Rand				150	≈ 42.7%
GA-1	Randomly generated TPs (all lengths)	363	93	93	100%
GA-2				85	≈ 91.4%
Rand				37	≈ 39.8%
GA-1	Both methods	726	444	444	= 100%
GA-2				341	≈ 76.8%
Rand				187	≈ 42.1%

Table 4.7: **The performance of the three test case generators on the five EFSM case studies.**

91.4%. However, the Rand approach exhibited a slightly worse performance than that observed on the GA group where the success rate decreased to approximately 39.8%.

By considering only the FTPs in both groups (GA and random), the proposed approach exhibited the best performance and triggered all of them (success rate = 100%). The alternative approach, GA-2, was outperformed by the proposed approach where the overall GA-2 success rate was approximately 76.8%. Finally, it was not surprising that Rand approach came last with a success rate of approximately 42.1%.

The results achieved of this experiment suggest that the proposed approach GA-1 is associated with the best performance and can potentially trigger FTPs derived from an EFSM model. The GA-2 approach exhibited better performance on the FTPs that were randomly generated than on the GA group of FTPs. This can be explained by considering the fact that FTPs in the random group were derived from ATM, Class 2 and Inres initiator EFSMs. However, for In-Flight and Lift EFSMs, none of the randomly generated TPs was feasible. By considering only the two groups of FTPs that were derived from these three EFSMs, it is clear that GA-2 exhibited better performance on the GA-group than that on the random group.

The results exhibited by the Rand approach on the GA group of FTPs show a trend that the TP fitness metric approach has the potential to produce FTPs that are relatively easy to trigger (by using a random test cases generator).

4.4.3 Design of the Second Experiment

In designing the second experiment, the aim was to study two questions. The first question is whether there is a relationship between the fitness metric value of an FTP and how much effort, in terms of time, is required by a test cases generator to trigger this FTP. The second question is to check whether the fitness metric of an FTP can predict the effort that is required by a test cases generator to trigger this FTP.

To answer these questions, there are three factors to be considered. The first factor concerns the considered test cases generator approaches. Since the proposed FTP test cases generator, GA-1, could trigger all the generated FTPs, this approach was selected to be used in this experiment. Furthermore, for comparison, it is useful to use another test cases generator approach to understand whether the same conclusion can be drawn for each question even though the test cases generation approach is different. For this, a constraint based testing approach was used. If the constraint based testing approach can trigger all the generated FTPs, then it can also be used in answering the two questions.

The second factor is related to the way in which the considered FTPs were generated. There are two groups of FTPs: (1) the GA group which includes FTPs that were generated by a GA search that implemented the TP fitness metric approach and (2) the random group which includes FTPs that were randomly generated. It is useful to understand whether the two considered questions can be answered regardless of the FTP generation method. Thus, the two questions were separately studied on the GA group of FTPs, on the random group of FTPs and on both groups of FTPs (all the FTPs).

The last factor is based on the observations from the previous experiment. For some FTPs that were associated with the same TP fitness metric value, the average number of GA-1 generations spanned a range of generations (see for example Figure 4.5a). Therefore, it is useful to understand whether clustering FTPs according to their TP fitness metric values can help with these questions. To investigate this, the FTPs in the GA group, the random group and all FTPs (both groups) were clustered by the same fitness metric value. Each cluster contained

FTPs with the same fitness metric value. Then for each cluster, the required time or generations to trigger was the average time or generations of the FTPs in this cluster.

For ease of reference, the proposed GA-1 approach will be denoted henceforth by *GA* while the constraint based testing approach will be denoted henceforth by *CBT*.

For the CBT approach, each FTP was first transformed to a set of constraints as described in Subsection 4.3.3. Then a solver, constrained nonlinear minimisation *fmincon* (Matlab, 1984-2010), was applied. A detailed description of the solver's parameters is provided in the Matlab's website. However, the values that were used are recorded here for the purpose of experiment replication. Each FTP was a set of constraints of two types: equality and inequality. The range of values that the solver can search was set to [0..1000] while the initial solutions of the considered variables were randomly generated in the range [0..1000]. For each FTP, the solver was called ten times to try to find the required test case and then the average time required by the solver in the ten tries was calculated.

In order to answer the experiment questions, statistical software was used (SPSS). For the first question, the Pearson correlation was selected. Although Spearman correlation can be also used, Spearman correlation was found to yield mostly higher correlation values. Therefore, Pearson correlation was selected since it gave the worst case correlation. Pearson correlation was performed between FTP fitness metric values and the CBT time, the GA time, and the GA generations that are required to trigger these FTPs. In Pearson correlations, there are two main outputs. The first output is the correlation coefficient r and the second output is the p value. The r value can be in the range [-1..1] where the range boundaries state a perfect correlation. Other values of r are classified to three categories (Cohen, 1988): (1) a small correlation when $0.10 \leq r \leq 0.29$, (2) a medium correlation when $0.30 \leq r \leq 0.49$ and (3) a large correlation when $r \geq 0.50$. The p value determines the confidence in the results where $p < 0.05$ denotes a statistically significant result. In this thesis the correlation value is only considered if the result is statistically significant (when $p < 0.05$).

Since there is only one considered factor (TP fitness metric value) to be used to estimate other factors (GA time and CBT time), a linear regression analysis can be performed. In linear regression, there are two variables: independent variable or the explanatory and dependent variable to be estimated. In linear regression there are two hypotheses: the null hypothesis which states that the independent variable has a zero impact on the dependant variable whereas the alternative hypothesis states that the independent variable does impact the dependent one. The important outputs of a linear regression are:

1. The coefficient of determination or *R Squared* (*RS*) which states how much the independent variable is capable of explaining the variance in the dependent variable. The range of *RS* values is [0..100]%.
2. The *F* ratio significance (*Sig*) which states the confidence $(1 - Sig)$ by which the null hypothesis can be rejected (usually when $Sig < 0.05$).
3. The line fit plot which shows the trend by which the estimation can be performed.

4.4.4 Results of the Second Experiment

This Section reports the experimental results obtained from applying the CBT approach to each group of FTPs. Then, the results of the correlation and linear regression analysis are presented.

4.4.4.1 CBT Performance on Both Groups of FTPs

The CBT approach was applied to each group of FTPs (351 FTPs in the GA group and 93 FTPs in the random group). For each FTP, the approach was applied ten times to try to solve the FTP's constraints and hence to find a test case that can exercise this FTP. Figure 4.9 shows the performance of CBT approach on both groups of FTPs. Each plot shows an FTP fitness metric value against the average time in seconds that is required by the CBT in ten tries to solve an FTP's constraints.

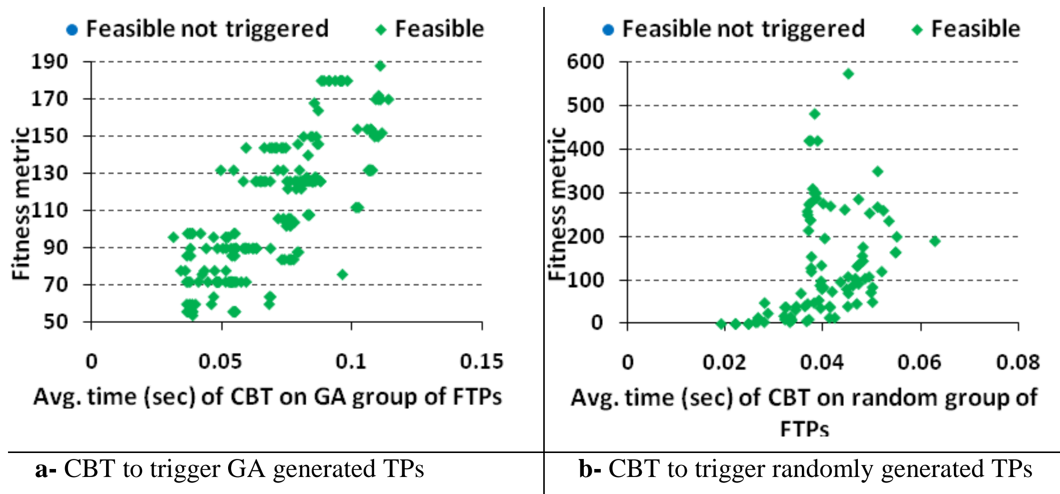


Figure 4.9: The CBT performance on both groups of FTPs. Plot *a* shows the performance on the GA group of FTPs while Plot *b* shows the performance on the random group of FTPs.

Figure 4.9a shows that CBT approach was successful in triggering all the FTPs that are included in the GA group. Furthermore, the maximum average time that was required for CBT to perform did not exceed 0.15 of a second. From Figure 4.9b, a similar observation can be noticed. The CBT approach successfully trigged all of the FTPs in the random group. Also, the maximum average time did not exceed 0.08 of a second[3]. The results achieved by the CBT approach suggest that this method can also be used as a second test cases generator approach to answer the considered two questions.

4.4.4.2 Correlation Study (Without FTP Clustering)

A study of the correlation among the FTPs fitness metric values and their corresponding GA average time (sec), GA average generations, and average CBT time (sec) are reported in Tables 4.8, 4.9 and 4.10.

Table 4.8 shows the correlations derived from the FTPs in the GA group. From this table, all the correlation values are statistically significant at $p < 0.01$. The achieved correlation shows that there was a strong positive correlation

[3] All experiments were conducted on a PC with Windows® XP Service Pack 3 OS, Intel® Pentium® 4 CPU 2.80 GHz 2.79 GHz and 1.24 GB of RAM

	Fitness metric	GA Gen.	GA time	CBT time
Fitness metric	1			
GA Gen.	0.791*	1		
GA time	0.798*	0.999*	1	
CBT time	0.864*	0.847*	0.851*	1

* Correlation is significant at the $p < 0.01$ level (2-tailed).

Table 4.8: **GA group of FTPs (no clustering) - Correlation among FTPs' fitness metric, GA average generations, GA average time and CBT average time.**

($r = 0.798$) between the fitness metric and the required GA time in seconds to trigger the FTPs. Similarly, there was a strong positive correlation ($r = 0.791$) between the fitness metric and the required GA generations to trigger the FTPs. The relationship between the GA time and GA generations was found to be almost perfect ($r = 0.999$). Furthermore, the correlations between the fitness metric and the required CBT time in seconds to trigger the FTPs was also found to be positive and strong ($r = 0.864$). This shows that for the considered test cases generators (GA and CBT), there was a strong agreement on their performances on the FTPs included in the GA group. Greater fitness metric values were associated with more GA time or CBT time.

Table 4.9 shows the correlations achieved from the FTPs in the random group. All the correlation values were found to be statistically significant at $p < 0.01$. The correlation between the fitness metric and the required GA time was positive and strong ($r = 0.650$). Similarly, the correlation between the fitness metric and the required GA generations was also positive and strong ($r = 0.650$). Thus, the correlation between the GA time and GA generations was almost perfect ($r = 0.999$). However, the correlation between the fitness metric and the required CBT time to trigger the FTPs was found to be positive but medium ($r = 0.363$). Similarly, the agreement in the performance between GA time and CBT time was found to be positive and medium ($r = 0.414$). The results achieved from the random group of FTPs suggest that greater fitness metric values were associated with more GA time and generations.

	Fitness metric	GA Gen.	GA time	CBT time
Fitness metric	1			
GA Gen.	0.650*	1		
GA time	0.650*	0.999*	1	
CBT time	0.363*	0.414*	0.414*	1

* Correlation is significant at the $p < 0.01$ level (2-tailed).

Table 4.9: **Random group of FTPs (no clustering) - Correlation among FTP fitness metric, GA average generations, GA average time and CBT average time**.

Table 4.10 shows the results of the correlation study achieved from both groups of FTPs. Similar to the previous results, all correlations were found to be statistically significant at $p < 0.01$. There were strong positive correlations between the fitness metric and both GA time ($r = 0.642$) and GA generations ($r = 0.654$). Also, the correlation between GA time and GA generations was almost perfect ($r = 0.999$). The correlation between the CBT time and the fitness metric was found to be positive and medium ($r = 0.454$). However, the correlation between the CBT time and the GA time was found to be positive and strong ($r = 0.757$). This indicates that more GA time was associated with more CBT time.

The results obtained from the correlation study raised a question about the difference in the correlations between the fitness metric and the CBT time which were observed on the GA group of FTPs ($r = 0.864$) and on the random group of FTPs ($r = 0.363$). This can be partially explained by the fact that the random group of FTPs did not include all the considered EFSMs (the In-Flight and Lift EFSMs were excluded since no randomly generated TP was an FTP). This, in turn, may mean that the random group is less representative. However, the latter reason cannot fully explain the difference in the performance of CBT approach on both groups. Since the CBT approach considers only two types of constraints: equality and inequality, these two factors can be used to provide some insights about the performance of the CBT approach.

The number of equality and inequality constraints was calculated for each FTP in each group. Then, a correlation study was performed again. For GA groups of FTPs, the total number of inequality constraints is 3035 whereas the

	Fitness metric	GA Gen.	GA time	CBT time
Fitness metric	1			
GA Gen.	0.654*	1		
GA time	0.642*	0.999*	1	
CBT time	0.454*	0.747*	0.757*	1

* Correlation is significant at the $p < 0.01$ level (2-tailed).

Table 4.10: **Both groups of FTPs (no clustering) - Correlation among FTP fitness metric, GA average generations, GA average time and CBT average time**.

total number of equality constraints is 385. This gives a ratio of approximately 7.9 and thus the prevalence of the inequality constraints is greater than that of equality constraints by approximately 7.9 times. Each FTP in the GA group has an average about 9 inequality constraints and 1 equality constraint. Similarly, for the random group of FTPs, the total number of inequality constraints is 291 and the total number of equality constraints is 332. The prevalence of the equality constraints is greater than that of inequality constraints by approximately 1.1 times. That is, each FTP in the random group has in average about 4 equality constraints and 3 inequality constraints.

From Table 4.11 and for the GA group of FTPs, the fitness metric was found to have a strong positive correlation (r = 0.842) with the number of inequality constraints. Nevertheless, the correlation between the fitness metric and the number of equality constraints was positive but small (r = 0.156). For the GA time, this was also found to have a strong positive correlation with the number of inequality constraints (r = 0.940). However, the correlation between the GA time and the number of equality constraints was found to be positive but small (r = 0.172). Similarly, the correlation between the CBT time and the number of inequality constraints was strong and positive (r = 0.849) but interestingly there was no significant correlation ($p > 0.05$) between the CBT time and the number of equality constraints.

For the random group of FTPs, the fitness metric was found to have a strong positive correlation with the number of equality constraints (r = 0.688). But it was found to have a positive small correlation with the number of inequality constraints (r = 0.261). The GA time was strongly correlated with the number of

FTPs		Fitness metric	GA Gen.	GA time	CBT time	Inequality Cons.	Equality Cons.
GA group of FTPs	**Inequality Cons.**	0.842*	0.938*	0.940*	0.849*	1	
	Equality Cons.	0.156*	0.177*	0.172*	0.088	-0.013	1
Random group of FTPs	**Inequality Cons.**	0.261*	0.197	0.201	0.619*	1	
	Equality Cons.	0.688*	0.951*	0.952*	0.357*	0.036	1
Both groups of FTPs	**Inequality Cons.**	0.436*	0.819*	0.828*	0.894*	1	
	Equality Cons.	0.553*	0.395*	0.381*	-0.031	-0.108	1

* Correlation is significant at the $p < 0.01$ level (2-tailed).

Table 4.11: FTPs (no clustering) - Correlation among FTP fitness metric, GA average generations, GA average time, CBT average time, number of inequality constraints and number of equality constraints.

equality constraints ($r = 0.952$) but it did not have a correlation with inequality constraints. Importantly, the CBT time was strongly correlated with the number of inequality constraints ($r = 0.619$) but it had a medium correlation with the number of equality constraints ($r = 0.357$).

Finally, for both groups of FTPs, the fitness metric was strongly correlated with the number of equality constraints ($r = 0.553$) but this has a medium correlation with the number of inequality constraints ($r = 0.436$). For GA time, this was strongly correlated with the number of inequality constraints ($r = 0.828$) but it had a medium correlation with equality constraints ($r = 0.381$). Importantly, the correlation between the CBT time and the inequality constraint was strong ($r = 0.894$) but there was no correlation between the CBT time and the number of equality constraints.

The results of the correlation study achieved by considering the two types of constraints show that the performance of the CBT approach was always correlated with the number of inequality constraints. In order to investigate this further, a close examination was conducted on the form of the equality constraints in the considered FTPs. It transpired that the majority of the equality constraints are of the form *parameter* = *constant*. Such constraints represent assignments that

the solver (CBT approach) has to apply before starting to find suitable values of the parameters in the inequality constraints. Therefore, the inequality constraints seem most likely to affect the time that is required by the solver. Another look at the correlations in Table 4.11 shows that in all cases the CBT time was strongly attached to the number of inequality constraints. For the GA group of FTPs, the prevalence of the equality constraints is minimal and so this is potentially the reason of there not being a correlation between the CBT time and the number of equality constraints. However, for the random group of FTPs, the prevalence of the equality constraints was dominant but the CBT time was still strongly correlated to the number of the inequality constraints.

In contrast, an equality constraint can be considered the hardest type for a GA search to satisfy. This explains why, for the random group of FTPs, the GA time was strongly attached to the number of equality constraints (since these were dominant). However, when the equality constraints were minimal, as the case in the GA group of FTPs, the GA time was strongly correlated with the other type (inequality).

The fitness metric seemed also to have statistically significant correlation with both types of constraints. When the number of equality constraints was minimal (GA group of FTPs), there was a small correlation between the fitness metric value and the number of equality constraints. Similarly, when the number of inequality constraints was minimal (Random group of FTPs) there was a small correlation between the fitness metric value and the number of inequality constraints. This can be explained by considering the fitness metric calculation. That is, the fitness metric penalises equality constraints with larger values than that of inequality constraints. Therefore, having more equality constraints results in greater fitness metric values. This is also the case in the random group of FTPs where the fitness metric was strongly correlated to the equality constraints. However, the equality constraints seem not to impact the CBT approach.

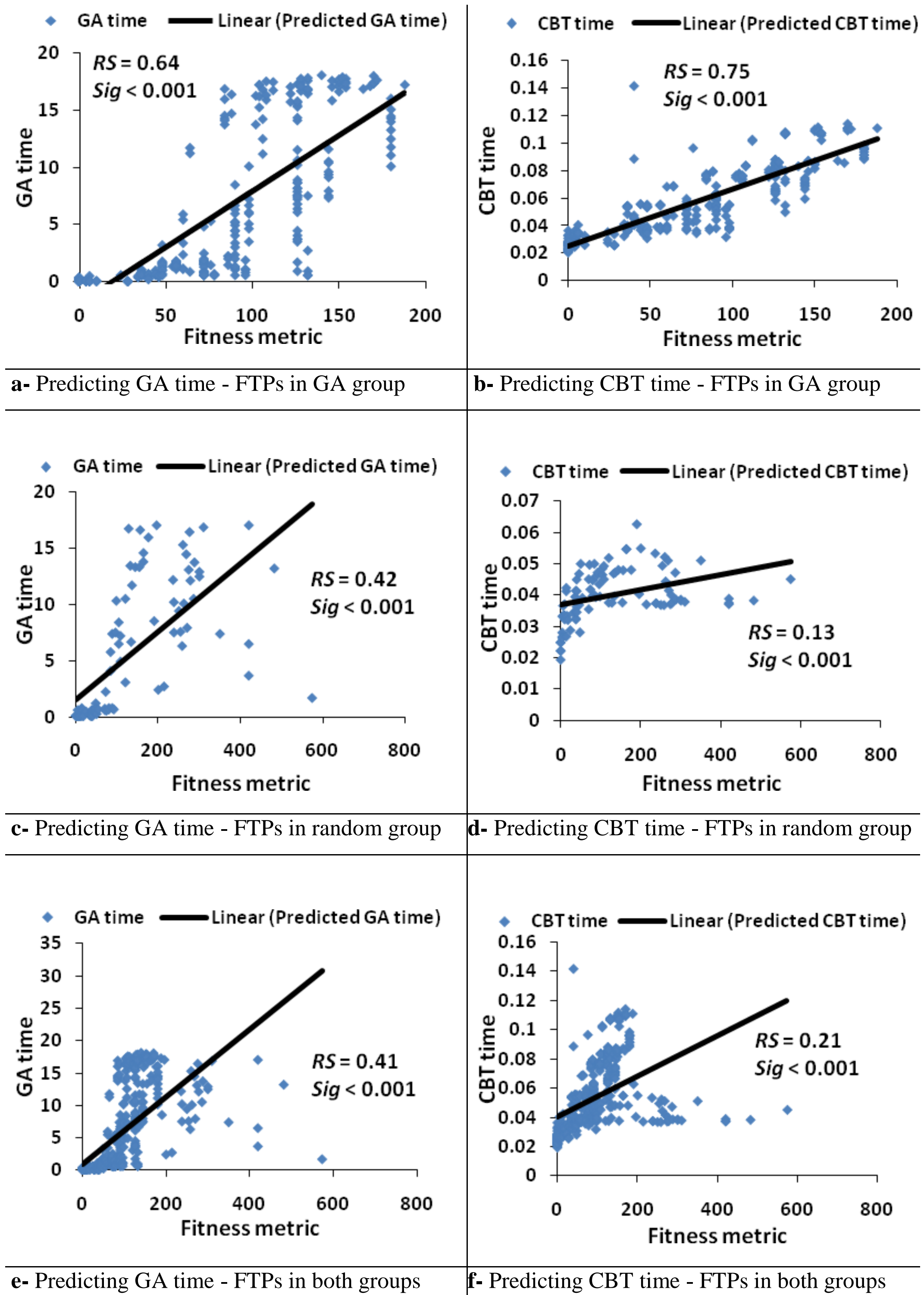
A very recent study by (Zhao et al., 2010) focused on determining factors that affect the efficiency of a search-based approach when generating test cases to trigger FTPs through an EFSM. The study concluded that the total number of numerical equality operators found in the guards of a given FTP has a vital role in

estimating the test efficiency. Nevertheless, this factor depends on the prevalence of such operators in the first place. By considering Table 4.11, if there are few of such operators, then the test efficiency is associated with the number of inequality constraints. Furthermore, when other test cases generators are considered, such as the CBT approach, the number of numerical equality operators does not seem to be that important.

The results of the correlation study can answer the first question about the relationship between the fitness metric and how much effort, in terms of time, that is required by a test cases generator to perform. However the answer does vary with the FTPs generation approach. For FTPs that were generated by a GA search which implemented the TP fitness metric approach, there was a strong positive correlation between the fitness metric and the time required by a test cases generator (CBT or GA) to trigger the FTPs. The strength of the relationship was almost the same even though different test cases generators were used. However, this was not the case when the FTPs were randomly generated. While the correlation between the fitness metric and the GA time was found to be positive and strong, the correlation was medium between the fitness metric and the CBT time (see Table 4.9). As aforementioned, this can partially be explained by considering that two EFSMs were excluded in the random group of FTPs. Therefore, the random group was not fully representative. Nevertheless, two other factors were found to affect the CBT time. These are the number of equality and inequality constraints in the considered FTPs. The correlation between the CBT time and the number of inequality constraints was found to always be greater than that between the CBT time and number of equality constraints. However, there were relatively few inequality constraints in the random group of FTPs and so CBT approach performed relatively similar on all the FTPs that are included in this group (see Figure 4.9b).

4.4.4.3 Regression Analysis (Without FTP Clustering)

The liner regression analysis was performed on each group of FTPs (GA and random) and then on all the FTPs from both groups. The independent variable was the fitness metric whereas the dependent variables were the GA time and CBT



a- Predicting GA time - FTPs in GA group

b- Predicting CBT time - FTPs in GA group

c- Predicting GA time - FTPs in random group

d- Predicting CBT time - FTPs in random group

e- Predicting GA time - FTPs in both groups

f- Predicting CBT time - FTPs in both groups

Figure 4.10: Fitness line fit plots of regression analysis (without clustering). Plots *a*, *c* & *e* are the prediction of GA time in seconds from FTPs in GA group, random group and both groups respectively. Similarly, Plots *b*, *d* & *f* are the prediction of CBT time in seconds from the same groups respectively.

time. The number of GA generations was not considered in this analysis since this was found to have almost a perfect correlation with GA time.

Figure 4.10 reports the results of the regression analysis in terms of *RS* and *Sig* values and also the fitness line fit plot. Figure 4.10a and Figure 4.10b show the predictions of GA time and CBT time respectively by using the GA group of FTPs. From Figure 4.10a, the regression analysis reported that the fitness metric contributes significantly to the prediction of the GA time ($Sig < 0.001$) and so the null hypothesis is rejected. Furthermore, the fitness metric explains 64% of the variance in the GA time. Figure 4.10b also shows that the fitness metric contributes significantly to the prediction of the CBT time ($Sig < 0.001$). The fitness metric here explains 75% of the variance in the CBT time.

Figures 4.10c and 4.10d report the regression analysis results obtained from the random group of FTPs for GA time and CBT time respectively. The fitness metric makes a significant contribution ($Sig < 0.001$) to the prediction of GA time (see Figure 4.10c). 42% of the variance in the GA time is explained by the fitness metric. However, for the CBT time, the fitness metric also makes a significant contribution ($Sig < 0.001$) but this explains only 13% of the variance in the CBT time (see Figure 4.10d).

The regression analysis results derived from both groups of FTPs for GA time and CBT time are shown in Figure 4.10e and Figure 4.10f respectively. From these figures, the fitness metric contributes significantly to the predictions of both GA time and CBT time. However, the fitness metric explains more of the variance in GA time (41%) than that of the CBT time (21%).

The results observed from the regression analysis lead to two important findings. When the FTPs are generated by using the GA search that implemented the proposed TP fitness metric approach, then the fitness metric has the potential to predict the effort of a test cases generation approach (explains from 64% to 75% of the variance in effort). However, when the FTPs are randomly generated, then the fitness metric has a relatively poor prediction capability of CBT time and medium prediction capability of GA time. When merging both groups of FTPs, the random group of FTPs seems to negatively impact the GA group of FTPs and

thus the prediction capability of the fitness metric on both groups of FTPs was similar to that observed on the random group of FTPs.

4.4.4.4 Correlation Study (Clustered FTPs)

For each group of FTPs (GA and random groups) and for all the FTPs (both groups) the FTPs were clustered according to their fitness metric values. For each cluster, the corresponding GA time, GA generations and CBT time values were averaged. The correlation study was then conducted again on the clustered FTPs in the GA group, the clustered FTPs in the random group and on all the FTPs after they were clustered.

Table 4.12 shows the correlation results obtained from the clustered FTPs in the GA group. All the achieved correlations were statistically significant at $p < 0.01$. The fitness metric had strong positive correlations with both the GA time ($r = 0.851$) and GA generations ($r = 0.847$) respectively. Furthermore, the correlation between the fitness metric and the CBT time was also positive and strong ($r = 0.904$). Moreover, more GA time or generations was strongly associated with more CBT time ($r = 0.919$). Compared to the correlation results obtained from the GA group of FTPs without clustering, these values clearly show an improvement in the strength of these correlations (see Table 4.8). Therefore, clustering the FTPs in the GA group has led to stronger associations among the considered factors.

For the clustered FTPs in the random group, Table 4.13 shows that all the achieved results were statistically significant at $p < 0.01$. The fitness metric was found to have a strong positive correlation with both the GA time and GA generations ($r = 0.582$). However, the fitness metric and the CBT time were in a small positive correlation ($r = 0.297$). Furthermore, the GA time and the CBT time were found to have a medium positive correlation ($r = 0.348$). Compared to the correlation results obtained from the random group of FTPs without clustering, all the correlations, apart from that between GA time and GA generations, have been worsened (see Table 4.9). This is different from the effect of clustering as observed on the GA group. In order to understand this, the number of the clusters in each group of FTPs was considered. The total number of FTPs in

	Fitness metric	GA Gen.	GA time	CBT time
Fitness metric	1			
GA Gen.	0.847*	1		
GA time	0.851*	1.000*	1	
CBT time	0.904*	0.919*	0.919*	1

* Correlation is significant at the $p < 0.01$ level (2-tailed).

Table 4.12: **GA group of FTPs (clustered) - Correlation among FTPs' fitness metric, GA average generations, GA average time and CBT average time.**

the GA group is 351 which include 46 unique fitness metric values. Thus the total number of clusters in the GA group is 46 clustered FTPs (approximately 87% reduction rate). In contrast, the total number of FTPs in the random group is 93 which include 62 unique fitness metric values. Thus, the total number of clusters in the random group is 62 clustered FTPs (approximately 33% reduction rate). By considering the performance of clustering on the GA group of FTPs, it is clear that clustering had less of an effect on the random group of FTPs. Thus, clustering cannot be expected to have a similar impact on the correlation results derived from both clustered groups of FTPs.

For all clustered FTPs (both groups), Table 4.14 shows that the fitness metric had a statistically significant medium positive correlations with both GA time (r = 0.482) and GA generations (r = 0.494). Furthermore, there was a statistically significant strong positive correlation between the GA time and CBT time (r = 0.655). However, there was no statistically significant correlation between the fitness metric and the CBT time. In Table 4.14 the correlation

	Fitness metric	GA Gen.	GA time	CBT time
Fitness metric	1			
GA Gen.	0.582*	1		
GA time	0.582*	1.000*	1	
CBT time	0.297*	0.350*	0.348*	1

* Correlation is significant at the $p < 0.01$ level (2-tailed).

Table 4.13: **Random group of FTPs (clustered) - Correlation among FTP fitness metric, GA average generations, GA average time and CBT average time**.

	Fitness metric	GA Gen.	GA time	CBT time
Fitness metric	1			
GA Gen.	0.494*	1		
GA time	0.482*	0.999*	1	
CBT time	0.047	0.642*	0.655*	1

* Correlation is significant at the $p < 0.01$ level (2-tailed).

Table 4.14: **Both groups of FTPs (clustered) - Correlation among FTP fitness metric, GA average generations, GA average time and CBT average time**.

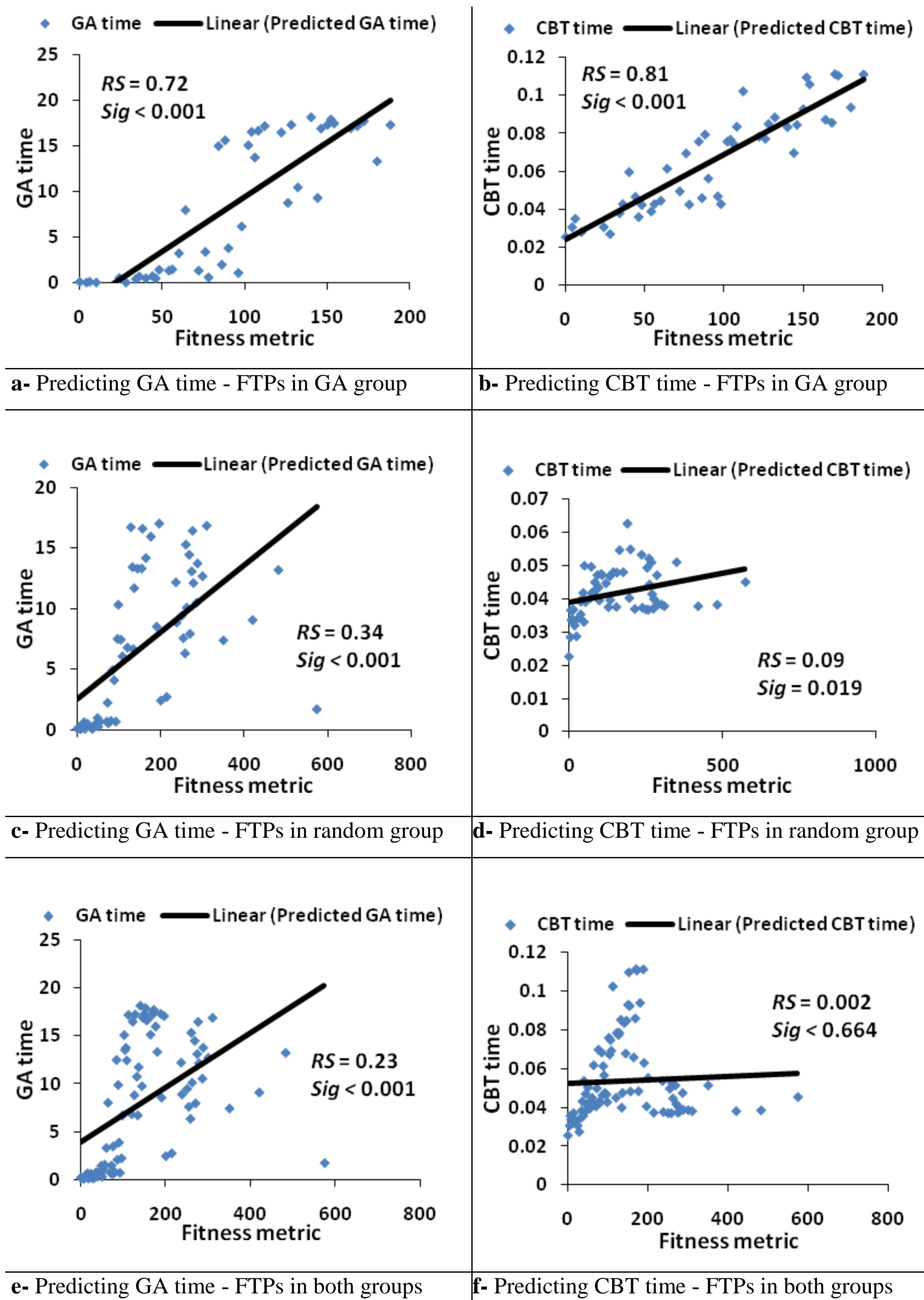
strength among the considered factors was worse than that observed before the clustering was applied (see Table 4.10). This indicates that clustering all the generated FTPs did not improve the correlation strength.

The results achieved from the correlation study, after clustering was applied, show that for the GA group of FTPs, the clustering technique can help strengthening the correlation between the fitness metric and both the GA time (or GA generations) and CBT time. A similar impact on the correlation between the GA time (or GA generations) and CBT time was also observed. Nevertheless, for the randomly generated FTPs, the clustering technique worsened the correlation strength among the fitness metric and both the GA time (or GA generations) and CBT time. It was also found to worsening the correlation strength between the GA time (or GA generations) and the CBT time. A similar negative impact of the clustering was also observed when all the generated FTPs were considered.

4.4.4.5 Regression Analysis (Clustered FTPs)

The linear regression was applied to each clustered group of FTPs and also to all the generated FTPs (both groups) after they were clustered.

Figure 4.11 reports the linear regression analysis of the fitness metric and the GA time and also of the fitness metric and the CBT time on each group. For the clustered FTPs in the GA group, Figure 4.11a shows that the fitness metric contributes significantly to the prediction of GA time (*sig* < 0.001). Furthermore, the fitness metric is found to explain 72% of the variance in the GA time. Similarly, Figure 4.11b reports that the fitness metric makes a significant



a- Predicting GA time - FTPs in GA group

b- Predicting CBT time - FTPs in GA group

c- Predicting GA time - FTPs in random group

d- Predicting CBT time - FTPs in random group

e- Predicting GA time - FTPs in both groups

f- Predicting CBT time - FTPs in both groups

Figure 4.11: Fitness line fit plots of regression analysis. Plots *a*, *c* & *e* are the prediction of GA time in seconds from clustered FTPs in GA group, random group and both groups respectively. Similarly, Plots *b*, *d* & *f* are the prediction of CBT time in seconds from the same groups respectively.

contribution to the prediction of the CBT time (*sig* < 0.001). 81% of the variance in the CBT time is explained by the fitness metric.

For the clustered FTPs in the random group, Figure 4.11c indicates a significant contribution of the fitness metric to the prediction of the GA time (*sig* < 0.001). However, the fitness metric explains only 34% of the variance in the GA time. For the prediction of CBT time, Figure 4.11d shows that the fitness metric did not actually contribute to the prediction of the CBT time and the null hypothesis cannot be rejected (*sig* = 0.019).

When all the FTPs were clustered, there is a significant contribution of the fitness metric to the prediction of the GA time. However, a small amount of the variance in the GA time is explained by the fitness metric (23%) (see Figure 4.11e). This is not the case for the prediction of the CBT time where the fitness metric is not found to impact the CBT time and the null hypothesis cannot be rejected (*Sig* = 0.664).

Compared to the linear regression analysis that was performed without FTPs being clustered (see Figure 4.10), the fitness metric is better able to predict both the GA time and CBT time when clustering was applied to the FTPs in the GA group. In contrast, when FTPs in the random group were clustered, the fitness metric is less able to predict the GA time. Furthermore, the fitness metric cannot be used to predict the CBT time. The same observations are also seen when all the FTPs were clustered. The results achieved from the linear regression, before and after FTP clustering was applied, lead to two findings. First, when FTPs are generated by a GA search that implemented the proposed TP fitness metric, then clustering improves the prediction capability of the fitness metric for both the GA time and CBT time. Second, if FTPs are randomly generated, then clustering worsens the prediction capabilities of the fitness metric for both the GA time and CBT time. The same negative impact on the prediction capability of the fitness metric can also be observed when all the generated FTPs are clustered.

4.5 Conclusion

Although the EFSM is a powerful modelling approach which has been widely applied, testing from this model is a challenging task. One part of the problem is related to finding a set of test cases that can follow the selected feasible transition paths through the model. Despite the fact that search-based testing techniques have proven to be effective in automating aspects of testing, previously these have mainly been applied to white-box testing.

This chapter addressed the problem of generating test cases that can trigger a given feasible path in an EFSM. The proposed approach treated transitions in an EFSM model as functions. Then the problem of test cases generation became a search for suitable test cases to be applied to a sequence of functions. The fitness of a test case has two components. The first is the function distance which measures how close a given input to a particular transition was to triggering this transition. The second component is the function approach level which determines how far the whole set of path inputs was to reaching the target (executing the last transition in a path).

The proposed test cases generation approach was applied to a set of subject TPs that were generated from five EFSM case studies. Also, an alternative test cases generation approach (taken from literature) and a random test cases generator were applied for comparison.

In the experiment the proposed approach was successful in triggering all the considered FTPs. Furthermore, it was found that when FTPs were generated by using the proposed TP fitness metric approach, a random test data generator was successful in triggering many of them. Moreover, the proposed approach was found to be superior to the other two used search-based test cases generators.

The chapter also studied the relationship between the fitness metric and the time which is required by two test cases generators: the proposed approach (GA) and a constraint based testing approach (CBT) to execute the FTP. Furthermore, the capability of the fitness metric to predict the required time to trigger a given FTP was also investigated.

The results showed that when FTPs are generated by using the proposed TP fitness metric approach (GA group of FTPs), there was a strong positive correlation between the fitness metric and the required time to execute FTPs by both approaches (GA and CBT). Also, the fitness metric showed a strong capability to predict the required time by both approaches. However, when FTPs were generated randomly, the correlation strength between the considered factors was weaker than that observed on the GA group of FTPs. Similarly, the predication capability of the fitness metric was also found to be much less than that observed on the GA group. Similar results to that achieved from the random group of FTPs were also achieved when all the FTPs were considered (both groups).

Finally, the correlation strength and the prediction capability were found to be stronger when the FTPs in the GA group were clustered by the same fitness metric values. Nevertheless, clustering was found to worsen the correlation strength and the predication capability on the random group of FTPs and so was when all the FTPs were considered.

Future work in this area is to investigate the performance of the proposed test cases generator when FTPs are associated with greater TP fitness metric values (i.e. producing TPs which have large fitness metric values but are still feasible). Another point is to redo the correlations and the predication capability on such set of FTPs.

Chapter 5: Generating Feasible Transition Paths for Testing from EFSMs with Counter Problem

5.1 Introduction

The extended finite state machine (EFSM) is a powerful approach for modelling that has been found to be suitable for state-based systems. Automatic test generation from an EFSM can be approached by generating a set of transition paths (TPs) that satisfy a test criterion and then finding test cases that exercise these paths. Nevertheless, generating feasible transition paths (FTPs) for model based testing is a challenging task and is an open research problem. One important problem is the existence of a transition with a guard that references a counter variable whose value depends on previous transitions. The presence of such transitions in paths can lead to infeasible paths and this has been reported by several authors (Chanson and Zhu, 1993, Derderian et al., 2010, Kalaji et al., 2010). Furthermore, previous studies have shown that the state problem can adversely affect methods that automate test generation in white-box testing (McMinn, 2005, McMinn and Holcombe, 2003, McMinn and Holcombe, 2005, Zhan and Clark, 2006). The problem is that it is not possible to derive an FTP to exercise such a transition without determining which other transitions are involved in setting the value of the counter variable and how many times they need to be called. This chapter proposes a novel approach based on data and control analysis to bypass the counter problem. It achieves this by automatically determining whether a given TP includes a transition whose guard references a counter variable. If so, the approach determines which other transitions must exist

and how many times they must occur so that the considered guard over the counter is satisfied. The proposed approach is evaluated by being used in a genetic algorithm to guide the search for FTPs.

The chapter starts by describing the problem. Then Section 3 reintroduces two case studies, Inres initiator and ATM EFSMs, that suffer from the counter behaviour. The proposed approach is described in Section 4. In Subsection 4.1, the dependencies representation is described, then Subsection 4.2 describes how to automatically determine which other transitions are required in a given TP that references a counter together with the algorithms that perform this task. A method to suggest the length of the generated TPs is then given in Subsection 4.3. The experiment is presented in Section 5 where Subsection 5.1 describes the experimental design and Subsection 5.2 reports the experimental results. Concluding remarks are in Section 6.

5.2 Problem Area

Testing from an EFSM can be based on coverage criteria such as state coverage, transition coverage and path coverage (Tahat et al., 2001). One approach to satisfying a criterion is to produce a set of paths through an EFSM that satisfies the criterion and then produce test cases (test data) to trigger the paths. However, a transition path (TP) can be infeasible as a result of transitions having guards (preconditions). For example, in a given TP a transition may set the value of variable x to 0 and a later transition can have a guard that requires $x > 0$. Such a conflict, as described in Chapter 3, can be statically determined and heavily penalised so that it can be avoided during the search. However, a TP can be infeasible due to a particular transition not being included a sufficient number of times. For example, if an EFSM has a counter variable c whose value is initially 0, a transition t_i that has a guard $c > 2$ cannot be exercised unless a transition t_j which updates the value of c has already occurred a sufficient number of times. Such variables are often called *counter variables* and are usually used to count how many times a transition is repeated. For example, the variable *attempts* in ATM

EFSM (Figure 5.2) plays the role of a counter where it is increased whenever transition t_2 is called (an incorrect *pin* was entered).

For an EFSM that contains a counter variable, a given TP may include a transition that has a guard over the counter variable and so there is a need to determine which other transitions are involved (those that affect the counter variable value) and how many times they must be called. The counter problem thus requires additional analysis. Unfortunately, this is a substantial mathematical problem (Chanson and Zhu, 1993) which is related to a complex dynamic behaviour that is difficult to determine (Derderian et al., 2010). Furthermore, counter variables are a significant problem for search based testing (McMinn, 2004, Harman, 2008, Kalaji et al., 2009c).

For example, (Harman, 2008) states that the problem is related to assigning a suitable value to the counter variable, however, such a value cannot be assigned directly but indirectly by calling a specific function and for a specific number of times. Thus, the problem also involves a determination of this function(s) and a decision about how many times it should be called.

Another problem associated with the counter is the need to reason about the adequate length of the test case (i.e. how many transitions in a TP) to be executed so that the counter can be assigned the required value (McMinn, 2004). Consider for example, a test case which comprises a sequence of calls to a set of functions. If the test case is short (in terms of the number of calls), then it may not allow a specific function (that updates the counter) to be called a sufficient number of times. Nevertheless, if the test case is long, it may simply complicate the generated test case by calling unnecessary functions.

As mentioned in Chapter 3, the problem of generating feasible transition paths (FTPs) is generally an undecidable problem (Chanson and Zhu, 1993, Dssouli et al., 1999, Hierons et al., 2009). Furthermore, developing good methods to derive FTPs from an EFSM is an open research problem (Duale and Uyar, 2004) and the existence of the counter behaviour is therefore an additional substantial obstacle.

The approach described in Chapter 3 (Kalaji et al., 2009a) proposes a TP fitness metric to generate FTPs that are likely to be feasible by using search.

Importantly, the TP fitness metric can be computed quickly and thus is suitable to be used in search. However, since the TP fitness metric does not consider the counter problem and so this requires additional analysis. This chapter proposes a novel approach based on control and data analysis to automatically determine whether in a given TP a particular transition's guard references a counter variable, which other transitions are involved (are required in this TP) and how many times they must be called. To this end, the approach presented in this chapter aims to form part of the solution to the following problem:

Given: a test adequacy criterion and an EFSM model that includes counter variables

Problem: generate a set of TPs that are feasible and satisfy the test criterion.

The primary contributions of this chapter are the following:

1. It proposes a method to bypass the counter problem by automatically determining whether a transition's guard references a counter, which other transitions are involved and how many times they have to be called
2. It proposes a method based on Dijkstra's algorithm that suggests a suitable length of TPs to be generated in the presence of the counter problem.
3. It shows that the proposed approach is effective in generating FTPs to satisfy the test criterion from EFSM models that suffer from the counter problem.
4. The chapter empirically validates the approach by using it with two EFSM case studies: an ATM model and the Inres initiator.

5.3 Case Studies

In this chapter, two EFSM case studies are used to validate the proposed approach. The first EFSM is the Inres initiator (Hogrefe, 1991) that is described in Chapter 3. This EFSM suffers from the counter problem through using the variable *counter* which is referenced by the guards of six transitions t_3, t_4, t_8, t_9, t_{10} and t_{11}. The second case study is an ATM system that represents an extension of the machine described in (Korel et al., 2002) which includes a counter variable *attempts* that is referenced by the guards of two transitions t_2 and t_3. The Inres

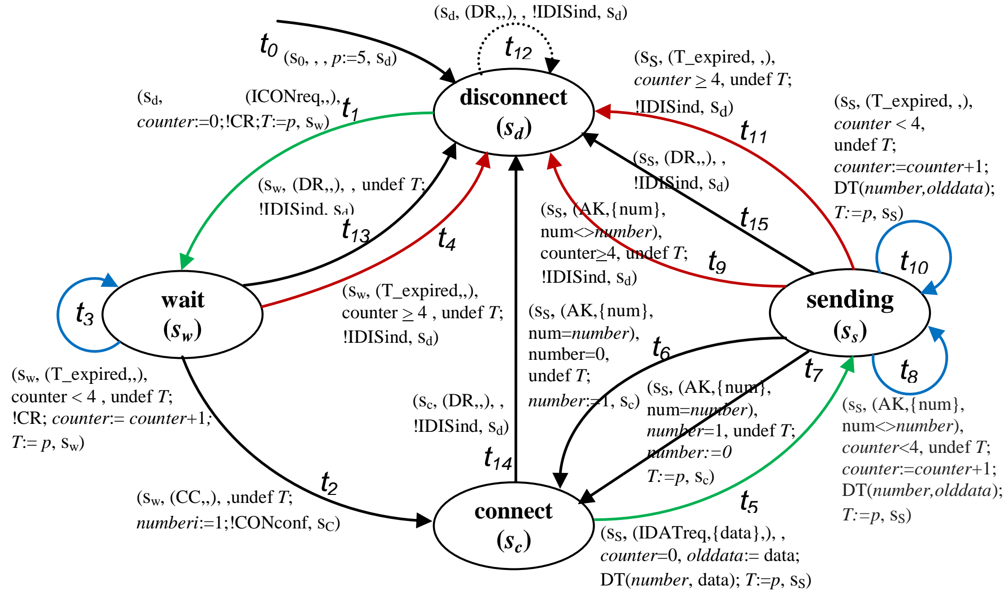


Figure 5.1: Inres Initiator EFSM. Initialisers, updaters and target transitions are coloured green, blue and red respectively. Transition t_{12} represents an 'escape' transition and is represented by a dashed arrow.

initiator and ATM EFSMs are shown in this chapter in Figure 5.1 and Figure 5.2 respectively.

5.4 The Proposed Approach

The TP fitness metric described in Chapter 3 estimates a given TP feasibility by analysing the dependencies among the TP's transitions. In order to do this, an EFSM's transitions are classified to two types: *affecting* and *affected-by*. A transition t_i is said to be affecting within a TP if it has an assignment operation that can affect the guard of a later transition (the affected-by). Based on this, any pair (t, t') which represents (affecting, affected-by) is assigned a penalty value depending on the type of the assignments and guards found in t and t' respectively. The assignment operation (op) is classified to three types $\{op^{vp}, op^{vv}, op^{vc}\}$ which denote that a variable v is assigned a value of an expression that depends on parameters, only context variables and a constant respectively. Also, a transition's guard is classified to five types $\{g^{pp}, g^{pc}, g^{vp}, g^{vv}, g^{vc}\}$ which denote a comparison

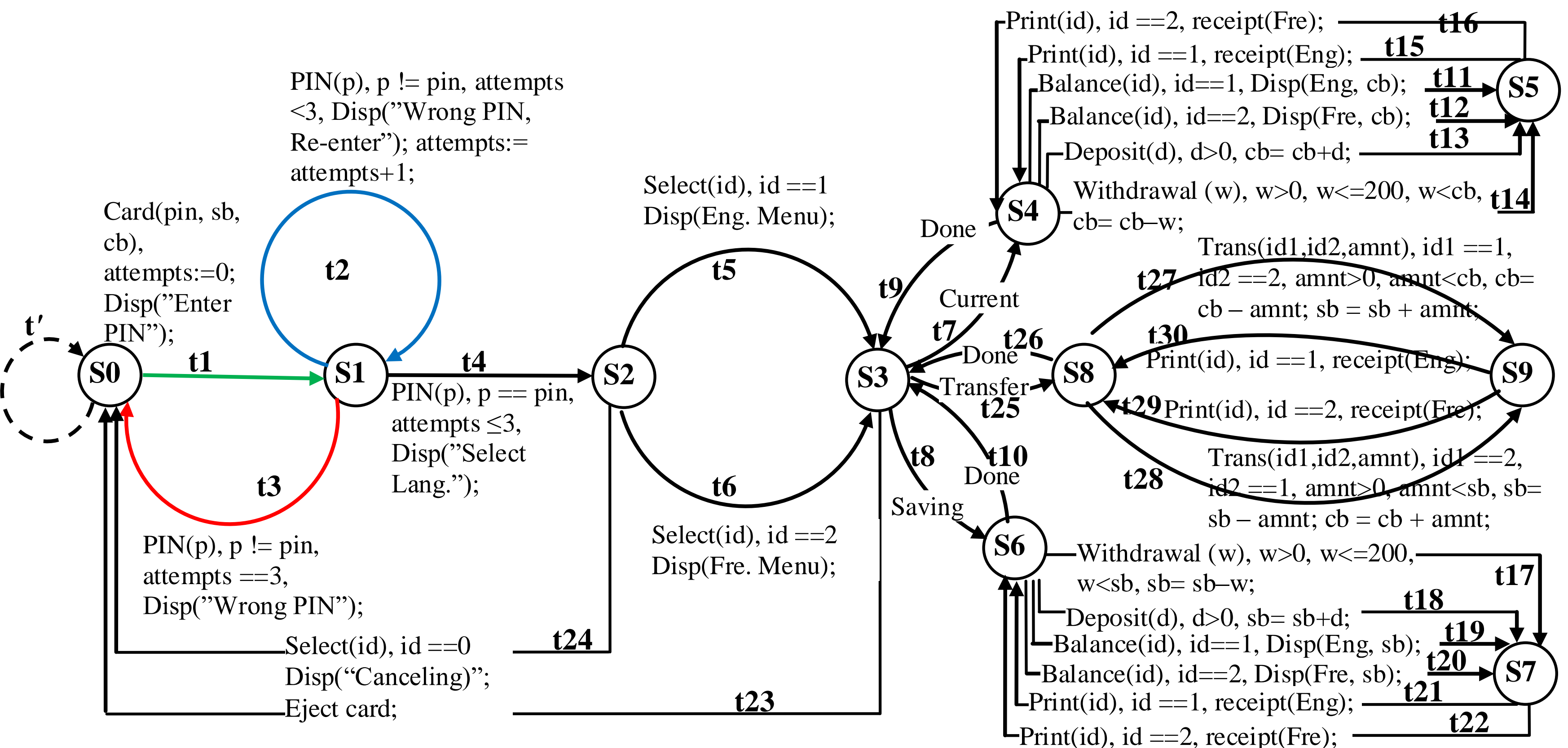


Figure 5.2: The EFSM Model of the ATM System. Initialisers, updater and target transitions are coloured green, blue and red respectively. Transition *t′* represents an ‘escape’ transition and is represented by a dashed arrow.

among only parameters, parameters and constants, variables and parameters, only variables, and variables and constants respectively.

The penalty value assigned to each pair (t, t') represents a numerical estimation of how difficult it is to satisfy the guard of t'. Problems can occur when a given TP includes a transition guard that references a counter variable and in this case the search may not receive the necessary guidance. Here, the problem is that in a given TP in order to execute a transition t' whose guard references a counter variable, the TP must first call another transition t (that updates the value of the counter variable) a certain number of times so that the guard of t' is satisfied.

For example, consider Inres initiator (Figure 5.1), in order to cover transition t_4, the *counter* should have a value of at least 4. Thus, t_1 should occur first (to initialise the variable *counter*), followed by t_3 (to update the value of the *counter*) four times. However, (t_3, t_3) forms a pair of (affecting, affected-by) because t_3 affects (updates) the value of the variable *counter* and also t_3 is affected-by (has a guard referencing the variable *counter*), thus each occurrence of t_3 followed by t_3 incurs a new penalty and therefore increasing the overall TP penalty. However, the search aims to minimise the TP fitness metric value and so the correct sequence of the required transitions is unlikely to be generated in this particular counter case. This description explains why the TP fitness metric, in certain cases, could not guide the GA search towards FTPs when the TP includes a transition whose guard references a counter variable.

To overcome this problem, the test criterion should include the required extra transitions together with the number of times they have to be called (Kalaji et al., 2010). Generally, this problem is a challenging mathematical task (Chanson and Zhu, 1993). However, in some cases it can be approached with an abstraction to generate acceptable solutions. This abstraction is related to the counter definition in an EFSM. In Chapter 3, assignment operations are classified to three types $\{op^{vp}, op^{vv}, op^{vc}\}$. In this chapter, the operation op^{vv} is further classified to four subtypes to cover the counter situations:

1. op^{v+c}: it increments the context variable v by a constant value
2. op^{v-c}: it decrements context variable v by a constant value

3. $op^{v \times c}$: it multiplies context variable *v* by a constant value

4. $op^{v/c}$: it divides context variable *v* by a none-zero constant value

Based on the classifications of counter situations, the following definitions can be provided:

Definition 5.4.1: A context variable *v* in an EFSM is a counter variable if there is a transition *t* with an assignment $op \in \{op^{v+c}, op^{v-c}, op^{v \times c}, op^{v/c}\}$ to *v*, and *v* is referenced in a guard of a transition *t'*.

Definition 5.4.2: A transition *t* is affecting a counter variable *v* if it assigns to *v* using an $op \in \{op^{vc}, op^{v+c}, op^{v-c}, op^{v \times c}, op^{v/c}\}$, however, *t* is affected-by a counter variable *v* if it has a guard that references *v*.

Definition 5.4.3: A transition *t* is an *initialiser* of a counter variable *v* if it assigns to *v* using an $op \in \{op^{vc}\}$,

Definition 5.4.4: A transition *t* is an *updater* of a counter variable *v* if it assigns to *v* using an $op \in \{op^{v+c}, op^{v-c}, op^{v \times c}, op^{v/c}\}$.

Definition 5.4.5: Given a counter variable *v*, a TP: $t_1, t_2, .., t_n$ and three transitions t_i, t_j and t_k from this TP where $i < j < k$, the triple (t_i, t_j, m) forms a sequence that can satisfy the guard of t_k if t_i is an initialiser of *v*, t_j is an updater of *v*, t_k is affected-by *v*, the path $t_i,..,t_k$ contains exactly *m* instances of transition t_j without other assignments to *v* and $m \geq 0$ is an integer that specifies the exact number of updater occurrences required so that the guard of t_k is satisfied.

For example, consider Inres initiator (Figure 5.1), the variable *counter* is a counter variable because there is a transition t_3 which assigns to this variable by using op^{v+c} and *counter* is referenced in the guard of transition t_4. Transition t_1 is an initialiser since it assigns to the *counter* by using op^{vc} while transition t_3 is an updater since it assigns to *counter* by using op^{v+c}.

Based on the above definitions, if a given TP is required to cover a transition t_k which is affected-by a counter variable, the problem can be approached by having a method to automatically determine all the possible triples of (initialiser, updater, updater times) that can satisfy the guard of t_k (Kalaji et al., 2010).

Guard	Representation
No guard	0
=	1
>	2
<	3
≥	4
≤	5
≠	6

Table 5.1: Guards representation as integers

Operation	Representation
No operation	0
$op^{v=c}$	1
op^{v+c}	2
op^{v-c}	3
$op^{v\times c}$	4
$op^{v/c}$	5

Table 5.2: Operations representation as integers

5.4.1 Dependencies Representation

For each counter variable v, its initialisers and updaters are determined. Also, for each transition that is affecting a counter, its operation type is recorded whereas for each affected-by a counter, its guard type is recorded.

Table 5.1 and Table 5.2 above show the integer representation of possible guard operators and possible assignment operations. Based on this representation, two matrices can be constructed: affecting a counter *aff* and affected by a counter *aff-by*. Each row in these matrices represents one transition and each column represents one counter variable. In an *aff* matrix, each cell comprises a tuple with two fields: one to record the operation code (see Table 5.2), and the other records the value of the constant that appears in this operation. Similarly, each cell of the *aff-by* matrix is a tuple with two fields: one to represent the guard code (see Table 5.1), and the other records the constant value referenced by this guard and thus it considers only the cases where a counter variable is compared to a constant.

Consider for example the Inres initiator shown in Figure 5.1. This EFSM has one counter variable: *counter* (which will be referred to by v_1). Figure 5.3 shows a part of the two matrices: *aff* and *aff-by* for this counter variable. From the *aff* matrix, t_1 and t_5 are initialisers as they assign to v_1 the constant value 0 (operation code =1: $op^{v=c}$) (see Table 5.2). Similarly t_3, t_8 and t_{10} are updaters (operation code = 2: op^{v+c}) with an assignment of constant value 1. From the *aff-by* matrix, t_3, t_4, t_8 and t_{10} are affected-by a counter. For example, t_4 has a guard

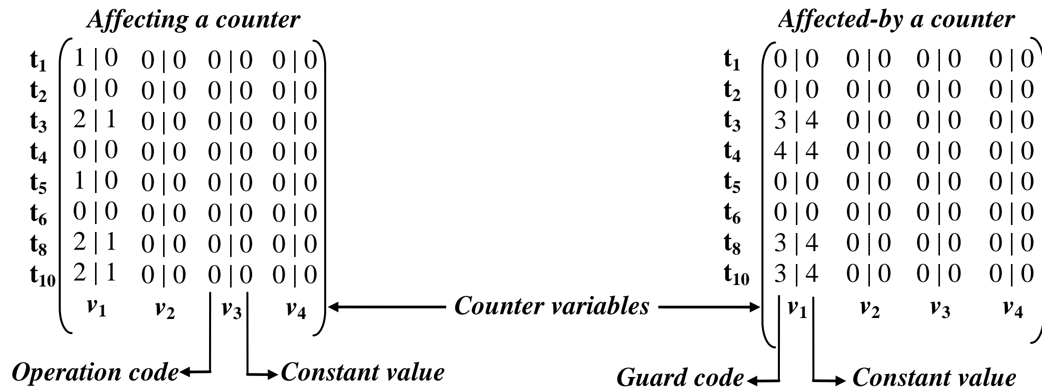


Figure 5.3: Example of affecting and affected-by (a counter) matrices for Inres initiator EFSM

code = 4 (≥) (see Table 5.1) and a constant 4. The existence of extra columns in Figure 5.3 is just for illustration purpose as it is possible to have more than one counter variable in a given EFSM.

5.4.2 Finding the Required Sequence of Transitions

This subsection defines the approach that determines a sequence(s) of transitions that is required to satisfy a given target guard over a counter variable. The proposed approach is applied through an algorithm which consists of four routines: *FindSequence*, *Validate, GuardCheck* and *IsTripleExisted*.

The first procedure, *FindSequence*, is the main one and it performs the following tasks:

1. Determining whether a target transition (to be covered), t_j, has guard(s) referencing a counter variable(s).
2. Scanning for all possible pairs of (initialiser, updater) that initially have the potential to satisfy the given target guard(s) over a counter variable(s)

The second function, *Validate*, is called from the main procedure to validate each given triple (initialiser, updater, updater_times) by:

1. Checking whether calling only the initialiser can satisfy the target transition guard and in this case updater_times will be set to 0
2. Setting a given triple as either invalid or valid and thus setting the number of times the updater must be occurred (updater_times > 0)

Procedure Find Sequences of Transitions

```
1.  input: tj, affecting matrix aff[], affected-by matrix aff-by[]
2.  output: a list LT of triples (initialiser, updater, updater_times)
3.  goal: determine a sequence of transitions to satisfy tj guard
4.  initialise variable: empty(LT); integer updater_times := 0;
5.  begin
6.     for vi := 1 to number_of_counter_variables
7.        if guard of tj references a counter_var_vi  then
8.        begin
9.           build a list LI of transitions that initialise vi
10.          build a list LU of transitions that update vi
11.          for every initialiser from LI
12.             for every updater from LU
13.                if validate (initialiser, updater, updater_times) then
14.                insert in LT the triple (initialiser, updater, updater_times);
15.       end;
16. end;
```

Figure 5.4: The algorithm which finds sequence of transitions

Function Guard Check

```
a1.  input: guard_code, guard_const, counter_var
a2.  output: boolean result
a3.  goal: check whether a guard is satisfied
a4.  initialise variable:  result := false;
a5.  begin
a6.     case guardCode of
a7.        0 : result := true;                                          // no guard
a8.        1 : if counter_var = = guard_const then result := true;     // equality
a9.        2 : if counter_var > guard_const then result := true;       // greater than
a10.       3 : if counter_var < guard_const then result := true;       // less than
a11.       4 : if counter_var ≥ guard_const then result := true;       // equal or greater than
a12.       5 : if counter_var ≤ guard_const then result := true;       // equal or greater than
a13.       6 : if counter_var ≠ guard_const then result := true;       // inequality
a14.    end;
a15. end;
```

Figure 5.5: The Guard_Check routine

The third function, *GuardCheck*, is used by function *Validate* to check if a given guard is satisfied. The *GuardCheck* is called to decide whether:

1. The target guard over the counter is satisfied and thus the triple is valid
2. The guard of an updater is still satisfied so more calls can be made to the updater.

The last function *IsTripleExisted* is called during the search phase to check whether one of the required triple exists within a given TP.

Figures 5.4, 5.5, 5.6 and 5.7 show a high level description of the algorithm that find a sequence of transitions to cover a particular transition t_j that has a guard referencing a counter variable. Given a transition to cover t_j in an EFSM, the algorithm finds a sequence or a set of sequences by first using *FindSequence* (Figure 5.4) to check if t_j has a guard which reference a counter variable(s) (Line. 6 & 7). This is accomplished by checking if t_j belongs to the matrix that contains all the transitions that are affected by counter variables. For each counter variable v_i referenced by t_j guard, the algorithm builds two lists: LI that contains all the initialisers and LU that contains all the updaters of v_i (Line 9 & 10). An initialiser of v_i is determined from the matrix that lists all the transitions that are affecting counter variables (transitions that have operation code =1). Similarly, updaters are detected from the same matrix with operation code in [2..5] (see Table 5.2). For each initialiser, every updater is considered (Line 11 & 12) to form a candidate triple of (initialiser, updater, updater times) where *updater_times* is an integer value that represents how many times an updater should be called after an initialiser. After a candidate triple is formed (Line 13), this needs to be checked for being valid and to set the value of *updater_times*. This is accomplished through the function *Validate* (Figure 5.6)

The function *Validate* checks whether a given triple is valid and so how many times the updater within this triple must be called. Firstly, the triple is set to be initially invalid (Line b4). Then the initialiser is called to initialise the counter variable v_i and so the value of v_i is now equal to the constant value of the initialiser (Line b5). Since calling only the initialiser may satisfy the guard of t_j, the status of t_j guard is checked by calling the function *GuradCheck* (Line b7). If t_j guard is achieved, then the *updater_times* is set to be 0 (there is no need to call the

Function Validate

```
b1.  input: tj, vi, aff[], affby[], a triple (initialiser, updater, updater_times)
b2.  output: boolean result; integer updater_times;
b3.  goal: determine how many times an updater should be called in order to satisfy the
     guard of tj over the counter variable vi
b4.  initialise variable: result := false;     // currently set the triple to be invalid
b5.  counter_var_vi := initialiser_const;      // apply the initialiser operation
b6.  begin
b7.    if guardCheck(tj_guard_code, counter_var_vi, tj_guard_const) then
b8.    begin
b9.      updater_times := 0;    // calling the initialiser alone can satisfy tj_guard
b10.     result := true;        // and so the triple is valid; exit the routine
b11.     exit;
b12.   end;
b13.   repeat                   // the updater must be called. Determine the updater times
b14.     first_branch_distance_of_tj_guard := abs( counter_var_vi − tj_guard_const);
b15.     if guardCheck(updater_guard_code, counter_var_vi, updater_guard_const) then
b16.       begin                // updater's guard is true, so call the updater
b17.         case updater_operation_code of
b18.           2 : counter_var_vi := counter_var_vi + updater_operation_const;
b19.           3 : counter_var_vi := counter_var_vi − updater_operation_const;
b20.           4 : counter_var_vi := counter_var_vi * updater_operation_const;
b21.           5 : counter_var_vi := counter_var_vi / updater_operation_const;
b22.         end;
b23.         updater_times := updater_times + 1;
b24.       end
b25.     else                   // cannot call the updater and thus the triple is invalid
b26.       exit;                // exit and return false
b27.     if guardCheck(tj_guard_code, counter_var_vi, tj_guard_const) then
b28.       begin                // the guard over the counter is satisfied,
b29.         result := true;    // triple is valid and no more calls to the updater is required
b30.         exit;              // return true and exit
b31.       end
b32.     else next_branch_distance_of_tj_guard := abs(counter_var_vi − tj_guard_const);
b33.   until (next_branch_distance_of_tj_guard ≥ first_branch_distance_of_tj_guard);
b34. end. // the value that can satisfy tj guard has been surpassed and the triple is invalid.
          // The loop is broken and no more calls to the updater are performed
```

Figure 5.6: The algorithm which validates a given triple

updater), the triple is set to be valid and the function is exited (Lines b9, b10 & b11). If t_j guard is not yet satisfied, a loop is entered to try to detect how many times an updater should be called to satisfy t_j guard (Line b13).

Since an updater can have a guard that controls its assignment operation (the assumption here is that an updater's guard only references the same counter variable), there is a need to check that this guard is always satisfied (through calling the function *GuardCheck*) every time an updater is called (Line b15). If the current value of v_i does not breach the updater guard, the updater can be called one more time. Depending on the updater operation code (see Table 5.2) an updating operation is executed over v_i by the updater (either Line b18, b19, b20 or b21). Since the updater is called, the *updater_times* is increased (Line b23). If the updater guard is breached (as a result of calling the initialiser or the updater itself) (Line b25), then the current triple is invalid. If t_j guard is satisfied due to calling the updater, then the triple is set to be valid, the number of times the updater must be called is recorded and the algorithm exits (Line b27 & b29). If t_j guard is not yet satisfied, there is a need to check whether more calls should be made to the updater.

Since a loop is utilised to determine how many times an updater must be called, it is necessary to avoid looping infinitely. Utilising a loop to perform the necessary calls has been proposed by (Harman, 2008), however, the loop was allowed to iterate *n* times where *n* is an arbitrary large value. The proposed method here manipulates this problem by firstly, the absolute branch distance of t_j guard is calculated after the loop is entered (Line b14) and then calculated again after the updater is called (Line b32). Finally, the first branch distance is compared to the next branch distance (Line b33). If the next branch distance is greater than or equal to the first branch distance, then calling the updater has caused the value of v_i to surpass the desired point and thus the guard of t_j is breached. Consequently, no more calls to the updater should be performed, the triple is invalid and the loop is broken. However, if the first branch distance of t_j guard is less than the next branch distance, then more calls to the updater are required and so forth.

Function Is Triple Existed

```
c1.  input: a TP, target transition position in the TP (t_pos), counter variable (v_c),
     triple (initialiser, updater, updater_times), aff[], aff-by[].
c2.  output: boolean result
c3.  goal: check whether the given triple exists in the given TP
c4.  initialise variables:  result:= false; integer k := -1; integer count :=0;
c5.  begin
c6.    for i := t_pos -1 downto 1               // scan the TP to detect the last initialiser
c7.      if t[i] is the initialiser then        // if the current transition is the initialiser
c8.        begin
c9.          k := i;                            // store the initialiser position in the TP
c10.         break;                             // and break the loop since the very last
c11.       end;                                 // initialiser is the important one
c12.   if k < 0 then                            // the initialiser is not found. The triple is
c13.     exit                                   // invalid. Exit and return false
c14.    if updater_times > 0 then               // the updater should occur at least once
c15.     begin
c16.       for i := k+1 to t_pos -1             // scan for the updater in the path between
c17.         if t[i] is the updater then        // the initialiser and the target transition
c18.           count := count + 1;              // count each updater occurrence
c19.         else if t[i] is in aff[v_c] then   // if there are other assignments, path is not
c20.           exit;                            // a definition clear. Exit and return false
c21.        if count := updater_times then      // if the updater occurs the required
c22.         begin                              // number of times, the triple is valid
c23.           result := true;                  // return true and exit
c24.           exit;
c25.         end
c26.        else exit;                          // updater did not occur as required. Exit
c27.     end                                    // and return false
c28.   else                                     // no need for the updater and so scan to
c29.     for i := k+1 to t_pos -1               // determine whether the path between the
c30.       if t[i] in aff[v_c] then             // initialiser and the target transition is
c31.         exit                               // definition clear for the counter variable
c32.   result := true;                          // if not, exit and return false, otherwise
c33. end                                        // return true.
```

Figure 5.7: The routine which verifies a triple existence in a TP

The algorithm that finds the required sequences has a polynomial running time $T(n) = O(n^2 \times v_c \times k)$ where n is the number of transitions in an EFSM, v_c is the number of the counter variables and k is the number of iterations performed by *validate* routine. The *IsTripleExist* routine used during a GA search has a linear running time $T(n) = O(n)$ where n is the number of transitions in a TP.

Naturally, there can be more than one possible triple (initialiser, updater, updater times) that may satisfy t_j guard. For such a case, a set of triples are linked by OR. If the guard of t_j references more than one counter variables, then triples that belong to different counter variables which are referenced by t_j guard are linked by AND.

For example, let's consider a TP that includes the transition t_4 in the Inres initiator (Figure 5.1). Since t_4 has a guard that references a counter variable, the algorithm first detects all the potential triples that may lead to the guard of t_4 being satisfied. From the two matrices *aff* and *aff-by*, there are two initialisers t_1 and t_5, also, there are three updaters t_3, t_8 and t_{10}. Thus there are six triples that can be initially considered. When *Validate* is called, these triples are set to be valid and so the final criterion (to generate a TP to cover t_4) will be: (t_4) *AND* ((t_1, t_3, 4) *OR* (t_1, t_8, 4) *OR* (t_1, t_{10}, 4) *OR* (t_5, t_3, 4) *OR* (t_5, t_8, 4) *OR* (t_5, t_{10}, 4)).

Once the final set of triples is ready, it is fed to the TP fitness metric to check whether the transition to cover (t_j) exists, furthermore, the triple (initialiser, updater, updater times) is present in this TP and the TP incurs the least possible penalty. This is accomplished by calling the function *IsTripleExisted* (Figure 5.7) which checks that the initialiser comes first (Lines c6, c7, c8 & c9), then the updater occurs certain number of times (Lines c14, c16, c17, & c18) and finally the target t_j. It also checks that the path from the initialiser to the first occurrence of the updater is definition clear for the related counter variable (Line c19). The occurrences of the updater need not be in successive transitions in the path and it is sufficient that the sub-path between each pair of occurrences of the updater is definition clear for the counter variable. Finally, the path from the last occurrence of the updater to the target t_j must be also definition clear for the same counter variable (line c19). If a given generated TP does not meet the criterion, then a large penalty value (INF) is assigned i.e. an infeasible TP.

5.4.3 Determining the Length of TPs

Short TPs may not allow the target transition (the transition whose guard references a counter variable) to be reached. Furthermore, if the target transition can be reached by a certain TP length, it is still possible that this length does not allow the updater transition to occur a sufficient number of times. Moreover, if the length is chosen to be longer than necessary, the generated TP may incur unnecessary complexity by including transitions that are not required.

A possible way to overcome this problem is to select a long TP length which can be considered adequate. The selection of such a length can partially be based on the previously derived triples (initialiser, updater, updater times). The maximum value of updater occurrences among all of the derived triples can be considered in computing the length. In addition to this value, the maximum length to reach any transition in the considered EFSM by starting from the initial state should also be taken into account.

In this way, any generated TP is likely to have extra transitions since the worst case scenario is considered. However, extra transitions in a given TP may not be desirable since it may lead to a given TP incurring a larger penalty and thus an unnecessary complexity. To avoid this problem, any EFSM can be extended with a cyclic transition t' which starts and ends at the first state. Such a transition, t', does not have any operations or guards and so it is an 'escape' transition (an 'escape' transition is a transition that has no guards and operations). The t' is placed in the first state to guarantee that any generated TP can make use of it. The idea of using such a transition is already discussed on the experimental results reported in Chapter 3. TPs that were generated by using the TP fitness metric from EFSM machines that contain 'escape' transitions (such as Inres initiator and ATM EFSMs) did not incur larger penalty values when they were longer. However, the existence of t' in any TP, means that such a TP cannot directly be applied to the considered implementation under test (IUT). The remedy of this problem is simple where any generated TP that includes instances of t' can easily be filtered by just removing t' from the transition sequence. In this way, the resultant TP includes only the actual transitions. For Inres initiator EFSM (Figure 5.1), this

already includes the transition t_{12} which represents t' (represented by a dashed arrow). However, for ATM EFSM, this was extended with t' that is represented by a dashed arrow in Figure 5.2.

Determining the shortest length to reach a given transition by starting from the first state can be performed by using Dijkstra's algorithm (Dijkstra, 1959). Dijkstra's algorithm can generally be used to determine the shortest path between two nodes in a given graph. The algorithm input is a graph (V, E) where V is a finite set of vertex (node) and E is a finite set of edges that connects the graph vertices. Also, the algorithm requires a finite set of cost, ω, which determines the cost associated with each edge of E. Each edge between two vertices represents the distance and thus the cost which the algorithm uses when searching the graph to produce the shortest (lowest cost) path between two vertices.

For the TP length, an EFSM transition diagram is the graph to be searched where states represent vertices and transitions represent edges. Since the aim is to find the shortest path, all transitions are given the same cost (i.e. $\omega_1 = \omega_2 =... \omega_n= 1$, where n is the total number of transitions). Furthermore, if two or more transitions are connecting the same two states, one transition is kept and the others are removed.

By specifying the start and end states for which a shortest path is required, the algorithm begins searching from the start state. For each state, the search includes two labels: a permanent label and a temporary label. A label at each state is used to store the lowest cost (the shortest distance) between this state and the start state. At any given time, any state can have only one label and therefore one cost. If the label is permanent, then it specifies the final value of cost, however, a temporary label represents an initial cost or a cost which is not yet a final one. Initially, all states but the start state are associated with temporary labels with an infinite cost value, however, the first state is associated with a permanent label of a cost value equal to zero and marked as a current state.

The next step is to search all the neighbour states of the current state by following the transitions that initiate from the current state. For each neighbour state, if the sum of the current state's cost and the cost of the transition which reaches to the considered neighbour state is less than the cost recorded in the

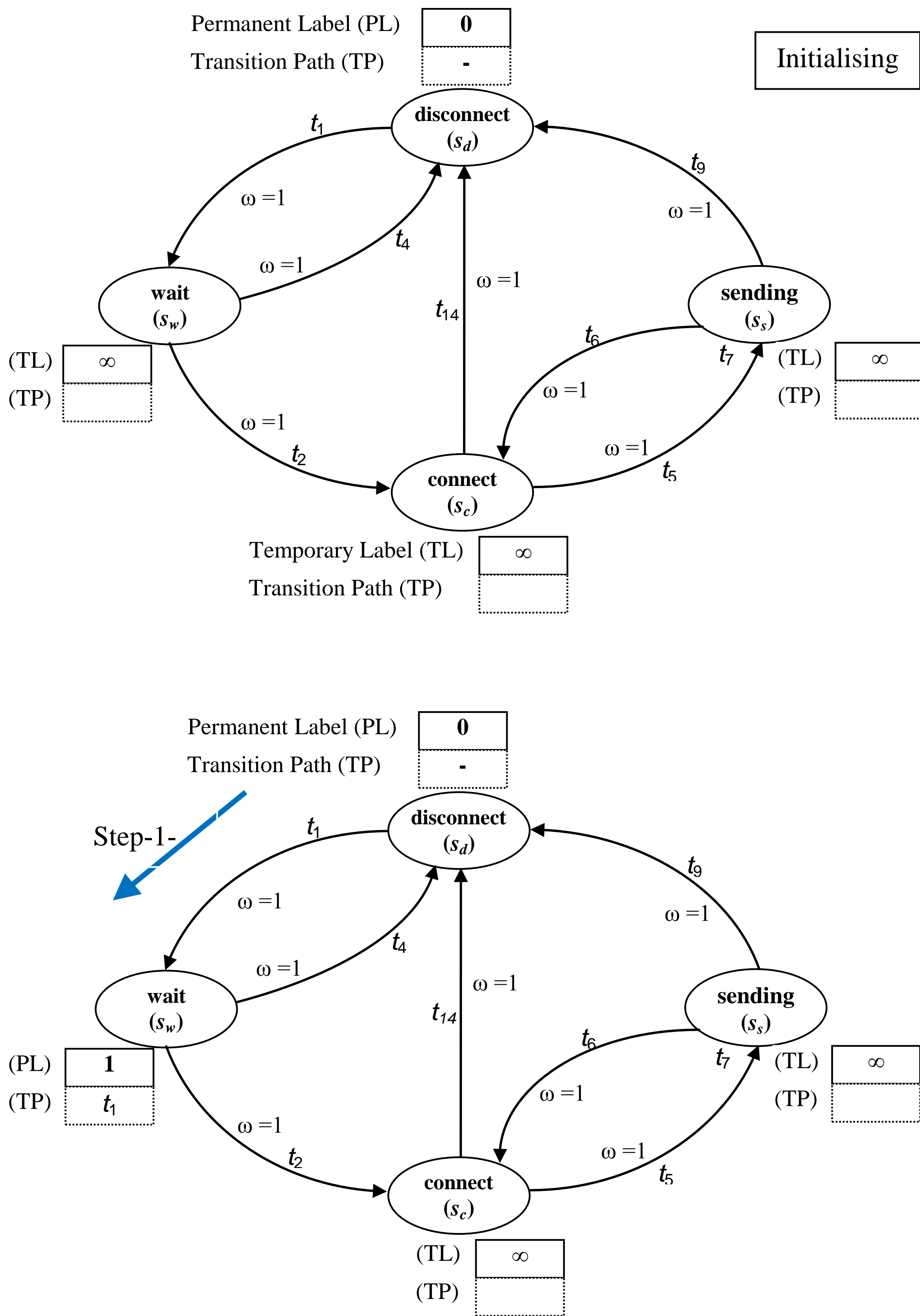
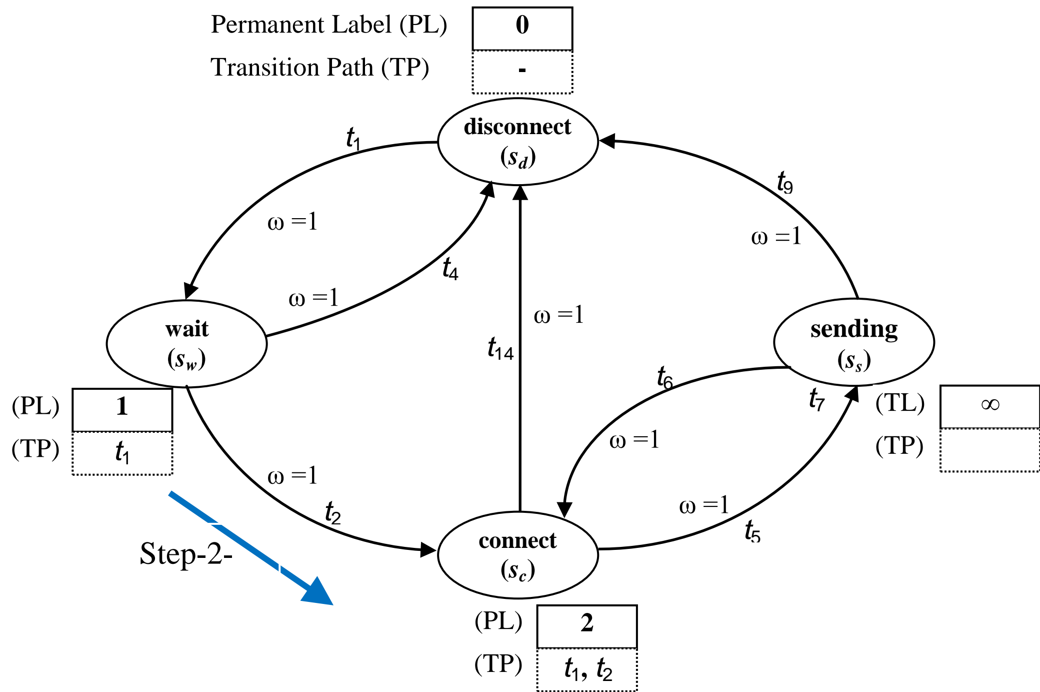


Figure 5.8: The initial and first steps of Dijkstra's algorithm between states S_d & S_s in Inres Initiator EFSM



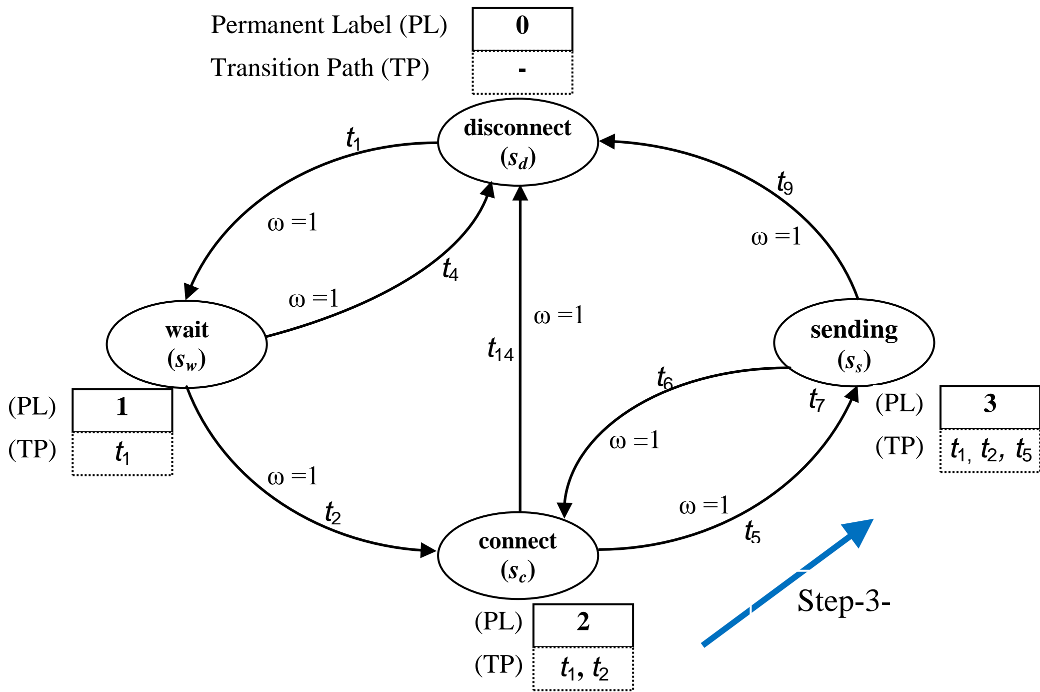


Figure 5.9: The second and last steps of Dijkstra's algorithm between states S_d & S_s in Inres Initiator EFSM

temporary label of the considered neighbour state, the cost recorded in the temporary label is updated to be equal to the sum of the values. Once all the neighbour states are explored, the current state is marked as visited so that it will not be visited again. Furthermore, the neighbour state which is associated with the lowest cost is set as the current state and its label becomes permanent. Now, the search starts to explore the neighbours of the new current state and so on.

For example, consider the Inres initiator EFSM for which a shortest path is required between the first state (s_d) and the last state (s_s). Figure 5.8 and Figure 5.9 show how the algorithm is applied step by step to the considered case.

For any transition whose guard references a counter variable, the proposed approach produces a set of triples that initially has the potential to satisfy the considered guard. However, to determine an adequate TP length, there are five factors to be considered when a TP references a counter variable:

1. The initial cost: The initialiser must occur once and so should the target transition (to be covered).
2. The updater cost: The cost associated with calling the updater and its number of occurrence. If the updater is cyclic, the cost of calling an updater is the number of updater times. However, if the updater starts and ends at different states, then the cost associated with calling an updater is calculated as (1 + Dijkstra (updater's end state, updater's start state)) × updater occurrences.
3. The starting cost: The cost to reach the initialiser from an EFSM's initial state as Dijkstra (initial state, initialiser's start state)
4. The middle cost: The cost to reach the updater (from the initialiser) as Dijkstra (initialiser's end state, updater's start state)
5. The end cost: The cost to reach the target transition (from the updater) as Dijkstra (updater's end state, target transition's start state)

The final length then is the sum of these considered costs. Naturally, for a given EFSM, there can be more than one TP which references a counter variable and so different lengths are required. Furthermore, the above factors may be insufficient to suggest a length to reach any transition in an EFSM. Thus, the final TP length can be calculated as shown in Equation 5.1.

EFSM	Target	Triple	Cost					
			Initial	Updater	Starting	Middle	End	Total
Inres	t_4	(t_1, t_3, 4)	2	4	0	0	0	6
		(t_1, t_8, 4)	2	4	0	2	0	8
		(t_1, t_{10}, 4)	2	4	0	2	0	8
		(t_5, t_3, 4)	2	4	2	2	0	10
		(t_5, t_8, 4)	2	4	2	0	0	8
		(t_5, t_{10}, 4)	2	4	2	0	0	8
	t_9 OR t_{11}	(t_1, t_3, 4)	2	4	0	0	2	8
		(t_1, t_8, 4)	2	4	0	2	0	8
		(t_1, t_{10}, 4)	2	4	0	2	0	8
		(t_5, t_3, 4)	2	4	2	2	2	12
		(t_5, t_8, 4)	2	4	2	0	0	8
		(t_5, t_{10}, 4)	2	4	2	0	0	8
ATM	t_3	(t_1, t_2, 3)	2	3	0	0	0	5

Table 5.3: **Calculating the TP length for the Inres initiator and the ATM EFSMs.**

Length* = *Max (*lengths of counter TPs*, *lengths to reach any transition*) (5.1)

The first part of Equation 5.1 can be calculated by using the considered five factors when a TP references a counter variable. The second part of this equation is calculated by using Dijkstra's algorithm between the initial state and each other state.

Table 5.3 shows the length calculation for the TPs that reference counter variables in Inres initiator and ATM EFSMs by using the considered five factors. Then, by using Equation 5.1, the suggested TP lengths for Inres initiator and ATM EFSMs are: $Length_{Inres}$ = Max (12, 4) = 12; $Length_{ATM}$ = Max (5, 6) = 6.

5.5 Experiment

This section describes the experiment design and reports the experimental results achieved from applying the proposed approach to two EFSM case studies: Inres initiator and ATM EFSMs that both suffer from the counter problem.

5.5.1 Experimental Design

In designing the experiment, the aim was to evaluate the effectiveness of the proposed approach by guiding a GA search towards a set of FTPs that satisfies the transition coverage test criterion and also bypasses the counter problem that was observed in Chapter 3.

Since the considered problem is the counter problem, it is important to consider the length of the generated TPs. To achieve this, the affecting and affected-by counter matrices were derived for each considered EFSM. Then for each transition that is affected-by a counter variable, the proposed approach generated candidate triples that were valid and potentially satisfied the considered transition's guard that is affected-by a counter variable. Dijkstra's algorithm was then applied and a set of proposed TP length was derived.

As described in Subsection 5.4.3, for Inres initiator EFSM the TP length is 12 transitions and for ATM EFSM the TP length is 6 transitions (see Table 5.3). Since Inres initiator originally had an 'escape' transition (t_{12} in Figure 5.1), other longer TP lengths can also be used. Thus, three TP lengths were considered when generating the Inres initiator subject TPs. These lengths are 13, 14 and 15 transitions where each specific length was used to generate a set of subject TPs that satisfy transition coverage test criterion. Similarly, ATM EFSM was instrumented with an 'escape' transition which starts and ends at the first state (t' in Figure 5.2) and therefore longer TP length can also be used. Thus, three TP lengths as 6, 12 and 15 transitions were used. For each specific length, a set of subject TPs was generated that satisfies transition coverage test criterion.

Also, the experiment aimed to understand how the proposed approach enhances the previously proposed TP fitness metric approach described in Chapter 3. Thus, when generating subject TPs, two approaches were used to guide the GA search. The first approach is the proposed approach which consists of the previously defined TP fitness metric and the technique to bypass the counter problem. The second approach is merely the previous approach that depends only on the TP fitness metric guidance (described in Chapter 3). This allows comparing the performance of the proposed approach with the previous one and

therefore understanding whether the proposed technique, presented in this chapter, enhances the TP fitness metric guidance to overcome the counter problem.

A GA search that implemented the previously defined TP fitness metric together with the proposed approach was applied to the two EFSM case studies to generate three sets of subject TPs from each EFSM where each set provides transition coverage test suite for the considered machine. During TP generation with the proposed technique, each TP was first checked for the existence of a particular transition that this TP is intended to cover. Then, if this TP included a transition's guard that references a counter variable, the test criterion (transition coverage) for this TP was modified to require extra transitions (triples) and the TP was rechecked against these additional requirements by using the *IsTripleExisted* routine (given in Figure 5.7). Any TP that violated these constraints was considered invalid and given a large fitness value (INF = 1×10^4). For the purpose of comparison, three similar alternative sets of subject TPs were generated from each EFSM by using only the guidance of the previously defined TP fitness metric. For ease of reference, the proposed approach will be denoted by (GA-1) whereas the previously defined TP fitness metric approach will be denoted by (GA-2).

Once the sets of subject TPs were generated, these were checked for being feasible by using the test cases generation technique described in Chapter 4. If a given TP was not successfully trigged, this was closely examined to determine whether or not it was an FTP.

The GA (FTPs and path test case generation) was implemented using the publicly available Genetic and Evolutionary Algorithm Toolbox (GEATbx) (Pohlheim, 1994-2010). A detailed description of each of the GEATbx parameters used is provided at the tool's website (Pohlheim, 1994-2010) and the values used in the experiment are record here for the purpose of experiment replication.

An integer valued encoding was used. The population size was 100. The selection method was linear-ranking with a selective pressure set to 1.8. Discrete recombination was used whereas mutate integer mutation was applied. For FTP generation, individuals consisted of a number of variables depending on the considered TP length. For Inres initiator, individuals consisted of 13, 14 or 15

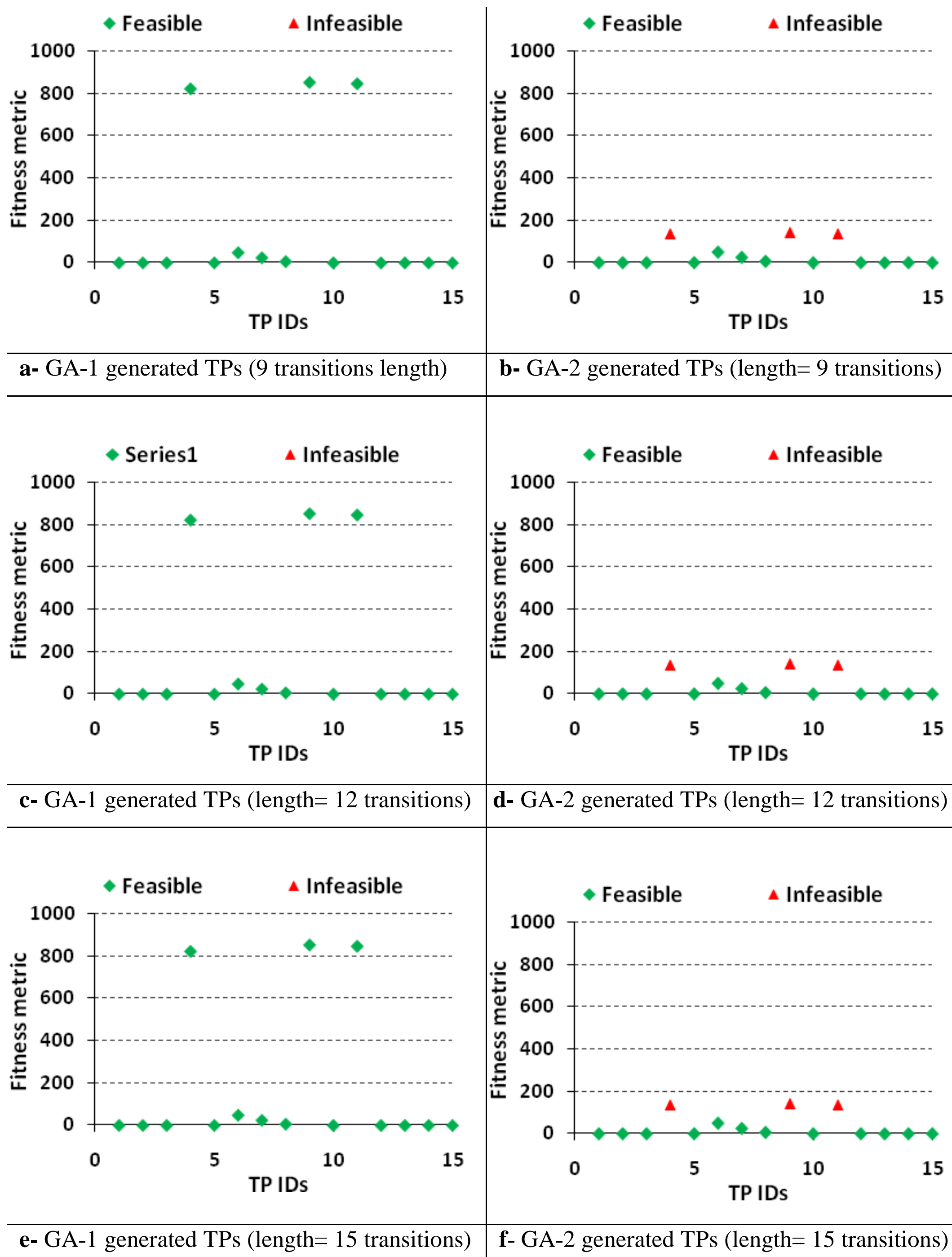
variables. For ATM, individuals consisted of 6, 12 or 15 variables. For path test case generation, an individual consisted of 20 variables that represent the number of inputs to be applied to a given FTP in order to be triggered. When a given TP required less than 20 inputs, extra inputs were ignored. The range of values allowed for each individual's variable was [0..1000] for path test case generation, [1..28] for Inres initiator FTPs generation and [1..60] for the ATM FTPs generation. An FTP generation search and a path test case generation search were given 1000 generations before being terminated. Finally, for path test case generation, the search was repeated 10 times for each subject TP.

5.5.2 Experimental Results

For each EFSM, there are two parts of the results achieved from the experiment. The first part reports three sets of subject TPs that were generated by the GA search (GA-1) which implemented the proposed approach described in this chapter. The second part reports the three alternative sets of subject TPs that were generated by the GA search (GA-2) which implemented the previously defined TP fitness metric (described in Chapter 3).

5.5.2.1 Results of the Inres Initiator EFSM

The GA search that implemented the proposed approach (GA-1) was applied to the Inres initiator EFSM to generate three sets of subject TPs where each set comprises 15 TPs that provide, if all subject TPs are FTPs, a transition coverage test suite for the Inres initiator EFSM. The first set contained subject TPs with length of 13 transitions whereas the second and third sets contained subject TPs with length of 14 and 15 transitions respectively. For the purpose of comparison, similar alternative sets of subject TPs were generated by a GA search that depended only on the guidance of the previously defined TP fitness metric (GA-2).



a- GA-1 generated TPs (9 transitions length)

b- GA-2 generated TPs (length= 9 transitions)

c- GA-1 generated TPs (length= 12 transitions)

d- GA-2 generated TPs (length= 12 transitions)

e- GA-1 generated TPs (length= 15 transitions)

f- GA-2 generated TPs (length= 15 transitions)

Figure 5.10: Inres EFSM TPs. The sets *a* & *b* have a TP length = 13, sets *c* & *d* have a TP length = 14 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are generated by using the proposed approach (GA-1). Sets *b*, *d* & *f* are the alternative sets that are generated by using the previously defined TP fitness metric approach (GA-2).

Figure 5.10 shows all the generated sets of subject TPs for Inres initiator EFSM by using the two approaches. The left column shows the sets *a*, *c* and *e* that were generated by the proposed approach (GA-1) whereas the right column shows the sets *b*, *d*, and *f* that were generated by the approach that uses the previous TP fitness metric (GA-2). Each generated TP, in any set, was assigned a unique ID that refers to the transition's number that this TP covers. Each set is plotted in terms of the ID of the generated TP against its best achieved TP fitness metric value.

For all the considered TP lengths (13, 14 and 15 transitions), the proposed approach generated subject TPs that were all FTPs (see Figures 5.10a, 5.10c & 5.10e). Furthermore, for each transition, all the derived TPs with corresponding ID have similar best TP fitness metric values. The TP fitness metric values associated with the TPs that did not suffer from a counter problem were generally low. However, the TPs that include transitions with guards that reference a counter variable were associated with greater TP fitness metric values. This applies to TPs with IDs 4, 9 and 11 where these TPs are associated with TP fitness metric values 820, 840 and 855 respectively. Nevertheless, these TPs were all feasible.

By considering the alternative sets, shown in Figures 5.10b, 5.10d & 5.10f, these were also similar to each other in terms of the TPs' IDs and their best achieved fitness metric values. However, there are three TPs in each set that are infeasible but associated with lower fitness metric values than the corresponding TPs produced using GA-1. This applies to TPs with IDs 4, 9 and 11 where these TPs are associated with TP fitness metric values 136, 142 and 136 respectively. A close inspection of these TPs showed that each TP includes a transition that references a counter variable where the required *updater* transition occurred insufficiently and consequently the TP is infeasible. This performance that was exhibited by GA-2 approach was also previously observed in the experimental results reported in Chapter 3.

By excluding the TPs that reference counter variables, all the generated sets of subject TPs (by either GA-1 or GA-2) were similar in terms of the TP's IDs and their best achieved fitness metric values (see Figure 5.10). This shows

Method	TP Length	Feasible	Infeasible	Avg. Fitness ≈
GA-1	9	15	0	173
GA-2		12	3	33
GA-1	12	15	0	173
GA-2		12	3	33
GA-1	15	15	0	173
GA-2		12	3	33
Total GA-1	9, 12, 15	45	0	173
Total GA-2	9, 12, 15	36	9	33

Table 5.4: **The Inres initiator EFSM generated TPs by the proposed approach (GA-1) and by the previously defined TP fitness metric approach (GA-2).**

that when the counter problem is absent, the proposed approach relies on the TP fitness metric guidance to evolve TPs that are associated with lower TP fitness metric values and therefore are likely to be feasible. Nevertheless, for TPs that reference counter variables, the performances of the two approaches were different.

Table 5.4 shows the summary of the results derived from Inres initiator EFSM. The average TP fitness metric value for each set of subject TPs that was generated by the proposed approach was approximately 173 and this is greater than that achieved by using the previously defined TP fitness metric approach where the average TP fitness metric was approximately 33. However, the proposed approach generated TPs that are all FTPs and thus had a success rate of 100%. This was not the case with the previously defined approach where the success rate was 80%. This shows that for this EFSM case study the proposed approach outperformed the previously defined TP fitness metric approach.

The fitness metric values of the infeasible TPs observed on each alternative set were the same (136, 142 and 136). By considering the TPs' IDs, the corresponding TPs generated by the proposed approach are associated with greater TP fitness metric values (820, 840 and 855). This suggests that the approach that depends only on the guidance of the TP fitness metric is unlikely to overcome the counter problem. That is, the search aims to minimise the TP fitness metric value, however, there exist alternative TPs that are associated with low

fitness metric values but correspond to infeasible TPs (such as the infeasible TPs observed on the alternative sets). Furthermore, the corresponding feasible TPs are associated with greater fitness metric values. Thus, the search that merely uses the guidance of the proposed TP fitness metric is unlikely to favour the TPs with greater TP fitness metric values and so is unlikely to work in such a situation. By using the proposed technique in this chapter, the search was directed to look for a subsequence of transitions that are required in the presence of the counter behaviour. Therefore, the search was successful to progress towards TPs that are feasible in the presence of the counter behaviour.

From Table 5.4, it is also clear that the TP length factor did not play a role in worsening the TP fitness metric value when this was longer. This observation was also previously spotted (Chapter 3) on the subject TPs derived from Inres initiator by using the TP fitness metric approach. As mentioned before, this machine is already facilitated with 'escape' transitions and so for longer TPs, the search tends to produce more instances of 'escape' transitions since these do not incur any penalty and thus do not worsen the overall TP fitness metric value. Nevertheless, the proposed TP length that is derived by using Dijkstra's algorithm was adequate to allow producing TPs that fulfil the counter requirements in terms of an adequate length that allows repeating an updater.

Since the best achieved TP fitness metric values were the same for the same TPs' IDs in each produced set of subject TPs, the subject TPs produced by each approach were grouped according to their best achieved TP fitness metric values. Then, these were plotted against the average number of generation required by the path test case generator in ten tries. Figure 5.11a shows the entire generated subject TPs by using the proposed approach (GA-1) whereas those generated by using the previous approach (GA-2) are plotted in Figure 5.11b.

From Figure 5.11a, it is clear that TPs that suffer from the counter behaviour where associated with greater fitness metric values but were easily triggered (required just one generation). For remaining TPs, there is a significant strong correlation (Pearson r = 0.976) between the average number of path test case generations and the associated fitness metric values. A similar observation is also exhibited by the TP fitness metric approach (GA-2) shown in

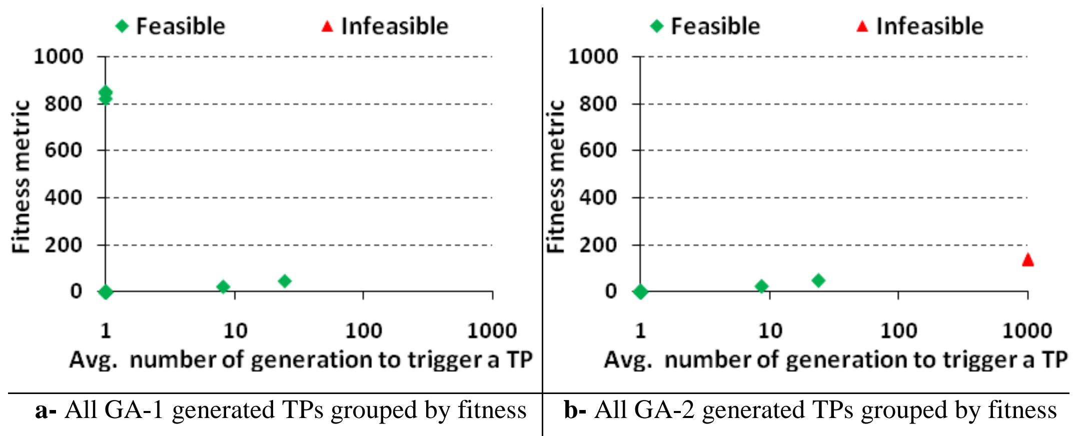


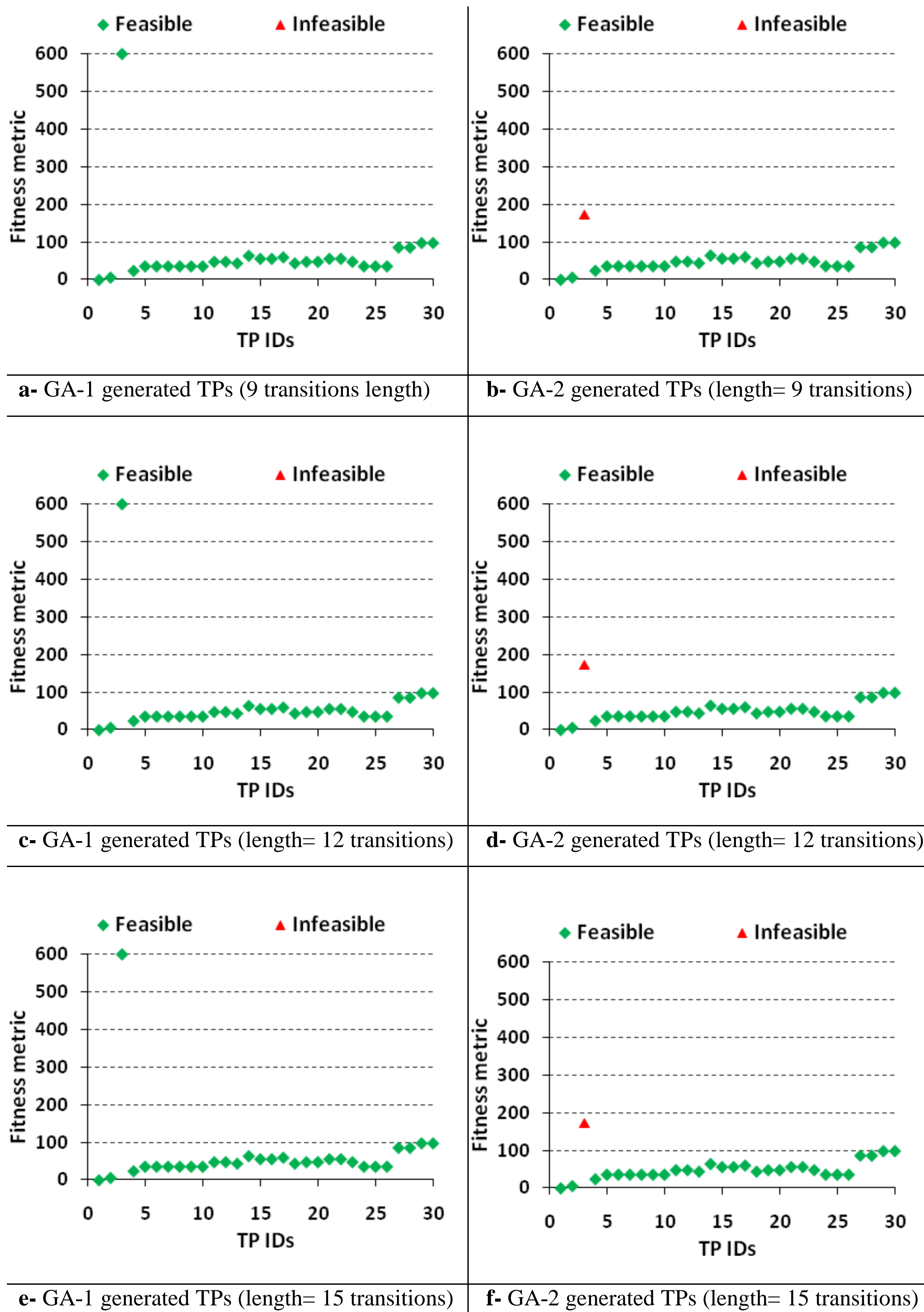
Figure 5.11: Inres initiator EFSM TPs. Each figure shows a TP fitness metric value vs. the average number of generations in ten tries to trigger this TP. Figure *a* plots all the generated TPs by using the proposed approach (GA-1). Figure *b* plots all the generated TPs by using the previously defined TP fitness metric approach (GA-2). For clarity, the horizontal axis is a logarithmic scale.

Figure 5.4b. This shows that the TP fitness metric values associated with TPs that suffer from the counter behaviour do not reflect the actual TP complexity in terms of how difficult such TPs can be triggered. This suggests that a further calibration is required for the TP fitness metric approach when evaluating TPs that reference counter variables.

5.5.2.2 Results of the ATM EFSM

The proposed approach (GA-1) was applied to the ATM EFSM to generate three sets of subject TPs. Each set provided, when all TPs are FTPs, a transition coverage test suite for the considered ATM machine. The first set contained subject TPs with length of 6 transitions whereas the second and third sets comprised subject TPs with lengths of 12 and 15 transitions respectively. Furthermore, the previously defined TP fitness metric approach (GA-2) was applied to generate three similar alternative sets of subject TPs for comparison purpose.

Figure 5.12 shows all the generated sets of subject TPs for ATM EFSM by using the two approaches. The left column shows the sets *a*, *c* and *e* that were generated by the proposed approach (GA-1). The right column shows the sets *b*, *d*,



a- GA-1 generated TPs (9 transitions length)

b- GA-2 generated TPs (length= 9 transitions)

c- GA-1 generated TPs (length= 12 transitions)

d- GA-2 generated TPs (length= 12 transitions)

e- GA-1 generated TPs (length= 15 transitions)

f- GA-2 generated TPs (length= 15 transitions)

Figure 5.12: ATM EFSM TPs. The sets *a* & *b* have a TP length = 9, sets *c* & *d* have a TP length = 12 and sets *e* & *f* have a TP length = 15. Sets *a*, *c* & *e* are generated by using the proposed approach (GA-1). Sets *b*, *d* & *f* are the alternative sets that are generated by using the previously defined TP fitness metric approach (GA-2).

and *f* that were generated by the approach that uses the previous TP fitness metric (GA-2). Each set is plotted in terms of the ID of the generated TP against its best achieved TP fitness metric value.

From the left column of Figure 5.12, all the sets of subject TPs generated by using the proposed approach (GA-1) were feasible. Furthermore, these sets, in terms of TPs' IDs and their best achieved fitness metric values, were alike. Nevertheless, every set of subject TPs generated by using the previous TP fitness metric approach (GA-2) contained one TP which is infeasible. This applies to all the GA-2 sets when the TP ID is 3. A close inspection showed that these TPs are infeasible since they reference a counter variable where the required *updater* did not occur sufficiently.

For the sets that were generated by the proposed approach (GA-1), TPs that reference counter variable were feasible but associated with greater fitness metric value than that observed on the alternative sets. This applied to the TPs with ID number 3. From the left column of Figure 5.12, the GA-1 TPs that referenced a counter variable were associated with TP fitness metric value of 600. However, in the alternative sets (shown in the right column of Figure 5.12) the fitness metric value was 172. This is consistent with the previous conclusion that the formerly proposed TP fitness metric approach is unlikely to produce FTPs when a given TP references a counter variable. This is because alternative infeasible TPs exist and are associated with lower fitness metric values and so the search is likely to progress towards such TPs. However, the results show that the proposed approach enhanced the TP fitness metric approach and guided the GA search towards TPs that were feasible though they were associated with greater TP fitness metric values.

Similar to the results achieved from Inres, if TPs that suffer from counter problem are excluded (TPs with ID =3), all the derived sets of subject TPs by using both approaches were alike. This shows that if the machine does not suffer a counter behaviour, the proposed approach functions by using the TP fitness metric approach. However, with the existence of counter variable, the proposed approach outperformed the previously defined TP fitness metric approach.

Method	TP Length	Feasible	Infeasible	Avg. Fitness ≈
GA	9	30	0	67
RA		29	1	52
GA	12	30	0	67
RA		29	1	52
GA	15	30	0	67
RA		29	1	52
Total GA	9, 12, 15	90	0	67
Total RA	9, 12, 15	87	3	52

Table 5.5: **The ATM EFSM generated TPs by the proposed approach (GA-1) and by the previously defined TP fitness metric approach (GA-2).**

Table 5.5 shows the summary of the results derived from the ATM EFSM. For each considered TP length, the proposed approach (GA-1) produced a set of TPs that were feasible and thus had a success rate of 100%. However, the alternative approach (GA-2) had a success rate of 96.6%. For the proposed approach, the best average achieved TP fitness metric value was 67 and this was the same for each set of subject TPs. Similarly, for the alternative approach, the best average achieved TP fitness metric value was 52 for each set of subject TPs. This shows that the proposed approach produced greater average fitness metric values than the alternative one. This emphasises that when the counter problem exists, there can be TPs that are associated with lower TP fitness metric values but they are infeasible.

Furthermore, for each approach (GA-1 & GA-2) the best average TP fitness metric in each set did not increase when the TP length increased. This shows that the existence of an 'escape' transition is important to avoid unnecessary complexities when producing TPs that have extra length. In order to validate this, Figure 5.13 shows the prevalence of the 'escape' transition, t', for each TP in the three sets of subject TPs that were derived from ATM EFSM by using the proposed approach. This figure shows that when TPs are longer, the search avoids extra penalties by targeting the 'escape' transition. Consequently, the longer the TP is the more likely the 'escape' transition is to be repeated. Although this trend is relatively clear in Figure 5.13, there are cases where the search did not particularly make use of the 'escape' transition t' (i.e. TP ID 22).

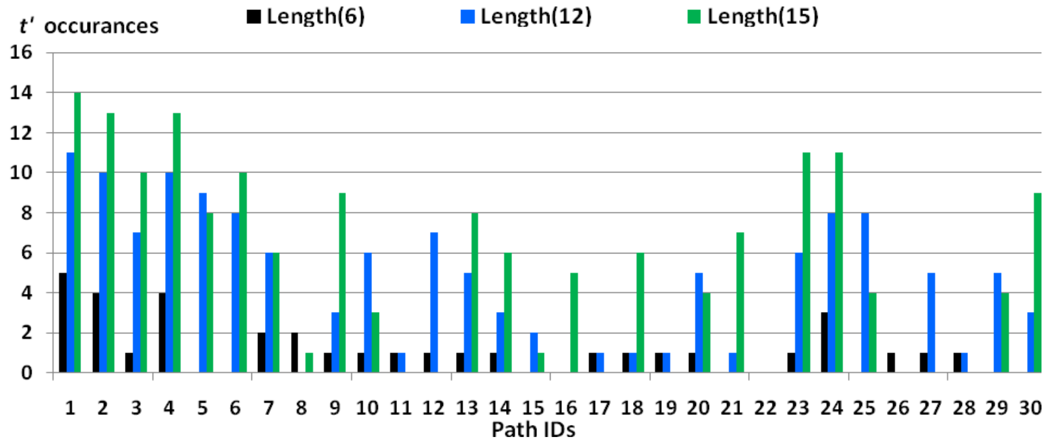
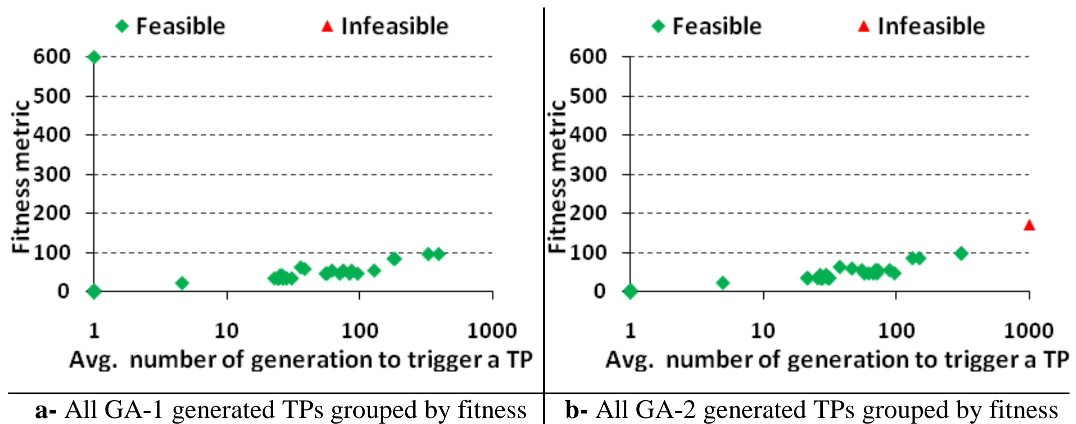


Figure 5.13: The prevalence of the 'escape' transition *t′* in the three sets of subject TPs derived from the ATM EFSM by using the proposed approach.

This can be explained by considering the ATM structure shown in Figure 5.2. Since there are already other transitions that represent 'escape' ones such as t_7, t_8, t_9, t_{10}, t_{25} & t_{26}, the search also targeted these transitions when TPs were longer.

However, the use of an 'escape' transition at the first state is still important even if the machine is already facilitated with other 'escape' transitions at other states. This can be verified by comparing the ATM results achieved by the TP fitness metric approach to the results derived by the same approach on Chapter 3 where the considered ATM did not have the particular 'escape' transition *t′*. By referring to (Table 3.8, pp. 89), the best average achieved TP fitness metric was 60, however as reported in Table 5.5, this value is dropped down to 52 even though the same approach was applied in both cases. This shows that the search can better use the 'escape' transition when it is presented at the first state. This finding suggests that when deriving FTPs by using the proposed TP fitness metric approach, the addition of an 'escape' transition can be useful in two aspects. Firstly, it helps the search in avoiding unnecessary transitions in the generated TPs and consequently it is likely to avoid extra complexity. Secondly, when the length of the generated TPs is difficult to reason about, the 'escape' transition at the first state can ease the problem.

By considering the length factor, the results achieved in the experiment show that the TP length of 6 transitions derived by using Dijkstra's algorithm was sufficient. Furthermore, longer TP length did not improve the performance of the alternative approach (i.e. increasing the chance by which the approach (GA-2) can



a- All GA-1 generated TPs grouped by fitness | **b-** All GA-2 generated TPs grouped by fitness

Figure 5.14: ATM EFSM TPs. Each figure shows a TP fitness metric value vs. the average number of generations in ten tries to trigger this TP. Figure *a* plots all the generated TPs by using the proposed approach (GA-1). Figure *b* plots all the generated TPs by using the previously defined TP fitness metric approach (GA-2). For clarity, the horizontal axis is a logarithmic scale.

produce a correct sequence of transitions to bypass the counter problem). This supports the observation that the proposed approach enhances the TP fitness metric approach in the presence of the counter problem.

Finally, since for each approach the generated sets of subject TPs were similar, each three sets that belong to the same approach (GA-1 or GA-2) were grouped by the TP fitness metric values. Then, these were plotted against the average number of generations required by the path test case generator in ten tries to trigger each TP. Figure 5.14 shows two plots where Figure 5.14a represents the subject TPs derived by the proposed approach (GA-1) and Figure 5.14b represents the subject TPs derived by the alternative approach (GA-2). From these Figures (5.14a & 5.14b), there is a significant strong correlation (Pearson r = 0.801) between the TP fitness metric value and the average number of generations that were required to trigger each FTP. However, this was not the case for the FTPs that suffer from the counter behaviour. For example, Figure 5.14a shows that the TP ID 3 is associated with a TP fitness metric value of 600 but this TP required only one generation to trigger. This finding supports the previous finding (from Inres initiator EFSM) where the TP fitness metric does not seem to reflect the actual complexity of given FTPs when these suffer from the counter problem.

EFSM	Method	Total TPs	FTPs	Avg. Fitness ≈	Success Rate
Inres	GA-1	45	45	173	100%
	GA-2		36	33	80%
ATM	GA-1	90	90	67	100%
	GA-2		87	52	96.6%
All	GA-1	135	135	Not applicable	100%
	GA-2		123	Not applicable	91.1%

Table 5.6: **Summary of the results achieved by the proposed approach (GA-1) and the previously defined TP fitness metric approach (GA-2) on generating FTPs from two EFSMs that suffer from the counter problem.**

This motivates a need for a further investigation into the proposed TP fitness metric approach (Chapter 3) when evaluating such TPs.

5.5.2.3 Summary of Results

Table 5.6 reports the summary of the results achieved from the experiment. From this Table, the proposed approach successfully guided a GA search towards FTPs and the counter problem was bypassed. Therefore, the proposed approach had a success rate of 100%. In contrast, the previously defined TP fitness metric approach (Chapter 3) did not overcome the counter problem even when TPs were allowed to be longer. The success rate associated with the latter approach was approximately 91.1%.

This provides evidence that the proposed approach in this chapter enhanced the previously defined TP fitness metric approach and consequently provided better guidance towards TPs that are feasible when the counter problem exists. Furthermore, the experimental results suggest that utilising Dijkstra's algorithm in the way described in this chapter can allow reasoning about the length of the generated TPs. Moreover, the results highlight the importance of facilitating a given machine with an 'escape' transition at the first state if the machine does not already have one. This has two benefits. First, it helps avoiding unnecessary complexity when the considered TP length (to be generated) is longer than necessary. Second, it can ease the problem of deciding about a suitable TP

length where it allows producing variable-length TPs by using a fixed TP length approach.

The results also show that the proposed TP fitness metric calculation is not always suitable when TPs suffer a counter problem. In particular, there are cases where the actual TP's complexity is not reflected by the overall assigned penalty. This, in turn, suggests the need for further investigation in order to calibrate the TP fitness metric.

5.6 Conclusion

The EFSM is a powerful modelling approach and has been widely applied when testing from state-based systems. Despite its popularity, testing from an EFSM is complicated by the presence of infeasible transition paths. The problem of generating feasible transition paths from an EFSM is generally uncomputable. One particular problem that causes paths to be infeasible is the existence of a counter variable whose value depends on previous transitions. Thus, in order to generate a feasible path to exercise a particular transition which has a guard that references a counter variable, there is a need to ensure that other transitions, that update the value of the counter, have occurred sufficiently often. Previous studies have shown that the counter problem can adversely affect methods that automate test generation from an EFSM.

This chapter presents an approach based on the analysis of data and control dependency to automatically determine whether a transition's guard references a counter variable, which other transitions are involved and the number of times they have to be called. The approach is then utilised the TP fitness metric approach to search automatically for feasible transition paths that satisfy a test criterion.

In the experiment, the proposed approach was utilised to guide a GA search to generate subject TPs from two EFSM case studies that suffer from the counter problem. For each EFSM three sets for subject TPs were derived with different TP lengths where the minimum adequate TP length was calculated by

utilising Dijkstra's algorithm. Each derived set of subject TPs provided transition coverage test suite for the considered EFSM. For the purpose of comparison, similar alternative sets of subject TPs were generated by using the guidance of the previously proposed TP fitness metric. The TPs generated by using the proposed approach were all feasible and so the approach had a success rate of 100%. However, in the alternative sets, TPs that suffer from the counter problem were found to be infeasible. The experimental results show that the proposed approach effectively guided the GA search towards TPs that were feasible and the counter problem was successfully bypassed.

Future work in this area could investigate other cases of the counter problem that might exist in order to extend the current approach so it includes these cases. Furthermore, the TP fitness metric values associated with FTPs that reference counter variables did not reflect the actual complexity a TP may have. This was observed by considering the average number of generations that a GA path test case generator required to trigger such FTPs. This shows that the penalty values, incurred as a result of repeating the updater for a certain number of times, may not be necessary. A further investigation on this would be useful to allow a more representative TP fitness metric value to be assigned to TPs that reference counter variables.

Chapter 6: Conclusions and Future Work

6.1 Conclusion of Achievements

The thesis's main aims and objectives were:

1. To identify challenges associated with testing from extended finite state machine (EFSM) models. Furthermore, highlighting limitations associated with current EFSM testing approaches.
2. To determine barriers to the application of search-based approaches to testing from EFSM models.
3. To propose an integrated search-based approach to automate the process of testing from EFSM models.

6.1.1 Problems Associated with Testing from EFSMs

Chapter 2 determined the importance of model based testing and particularly the wide use of an EFSM as a modelling approach. Furthermore, the chapter identified the main two problems associated with EFSM testing:

1. Since an EFSM combines both control and data aspects of a system, generating feasible transition paths (FTPs) from an EFSM to satisfy a given test adequacy criterion is a substantial problem.
2. When there is a set of FTPs that are derived from an EFSM, these FTPs require suitable input values in order to be triggered. Nevertheless, finding such values is a challenging task since the input domain can be considerably large, while the suitable input values may constitute just a very small subset of the input domain. This is because FTPs usually

include guards narrowing the acceptable range of input values to a certain small sub-ranges.

The chapter also determined that many current EFSM testing approaches tackled these problems by:

1. Converting an EFSM to an FSM (by either expanding an EFSM or abstracting an EFSM) to avoid the path feasibility problem and thus apply the widely studied FSM-based testing techniques. However, this approach may either lead to the state explosion problem (when expanding) or the path feasibility problem (when abstracting).
2. Rewriting an EFSM to form another EFSM in which the path feasibility problem does not exist. Such approaches are either limited to certain EFSMs or are not automated.
3. Using symbolic execution and constraint satisfaction techniques to study the path feasibility and also to produce test cases to trigger the generated paths. However, such approaches inherit the applicability limitations of both symbolic execution and constraint satisfaction.

Moreover, Chapter 2 showed that while search-based approaches have proven efficient in automating aspects of testing, these have received little attention when testing from an EFSM. The search-based testing approaches are particularly suitable for automating the test from an EFSM because:

1. The complex nature of the problems associated with an EFSM testing in terms of feasible path generation and test cases generation where:
    a. The search space for the path generation problem is large (theoretically, there can be an infinite number of paths that can be formed from a given EFSM).
    b. The search space for the problem of test cases generation to trigger a given path can also be considerably large.
2. Search-based approaches have proven efficient in automating aspects of white-box testing.

The proposed approach in this thesis handled testing from an EFSM by generating feasible transition paths (FTPs) that satisfy a given test criterion, then generating test cases (test data) that can exercise the generated FTPs.

6.1.2 Generating Feasible Transition Paths (FTPs)

For an EFSM, there can be a large number of transition paths that can be formed. Due to the complex nature of an EFSM model, some transition paths may be infeasible. When testing from an EFSM according to a given test criterion, there is a need to generate a set of transition paths to satisfy this criterion. Chapter 3 proposed a novel search-based approach to generate transition paths that are *likely* to be feasible. To facilitate the search for such FTPs, transitions in an EFSM were classified into two types: *affecting* and *affected-by*. An affecting transition is one which has an assignment operation that affects a guard of another transition (the affected-by).

Furthermore, assignment operations were classified into three types that represent a context variable being assigned a value based on (1) a parameter value, (2) a constant value and (3) context variables value. Similarly, guards were classified into five types that represent a comparison that involves (1) only parameters, (2) parameters and constants, (3) variables and parameters, (4) only variables, and (5) variables and constants. The fitness metric was then calculated for each transition path by detecting all pairs of (affecting, affected-by). Then for each pair of (affecting, affected-by), and depending on the assignment and guard types that appear in the (affecting, affected-by) respectively, a penalty was assigned. The penalty is a numerical estimation of how difficult it is to satisfy the guard of the affected-by transition by considering the type of the affecting transition's assignment. Furthermore, transitions that are not affected by any transition but have guards (such as comparison among parameters), these are also penalised.

An algorithm was then proposed to calculate the fitness metric of a given transition path. Importantly, the proposed TP fitness metric can be computed quickly and thus is suitable to be used in a search-based approach. Furthermore, to apply a search-based approach, a TP encoding method was adapted from the literature. A GA search that implemented the proposed TP fitness metric approach was applied to five EFSM case studies. For each case study, three different TP lengths were used. Also, a random path generator was applied for the purpose of

comparison. Experimental results showed that generating FTPs from the considered machines is not an easy task. However, the proposed search-based approach was found to generate FTPs that are mostly feasible regardless of the TP length. However TPs that were found to be infeasible, suffered from the counter problem.

6.1.3 Generating Test Cases to Trigger FTPs

In order to apply a test to an EFSM, FTPs that were previously derived (to satisfy a test criterion) need to be triggered. Since a transition can require some input values in order to be executed, it is necessary to provide such values. However, a transition can also have guards that need to be satisfied in order for this transition to be fired. Furthermore, for a given FTP, there is a need to find a test case that can trigger each transition in this path. Such a test case should consider the effects of triggering the earlier transitions in the path on the later transitions. These effects take place through the machine context variables. This, in turns, makes the process of finding a suitable test case to trigger a given FTP difficult.

Chapter 4 proposed a search-based approach that can automatically generate test cases to trigger given FTPs through an EFSM. The approach considered an EFSM transition as a function. Thus, the problem of test cases generation was formulated as finding input values to take a sequence of functions. These functions interact with each other through a set of global variables (the machine's context variables). Then, a fitness function to facilitate the search in the presence of function calls was derived. The fitness calculation was performed through two steps. First, a function distance was calculated to determine whether a given function was achieved or how close the applied inputs were to executing the function. The function distance was calculated by using a method proposed in the literature for calculating the fitness for nested conditions. Then for each guarded function in the path, a function approach level was measured. The function approach level determines how close a test case was to triggering the last transition in the path and calculated by subtracting one from the number of guarded transition away from the last transition. The final fitness was then

composed of two components: the normalised function distance and the function approach level at the transition where the execution flow was diverted.

Subject TPs that were previously derived from the five EFSM case studies by using the TP fitness function approach and random path generator were the experiment subjects. Furthermore, the experiment used two other search-based approaches to find an FTP's test case for comparison. The first approach was taken from the literature while the second approach was a random test case generator. The experimental results reported that the proposed test cases generation approach successfully generated test cases that triggered all the used FTPs. Also, the proposed approach was found to be superior to the other two search-based approaches.

Chapter 4 also studied the relationship between two factors: the FTP's fitness metric value and the effort, in terms of time, that is required to trigger the FTP. Furthermore, an investigation of the FTP's fitness metric capability to predict such an effort was also studied. In the second experiment, a constraint satisfaction approach was used together with the proposed search-based FTP test cases generator. The subject FTPs were the same ones that were generated in Chapter 3.

The salient results of the second experiment stated that when FTPs were generated by the proposed search-based approach that implemented the TP fitness metric, there was significantly positive strong correlation. The correlation was between the FTP's fitness metric value and the effort that was required to trigger the FTP. This correlation was found to be strong regardless of the applied test case generator. Furthermore, the study found that the FTP fitness metric value can significantly predict the time that is required by a test cases generator to trigger an FTP. The predication capability of the fitness metric together with the relationship between the fitness metric and the effort of a test cases generator were found to be stronger when the subject FTPs were clustered by the same fitness metric values.

For FTPs that were randomly generated, the correlation between the FTP's fitness metric value and the effort that was required to trigger the FTP by the proposed test cases generator was found to be statistically significant and strongly positive. However, there was a small positive correlation between the FTP's

fitness metric value and the time required by the constraint satisfaction approach to trigger the FTP. Furthermore, the capability of the FTP fitness metric to predict the required time to trigger an FTP was found to be significant but explained less variance in the required time by the proposed test cases generator. The predication capability of the FTP fitness metric was much less for the constraint satisfaction approach. Finally, FTP clustering was found to worsen both correlation strength and the predication capability among the considered factors.

6.1.4 By-Passing The Counter Problem When Generating FTPs

A transition path through an EFSM can be infeasible due to an *opposition* between guards or a guard and an assignment. Such oppositions can be static when guards and assignments involve constants. When there is a static opposition, this can be captured by the proposed TP fitness metric approach and harshly penalised so that the search can avoid it. However, oppositions between guards or a guard and an assignment can be dynamic where a certain infeasible path would be feasible were certain transitions to occur sufficiently often. Such transitions update the value of an internal variable (a context variable) so that the guard of a later transition is satisfied and the path becomes feasible. In the EFSMs studied, such a variable plays the role of a counter. Therefore, when a path has a transition's guard that references a counter variable, there is a need to determine which other transitions have to occur earlier and also how many times they should occur so that the considered guard is satisfied.

Chapter 5 proposed a novel approach to determine whether a given EFSM contains a counter variable. If so, the EFSM transitions were classified to two types: affecting a counter and affected by a counter. Transitions that affect a counter can be either *initialisers* which initialise the counter variable with a constant or *updaters* that update the counter variable by using one of the $\{+, -, \times, /\}$ operations and a constant. Transitions that are affected by a counter contain a guard that references a counter variable.

Then, an algorithm was defined that can produce triples of the form (initialiser, updater, updater times). These triples state which particular initialise should come first, which particular updater should follow and how many times the updater should be repeated. The algorithm was then combined in the TP fitness metric approach to inform the search when a path should meet an extra requirements to be potentially feasible.

Since the counter problem required a given path to have an adequate length to allow certain transitions to occur sufficiently often, the path length was also studied. Dijkstra's algorithm was utilised to suggest a path length that can be considered sufficient (long enough). However, when a path length is longer than that actually required, extra complexity can be introduced by calling unnecessary transitions. This problem was overcome by adding an 'escape' transition at the first state of an EFSM so that an extra length can potentially be occupied by instances of the 'escape' transition.

The approach was then applied to two EFSM case studies that suffer from the counter problem. Also, the original TP fitness metric approach was applied for the purpose of comparison. The experimental results reported that the proposed approach in Chapter 5 significantly enhanced the performance of the TP fitness metric approach and the counter problem was successfully surpassed. However, the generated FTPs that referenced a counter variable were found to be associated with greater TP fitness metric values. These values did not seem to reflect the FTP complexity in the same way that was observed when FTPs did not suffer from the counter problem.

6.2 Points for Future Work

There are many directions in which the work of this thesis can be extended. These directions are listed in the next subsections.

6.2.1 The Fitness Metric

The proposed TP fitness metric is designed to penalise guards according to their types. For example, consider the flowing two sets of nested guards:

if $p_1 > c_1$ then
    if $p_2 > c_2$ then
        if $p_3 > c_3$ then
            // target
	if $p_1 > c_1$ then
	    if $p_1 < c_2$ then
	        if $p_2 > c_3$ then
	            // target

a- Case study of penalty assignment	**b-** Equivalent penalty but yet harder to satisfy

Figure 6.1: Two sets of nested guards that are assigned the same TP fitness metric value. However, the second set is harder to be satisfied.

Both sets (Figure 6.1a and Figure 6.1b) have guards involving parameters and constants. The fitness metric mechanism gives both sets the same penalty value. However, in the first set, each input parameter is bounded once while the second set has one input parameter which is bounded twice. The first set is likely to be much easier to achieve than the second set. This is because, in the first set the range of acceptable values of p_1 is much larger than that in the second set where this range is narrowed down to, possibly, a tiny range. Therefore, it would be useful if the fitness metric could consider how many times the same variable is constrained when the comparison operator is not the equal one. This could improve the path's complexity reflected by the TP fitness metric.

Another point of fitness metric improvement could be the case when a transition path suffers from the counter problem. The current fitness metric approach is to penalise each pair of (affecting, affected-by) with a certain number of points. However, when there is a counter variable, there is a need for a certain transition, an updater, to occur a specific number of times. Since an updater can be affecting and affected-by at the same time, each occurrence of the updater will incur extra penalty. Nevertheless, such extra penalty might not actually reflect an

// Transition t
if $v < c_1$ then
    $v = v + 1$
	// Transition t'
	if $v \geq c_1$ then
	    // target

a- A transition that is affected-by and also affecting a counter variable	**b-** A transition that is only affected by a counter variable

Figure 6.2: A counter problem and a TP fitness metric mechanism.

// *Transition* t_1
if $p_1 > c_1$ then
    if $p_2 > c_2$ then
        if $p_3 > c_3$ then
            $v = v + p_1$ // *target*-1
 	
// *Transition* t_2
if $p_1 > v$ then
    $v = p_1$ // *target*-2
	double path (int p_1, int p_2, int p_3, int p_4)
	{ if $p_1 > c_1$ then
	    if $p_2 > c_2$ then
	        if $p_3 > c_3$ then
	            $v = v + p_1$ // *target*-1
	            if $p_4 > v$ then
	                $v = p_4$
	                return 0 // *main target*
	    // *else return normalised(branch distance)*
	    // *+ approach level at the false branch*}

a- Two guarded transitions	**b-** A path fitness calculation by using ‘bushing’

Figure 6.3: Calculating a path fitness by using only Wegener et al. approach

additional path’s complexity as a result of repeating the updater. Consider for example transition *t* and *t′* shown in Figure 6.2. Transition *t* is an updater and also affected by a counter while transition *t′* is just an affected-by a counter. Transition *t* however should occur for a number of times so that the guard of *t′* is satisfied. While, *t* is repeated, a new pair of (affecting, affected-by) is formed by (updater, updater) and thus new penalty is incurred. However, these repetitions do not necessarily increase the complexity. Therefore, developing a method (i.e. a filter) that drops penalties assigned to (updater, updater) might help when FTPs suffer from the counter problem.

6.2.2 FTPs Test Cases Generations

The approach presented in Chapter 4 manipulates a given feasible transition path as a sequence of functions. Then, the problem is tackled by triggering the functions in their order in the considered path. Another potential way to calculate the fitness of a path is to merge the path’s functions altogether in one function and then compute the fitness by using the approach of (Wegener et al., 2001). Consider for example transitions t_1 and t_2 shown in Figure 6.3a. The fitness of the path $t_1 t_2$ can be calculated as shown in Figure 6.3b. The main idea is to nest the guards of the two transitions in one transition by considering the order of both guards and assignments.

6.2.3 The Normalisation Function During FTPs Test Cases Generation

The proposed fitness function that evaluates a given test case to trigger an FTP consists of two components: the *function distance* and the *function approach level*. Since it is important to reward a test case that achieves more transitions, the function distance is normalised to a value in the range [0..1]. Furthermore, in order to calculate the function distance for each transition in a path, there are two components to be calculated: the *branch distance* and the *approach level*. Again, the branch distance is normalised to a value in the range [0..1] so that the approach level can reward a test case that exercises more branches (guards) in a transition. This shows that the proposed fitness function that evaluates an FTP's test case uses the normalisation function a number of times depending on the number of the guarded transitions in a given FTP. A recent study (Arcuri, 2010) focused on the way in which a normalisation function is implemented. The study compared the commonly used form of the normalisation function (Equation 6.1) to a new form proposed by the study (Equation 6.2).

$$\textbf{\textit{norm}}(x) = 1 - \alpha^{-x} \quad ; \textit{where } \alpha > 0 \textit{ is a constant} \qquad (6.1)$$

$$\textbf{\textit{norm}}(x) = x / (x + \beta) \quad ; \textit{where } \beta > 0 \textit{ is a constant} \qquad (6.2)$$

The study argued that the commonly used form requires running the *math* library (to calculate the *power* function) every time the *norm* function is called. Furthermore, the conventional *norm* function can be more vulnerable to precision error. By considering the fact that the proposed fitness function is calculated a large number of times during FTPs' test cases generation, the efficiency of the proposed fitness function could be improved by using the one proposed in the study (Equation 6.2).

6.2.4 Complex FTPs' Test Cases Generation

One important aspect of the proposed TP fitness metric is to target transition paths that are potentially associated with the least fitness metric values. Therefore, such paths are likely to be less complex and thus relatively easy to trigger. However,

// Transition t_1
if $p_1 > c_1$ then
    if $p_1 < c_2$ then
        // target
	// double stretched (int p_1, int aux_1, int aux_2)
	{ if ($p_1 + aux_1$) > c_1 then
	    if p_1 < ($aux_2 + c_2$) then
	     if (aux_1+ aux_2 == 0) then
	        // target

a- A transition with nested guards	**b-** Calculate the fitness by using 'Stretching'

Figure 6.4: Calculating a path fitness by using 'Program Stretching' approach (Ghani and Clark, 2009b).

complex paths could be better able to reveal errors than simple ones. The TP fitness metric approach can be adjusted to produce paths with large TP fitness metric values but they are still likely to be feasible. However, the problem that may arise is to generate test cases that can trigger such complex paths (FTPs that have large TP fitness metric values).

One important technique to overcome this problem is the approach of 'program stretching' proposed by (Ghani and Clark, 2009b). Consider for example transition t_1 shown in Figure 6.4 which has two nested IF statements. The idea of program stretching is to use auxiliary variables in each IF statement. Each auxiliary variable has initially a large value which makes each guard initially satisfied. Then, the target becomes to have the sum of these auxiliary variables equal to zero while the guards still hold (Figure 6.4b). The 'program stretching' approach was found to enhance the search-based testing in the presence of difficult guards and thus fulfil the problem of test cases generation for complex FTPs.

6.2.5 Calibrating the TP Fitness Metric

The penalties that were used by the TP fitness metric when generating transition paths are by no means definitive. Although these penalties were found to be potentially reasonable to reflect paths' complexity, a further calibration would be useful. A possible way to achieve this is to use feedbacks from a test cases generator and reflect these on the penalties as shown in Figure 6.5. Such an approach could be used to learn about how much a path's complexity, in terms of the time required to be triggered, is actually reflected by the associated TP fitness metric value. Then this information is reflected on the used penalties in order to

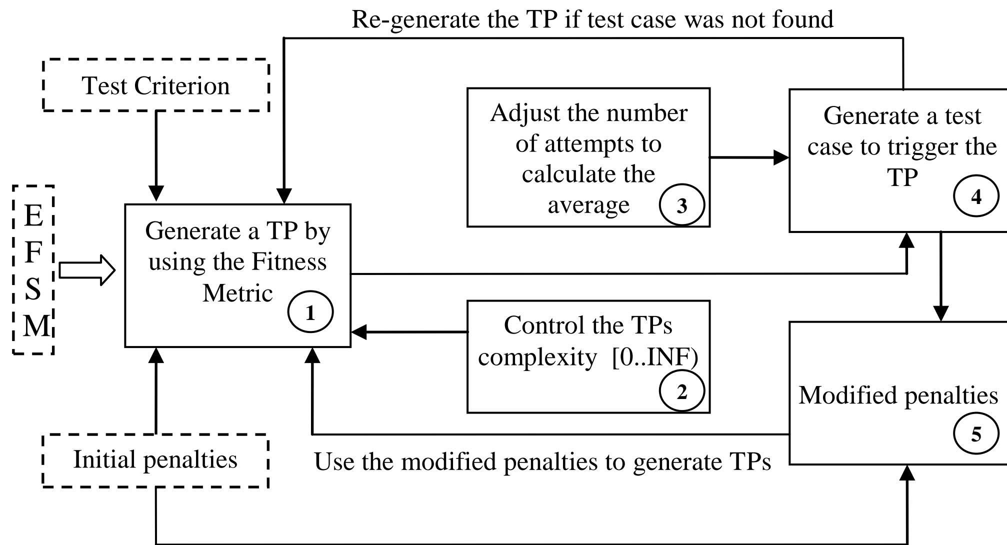


Figure 6.5: An iterative approach to calibrate the TP fitness metric

calibrate and thus the TP fitness metric. Figure 6.5 shows five stages: the first stage is to generate a TP by using the initial penalties while stage 2 can be used to determine the desirable range of TP complexity. The number of attempts that are allowed for a test cases generator to trigger a TP can be adjusted to decide how the average time is calculated (Stage 3). The recorded average time is then used to modify the penalties by considering their initial values (Stage 5). Finally, the modified penalties are used to generate new TPs, and so on.

6.2.6 An Iterative Approach

The proposed approach has the potential to be implemented as an iterative process as shown in Figure 6.6. Such a process starts by generating a path in the first stage while the complexity of the generated path can be controlled by using the second stage. Then, in the third stage, the path is checked for the counter problem. If the counter problem is detected, the path may be regenerated. The next stage, number 4, is to accept the generated path if its estimated time (to be triggered) fits in the allowed time (i.e. a time dedicated for testing). If this path is not accepted, a request to regenerate the path is sent. Similarly in the final stage, if a test case cannot be found to trigger the path, a request to regenerate the path is sent. Such

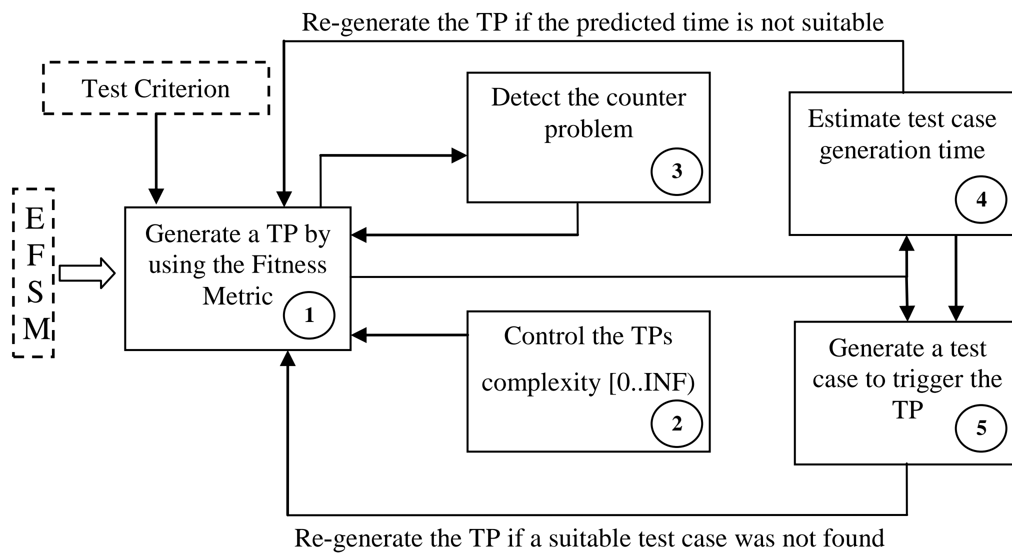


Figure 6.6: An iterative search-based approach to test from an EFSM

an iterative search-based approach can be useful to provide important flexibility to the testing process.

6.3 General Conclusion

The work of this thesis investigated the application of search-based approaches to the domain of EFSM testing. An integrated search-based approach was then developed for the purpose of an automating testing from an EFSM. The proposed approach comprised of three stages where each stage tackled one important problem associated with EFSM testing.

A set of experiments showed the value of the proposed approach in automating the process of testing from EFSM models.

References

Aho, A. V., Dahbura, A. T., Lee, D. & Uyar, M. U. (1991) An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours. *Communications, IEEE Transactions on,* 39**,** 1604-1615.

Andrews, A. A., Offutt, J. & Alexander, R. T. (2005) Testing Web applications by modeling with FSMs. *Software and Systems Modeling,* 4**,** 326-345.

Apfelbaum, L. & Doyle, J. (1997) Model Based Testing. *10th International Software Quality Week***,** pp. 296-300. May 1979.

Arcuri, A. (2010) It Does Matter How You Normalise the Branch Distance in Search Based Software Testing. *Third IEEE International Conference on Software Testing, Verification and Validation (ICST10)***,** pp. 205-214. IEEE Press, April 2010.

Baresel, A., Binkley, D., Harman, M. & Korel, B. (2004) Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. *ACM SIGSOFT international symposium on Software testing and analysis***,** pp. 108-118. ACM Press, July 2004.

Beizer, B. (1990) *Software testing techniques 2nd ed.*, Van Nostrand Reinhold.

Bertolino, A. (2007) Software Testing Research: Achievements, Challenges, Dreams. *Future of Software Engineering (FOSE '07)***,** pp. 85-103. IEEE Press, May 2007.

Bochmann, G. V. (1990) Specifications of a simplified transport protocol using different formal description techniques. *Computer Networks and ISDN Systems,* 18**,** 335-377.

Boehm, B. W. (1981) *Software Engineering Economics,* NJ, Prentice Hall PTR.

Bourhfir, C., Dssouli, R. & Aboulhamid, E. M. (1996) Automatic Test Generation for EFSM-based Systems. Technical report. University of Montreal, TR-1043.

Browne, M. C., Clarke, E. M., Dill, D. L. & Mishra, B. (1986) Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Transactions on Computers,* 35**,** 1035-1044.

Budkowski, S. & Dembinski, P. (1987) An introduction to Estelle: a specification language for distributed systems. *Computer Networks and ISDN Systems.,* 14**,** 3-23.

Chanson, S. T. & Jinsong, Z. (1994) Automatic protocol test suite derivation. *13th IEEE Networking for Global Communications (INFOCOM '94),* 2**,** pp. 792-799. IEEE Press, June 1994.

Chanson, S. T. & Zhu, J. (1993) A unified approach to protocol test sequence generation. *12th Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future (INFOCOM '93),* 1**,** pp. 106-114. IEEE Press, April 1993.

Cheng, K.-T. & Krishnakumar, A. S. (1996) Automatic generation of functional vectors using the extended finite state machine model. *ACM Transactions on Design Automation of Electronic Systems.,* 1**,** 57-79.

Chow, T. S. (1978) Testing Software Design Modeled by Finite-State Machines. *Software Engineering, IEEE Transactions on,* SE-4**,** 178-187.

Clark, J., Dolado, J. J., Harman, M., Hierons, R. M., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M. & Shepperd, M. (2003) Reformulating software engineering as a search problem. *Software, IEE Proceedings -,* 150**,** 161-175.

Clarke, E. (2008) The Birth of Model Checking. *25 Years of Model Checking.* Berlin, Springer.

Clarke, L. A. (1976) A System to Generate Test Data and Symbolically Execute Programs. *Software Engineering, IEEE Transactions on,* SE-2**,** 215-222.

Cohen, J. (1988) *tatistical power analysis for the behavioral sciences,* New Jersy, Lawrence Erlbaum Associates.

Dahbura, T. A., Sabnani, K. K. & Uyar, M. U. (1990) Formal methods for generating protocol conformance test sequences. *Proceedings of the IEEE,* 78**,** 1317-1326.

Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C. & Horowitz, B. M. (1999) Model-based testing in practice. *International Conference on Software Engineering***,** pp. 285-294. IEEE Press, May 1999.

Darringer, J. A. & King, J. C. (1978) Applications of Symbolic Execution to Program Testing. *Computer,* 11**,** 51-60.

David, H. & Amnon, N. (1996) The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.,* 5**,** 293-333.

Dechter, R. & Pearl, J. (1989) Tree clustering for constraint networks (research note). *Artificial Intelligence,* 38**,** 353-366.

Demillo, R. A. & Offutt, A. J. (1991) Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on,* 17**,** 900-910.

Derderian, K., Hierons, R. M., Harman, M. & Guo, Q. (2005) Generating feasible input sequences for extended finite state machines (EFSMs) using genetic algorithms. *The Genetic and Evolutionary Computation Conference (GECCO)***,** pp. ACM Press, June 2005.

Derderian, K., Hierons, R. M., Harman, M. & Guo, Q. (2006) Automated Unique Input Output Sequence Generation for Conformance Testing of FSMs. *The Computer Journal,* 49**,** 331-344.

Derderian, K., Hierons, R. M., Harman, M. & Guo, Q. (2010) Estimating the feasibility of transition paths in extended finite state machines. *Automated Software Engineering,* 17**,** 33-56.

Dijkstra, E. W. (1959) A note on two problems in connexion with graphs. *Numerische Mathematik* 1**,** 269–271.

Dijkstra, E. W. (1970) Notes on Strcutred Programming. Technical Report 70-WSK03. Eindhoven, Technological University.

Dssouli, R., Saleh, K., Aboulhamid, E. M., En-Nouaary, A. & Bourhfir, C. (1999) Test development for communication protocols: towards automation. *Computer Networks,* 31**,** 1835-1872.

Duale, A. Y. & Uyar, M. U. (2004) A method enabling feasible conformance test sequence generation for EFSM models. *Computers, IEEE Transactions on,* 53**,** 614-627.

Duale, A. Y., Uyar, M. U., Mcclure, B. D. & Chamberlain, S. (1999) Conformance testing: towards refining VHDL specifications. *IEEE Military Communications Conference Proceedings (MILCOM 1999),* 1**,** pp. 140-144. IEEE Press, October 1999.

El-Far, I. K. & Whittaker, J. A. (2001) Model-based software testing. IN MARCINIAK, J. J. (Ed.) *Encyclopedia on Software Engineering.* Wiley.

Elfriede, D., Jeff, R. & John, P. (1999) *Automated software testing: introduction, management, and performance*, Addison-Wesley Longman Publishing Co., Inc.

Ferrante, J., Ottenstein, K., J. & Warren, J., D. (1987) The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems,* 9**,** 319-349.

Fraser, G., Wotawa, F. & E. Ammann, P. (2009) Testing with model checkers: a survey. *Software Testing, Verification & Reliability,* 19**,** 215-261.

Ghani, K. & Clark, J. A. (2009a) Automatic Test Data Generation for Multiple Condition and MCDC Coverage. *Fourth IEEE International Conference on Software Engineering Advances (ICSE '09)***,** pp. 152-157. IEEE Press, September 2009.

Ghani, K. & Clark, J. A. (2009b) Widening the Goal Posts: Program Stretching to Aid Search Based Software Testing. *1st International Symposium on Search Based Software Engineering***,** pp. 122-131. IEEE Press, May 2009.

Gonenc, G. (1970) A Method for the Design of Fault Detection Experiments. *Computers, IEEE Transactions on,* C-19**,** 551-558.

Harman, M. (2008) Open Problems in Testability Transformation. *1st IEEE International Conference onSoftware Testing Verification and Validation Workshop (ICSTW '08)***,** pp. 196-209. IEEE Press, April 2008.

Harman, M. & Clark, J. (2004) Metrics are fitness functions too. *10th International Symposium on Software Metrics (METRICS'04)***,** pp. 58-69. IEEE Press, September 2004.

Harman, M. & Danicic, S. (1997) Amorphous program slicing. *5th International Workshop on Program Comprehension (WPC '97)***,** pp. 70-79. IEEE Press, May 1997.

Harman, M. & Hierons, R. M. (2001) An overview of Program Slicing. *Software Focus,* 2**,** 85-92.

Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A. & Roper, M. (2004) Testability transformation. *Software Engineering, IEEE Transactions on,* 30**,** 3-16.

Harman, M., Hu, L., Hierons, R. M., Baresel, A. & Sthamer, H. (2002) Improving Evolutionary Testing By Flag Removal. *Genetic and Evolutionary*

Computation Conference (GECCO '02), pp. 1359-1366. Morgan Kaufmann, 2002.

Harman, M. & Jones, B. F. (2001) Search-based software engineering. *Information and Software Technology,* 43**,** 833-839.

Harman, M. & Mcminn, P. (2009) A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search. *IEEE Transactions on Software Engineering. In press*.

Hierons, R. M. (2004) Testing from a nondeterministic finite state machine using adaptive state counting. *Computers, IEEE Transactions on,* 53**,** 1330-1342.

Hierons, R. M., Bogdanov, K., Bowen, J. P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A. J. H., Vilkomir, S., Woodward, M. R. & Zedan, H. (2009) Using formal specifications to support testing. *ACM Computing Surveys,* 41**,** 1-76.

Hierons, R. M. & Harman, M. (2004) Testing conformance of a deterministic implementation against a non-deterministic stream X-machine. *Theoretical Computer Science,* 323**,** 191-233.

Hierons, R. M., Harman, M. & Fox, C. J. (2005) Branch-Coverage Testability Transformation for Unstructured Programs. *The Computer Journal,* 48**,** 421-436.

Hierons, R. M., Kim, T.-H. & Ural, H. (2004) On the testability of SDL specifications. *Computer Networks,* 44**,** 681-700.

Hierons, R. M., Sadeghipour, S. & Singh, H. (2001) Testing a system specified using Statecharts and Z. *Information and Software Technology,* 43**,** 137-149.

Hogrefe, D. (1991) OSI formal specification case study: the Inres protocol and service. *Technical Report* IAM-91-012. University of Bern, Institute of Computer Science and Applied Methematics.

Holland, J. H. (1975) *Adaptation in natural and artificial systems,* Ann Arbor, The University of Michigan Press.

Holzmann, G. J. (1991) *Design and Validation of Computer Protocols,* New Jersey, Prentice-Hall, Englewood Cliffs.

Itu-T (1994) Recommendation Z.100-Specification and Description Language(SDL). Geneva.

Jaffar, J. & Maher, M. J. (1994) Constraint Logic Programming: A Survey. *Journal of Logic Programming,* 19/20**,** 503-581.

Jolliffe, I. T. (2002) Principal Component Analysis. *Springer Series in Statistics* 2nd ed. New York, Springer.

Jones, B. F., Eyres, D. E. & Sthamer, H. H. (1998) A Strategy for using Genetic Algorithms to Automate Branch and Fault-based Testing. *The Computer Journal,* 41**,** 98-107.

Kalaji, A. S., Hierons, R. M. & Swift, S. (2009a) Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM). *2nd IEEE International Conference on Software Testing, Verification, and Validation (ICST' 09)***,** pp. 230-239. IEEE Press, April 2009.

Kalaji, A. S., Hierons, R. M. & Swift, S. (2009b) A Search-Based Approach for Automatic Test Generation from Extended Finite State Machine (EFSM). *Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC-PART)***,** pp. 131-132. IEEE Press, September 2009.

Kalaji, A. S., Hierons, R. M. & Swift, S. (2009c) A Testability Transformation Approach for State-Based Programs. *Search Based Software Engineering, 2009 1st International Symposium on***,** pp. 85-88. IEEE Press, May 2009.

Kalaji, A. S., Hierons, R. M. & Swift, S. (2010) Generating Feasible Transition Paths for Testing from an Extended Finite State Machine with the Counter Problem. *3rd IEEE International Conference on Software Testing Verification and Validation Workshops (ICSTW' 10)***,** pp. 232-235. IEEE Press, April 2010.

Keum, C., Kang, S., Ko, I.-Y., Baik, J. & Choi, Y.-I. (2006) Generating Test Cases for Web Services Using Extended Finite State Machine. *Testing of Communicating Systems.*

Kim, Y. G., Hong, H. S., Bae, D. H. & Cha, S. D. (1999) Test cases generation from UML state diagrams. *Software, IEE Proceedings -,* 146**,** 187-192.

King, J. C. (1976) Symbolic execution and program testing. *Communications of the ACM,* 19**,** 385-394.

Kirkpatrick, S., Gelatt, J. C. D. & Vecchi, M. P. (1983) Optimization by Simulated Annealing. *Science,* 220**,** 671-680.

Koh, L.-S. & Liu, M. T. (1994) Test path selection based on effective domains. *International Conference on Network Protocols (ICNP '94)*, pp. 64-71. IEEE Press, October 1994.

Kohavi, Z. (1978) *Switching and finite automata theory,* New York, McGraw-Hill.

Korel, B. (1990) Automated software test data generation. *Software Engineering, IEEE Transactions on,* 16**,** 870-879.

Korel, B., Tahat, L. H. & Vaysburg, B. (2002) Model based regression test reduction using dependence analysis. *International Conference on Software Maintenance (ICSM '02)*, pp. 214-223. IEEE Press, October 2002.

Lee, D. & Yannakakis, M. (1994) Testing finite-state machines: state identification and verification. *Computers, IEEE Transactions on,* 43**,** 306-320.

Lee, D. & Yannakakis, M. (1996) Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE,* 84**,** 1090-1123.

Lefticaru, R. & Ipate, F. (2008) Functional Search-based Testing from State Machines. *1st International Conference on Software Testing, Verification, and Validation (ICST '08)*, pp. 525-528. IEEE Press, April 2008.

Lorenzoli, D., Mariani, L. & Pezz, M. (2008) Automatic generation of software behavioral models. *30th international conference on Software engineering*, pp. 501-510. ACM Press, May 2008.

Matlab (1984-2010) The Math Works- Optimization Toolbox: http://www.mathworks.com/access/helpdesk/help/toolbox/optim/ug/fminc on.html.

Mcminn, P. (2004) Search-based software test data generation: a survey. *Software Testing, Verification & Reliability,* 14**,** 105-156.

Mcminn, P. (2005) Evolutionary Search for Test Data in the Presence of State Behaviour. University of Sheffield.

Mcminn, P., Binkley, D. & Harman, M. (2009) Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering and Methodology,* 18**,** 1-27.

Mcminn, P., Harman, M., Binkley, D. & Tonella, P. (2006) The species per path approach to SearchBased test data generation. *International Symposium on Software Testing and Analysis (ISSTA '06)***,** pp. 13-24. ACM Press, July 2006.

Mcminn, P. & Holcombe, M. (2003) The State Problem for Evolutionary Testing. *Genetic and Evolutionary Computation - GECCO 2003.* Berlin, Springer-Verlage.

Mcminn, P. & Holcombe, M. (2005) Evolutionary testing of state-based programs. *Conference on Genetic and Evolutionary Computation (GECCO '05)***,** pp. 1013-1020. ACM Press, June 2005.

Michael, C. C., Mcgraw, G. & Schatz, M. A. (2001) Generating software test data by evolution. *Software Engineering, IEEE Transactions on,* 27**,** 1085-1110.

Myers, G. J. (2004) *The Art of Software Testing,* New Jersey, John Wiley & Sons.

Nilsson, R., Offutt, J. & Mellin, J. (2006) Test Case Generation for Mutation-based Testing of Timeliness. *Electronic Notes in Theoretical Computer Science,* 164**,** 97-114.

Nist (2002) National Institution of Standard and Technology. The economic impacts of inadequate infrastructure for software tesing. May 2002. Palnning Report 02-3.

Omg (2002) Unified Modeling Language (UML), Version 1.4. November 2002.

Petrenko, A., Bochmann, G. V. & Yao, M. (1996) On fault coverage of tests for finite state specifications. *Computer Networks and ISDN Systems,* 29**,** 81-106.

Petrenko, A., Boroday, S. & Groz, R. (2004) Confirming configurations in EFSM testing. *Software Engineering, IEEE Transactions on,* 30**,** 29-42.

Pohlheim, H. (1994-2010) GEATbx - Genetic and Evolutionary Algorithm Toolbox for Matlab.

Ramalingom, T., Thulasiraman, K. & Das, A. (1996) Context independent unique sequences generation for protocol testing. *Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation (INFOCOM '96),* 3**,** pp. 1141-1148. IEEE Press, March 1996.

Ramalingom, T., Thulasiraman, K. & Das, A. (2003) Context independent unique state identification sequences for testing communication protocols modelled as extended finite state machines. *Computer Communications,* 26**,** 1622-1633.

Rapps, S. & Weyuker, E. J. (1985) Selecting Software Test Data Using Data Flow Information. *Software Engineering, IEEE Transactions on,* SE-11**,** 367-375.

Rivest, R. & Schapire, R. (1993) Inference of finite automata using homing sequences. *Machine Learning: From Theory to Applications.* Springer Berlin / Heidelberg.

Sadiq, M. S. & Habib, Y. (1999) *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems,* Los Alamitos, CA, IEEE.

Sarikaya, B. & Bochmann, G. (1984) Synchronization and Specification Issues in Protocol Testing. *Communications, IEEE Transactions on,* 32**,** 389-395.

Sarikaya, B., Bochmann, G. V. & Cerny, E. (1987) A Test Design Methodology for Protocol Testing. *Software Engineering, IEEE Transactions on,* SE-13**,** 518-531.

Shih, C.-H., Huang, J.-D. & Jou, J.-Y. (2005) Stimulus generation for interface protocol verification using the nondeterministic extended finite state machine model. *Tenth Annual IEEE International High-Level Design Validation And Test Workshop***,** pp. 87-93. IEEE Press, November 2005.

Shuhao, L., Ji, W. & Zhi, C. (2004) Property-oriented test generation from UML Statecharts. *Automated Software Engineering, 2004. Proceedings. 19th International Conference on***,** pp. 122-131.

Sidhu, D. P. & Leung, T. K. (1989) Formal methods for protocol testing: a detailed study. *Software Engineering, IEEE Transactions on,* 15**,** 413-426.

Sinha, A., Paradkar, A. & Williams, C. (2007) On Generating EFSM Models from Use Cases. *Sixth International Workshop on Scenarios and State Machines (SCESM '07)***,** pp. 20-26. IEEE Press, May 2007.

Srinivas, M. & Patnaik, L. M. (1994) Genetic algorithms: a survey. *Computer,* 27**,** 17-26.

Tahat, L. H., Vaysburg, B., Korel, B. & Bader, A. J. (2001) Requirement-based automated black-box test generation. *25th Annual International Computer*

*Software and Applications Conference (COMPSAC '01)***,** pp. 489-495. IEEE Press, October 2001.

Tai, K.-C. (1984) A program complexity metric based on data flow information in control graphs. *7th International Conference on Software Engineering (ICSE '84)***,** pp. 239-248. IEEE Press, September 1984.

Tracey, N., Clark, J. & Mander, K. (1998a) Automated program flaw finding using simulated annealing. *ACM SIGSOFT International Symposium on Software Testing and Analysis***,** pp. 73-81. ACM Press, March 1998.

Tracey, N., Clark, J. & Mander, K. (1998b) The Way Forward for Unifying Dynamic Test-Case Generation: The Optimisation-Based Approach. *The IFIP International Workshop on Dependable Computing and Its Applications (DCIA)***,** pp. 169-180. January 1998

Tracey, N., Clark, J., Mander, K. & Mcdermid, J. (1998c) An automated framework for structural test-data generation. *13th IEEE International Conference on Automated Software Engineering***,** pp. 285-288. IEEE Press, October 1998.

Tretmans, J. (2008) Model Based Testing with Labelled Transition Systems. IN HIERONS, R. M., BOWEN, J. P. & HARMAN, M. (Eds.) *Formal Methods and Testing: An Outcome of the FORTEST Network Revised Selected Papers.* Berlin Heidelberg, Springer.

Tsang, E. P. K. (1993) *Foundations of Constraint Satisfaction,* London and San Diego, Academic Press.

Turner, K. J. (1993) *Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL,* West Sussex, John Wiley & Sons.

Ural, H., Saleh, K. & Williams, A. (2000) Test generation based on control and data dependencies within system specifications in SDL. *Computer Communications,* 23**,** 609-627.

Ural, H. & Yang, B. (1991) A test sequence selection method for protocol testing. *Communications, IEEE Transactions on,* 39**,** 514-523.

Wang, C.-J. & Liu, M. T. (1993) Generating test cases for EFSM with given fault models. *Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future (INFOCOM '93),* 2**,** pp. 774-781. IEEE Press, March 1993.

Wegener, J., Baresel, A. & Sthamer, H. (2001) Evolutionary test environment for automatic structural testing. *Information and Software Technology,* 43**,** 841-854.

Weiser, M. (1981) Program slicing. *5th International Conference on Software Engineering (ICSE '81)***,** pp. 439-449. IEEE Press, September 1981.

Wenzel, I., Kirner, R., Rieder, B. & Puschner, P. (2009) Measurement-Based Timing Analysis. *Leveraging Applications of Formal Methods, Verification and Validation.* Berlin & Heidelberg, Springer.

Whitley, D. (1989) The *GENITOR* algorithm and selection pressure: why rank-based allocation of reproductive trials is best. *Third International Conference on Genetic Algorithms***,** pp. 116-121. Morgan Kaufmann, 1989.

Whitley, D. (1999) A free lunch proof for Gray versus binary encodings. *Genetic and Evolutionary Computation Conference (GECCO '99)***,** pp. 726-733. Morgan Kaufmann, July 1999.

Wong, W. E., Restrepo, A. & Choi, B. (2009) Validation of SDL specifications using EFSM-based test generation. *Information and Software Technology,* 51**,** 1505-1519.

Xu, B., Qian, J., Zhang, X., Wu, Z. & Chen, L. (2005) A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes,* 30**,** 1-36.

Zhan, Y. & Clark, J. A. (2006) The state problem for test generation in Simulink. *Genetic and Evolutionary Computation Conference (GECCO '06)***,** pp. 1941-1948. ACM Press, July 2006.

Zhang, J. (2008) Constraint Solving and Symbolic Execution. *Verified Software: Theories, Tools, Experiments.* Berlin / Heidelberg, Springer.

Zhang, J., Xu, C. & Wang, X. (2004) Path-Oriented Test Data Generation Using Symbolic Execution and Constraint Solving Techniques. *Second International Conference on Software Engineering and Formal Methods (SEFM '04)***,** pp. 242-250. IEEE Press, September 2004.

Zhao, R., Harman, M. & Li, Z. (2010) Empirical Study on the Efficiency of Search Based Test Generation for EFSM Models. *3rd IEEE International Conference on Software Testing Verification and Validation Workshops (ICSTW' 10)***,** pp. 222-231. IEEE Press, April 2010.