Test Generation from an Extended Finite State Machine as a Multiobjective Optimization Problem

by

Seyedeh Nesa Asoudeh Khalajani

A dissertation submitted to the Faculty of Graduate and
Postdoctoral Affairs in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

The Ottawa-Carleton Institute For

Electrical and Computer Engineering (OCIECE)

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario

© 2016, Seyedeh Nesa Asoudeh Khalajani

Abstract

Extended Finite State Machines are widely used in different phases of software development including software testing. In this Ph.D. dissertation, we argue that test generation from an Extended Finite State Machine (EFSM) can be considered as a multi-objective optimization problem. When a test engineer generates tests from an EFSM he/she typically considers several objectives. We propose a search-based approach to generate test suites from an EFSM, accounting for multiple (potentially conflicting) such objectives. We aim at maximizing coverage of the EFSM test model and maximizing feasibility of the generated test suite so that its test cases can actually execute, while minimizing similarity between these test cases since this has been shown to increase fault detection, as well as minimizing overall cost. Therefore, we have defined a multi-objective genetic algorithm that searches for optimal test suites based on four fitness functions. In doing so, we create an entire test suite at once as opposed to creating a test suite one test case one at a time, which we argue is a suboptimal test suite generation procedure. Our approach is evaluated on different case studies, showing interesting results. We also investigate different ways of improving our solution and analyze impact of those improvements.

Acknowledgements

First and foremost, I would like to express my deep and sincere gratitude to my supervisor, Dr. Yvan Labiche for his earnest and devoted support, encouragement, continuous feedback and lots of good ideas throughout this work. Accomplishing this work would have been impossible without his supervision and assistance.

I would also like to thank my friends and colleagues in SQUAL Lab who were great companions during this journey.

Furthermore, I would like to thank Dr. Hadi Hemmati, Dr. Guy-Vincent Jourdan, Dr. Olga Baysal and Dr. Dorina Petriu, the thesis examination board, for revising this work and providing useful help and feedbacks.

Last but not least, I would like to thank my family: My Parents for their unconditional love, motivation and support during these years. My sister Nasim, and my brother-in-law Farzad who were always there for me. I could not have achieved this without their unlimited help and encouragements.

This work is dedicated to my selfless mother

Nahid Vahedi

And my supportive father

Mohammad Asoudeh

Table of Contents

Abstract	i
Acknowledgements	ii
Table of Contents	iv
List of Tables	viii
List of Figures	ix
List of Acronyms	xi
1 Introduction.....	1
2 Background and related work	7
2.1 Background on state machines.....	7
2.1.1 Finite State Machines.....	8
2.1.2 Extended Finite State Machines	13
2.2 Background on Genetic Algorithms.....	15
2.3 Related works.....	18
2.3.1 Principles and methods for testing from (extended) finite state machines	18
2.3.2 Achieving scalable model-based testing through test case diversity	20
2.3.3 Generating feasible test paths from extended finite state machines using genetic algorithms	22
2.3.4 Multi-Objective Search Based Testing	23
2.3.5 SBST and structural testing	27
2.3.6 SBST—state of the art	28
2.3.7 Differences with our work	29
2.4 Summary	30
3 Test generation from an EFSM.....	32
3.1 Chromosomes and genes.....	32
3.2 Mutation Operator	35
3.3 Crossover operator	38

3.4	Fitness functions.....	39
3.4.1	Feasibility	39
3.4.2	Similarity	41
3.4.3	Coverage	42
3.4.4	Cost.....	42
3.5	Genetic algorithm parameters	42
3.6	Multiple-objective GA implementation	43
3.7	Empirical Evaluation.....	44
3.7.1	Case study systems	45
3.7.2	Experimental design	47
3.7.2.1	Answering RQ 1—How do whole and incremental test suite constructions compare?	47
3.7.2.2	Answering RQ 2—Are test suites feasible?	48
3.7.2.3	Answering RQ 3—How does our GA compare to a random search?.....	48
3.7.2.4	Answering RQ 4—How does our GA compare to other test suite construction techniques?	50
3.7.3	Stochastic aspect of our GA.....	51
3.7.4	Qualitative analysis of some results.....	51
3.7.5	Results for RQ 1—How do whole and incremental test suite constructions compare?.....	54
3.7.6	Results for RQ 2—Are test suites feasible?.....	56
3.7.7	Results for RQ 3—How does our GA compare to a random search?.....	57
3.7.8	Results for RQ 4—How does our GA compare to other test suite construction techniques?.....	58
3.8	Threats to validity	59
3.9	Summary	61
4	Measuring similarity in a test suite	62
4.1	Applying existing measures to our Multi-Objective GA	62
4.2	Methods from life sciences	64
4.3	Comparing different measures using statistical analysis.....	68

4.3.1	Research questions.....	69
4.3.2	Statistical Procedure	69
4.3.3	Analysis of Variance (ANOVA).....	70
4.3.4	Using ANOVA in our experiments.....	72
4.3.5	Results.....	73
4.3.5.1	Effect on cost.....	74
4.3.5.2	Effect on mutation score	76
4.3.5.3	Qualitative analysis of results	77
4.3.6	Threats to validity	79
4.4	Summary	80
5	Counter Problem.....	82
5.1	A Taxonomy of counter problems	83
5.2	Real examples matching synthetic ones.....	87
5.3	The counter problem and test generation from an EFSM	89
5.3.1	Extending our GA using Kalaji et al.'s solution.....	89
5.3.2	Extending the GA with a new solution.....	91
5.3.3	Genetic Algorithm parameters.....	92
5.4	Empirical evaluation	93
5.4.1	Counter problem level1	94
5.4.2	Counter problem level2	94
5.4.3	Coutner problem level3	96
5.5	Summary	97
6	Conclusion and future work.....	98
6.1	Conclusion	98
6.2	Future work.....	103
	References.....	106
	APPENDICES	116
A.	Actions and guard conditions of Qui-Donc EFSM.....	116

B.	Actions and guard conditions of Class II protocol state machine	117
C.	Guard conditions of ordered set case study	118
D.	Test suites used in chapter 3	119
E.	Results of statistical analysis for section 4.3	124
F.	Examples of EFSMs having different types of counter problem.....	130

List of Tables

Table 1- Matrix used to implement our encoding of transition sequences	35
Table 2- Tabular representation of state machines used in our encoding.....	35
Table 3- Answering research questions with case studies	44
Table 4- Properties of state machine used in case studies	45
Table 5- Selected test suites from the Pareto front for the Transport Protocol case study	56
Table 6- Results of mutation analysis (Cruise Control).....	59
Table 7- Results of mutation analysis (Transport Protocol)	59
Table 8-Sample ANOVA table	71
Table 9- Mean and variance of cost values.....	73
Table 10 – Mean and variance of mutation scores	76

List of Figures

Figure 1 -FSM of the Qui-Donc system [1].....	11
Figure 2- EFSM of the Qui-Donc System [1].....	15
Figure 3- Chromosome encoding.....	33
Figure 4- A simple state chart used to present our encoding of transition sequences	35
Figure 5 – Exchanging transitions between test paths in the same test suite.....	38
Figure 6-Chromosome mutation: crossover between genes (test cases/paths) of the same chromosome	39
Figure 7-Exchanging genes between two chromosomes	39
Figure 8- EFSM for the Cruise Control system	46
Figure 9- EFSM of a class II transport protocol	47
Figure 10- Sample Pareto front contents for Cruise Control	52
Figure 11- Cruise Control: Pareto front when only optimizing for cost and coverage.....	53
Figure 12- Number of adequate not absolutely infeasible test paths	58
Figure 13 – Box plot comparing costs	74
Figure 14- Box plot comparing mutation scores.....	76
Figure 15-Relation between cost of test suites and method used to compute similarity ..	78
Figure 16 – Level1 of the counter problem (synthetic example).....	85
Figure 17 - Synthetic EFSM with a level2 counter variable.....	86
Figure 18 - Synthetic EFSM with multiple counters (level3).....	87
Figure 19 - EFSM of a simple stack	88
Figure 20 – EFSM of a bounded data structure [64]	88

List of Acronyms

ADW	Average Distance Within population
ANOVA	Analysis Of Variance
EA	Evolutionary Algorithm
EFSM	Extended Finite State Machine
FDR	Fault Detection Rate
FSM	Finite State Machine
GA	Genetic Algorithm
KW	Kosman's diversity Within population
LCM	Least Common Multiplier
MBT	Model Based Testing
MOGA	Multi-Objective Genetic Algorithm
RQ	Research Question
SBSE	Search Based Software Engineering
SBST	Search Based Software Testing
SUT	System Under Test

1 Introduction

Model-based testing is an approach to automate black box testing [1]. Applications of model-based testing (MBT) to automate test generation and execution have increased very rapidly. According to Legard [2] there are many reasons that explain why MBT turned into a widely used test automation technique:

- Software complexity is increasing continuously and user's tolerance of defects is much less than before. This requires functional testing to become more effective at detecting bugs.
- A large proportion of cost and time of software development is spent on software testing. This is a great motivation to invest in the development of methods like MBT, which can automate test design and execution.
- MBT and supporting tools (commercial and open source) have become mature enough to be used in a wide range of application domains. There is empirical evidence that MBT can result in good return on investment [2].

A model to be used in model based testing should be as concise and as precise as possible. A concise model can accelerate the modeling and validation but the model should also be precise enough to describe the System Under Test (SUT) behavior that is to be tested. One of the expected benefits of MBT is increasing quality of functional requirements. Undefined or ambiguous behavior in the specification can be detected during modeling for test generation [2]. Also automatic test generation can help in systematic coverage of functional behavior.

Transition based models like state machines are widely used by software engineers. Although finite-state machines (FSMs) can be used to describe the control structures of small systems, they become rapidly too large and unmanageable as the size of the system increases [3]. Extended finite state machines (EFSMs) reduce the size of the model using data variables which represent some aspects of a state. Two other important notions in an EFSM are actions and guard conditions. There are different types of actions on transitions and states (performing I/O, triggering an event, etc) but in this

thesis by actions we mean those operations that modify data variables. Guard conditions are Boolean expressions that should evaluate to True in order to enable a transition. Again in this thesis by guard conditions we mean the ones that depend on data variables. Testing a software implementation (IUT, i.e., implementation under test) against its specification in the form of an EFSM, and protocol testing are examples of these applications of EFSMs in model-based testing. It can also be of great help in the challenging process of software integration [4] in cases where black box components need to have specific state based behaviour to be integrated with the whole system. The latter application is becoming more and more important in the era of cloud computing where concepts like Software as a Service or Business Process as a Service are quite common [5]. Those services need to be verified before being integrated in a specific software application. As the number of such off the shelf components is increasing possibly uncontrollably an automated verification process against integration goals abstracted in an EFSM can facilitate and accelerate elicitation, verification and integration of the right component.

In this dissertation we propose, implement and experiment with a new idea in the field of model (EFSM) based testing. Our thesis is that state-based testing from an EFSM is a multi-objective optimization problem. Indeed, when a test engineer manually generates tests from an EFSM he/she considers several objectives. Avoiding infeasible paths (first objective) caused by conflicts between actions and guard conditions is one of those objectives: one wants test cases that can actually be executed. Generating feasible test paths from an EFSM is similar to the path sensitization problem, which is to find test inputs to execute a path in a program (white box testing), and is known to be undecidable [6]. Since exhaustively testing every possible behaviour specified in the EFSM is typically practically unfeasible, because of too many such behaviour, the test engineer is looking for a mechanism to select a relatively small subset of the entire set of possible behaviour which is, in some sense, representative of the entire set. To that end, the test engineer relies on a selection criterion (or several criteria at once) to decide which elements of the EFSM test model need to be exercised and to decide when to stop testing. Typically one decides to exercise all the transitions of the EFSM, although other selection criteria exist [6]: transition coverage has proved to be a good minimum to aim at in test

generation from an EFSM [1]. A second optimization objective is therefore to ensure adequacy, i.e., that a test suite satisfies a selection criterion, i.e., that the test suite exercises the elements prescribed by the criterion (or criteria). Since a large number of adequate test suites can typically be devised for a selection criterion, the test engineer is also looking for additional guidance for selecting test cases. Two such additional objectives are cost and diversity. Intuitively, increasing diversity among the test cases of a test suite (e.g., avoiding running the same path/subpath several times) should increase the fault detection effectiveness of the test suite. (Note that this has also been experimentally confirmed [3]). Considering diversity without accounting for cost may result in very expensive test suites because the technique may then traverse very long paths to avoid having similar test paths, assuming the length of a test case is a surrogate measure of cost. Therefore, cost, coverage, feasibility and similarity are four objectives that we believe should be considered together during test generation from an EFSM.

There might be other objectives of interest to a test engineer. We however believe that the four above-mentioned objectives are the most important ones.

It is also worth noting that these objectives are competing with one another: increasing diversity between test cases, i.e., varying the ways the EFSM graph is traversed, may increase cost; ensuring feasibility may require taking detours in the EFSM graph instead of shortest paths, thereby increasing cost. As a result, accounting for these objectives when creating a test suite will require identifying tradeoffs.

We also argue that these objectives must be considered together when creating an entire test suite (made of test cases) at once, rather than considered together when creating test cases one at a time, one after the other. The latter construction approach is a greedy way of constructing a test suite that satisfies the objectives and may therefore not be optimal with respect to one or more of these objectives. For instance, assuming a number of test cases have already been created and the test suite they compose is not adequate for a selection criterion, one needs to create at least one new test case. The current (inadequate) test suite is only a local optimum with respect to cost and diversity. A newly created test case, necessary to get closer to adequacy, may end up being very similar to a test case that already exists in the test suite. The one-by-one test case creation

procedure does not challenge decisions already made when constructing the current test suite. The one-by-one test case creation procedure is a greedy heuristic that makes locally optimal choices.

We are instead looking for a solution that would produce a more optimal solution. We are therefore turning our attention to meta-heuristic search algorithms. As there is no single unique (optimum/tradeoff) test suite as a solution to our multi-objective optimization problem, single state algorithms like simulated annealing that work out a single solution at a time do not seem to be a good choice. Also, since exchanging test paths between different test suites can result in better test suites in terms of feasibility and similarity, methods like evolutionary algorithms, which do not allow exchanging data between different solutions are not a good choice either. Therefore, this thesis is "a multi-objective genetic algorithm to generate test suites from an EFSM".

Search-based software engineering (SBSE) [7] has emerged in the field of software engineering since the nature of many software engineering problems makes them perfect for the application of meta-heuristic search techniques [8]. Search-based methods can be very helpful in solving problems in which there are tradeoffs between different constraints. They can provide solutions in cases where finding an optimal solution is either theoretically impossible or practically infeasible. One of the first applications of search-based software engineering has been in software testing and dates back to 1976 [9]. In Search-Based Software Testing (SBST) a meta-heuristic search method such as a Genetic Algorithm (GA) is used to automate or partially automate a software testing task [10].

This Ph.D. dissertation is a novel approach to model (EFSM) based testing that applies search-based software testing. A multi-objective genetic algorithm is proposed and implemented to generate test suites from an EFSM. Four objective (fitness) functions are used to search for a test suite that is most likely feasible, minimizes cost, minimizes similarity between its test paths, and is adequate (or as adequate as possible) for a test selection criterion. Feasibility pertains to the capability to find inputs such that generated test cases can actually execute; we aim at increasing the chances of feasibility without actually identifying for sure whether test cases are feasible during their construction. Test

suite cost needs to be minimized for obvious reasons. We want to minimize the similarity between test cases of the same test suite since this notion has been related to effectiveness at finding faults [3]: the less similar test cases the more they find faults. As always in software testing, we aim at producing adequate test suites for known adequacy criteria (we use the all-transitions criterion).

To the best of our knowledge, this is the first attempt to use those four objectives together in the context of state-based testing (SBT). Another important contribution of this work is that we attempt to build an entire test suite at once as opposed to constructing test cases one at a time until adequacy is reached, as has been done in the past for SBST. We argue that the latter is a kind of greedy search for an adequate test suite and therefore can only lead to a sub-optimal solution to the problem of adequate test suite construction.

We implemented our solution as a multi-objective GA using the MATLAB global optimization toolbox [11]. We used empirical evaluations to verify if we have been successful in achieving optimization objectives mentioned above. We show that generating an entire test suite rather than doing so incrementally results in improvements in terms of cost and dissimilarity, without having negative effect on (expected) feasibility and adequacy; Test suites generated by our multi-objective GA were indeed feasible. Also by using diversity as an objective function we find test suites that are less expensive, without sacrificing effectiveness at detecting faults. We compared test suites generated by our GA with randomly generated test suites; this showed the search space is so large or complex that a random search is not adequate and we have to resort to a GA. We also investigated different ways of improving our multi-objective GA. We tried different methods of computing similarity within a test suite and compared different flavors of our GA. We also investigated the effect of the counter problem on our approach and proposed and implemented improvements over existing solutions.

To summarize, the main contributions of this thesis are:

- Defining test generation from an EFSM as a multi-objective optimization problem and putting diversity, feasibility, coverage and cost as four objectives together for the first time;

- Proposing and implementing a solution to the problem above as a multi-objective genetic algorithm and evaluating the solution using different case studies;
- Generating whole test suites rather than individual test cases;
- Implementing and comparing different methods of computing diversity in a test suite including two methods from life sciences which to the best of our knowledge haven't been used in the area of model based testing before;
- Investigating the effect of the counter problem, which is a well know issue in the area of EFSM testing, on our multi-objective optimization approach.

This Ph.D. dissertation is structured as follows, after discussing background and related work in chapter 2, we describe our approach to test generation from an EFSM in chapter 3. In chapter 4 we explain how we extended our GA with different similarity measures. Chapter 5 discusses the counter problem. We conclude this dissertation and propose different routes for future work in chapter 6.

2 Background and related work

In this chapter we discuss basic concepts and related works, which will be used in subsequent chapters where we explain our approach. We start by a brief introduction to state based testing (section 2.1) and genetic algorithms (section 2.2). We discuss the related works in section 2.3.

2.1 Background on state machines

Model based testing is an approach that aims at increasing the quality of a test suite during black box testing [1]. Test generation is always based on a model of the expected behavior of a system under test. In classical manual testing this model is usually just an informal model in the test engineer's mind. In model based testing this model is made explicit. Therefore, test objectives are clearly captured in a test model, which results in the generation of a precise test suite by generating it from an explicit test model rather than (only) what a test engineer may have in mind. This can also facilitate test automation, which reduces the cost of testing, and results in test suites that are more effective at detecting faults by systematically covering test models. This is due to the fact that automation facilitates generation of a test suite that addresses all the desired behaviour captured in the test model and therefore results in better detection of faults.

Models used in model based testing can be created by a test engineer for testing purposes or can come from other development phases. These two approaches have some advantages and disadvantages. Reusing models created during (forward) development purposes can save time: the test model already exists. However this can have a negative effect on testing effectiveness as the essence of testing is finding differences between the behavior of a system under test and the behavior expected by tests. The lack of independence between the test model and a model used to develop the SUT can result in undetected faults. Therefore developing a separate model for testing purposes is very common [1], but comes with additional cost.

In either of these two approaches, state machines are widely used [1], [12]. A state machine is an ideal notation to use in modeling the expected SUT behavior. In this

section we explain how state machines are used in model based testing, which is commonly known as state-based testing.

2.1.1 Finite State Machines

Using a Finite State Machine (FSM) is the simplest approach to model state-based behavior. Formally, an FSM can be defined as a tuple: $M = (I, O, S, s_i, S_f, \delta, \lambda)$ where, I , O , and S are finite and nonempty sets of input symbols, output symbols, and states, respectively, s_i is the initial state (s_i belongs to S), S_f is a set of one or more final states (S includes S_f), and δ and λ are two functions defined as follows:

$\delta : S \times I \rightarrow S$ is the state transition function. If $\delta(s_i, I) = s_j$, s_i is called the source of the transition, s_j is called the target of the transition, and I is called the trigger of the transition;

$\lambda : S \times I \rightarrow O$ is the output function.

An FSM can be represented graphically as a graph where each node in the FSM graph corresponds to a distinct state of S and the arcs of the graph represent the FSM transitions as defined by δ and λ : there is an arc (transition) from node n_1 , representing state s_1 (in S), and node n_2 , representing state s_2 (in S), labeled with input symbol i_s (in I) and output symbol o_s (in O) if and only if $\delta(s_1, i_s) = s_2$ and $\lambda(s_1, i_s) = o_s$.

A test case derived from an FSM is typically a sequence of transitions generated by traversing the FSM graph. In the rest of this document, we use the term FSM to refer to either the formal, tuple representation or the equivalent graph representation.

In the rest of this subsection we will use the Qui-Donc (Who then?) example from [1] to illustrate how an FSM can be used as a test model. Qui-Donc is a phone service provided by France Telecom that allows clients to enter a telephone number and find the corresponding name and address. After a user dials Qui-Donc phone number, the system responds with a *WELCOME* message and asks the user to press the star key to continue. The welcome message is repeated up to three times with a time out of six seconds between them and if the star key is not pressed, the system plays the *NOTALLOW* message. If the system detects the star key it gives the *ENTER* message. This message is

also repeated three times with a longer time out of 20 seconds. If the system does not detect any number or hash or pound (#) key it terminates the call with the *BYE* message. If the system detects some digits followed by the pound key and the number is one of the registered emergency numbers in France (15 for ambulance, 17 for police and 18 for fire brigade) the explanation of that emergency number is played followed by a message that asks the user to press the star key to perform another search. Otherwise, if the entered number does not contain ten digits, the *ERROR* message is played. If the number entered by the user has ten digits, the system looks for it in the white pages. If the number is not in the white pages the *SORRY* message is played and the system resends the *ENTER* message. If the number exists in the white pages, the *NAME* message is given and the system enters the information menu where the *INFO* message is played. In the information menu if the user selects number 1, then the system spells out the name and returns to the information menu. If number 2 is selected, the system gives the address and returns to information menu. If the star key is pressed the system goes back to the number-entry procedure with the *ENTER* message. If no key is pressed the *INFO* message is repeated three times with a time out of six seconds and then the call is terminated with the *BYE* message. In addition to all of these scenarios the user can hang up at any moment.

Let us now create a test model from these requirements. We will start by modeling the system with an FSM. The first step is to abstract out non-relevant details to keep the FSM small. Specifically, there are 10^{10} possible 10-digit numbers (considering only valid inputs of the FSM) and we cannot afford testing all of them. We plan to find logical faults such as playing the wrong message or entering an incorrect menu. For testing purposes we only consider the four keys that Qui-Donc considers as special inputs (1, 2, * and #) as well as four different telephone numbers [1]:

- numEmerg (18): We selected the fire brigade number as the representative of emergency numbers. We assume that if the system works correctly with one of the emergency numbers it will work correctly with all of them.
- Num1 (03 81 11 11 11): This number represents numbers that do not exist in the white pages.

- Num2 (03 81 22 22 22): This number represents numbers that are in white pages and have a corresponding name and address.
- Bad (12 34 56 78 9): This number has nine digits and represents numbers with the wrong format. We assume the system will have the same behavior for all numbers that do not have exactly 10 digits.

Different assumptions, leading to a different identification of such numbers could obviously be made. We consider these are sufficient for this section of the thesis. They are also a good illustration of the abstracting activity that is required when one creates a state model for testing purposes, as opposed to reusing a state model that was originally created for forward development purposes.

These abstractions result in the FSM presented in Figure 1. Each transition in the figure is labeled with a single *input/output* pair, with the following syntax: *input / output*. The system is initially in the **Start** state, which is denoted by a double lined oval in the figure. As described earlier, the Qui-Donc system starts working, after a user has dialed Qui-Donc phone number, by playing the welcome message. This triggers transition from state **Start** to state **Star1**. In the **Star1** state, the system is waiting for the user to press the star (*) key. If the star key is pressed the system moves to the **Enter1** state; otherwise, it waits for a specified amount of time (six seconds) and moves to the **Star2** state after replaying the Welcome message: the input symbol of the transition is *wait* and the output symbol of the transition is *WELCOME*. The waiting step is repeated at most two additional times, which causes the system to move to state **Star3** and then state **Start**. After the third timeout period, the NotAllow message is played and the call is terminated: the system goes back to the **Start** state. At any time (before the last time out period has elapsed) if the user presses the star key the system moves to the **Enter1** state. In this state, the system waits for at most three periods of timeouts and moves to the **Enter2**, **Enter3** and **Start** states respectively. While in state **Enter1**, **Enter2** or **Enter3**, the system is waiting for the user to dial some digits followed by the pound (#) key. If the user dials two digits followed by the pound key (represented by numEmerg in the current test model) the transition to the **Emerg** state is triggered and the system plays the FIRE message. If the number dialed by the user has a wrong format (i.e., it belongs to the *bad*

category of inputs), the ERROR message is played and the system stays in (or moves to) state **Enter1**. If the number has a correct format but is not in the white pages (i.e., it belongs to the *num1* category of inputs), the SORRY message is displayed and the system stays in (or moves to) state **Enter1**. If the number has a correct format and it exists in the white pages (i.e., it belongs to the *num2* category of inputs), the system returns the corresponding contact with the NAME message and goes to the **Info1** state. In this state the system plays the info menu and waits for the user to dial either one or two.

Figure 1 -FSM of the Qui-Donc system [1]

In the information state if the user selects number 1, then the system SPELLs out the name and returns to the **Info1** state. If number 2 is selected, it gives the address with the ADDR message and also returns to **Info1**. If the star key is pressed the system goes back to the **Enter1** state with the *ENTER* message. If no key is pressed the *INFO* message is repeated three times with a time out of six seconds and the system transitions to the **Info2**, **Info3** and **Start** states respectively. After the third timeout period the BYE message is played and the call is terminated: the system goes back to the **Start** state.

This kind of FSM is called a mealy machine [13]. A mealy state machine has a finite set of states connected with a set of transitions; Each transition must have exactly one input and one output (the output can be null); The FSM should have exactly one initial state and may optionally have one or more final states. When a mealy machine is used as a test model each transition represents one step in a test case. The *input* represents the stimulus sent to the SUT and the *output* represents the expected response [1].

Having an FSM as test model in hand, test cases can be generated by simply traversing the graph representing the FSM. Different selection criteria can be used in state based testing to decide which elements of the FSM test model need to be exercised and identify when to stop testing. The simplest criteria are all-states (i.e., exercising each state in the FSM at least once), all-inputs (i.e., exercising each input symbol at least once) or all-outputs (i.e., exercising each output symbol at least once). These three criteria are easy to satisfy and in general do not provide enough confidence in whether the code correctly implements the FSM behaviour: e.g., satisfying any of these three criteria does not necessarily ensure that all the transitions, and therefore important behaviour, are exercised. Another criterion that is known to be a good minimum [1] to consider in test generation from an FSM is all-transitions: i.e., each transition of the FSM needs to be exercised at least once. An adequate test suite for the all-transitions criterion, i.e., a test suite that exercises each transition at least once, can be constructed by using well-known graph traversal algorithms such as the ones solving the Chinese postman problem.

2.1.2 Extended Finite State Machines

The FSM in Figure 1 may be considered too complex for such a small system, even after all the abstractions mentioned previously. The main reason for this complexity is that three levels of timeout cause a lot of duplications and make the FSM graph cluttered. This complexity can be somewhat reduced by using an Extended Finite State Machine (EFSM) in which a context variable is used to count the number of timeouts.

An Extended Finite State Machine (EFSM) facilitates modeling more complex state-based behaviour than an FSM because it allows abstracting a number of concrete states (each corresponding to one value of a state variable) into one abstract state (that corresponds to a set of state variable values). Like Petrenko et al [14] we define an EFSM as a Mealy FSM extended with input and output parameters, context variables, actions and guard conditions defined over context variables and input parameters. An EFSM M can be defined over the sets X , Y , R , V , and the associated valuation domains where X and Y are sets of input and output symbols, and R and V are disjoint sets of parameters and context variables respectively. M is represented by the pair (S, T) , where S is a finite set of states and T is a finite set of transitions between states in S and each transition t in T is a tuple (s, x, G, A, y, s') where:

- s and s' , states in S , are the initial and final states of the transition, respectively;
- x in X is the input symbol of the transition;
- y in Y is the output symbol of the transition;
- G is the guard condition of the transition;
- A is the action of the transition.

The semantics of a transition (s, x, G, A, y, s') is that, assuming the current state is s and input symbol x is received, the transition fires only if guard condition G is true; when the transition fires, the output symbol is generated, action A is executed and the new state is s' . If guard condition G is false then the input symbol x is dropped (ignored).

An EFSM can be considered as a compressed format of an FSM. If all the parameters and context variables have finite domains it is possible to unfold an EFSM into an FSM. In this case test generation methods based on an FSM can be used. However, in most practical situations unfolding can result in state space explosion [14]. This is due to the fact that the size of the state space grows exponentially with the number of possible values a context variable can take and that a EFSM can have more than one state variable.

An EFSM typically has a smaller number of states than an equivalent FSM. This is due to the fact that an EFSM introduces a level of abstraction using context variables and parameters [1]. This results in a state machine with a fewer number of (visible) states, which is easier to grasp. However, the addition of guard conditions and actions on transitions can make the EFSM more difficult to understand. Also, this can make a traversal of the graph representing the EFSM infeasible, i.e., a sequence of transitions that cannot execute because one cannot find inputs to execute it (because of some guard conditions being false).

Choosing between an FSM and an EFSM as test model is a decision to be made by the test engineer, based on the test objectives as well as size and complexity of the problem at hand.

Figure 2 presents how the Qui-Donc system can be modeled as an EFSM. Similarly to Figure 1, the system is initially in the **Start** state, which is denoted by a double lined oval in the figure. By introducing an internal variable which represents the number of elapsed time outs during different phases of the interaction with the user (waiting for the star key, waiting for the user to dial the phone number, waiting for the user to select an option from the info menu), the corresponding three states of the FSM in Figure 1 (e.g., **Star1**, **Star2**, **Star3**) are merged into a single state of the EFSM in Figure 2 (e.g., **Star**). The addition of a variable counting the number of time outs results in the addition of guard conditions and actions. A complete code listing of those actions and guard conditions is available in appendix A. The number of EFSM states (five) and transitions (17) is smaller than in the corresponding FSM (eleven states and 37 transitions).

Figure 2- EFSM of the Qui-Donc System [1]

2.2 Background on Genetic Algorithms

A genetic algorithm (GA) is a search algorithm based on the notion of natural selection and genetics. A GA supports the survival of the fittest principle. A GA has proved to be a valid approach to problems requiring efficient and effective search [15]. They are computationally simple while successful in searching for improvement in the solution space.

The first step in a genetic algorithm is coding the parameters of the optimization problem as a finite-length string. This encoded string is called a chromosome and each character of the chromosome, is called a gene. Note that the structure of a gene can be something else than a character: it can be as simple as an integer value, or as complex as a structure (in the sense of the C programming language). Other steps after the encoding are:

1. Generating an initial population: This is a set of randomly generated chromosomes and usually has a predefined size.

2. Evaluation: Each chromosome is evaluated based on a fitness function, which quantifies how good each solution is. In the rest of the thesis we will use the terms fitness function and objective function interchangeably.
3. Selection: Parents for the next generation are selected from the current solutions. Chromosomes with a better fitness value have higher chance of being selected.
4. Reproduction: Selected parents exchange their genetic material (genes) to produce offspring. Reproduction helps propagate good characteristics from one generation to another.
5. Mutation: A randomly selected gene of a chromosome is altered. Mutation helps avoid local optima. Typically a rate of 0.01 to 0.05 is used for mutation [15].
6. Repeat: Steps 2 through 5 are repeated until the algorithm finds an optimum or the best solution does not improve in a number of consecutive generations.

Genetic algorithms can also be used in multi-objective optimizations. It is possible to customize traditional GAs and use them in multi-objective optimization by defining specialized fitness functions and methods that promote solution diversity [16].

Konak et al. have defined three different categories of fitness functions that are used by multi-objective GAs [16] :

- Weighted sum approaches: In this approach a multi-objective problem is converted into a single objective one through a weighted sum of the fitness functions. The biggest challenge of this method is selecting appropriate weights for different objective functions.
- Altering objective functions: The oldest method that belongs to this category is vector evaluated GA (VEGA) [17]. To solve a multi-objective problem with N objective functions, VEGA randomly divides the population into N equal sized sub-populations ($P_1, P_2, \dots, P_i, \dots, P_N$). Each chromosome in sub-population P_i is evaluated based on fitness function F_i . After proportional selection (based on

fitness values) from each sub-population, sub-populations are merged and crossover and mutation operators are applied to the combined population.

- Pareto-ranking approaches: In Pareto-ranking, the whole population is ranked based on the Pareto-dominance rule. The fitness of each chromosome is determined based on its rank in the population.

Pareto-ranking approaches have been the privileged approaches for multi-objective GAs for a number of years because they do not have the drawbacks of the two other kinds of approaches (e.g., finding weights, separate single optimizations); We will therefore follow this trend.

In a Pareto Ranking multi-objective GA, it is important to keep the population as diverse as possible so that solutions are uniformly distributed over the Pareto front. Several approaches to maintain the diversity of the population in multi-objective GA have been proposed. One of them is fitness sharing [18] in which the fitness of chromosomes belonging to crowded areas of the population is reduced artificially. Another category is cell-based density [19]. In a multi-objective problem with N objective functions the phenotype space is split into N dimensional cells. The number of solutions in each cell is used as a fitness measure, which is used in another GA in which each cell is treated as a chromosome during the selection process. A cell with fewer individuals has a higher chance of being selected than a crowded one. When a specific cell is selected, solutions in that cell are randomly selected for the mutation and crossover operations.

Our optimization is multi-objective [20], using a Pareto ranking approach. We have four different, possibly conflicting objectives, whereby a solution is characterized by a vector $\vec{u} = (u_1, \dots, u_k)$ of fitness function values (in our case, $k=4$). A vector (solution) is said to dominate another vector $\vec{v} = (v_1, \dots, v_k)$ (denoted by $\vec{u} \prec \vec{v}$) if and only if u is partially less than v : $\forall i \in \{1, \dots, k\}, u_i \leq v_i \wedge \exists i \in \{1, \dots, k\} : u_i < v_i$.

The objectives being optimized will often conflict, which places a partial ordering on the search space. GAs are well suited to this task [21] as they rely on a population of

solutions: Individuals in the population can represent solutions that are close to an optimum and represent different tradeoffs among the various objectives.

For a given multi-objective problem, the Pareto (optimal) front refers to optimal solutions whose corresponding vectors are non-dominated by any other solution vector. Because a GA works on a population of individual solutions it is possible to find many solutions in the Pareto optimal set, which thus present the many possible tradeoffs between the various objectives. The final decision as to which solution to finally select from the Pareto front is left with a decision maker, rather than the optimization algorithm.

2.3 Related works

The related works section first discusses the general problem of state-based testing (section 2.3.1). We then discuss two important pieces of work which we reuse in our solution: a study of test cases similarity in the context of state-based testing (section 2.3.2); an attempt to create feasible traversals of a state machine graph without computing actual values that can execute these paths (section 2.3.3). Then we discuss related work on search-based software testing in sections 2.3.4 to 2.3.6. Last (section 2.3.7) we outline the main differences between our work and the literature in the domain.

2.3.1 Principles and methods for testing from (extended) finite state machines

Lee and Yannakakis [13] have performed a survey on different testing techniques based on mealy state machines. Two types of testing problems have been discussed in the paper. The first one is the state identification problem in which the transition diagram of a Finite State Machine is given but it is not known in which state the implementation is. The second type is conformance testing. In conformance testing a specification finite state machine and a black box implementation are given. The goal is verifying whether the implementation conforms to the specification. The solution to this problem is a test sequence which is called checking sequence.

Their survey also addresses testing of Extended Finite State Machines. As the general problem of testing an EFSM is not decidable they have focused on deterministic

EFSMs, which can be reduced into an equivalent FSM through a process called flattening. However, flattening an EFSM to an FSM can result in a large number of states and in some cases to state explosion. The authors propose a procedure for minimization and reachability analysis of EFSMs.

The testing strategies put into place when testing from a state machine, or a set of communicating state machines, heavily depends on the kind of behavior specification the state model contains. When the state machine does not have actions (or activities) on transitions or states, or guard conditions, then many structural approaches exist to automatically generate feasible test cases (any traversal of the graph representing the state machine is feasible), such as the W-method, the Wp-method or the UIO-method [22]. A set of communicating state machines can under some conditions (so as to avoid a state space explosion) be transformed into a larger extended finite state machine from which the abovementioned techniques can be used [23]. Alternatively, techniques specific to communicating state machines exist (e.g., [24]). Other techniques involve symbolic execution (e.g., [25], [26]) or a meta-heuristic search (e.g., [27]). Alternatively, one can consider testing techniques for labeled transition systems (e.g., [28]).

When the state machine (or a set of communicating state machines) has actions (or activities) and/or guards that are all linear [29], automated test case construction is also feasible: e.g., [1], [29], [30], [31], [32].

In all those cases, the testing technique is typically offline since it is possible to statically analyze the test model and create feasible test cases prior to executing them. There are exceptions, such as UPPAAL-TRON [32], which is a technique (and a tool) to online test a real-time embedded software from a timed automaton (or network of timed automata) specification.

In those approaches, generated test cases have the form prefix-target-suffix [13], where *target* is a test objective to achieve, such as a transition to reach, *prefix* is a path in the state machine that allows the test to reach the target, and *suffix* is a sequence that allows for instance to identify the state that was reached after triggering the target. Test cases are therefore typically created one at a time, each test case achieving one test

objective (i.e., reaching and testing one target, such as a transition). Instead, in this thesis, we try to generate an entire adequate test suite in one optimization step.

When actions (or activities) and guards are specified with a more complex language, offline testing is typically not possible since it is very difficult (or even impossible) to statically analyze both the state-based behavior and the pieces of code that specify actions and activities, which can be as complex as any piece of Java/C/C++ code, to create feasible test cases: e.g., Conformiq [26] uses a Java-like language for specifying actions; Instead, online testing is necessary to simulate state changes, and more so those pieces of code to identify the resulting state and therefore identify what can be the next event to send to the implementation/simulation of the state-based behavior (e.g., [33]).

2.3.2 Achieving scalable model-based testing through test case diversity

Model based testing (MBT) is widely used in academia and industry. However, for large industrial systems MBT results in very (too) large test suites, which are hard to completely execute because of limited time and budget of software development projects. This scalability issue is one of the main obstacles to the transfer of MBT in to industrial projects [34].

Hemmati et al [3] have proposed a similarity-based selection technique for state machines that addresses this scalability issue. The proposed approach uses the similarity between test cases in a test suite as fitness measure to be used by an optimization algorithm that aims at minimizing similarity. Increasing diversity in the test suites increases fault detection rate (FDR). However, as we will see in section 3.7.4, this results in an increase in the cost of test suite as well. Hemmati et al experimentally show that the relation between test suite size and FDR is a linear relation with an elbow point, after which increasing test suite size (for example reaching adequacy based on a selection criteria) does not result in a significant increase in FDR. They propose this point as a good trade-off between FDR (test suite size) and cost.

A similarity function between two test paths (tp_i , tp_j) is defined as “the number of identical transitions in tp_i and tp_j divided by the average length of tp_i and tp_j .” [3] The

similarity measure for a test suite S_n of size n , i.e., comprised of n test cases, is defined as the sum of pair-wise similarities values between the test paths in the test suite.

$$SimMsr(S_n) = \sum_{tp_i, tp_j \in S_n \text{ and } i > j} SimFuc(tp_i, tp_j)$$

The similarity-based test case selection procedure is composed on three steps:

1. Encoding of test cases to compute similarity
2. Calculating pair-wise similarities among test cases
3. Selecting test cases with minimum pair-wise similarity

Based on the fact that it is possible to use different methods for each one of those step the authors performed an experiment to compare the results of using several of those different techniques. Regarding the first step, they have considered four different encodings for a test path, i.e., the amount of information in test path that is used to compute similarities:

1. State-based (SB): Each test path is a sequence of unique state identifiers.
2. Transition-based (TB): Each test path is a sequence of unique transition identifiers.
3. Trigger-guard base (TGB): Each transition is identified by its trigger and its guard. Therefore, two different transitions that have the same trigger and guard will be considered the same. Transitions without a guard or trigger are identified by their transition identifier.
4. State-trigger-guard-based (STGB): This encoding has the highest level of details because it contains both state and trigger-guard information.

With respect to similarity calculation, two types of similarity functions have been considered. Set-based functions include: hamming distance, Jaccard index, Gower-Legendre measure, Sokal-Sneath measure and counting function. Sequence-based functions include: Levenshtein distance, Global alignment, Needleman-Wunsch measure, local alignment and Smith-Waterman similarity function. The reader interested in those functions is referred to [3].

Different minimization algorithms have been examined for the third step, including greedy-based algorithms, clustering-based algorithms and search-based techniques like hill climbing or genetic algorithms (GA).

The effects of those choices have been compared in a series of case studies that resulted in a total of 320 different similarity based selection techniques. By comparing the effectiveness at detecting faults and the cost of generated test cases, they conclude that a specific type of evolutionary algorithms ((1+1) EA), which is a simplified version of genetic algorithms, and the Gower-Legendre (set based) similarity measure is the best combination of optimization technique and similarity measure.

2.3.3 Generating feasible test paths from extended finite state machines using genetic algorithms

Kalaji et al [30] have proposed a search-based approach to (i) generate (likely) feasible test paths from extended finite state machines and (ii) generate input sequences that trigger those test paths. The notion of feasibility relates to the capability to find test inputs for a test path such that we can execute the test path: a test path is a traversal of the graph representing the state machine; this test path is feasible if we can find test inputs (e.g., input arguments to triggering events) such that the path can actually execute as specified. Any traversal of the state machine graph is not necessarily feasible because of guard conditions.

To achieve the first goal, i.e., generating (likely) feasible test paths, they use a genetic algorithm (GA). Their objective is to satisfy the all-transitions test selection criterion. For each transition, the GA is used to find a (likely) feasible test path that executes it. The fitness function estimates the feasibility of the path based on the dataflow dependencies among its transitions.

Whenever, there is a dependency between two transitions t and t' (with t being before t' in the test path) the transitions are called affecting (resp. affected-by) transitions, resulting in an affecting/affected-by pair, and a penalty value is assigned to this pair based on the type of dependency between the elements of the pair. The penalty is assigned considering three factors: the guard type, the guard operator, the operation type

of the affecting transition; and the penalty is an estimate of the impact of those factors on feasibility, i.e., on the capability to find test inputs to execute the path. For example the ‘equals’ guard type (=) gets a higher penalty than the ‘greater than’ or the ‘less than’ (< or >) guard types.

As mentioned earlier they have also proposed a method for generating an input sequence that triggers a test path: their second goal. To achieve this, they transform each test path into code using nested IF statements. They use another GA to find input values that cover this piece of code. The fitness function of this GA is an adaptation of similar work for finding test data for white box testing [35], and is based on the notions of branch distance and approach level. The branch distance represents how close an input value is to satisfying a specific predicate (in their case a guard condition) [36]. The approach level measures how close a test path is to executing a specific statement (in their case a transition).

Kalaji et al report a success rate of 96.6 % when generating feasible test paths and a rate of 100% when finding input values that trigger those paths. 3.4% of the paths were not feasible because of the counter problem which we will discuss in section 5.

Regarding input sequence generation, Lefticaru et al [37] have also proposed an approach that uses a GA. They employed a GA to generate actual test values that trigger a given feasible path in an EFSM where transition guards are described as algebraic predicates. Each chromosome in their GA is a list of input parameters corresponding to all the transition actions of a given path in the order they appear in the path. Similar to Kalaji et al they used a combination of approach level and branch distance as fitness function.

2.3.4 Multi-Objective Search Based Testing

According to a recent analysis of SBST research agenda by Harman et al [38], multi-objective search based methods are frequently used in test suite selection and prioritization but in case of test data generation most of the existing approaches are single objective. Similar to our discussion in section 1, they believe test generation is multi-

objective and it is very improbable that practicing software testers consider only a single objective.

One of the few existing multi-objective test generation techniques is the one proposed by Yano et al [39]. They have proposed a multi-objective approach for test case generation from EFSMs (MOST). Their approach considered two objectives, which are coverage criterion and the solution length. Like Kalaji et al [30] they create one transition at a time. To avoid generating infeasible paths an executable version of the EFSM model was used. They have used State Machine Compiler (SMC tool) to generate executable versions of their EFSMs. SMC accepts as input an EFSM and returns code implementing the EFSM behavior in a variety of programming languages. They only considered paths triggered during model execution as candidate solutions. Using an executable version of the test model they obtained transition path and the data that triggers the path at the same time. MOST has two objective functions, which are test purpose coverage (a transition in the EFSM) and minimum length of input sequence. Similar to Kalaji et al [30] the first objective function was defined in terms of approach level and branch distance from structural testing and the second one was simply the length (in terms of transitions) of the input sequence.

Ferrer et.al. [40] have also worked on multi-objective test generation. Similar to MOST their two objectives were maximizing coverage and minimizing cost. However, they addressed structural testing rather than model based testing. They compared two different solutions for multi-objective test data generation. The first one was a direct multi-objective approach and the second one was a combination of a single objective algorithm followed by a multi-objective test case selection optimization. They have used branch coverage as the adequacy criterion. For the direct multi-objective solution they have used five different multi-objective algorithms, which are NSGA-II (Non-dominated Sorting Genetic Algorithm) [41], MOCell (Multi-Objective Cellular Genetic Algorithm) [42], SPEA2 (Strength Pareto Evolutionary Algorithm) [43], PAES (Pareto Archived Evolutionary Strategy) [44] and a random search (RNDMulti). The second solution started with a single-objective test data generator, which covered only one branch of the program at a time. After that they used a multi-objective formalization of the test case

selection problem proposed by Harman and Yoo [45]. After deleting repeated test cases they used NSGA-II to solve a multi-objective test case selection problem. Like most search-based structural testing techniques their objective function was a composition of branch distance and approximation-level. The single-objective algorithms they used were an evolutionary strategy and a genetic algorithm. To experimentally compare different approaches and different search-based algorithms they designed an automatic program generator that generates synthetic programs similar to real-world programs. They used 800 synthetic programs with all branches reachable as well as 13 real programs in their experiments. GA and MOCeII had the best results in terms of convergence towards the optimal Pareto front. MOCeII had better results for programs with low nesting degree (one and two), which means there are at most two branching conditions nested in one another, while GA was more successful for programs with high nesting degree.

Langdon and Harman [46] have modeled higher order mutation testing as a multi-objective optimization problem. Mutation testing is a fault based software testing techniques. Faults are intentionally seeded in the program under test to create mutants. A test input that can distinguish between the original program and the mutant by generating different outputs for each of them is called to kill the mutant. In higher order mutation testing more than one change is applied to the original program to create the mutants. Based on the fact that ninety percent of faults that are not detected by testing are complex they aimed at higher order mutants. They used search based testing to generate higher order mutants which are both hard-to-kill and realistic. They aimed at studying the relationship between a mutant's syntax and its semantics and they set search objectives accordingly. Their first objective was looking for mutants with similar semantics to the original program and that would pass most of the test cases (minimize semantic distance). The second objective was minimizing syntactic distance. They measured syntactic distance as the number of changes to create the mutant weighted by the actual changes. To automatically remove equivalent mutants from the population of mutants they have treated semantic distance of zero as a special case. (An equivalent mutant is a mutant that cannot be distinguished from the original program through testing, i.e., a test case that could distinguish the two programs does not exist.) They assigned to those mutants very poor fitness values. They have experimented with four different benchmark programs.

Based on the results reported in the paper, higher order mutation testing using genetic programming is able to reveal complex faults designated by higher order mutants that cannot be represented by first order mutants. Those higher order mutants were harder to kill than any first order mutant. In addition they found that although it is believed [47] that less than one percent of mutants are equivalent, for the four benchmark programs they experimented with, 16%, 14%, 34% and 49% of first order mutants were potentially equivalent mutants.

The problem of test generation considering domain specific quality criteria and trade-offs between them was also modeled as a multi-objective optimization problem. The motivation for this research has been the fact that today, even non-software products include software as a large and important component. The quality of such products depends on a series of trade-offs and software quality is just one of them. Marculescu et al [48] proposed an Interactive Objective Re-Weighting (IORW) system where domain specialists can guide the search-based test generation process by interactively re-weighting quality goals. The proposed system is composed of two main components. The inner component is the one that can be fully automated and deals with generating test cases and evaluating their fitness. The outer component is designed to be interactive. A set of test cases is presented to the domain specialists and, depending on those test cases, they decide if the current weights of objective functions are reasonable or they need to be changed. To reduce the number of solutions that need to be manually evaluated, human interaction is done every n generations rather than every generation. Their system has been evaluated using an industrial case study where the system under test (SUT) is controlling an electric motor, which is powering a mechanical arm. Because of the limitations of the motor and to avoid damages in what the mechanical arm is handling the SUT is an average filter that converts sharp changes in the input to smoother signals. Based on this desired functionality three domain specific objectives were defined for the system [48]:

- Discontinuity: Represented the number of discontinuities in a test case and needs to be maximized.

- Upper limit: This objective was designed to measure how close a given value is to the upper limit for the input signal. This objective was to be maximized as well.
- Length: Each input signal had to be within a predefined range. Solutions outside that range were assigned a positive cost. That cost had to be minimized.

Using these domain specific objectives and IORW system, domain specialists were involved in the search based test generation process. Once every 250 optimization steps a domain specialist interacted with the system and adjusted the weights based on the importance of each objective. At different iterations different weights were assigned to different objectives based on the importance of each objective.

2.3.5 SBST and structural testing

SBST has been mostly used in structural testing [10], initially considering test data generation a single-objective optimization problem, focussing on one test objective (e.g., a branch in a control flow graph) at a time. Different avenues have been researched to improve such test data generation. On the one hand, Harman et al [49] proposed the first formulation of test data generation as a multi-objective problem. On the other hand, Michael et al [50] extended a single-objective search to build an adequate test suite achieving many test objectives at once, instead of searching for one solution for each test objective separately, and iterating over objectives: The authors do build an adequate test suite one test case at a time instead of an entire test suite in one search activity; however, as opposed to other approaches, which consider each new test case construction a brand new, from scratch, search, the GA-supported search for a new test case is guided by what has already been covered with already created test cases. A third avenue is to create a whole adequate test suite in one general multi-objective search, instead of creating a test case for each test objective [50] .

2.3.6 SBST—state of the art

Harman presented a recent survey on SBST as a keynote at the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015) [38]. His analysis shows an exponential growth in the cumulative number of publications on SBST since 1990. Based on the trend extracted from the SBSE repository [51] there will be more than 1,700 papers on SBST by 2020. Search based software testing is now sophisticated enough to be used in industry. SBST has been mostly used for structural testing [38]. Wherever favorable properties of test data can be modeled as one or more fitness functions SBST can be used as an approach that searches the system input space for best solutions. SBST has been used in functional testing, safety testing, security testing, robustness testing, integration testing, service based testing, temporal testing, exception testing, combinatorial interaction testing, software product line testing, state-based testing and mutation testing [38]. According to Harman, open problems and challenges in SBST that need more consideration are:

1. Testing non functional requirements;
2. Search based test strategy identification (searching for strategies for finding test cases);
3. Multi-objective test data generation: Although there has been some work on regression testing, there is a lot to be done in the area of multi-objective test data generation.

Ali et al, have also performed a systematic literature review on the application and empirical investigation of search based test case generation [52]. Aiming at the assessment of existing evidence on cost and effectiveness of SBST they selected 122 relevant articles published over the period of 1996-2007. After excluding papers that did not address automatic test case generation or did not report any empirical evaluation they had 64 papers to review and classify. One of the aspects they considered was the meta heuristic search algorithm used in SBST. The most common algorithm was genetic algorithm (GA) and its extensions (73 percent). They referred to a number of reasons for the popularity of GA in SBST which include, large volume of publications on the

application of GAs to different problems, different parameter settings for GAs which help customizing them for a specific problem, better performance compared to local search algorithms as well as great tool supports and lots of frameworks that implement GAs.

The current state of the art on SBST, the need for multi-objective test generation techniques as well as the popularity of GAs, justify the solution proposed in this thesis: a multi-objective genetic algorithm that automatically generates whole state-based test suites.

2.3.7 Differences with our work

Although this will become clearer as we describe our approach in subsequent chapters of this document, we can outline the main differences between our work and previous work in the domain of search based software testing.

Unlike most of the methods mentioned in section 2.3.1, our approach aims at test generation from EFSMs rather than FSMs. There is no need to flatten an EFSM in to an FSM in our approach; therefore, state explosion can be avoided. Also we are using a meta-heuristic search which has proved to be a reasonable approach to tackle undecidable problems [10].

While Kalaji et al [30] (sections 2.3.2 and 2.3.3) create one test case at a time, our approach generates an entire test suite in one search activity. Unlike Kalaji [30] and Hemmati [3] which aim at single objective, feasibility and similarity, respectively, we aim at multiple objectives at the same time. We model test generation from an EFSM as a multi-objective optimization problem. The multi-objective genetic algorithm searches for whole test suites with maximum coverage and feasibility as well as minimum cost and similarity. Different solutions on the Pareto front found by a multi-objective GA provide trade-offs between different objectives and give the software testing specialist the opportunity to choose the best solution for a given EFSM testing problem.

As mentioned in section 2.3.4, there are few multi-objective search based approaches that address test data generation. Yano et al [39] have also worked on test generation from EFSM. However, similar to Kalaji et al, their approach generates one test

path at a time and they address only two objectives, which are cost and feasibility. On the other hand, our approach generates whole test suites and addresses coverage and diversity in addition to cost and feasibility. Ferrer et al have also proposed a multi-objective test generation but their solution falls in the category of structural testing rather than model-based testing. They also address only two objectives, which are cost and coverage, while we generate whole test suites that present best tradeoffs between four different objectives. Longdon and Harman [46] have proposed a multi-objective solution in another area of software testing which is mutation testing. Multi-objective testing approach proposed by Marculescu et al [48] addresses another aspect of software testing which is engaging domain specialists, who are not necessarily software engineers, in the process of software verification and validation.

There has been a great volume of SBST research on structural testing while we consider state-based testing. The application of SBST to functional testing and specifically state-based testing has been very limited and has not yet considered more than two objectives, to the best of our knowledge.

To summarize: (1) With respect to the construction of an entire test suite, we conduct functional testing rather than structural testing, and we use four objective functions together; (2) In the domain of SBT, we create an adequate test suite at once rather than test cases one at a time to achieve sub-objectives. We believe that creating test cases one at a time is a sub-optimal strategy/optimization, and that our construction of a complete test suite is a more global optimization; (3) We use a multi-objective algorithm to simultaneously account for test case diversity within the test suite, feasibility of individual test cases, cost and coverage of the entire test suite.

2.4 Summary

In this chapter we discussed background and related work. We started by describing state machines and genetic algorithms, which are the main basic concepts we use in this thesis. We then discussed related work. As contributions of this thesis fall into the category of state based testing we used the survey by Lee et al [13] as a framework to classify different existing approaches to state based testing. Then we described to main

papers that actuated this thesis [3], [30]. We continued by reviewing papers on multi-objective testing and search based software testing that are two most current fields of research that this thesis falls into. We concluded this chapter by explaining the differences between our approach and current existing solutions.

3 Test generation from an EFSM

This Chapter presents our solution to the problem of generating a test suite from an EFSM using a multi-objective genetic algorithm. We will start by describing our encoding of the problem solutions, i.e., what a chromosome and its genes represent (section 3.1) and then we discuss the other main components of the genetic algorithm: mutation and crossover operators (sections 3.2 and 3.3, respectively) as well as the fitness functions (section 3.4). We continue by explaining parameter settings of our GA as well as its implementation in sections 3.5 and 3.6. Empirical evaluation procedures as well as its results are presented in section 3.7. We discuss threats to validity in section 3.8.

3.1 Chromosomes and genes

A chromosome is a solution to the optimization problem, which means, in the context of our multi-objective GA, a chromosome is a test suite. The genes are the elements that compose a chromosome (test suite). A gene is therefore a test case. As we are using an EFSM as test model, each test case is a path in an EFSM. We use a transition-based representation of paths in an EFSM in our solution. Therefore each gene is a sequence of transitions of the state model. One of the objectives of test suite construction is to achieve a certain level of coverage according to a selection criterion; we want to obtain an adequate test suite for that criterion. Since different adequate (i.e., satisfying a criterion) test suites usually exist and have varying number of test cases, our GA has chromosomes of variable length (i.e., variable number of test cases).

Using a variable length chromosome can lead to a phenomenon known as bloat [53], whereby chromosomes get unreasonably bigger and bigger because small negligible improvements in an objective value are obtained with larger solutions. To control for bloat we have put a limit on the total number of transitions in a test suite, similarly to others [54]. Following a trial and error procedure: for each case study we ran the GA a couple of times and observed the number of transitions in adequate test suites. We specified a limit of twice the number of transitions in the biggest adequate test suite.

Figure 3- Chromosome encoding

For instance, Figure 3 shows a chromosome (test suite) with five genes (test paths). There are five genes (g_1 to g_5) in the chromosome, representing test paths of length lg_1 , lg_2 , lg_3 , lg_4 , and lg_5 , respectively, the length of a gene being the number of transitions in the test path. Each t_i is uniquely identifying a transition in the state model. Note that the order of the genes (test paths) in the chromosome is not relevant: i.e., $\{g_1, g_2, g_3, g_4, g_5\}$ and $\{g_2, g_5, g_1, g_4, g_3\}$, for instance, represent one and only one chromosome (test suite).

In order to make any sequence of numbers a valid test path we used a slightly modified version of Kalaji et al's [30] encoding. In their encoding each test path is a sequence of integers, each integer representing a transition. In an EFSM with k states each having $n_1, n_2, n_3, \dots, n_k$ outgoing transitions respectively, they calculate the lowest common multiplier (LCM) of $n_1, n_2, n_3, \dots, n_k$. Then for each state a range (r_i) is defined as $r_i = \text{LCM}/n_i$. Each transition is represented by an integer between one and LCM. This number is divided by the corresponding range (r_i) to find the transition it defines. In our encoding we used the idea of LCM but in a different way. Instead of computing range values (r_i) we create a matrix for each EFSM. This matrix has LCM columns and k rows (each representing a state in the EFSM). For each state we define a mapping between outgoing transitions and values between one and LCM (similar to Table 1). Because the number of outgoing transitions can be smaller than LCM this is a one-to-many mapping, i.e., a transition can be mapped to more than one value between one and LCM. We used this matrix together with a tabular representation of the state machine to implement our encoding. In the tabular representation (similar to Table 2) each column represents one

transition. The first row corresponds to the source state and the second row corresponds to the target state of each transition.

Figure 4 shows a simple state machine. It has three states. Two of them (S1, and S2) have three outgoing transitions and one of them (S3) has two outgoing transitions. Therefore, LCM for this state machine is six. Table 1 presents the matrix that is used in our encoding. It has six columns of data and three rows. Each cell presents the outgoing transition that will be taken for a specific number between one and LCM. Table 2 presents the tabular representation of the state machine in Figure 4. Because it has eight transitions, the table has eight columns. Suppose for instance that sequence $\langle 1,3,4,2,6,1 \rangle$ is randomly generated. This sequence represents a transition sequence that starts in the initial state S1. From this state, transition number 1 is selected: this is T1 for state S1 (Table 1), which leaves the state as S1 (Figure 4). Transition number 3 is then selected: this is T3 for state S1 (Table 1), which changes the state into S3 (Figure 4). Transition number 4 from S3 is T8, which brings the state to S2. Transition 2 from S2 is T5, which leaves the state as S2. Transition 6 from S2 is T6, which brings the state to S3. Transition 1 in S3 is T7, which brings the state to S1. Sequence $\langle 1,3,4,2,6,1 \rangle$ therefore specifies transition sequence T1, T3, T8, T5, T6, T7.

This encoding facilitates the random construction of valid traversals of the graph representing the EFSM. A test path is a valid traversal of the EFSM in the sense that this is a path in the graph representing the EFSM. This traversal may however not be feasible, i.e., we may not be able to find inputs to execute it. This is the reason why we use an objective function to increase chances of feasibility (see below). Our encoding of a chromosome is then a set of gene encodings.

Figure 4- A simple state chart used to present our encoding of transition sequences

Definition 1: A **valid test case** is a valid traversal of the EFSM graph. For instance, any depth-first or breadth-first algorithm applied on an EFSM graph generates valid test cases.

Definition 2: A **feasible test case** is a valid test case for which one can find test inputs (e.g., for signals triggering transitions) such that this test path can actually execute.

Table 1- Matrix used to implement our encoding of transition sequences

	1	2	3	4	5	6
S1	T1	T2	T3	T1	T2	T3
S2	T4	T5	T6	T4	T5	T6
S3	T7	T8	T7	T8	T7	T8

Table 2- Tabular representation of state machines used in our encoding

	T1	T2	T3	T4	T5	T6	T7	T8
Source	S1	S1	S1	S2	S2	S2	S3	S3
Target	S1	S2	S3	S1	S2	S3	S1	S2

3.2 Mutation Operator

Since in our multi-objective GA a chromosome has many characteristics (e.g., number of genes, length of each gene/test path, specifics of each gene/test path), a chromosome can be mutated in many different ways. To identify possible mutation operators, we systematically considered every single characteristic of a chromosome and investigated ways to mutate them.

We have defined seven different mutation operators. During each mutation one of the following operators is selected randomly with equal probability:

- (1) Adding a gene to a chromosome, i.e., a randomly generated gene (i.e., test path) is added to the chromosome (test suite). Random generation of a gene (test path) proceeds as discussed by others [30] as this ensures we obtain valid traversals of the graph representing the EFSM, starting from the initial state. During this random construction, we randomly select the length of the test path to be constructed between one and twice the length of the shortest path that covers all the states in the EFSM. If such a path does not exist, i.e., it is not possible to tour all the states in one path (i.e., the graph is disconnected), the length of the shortest path that tours the maximum number of states is considered. This is arguably a heuristic and it may happen that no feasible path (i.e., one can find inputs to execute it), even of that length can indeed feasibly reach some state or transition. However, we believe this leads to long enough newly randomly generated paths that other mutation or crossover operators can extend. We add this new gene (test path) to the chromosome (test suite) being mutated. In case the randomly generated gene is a duplicate of a gene already in the chromosome, we repeatedly try to generate a new one with the procedure above until the new gene is different from an existing gene;
- (2) Removing a gene from a chromosome, i.e., a randomly selected gene (test path/ test case) is removed from the chromosome (test suite);
- (3) Replacing a gene, i.e., a randomly selected gene is replaced by a new randomly generated gene, following the procedure we discussed above in (1);
- (4) Altering a test path (gene), i.e., a randomly selected gene (i.e., test path) is mutated by removing a randomly selected transition (and the subsequent transitions) from the path and replacing those by a new (sub-)path of the same length as what is removed. Removing a transition (t_i) from a test path, results in a new path which has the same sequence of transitions before t_i 's source state and continues with a different, randomly generated sequence after t_i 's source state. Because of the encoding we use it is still a valid path in the EFSM. It is just equivalent to selecting a transition other than t_i in the state from which t_i is an outgoing transition. For the example in Figure 4, this means replacing an encoded sequence like $\langle 1,3,4,2,6,1 \rangle$ which represents path $\langle T1, T3, T8, T5, T6, T7 \rangle$ (see above) with the sequence

$\langle 1, 3, 4, \mathbf{1}, \mathbf{5}, \mathbf{4} \rangle$, which represents path $\langle T1, T3, T8, \mathbf{T4}, \mathbf{T2}, \mathbf{T4} \rangle$, assuming the forth transition is selected as mutation point and numbers 1, 5 and 4 are randomly selected;

- (5) Appending a transition to a test path (gene), i.e., a randomly selected gene is mutated by randomly adding a transition to its end, ensuring the entire path is valid. If a path ends with a terminal state, i.e., no outgoing transition, the path cannot be mutated with this operator. Therefore we are both changing the length of a test path as well the sequence of transitions in it;
- (6) Changing a transition of a test path (gene), i.e., a randomly selected gene is mutated by randomly replacing one of its transitions with another one between the same two states (if another such transition exists). Unlike the previous operator this mutation does not change the length of a test path;
- (7) Exchanging material between genes of the same chromosome (Figure 6), i.e., exchanging randomly selected transitions (in fact sub-paths) between randomly selected test paths (genes) from the same test suite (chromosome), while ensuring that results are valid traversal of the graph representing the EFSM. To achieve this, we reuse a published procedure [30] which, once again, because we reuse their encoding, results in valid traversals. We note that when Kalaji et al. used this operator they were searching (by means of a GA) for a test case that reaches a specific transition. In their context, this operator is a crossover operator since the population manipulated by the GA is a population of test paths: the crossover exchanges material between chromosomes (test paths) of the population. In our case, the test paths involved in this operator are the genes of a chromosome. The operator therefore alters only one chromosome, and we classify it as a mutation operator in our context. We select a number which is smaller than the length of the shorter test path as the crossover point. We then exchange transition sequences after this point between two test paths. Again because this is done on the encoded version of test paths this results in valid test paths.

Figure 5 – Exchanging transitions between test paths in the same test suite

3.3 Crossover operator

Crossover creates two new chromosomes from two existing (parent) chromosomes by exchanging some genetic information, i.e., genes, from/between those parents. Once again, we shall ask ourselves what information can be exchanged between two (parent) chromosomes. At first sight, similarly to traditionally-defined two-point crossover operators [15], we can exchange two genes (test paths) between two parent chromosomes (test suites) C_i and C_j (Figure 7). Our first crossover operator is a two-point crossover [15] in which the two points identify one gene in each parent instead of the more general case where they identify a sequence of genes. Given that we have variable length chromosomes, we ensure the crossover points are valid: e.g., if a parent has five genes and the other parent has ten genes, the crossover points identify one of the first five genes in both parents. We randomly select two parent chromosomes (test suites), C_i and C_j , and one test path (gene) from each of the parents (at the same index in the sequence of genes of the parents). We remove the selected test path (gene) from C_i and add it to C_j , and similarly remove the selected test path (gene) from C_j and add it to C_i . As a chromosome represents a test suite, the order of its test paths (genes) does not matter. Therefore, we add the test path extracted from C_i (resp. C_j) to the end of C_j (resp. C_i).

A second crossover operator exchanges genetic information at a lower level of details, specifically at the transition level, between test paths. We select two parent chromosomes, select one gene (test path) in each of those parents, and exchange transition sequence information between those two genes. This second crossover operator works identically to the last mutation operator we discussed, except that the two genes

(test paths) are selected from different chromosomes (test suites), whereas for mutation purposes the two genes belonged to the same chromosome.

When crossover happens, with a specific probability, these two crossover operators have an equal probability of occurrence.

Figure 6-Chromosome mutation: crossover between genes (test cases/paths) of the same chromosome

Figure 7-Exchanging genes between two chromosomes

3.4 Fitness functions

As previously mentioned, we are considering four fitness functions: Feasibility, which needs to be maximized; Similarity, which needs to be minimized; Coverage, which needs to be maximized; and Cost, which needs to be minimized.

3.4.1 Feasibility

To determine the **feasibility** of a test path, we reuse previous work that relies on an analysis of data flow dependencies between the transitions of a test path [30]. Different types of data flow dependencies that might exist between two transitions of a test path are assigned penalty values based on their potential effect on feasibility. This information is used to obtain a feasibility measure for each gene (test path) in a

chromosome (test suite) and then we obtain a feasibility measure for the chromosome as the sum of the feasibility values of its genes. This feasibility of a chromosome c_i , made of n genes is then:

$$feasibility(c_i) = \sum_{j=1}^n feasibility(g_j);$$

Where, feasibility (g_j) is the feasibility of gene g_j and is obtained by using the approach proposed by Kalaji and colleagues [30], which they showed is indeed a surrogate measure of feasibility. However, actually trying to find inputs such that g_i executes is the only way to identify whether g_i is indeed feasible.

The data flow analysis of Kalaji et al. [30] can identify if a test path is definitely infeasible when for instance the path assigns a constant to a variable and a following guard condition requires the variable to equal another constant. In such a case, the penalty assigned to the path is a very large (simulating infinity) value. Given our feasibility metric, if a test suite has a feasibility value greater than this very large value, we know it contains a test path that is definitely infeasible.

It is important to note that feasibility (g_i) is only an estimation of the feasibility to find input values for the corresponding path such that the model (EFSM) can actually be executed. If there is no abstraction difference between the model and the code, feasibility at the model level ensures feasibility of creating a test script that can actually execute on the code. Otherwise, a feasible test path at the model level will need to be adapted into an executable test script: e.g., one transition in the (abstract) EFSM test model needs to be realized as several method calls at the source code level. Similarly to others, we do not consider feasibility issues due to different abstraction levels between the model and the code. Also we do not claim that this is an exact measure of feasibility but it is a good surrogate measure of feasibility, others have used it, and it can lead the GA towards better solutions in terms of actual feasibility. The penalty is in direct relation with 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 guard conditions (i.e. they are not dependent on context variables but include other type of conditions such as comparison among parameters), are also penalized.

Last but not the least, penalty values can be computed quickly and thus are suitable to be used in a search-based approach.

3.4.2 Similarity

Our intuition, as well as others' [3], is that test paths should be as dissimilar as possible to increase effectiveness at fault detection. We are therefore interested in computing the **similarities** between pairs of test paths (genes) to obtain a similarity value for a test suite (chromosome). Different similarity measures can be used as fitness function. One of the measures [3], which is not limited to identical length sequences (we have variable length chromosomes), is the Levenshtein distance [55]. Although not the recommendation of Hemati et al. (recall section 2.3.2) this is the best measure of similarity they studied [3] that supports variable length sequences. To change this distance measure into a similarity measure, similarly to Hemmati et al. [3] we reward each match between two sequences by one point and penalize mismatches and gaps by simply ignoring them (i.e., assigning no point). More sophisticated measures (e.g. Needleman-Wunsch), which penalize mismatches and gaps, can be used as well. To obtain a similarity measure for a test suite (i.e., chromosome), we proceed similarly to Hemmati et al. [3]. The similarity measure for a test suite (chromosome) is the sum of similarity measures computed for each pair of test paths (genes) in that chromosome. The fitness function needs to be minimized, and is defined as follows:

$$similarity(c_i) = \sum_{pairs\ of\ genes\ (gj, gk)} similarity(gj, gk).$$

We note that our measure of similarity is not the best one as per the experiments reported by Hemati et al. (recall section 2.3.2). We however selected it since they experimented with it, showing it performs well, and more importantly (i) it allows us to account for test paths of variables lengths and (ii) we believe a sequence-based measure like the Levenshtein distance is more appropriate in our context than set-based distances like Gower-Legendre since we measure test cases that are paths (sequence of states and transitions) in an EFSM. We will revisit measurement of test cases and test suite similarity in section 4.

3.4.3 Coverage

Different **coverage** criteria can be used as objective function. Although more demanding, but also more effective (at finding faults) criteria exist [56] [57], we selected the all-transitions criterion. Transition coverage is considered a good minimum to aim for in state-based testing [1]. The fitness function is to minimize the number of distinct transitions uncovered by a test suite (chromosome).

3.4.4 Cost

The last objective is reducing the **cost** of a test suite as much as possible. A usual surrogate measure of cost is the size of a test set [58]. The underlying assumption is that test cost is proportional to the test set size. The notion of cost in the context of testing is complex as one may want to consider time to market or computer time usage as part of the equation [59]. Different test may also require different set up and tear down, resulting in different costs. In our context, test set size can be simply measured by counting the number of test sequences in a test set (e.g., [60]). However, not all test sequences contain the same number of transitions triggered. We could go further and say that not all sequences take the same time to execute. But without going to that level of detail, we could measure cost/size as the number of transitions triggered in the test set, summing up the number of transitions triggered in each test sequence. This objective function is to be minimized.

3.5 Genetic algorithm parameters

There are a number of factors that highly affect the success of a GA. The most important ones are the population size, the mutation and crossover rates, the Pareto Fraction, and the stopping criterion.

We selected a population size of 200, which conforms to what has been suggested in the literature [61]: i.e., a value in range [30, 80] (we selected 50) multiplied by the number of fitness functions. Based on results from previous studies [62] we selected a crossover rate of 0.7 and a mutation rate of 0.01. The Pareto Fraction parameter, or Pareto set, controls elitism in a multi-objective GA since it limits the number of

individuals on the Pareto set (elite or tradeoff members). Based on a previous study [61], which suggests setting the maximum size of the Pareto set such that the ratio of the Pareto set over the entire population is between $\frac{1}{4}$ and 4, we set the maximum size of the Pareto set to 0.35 of the entire population. With respect to the stopping criterion, we reuse the one of the toolbox we relied on (see below). Specifically, we stopped the GA if the observed average change in any fitness function value over 10 generations was less than e^{-4} . This was the default setting of the MATLAB optimization toolbox and we did not change it. The MOEA used by the toolbox is NGSA-II [63].

The initial population is generated randomly, i.e., we start with a set of 200 randomly generated test suites. Each test suite has a random number of randomly generated valid traversals (of variable length) of the graph representing the EFSM. When generating a test suite, we first randomly select the number of transitions N this test suite will have, between one and the maximum number of transitions we defined earlier to control for bloat (section 3.1). We then incrementally create test paths as random traversals of the state machine graph until the cumulative number of transitions in those test paths reaches N . For the purpose of creating test paths, we add a reset transition from each state to the start state. When creating a test path we then iteratively, randomly and uniformly select an outgoing transition (including the reset) from the current state. Admittedly, this can encourage the generation of short sequences, especially when states have a small number of outgoing transitions as specified in the EFSM. Our experiment results show however this does not happen thanks to objectives which favour diversity.

3.6 Multiple-objective GA implementation

To implement and run our genetic algorithm we tailored the multi-objective genetic algorithm framework included in the MATLAB Global Optimization Toolbox [11]. Because the chromosomes in our GA are of variable length and have a specific structure, we implemented our own population creation function, our mutation and crossover operators, as well as our fitness functions, as described earlier in this section.

3.7 Empirical Evaluation

To evaluate our test suite construction technique, we used two different case study EFSMs, modeling real world applications, to answer the following questions:

RQ 1. How do whole and incremental test suite constructions compare?

Are there any benefits in terms of cost for instance, in generating a whole test suite at once rather than test paths separately in an incremental manner?

RQ 2. Are test suites feasible?

Since we use a surrogate measure of feasibility during optimization, do we obtain actually feasible test suites? In other words, can we find inputs such that test suites in the Pareto front can actually execute?

RQ 3. How does our GA compare to a random search?

Is our multi-objective GA successful, compared to random generation for instance, in addressing our optimization goals? In other words, is the search space so large or complex that a random search is not adequate and we have to resort to a GA?

RQ 4. How does our GA compare to other test suite construction techniques?

How do test suites created by our GA compare to test suites created by existing approaches in terms of effectiveness at detecting faults?

The first EFSM models a simple Cruise Control system, which we have used in the past (e.g., [56], [64]) and was originally described in a textbook [65], while the second EFSM is a simplified model of a class II transport protocol we obtained from others [30]. They are used to answer our research questions as summarized in Table 3.

Table 3- Answering research questions with case studies

	RQ1	RQ2	RQ3	RQ4
Cruise Control	X			X
Transport Protocol	X	X	X	X

Section 3.7.1 introduces the two case studies while section 3.7.2 discusses how we used them to answer the research questions (experimental design). Before answering research questions in sections 3.7.5 to 3.7.8, we will perform a qualitative analysis of

some of the results we obtain with the Cruise Control case study in section 3.7.4 and discuss stochastic aspects of our multi-objective GA in section 3.7.3.

3.7.1 Case study systems

We selected two different case study systems for which we have a state model (FSM or EFSM) and source code: a simple Cruise Control and a realistic transport protocol. Some structural properties of the corresponding state machines are summarized in Table 4.

Although these two case studies represent typical cases where a state machine is used to model the behavior, they are admittedly small. However, similar behaviors are usually modeled using state machines in UML-based development [66], [65] and in industrial case studies reported in the literature [67], [68]. Additionally, it is very uncommon in practice to model subsystems or entire systems using state machines, as this is far too complex in realistic cases; there is one exception though, when one abstracts out from low level behavioural details of a (sub-)system in order to simplify the (E)FSM, in which case the state model used for testing purposes is (much) simpler than the actual implemented state-based behaviour. Last, the second case study is a representative sample of a series of case studies used by others [30].

Table 4- Properties of state machine used in case studies

Case study	Number of transitions	Number of states	Number of guarded transitions	Number of events	Number of transitions with actions
Cruise Control	28	4	0	7	0
Class II protocol	21	6	11	13	11

We selected a simple model of a Cruise Control as a first case study because, although it does not have guards or actions (Figure 8) and therefore any traversal of the state machine graph is a feasible test path (as a consequence we have really three fitness functions), it allowed us to check the correctness of our approach and focus on the three other fitness functions. With only three fitness functions we can furthermore plot results and learn from those results in a qualitative analysis: e.g., we can visually observe the result of the competition between the fitness functions. Additionally, we can observe the benefits of generating a whole test suite instead of one test path at a time (RQ1).

Figure 8- EFSM for the Cruise Control system

The second EFSM is a simplified model of Class II transport protocol [30] that models connection establishment, data transfer, end-to-end flow control and segmentation: Figure 9. The reader interested in more technical details about this model is referred to appendix B, where a complete description of the states, transitions, guards and actions is available. We selected this model because of three reasons: (i) There are 11 guard conditions and 11 actions in the model, so not every valid traversal of the state graph is feasible; (ii) It is possible to automatically find feasible test paths in this state machine [30]; (iii) We have information on cost and feasibility of an adequate test suite from others [30], which can be used as a basis of comparison with our approach.

Figure 9- EFSM of a class II transport protocol

3.7.2 Experimental design

3.7.2.1 Answering RQ 1—How do whole and incremental test suite constructions compare?

We simulated the construction of an all-transitions adequate test suite for Cruise Control and Transport Protocol by following a procedure similar to many existing approaches (recall section 2.3.1 on related work) whereby we created a test case for each transition: each test case is made of a prefix followed by the target transition, the prefix being a shortest path to reach that transition. In the case of Transport Protocol, we made sure generated test cases were feasible test paths. We intend to compare this test suite, which we refer to as T_{trad} , with adequate test suites produced by our GA in terms of cost, similarity and effectiveness at finding faults. We followed this procedure based on the fact that in any genetic algorithm that creates one test path at a time, different executions will be independent of each other. Therefore any specific execution of the GA would not necessarily have any information from a previous execution where the goal was to cover another transition. Arguably, T_{trad} may have redundant test cases, as pointed out by others (e.g., [30]): e.g., a test path targeting a transition may be a sub-path of a longer test path targeting another transition. To make the comparison more objective, we then removed the redundancy by iteratively removing any test path t_i that is a sub-path of another test

path t_j in the test suite, until no further test path deletion is feasible. We obtained a test suite we refer to as T_{red} .

3.7.2.2 Answering RQ 2—Are test suites feasible?

As discussed earlier, every traversal of the state graph of the Cruise Control case study is feasible. So we only used the Transport Protocol case study to answer research question RQ 2. Using this case study, we investigated the feasibility of a sample of test suites in the Pareto front we obtain with our GA by manually trying to identify test inputs that would make their test paths to execute.

Specifically, at the end of a typical run of our GA, we ranked the test suites in the Pareto front in increasing order of (lack of) coverage: the first ones we obtained were adequate test suites, i.e., 100% coverage of all the transitions of the EFSM. We then randomly selected one sample test suite for each of the first six values of (lack of) coverage: 0, 1, 2, 3, 4 and 5 (these are the numbers of un-covered transitions by the test suites). For each of the six test suites, we manually tried to find input values such that their test cases can actually execute.

3.7.2.3 Answering RQ 3—How does our GA compare to a random search?

We only used the Transport Protocol case study to answer research question RQ 3, because the model has guards and actions and therefore not all the paths are feasible, and compared our test suite generation technique to a random test suite generation approach. To have an as unbiased as possible comparison we performed two comparisons, each with one specific random generation procedure.

In the first random generation procedure, we considered the following requirements: (i) the random generation should investigate as many solutions in the search space as the number of solutions a typical GA run investigates; (ii) the random generation should investigate solutions that are comparable in terms of cost with solutions the GA investigates. Without (i) we would favour one technique over the other since one technique could easily perform better than the other simply because it considers

more solutions. Since increasing coverage, diversity, and (likely) feasibility can be achieved by increasing cost, we would also favour one technique over the other without (ii).

To account for these two requirements of the search, we monitored several executions of our GA. As there were not too much variation in terms of number of new solutions constructed and average cost of test suite we selected one typical execution of our GA. From the typical execution of our GA we collected the following information: the number of new solutions being constructed, the cost (cumulative number of transitions) of each such new solution. With the latter we obtained the distribution of the number of test suites of specific cost values.

We then randomly generated as many solutions (test suites) as the number investigated by the typical GA and selected an overall cost for each randomly created test suite by randomly sampling the abovementioned distribution. Given an overall cost value, we generated test paths in the test suite by following the same procedure as the one we used to create an element of the initial population of our GA (section 3.5).

Since random generation can easily generate many more test suites than our GA in the same amount of time, we considered a second random generation procedure where the GA and the random test generation are given the same amount of time to search for test suites. Similarly to the previous comparison, we want random test suites that are comparable in terms of cost to the ones generated by the GA. This time, randomly generating a test suite takes a randomly generated number of test cases/paths (tp), to be selected between the smallest and largest Pareto front test suite sizes obtained from a typical GA run, and a randomly generated cost (tc), i.e., number of transitions exercised in those tests, to be selected between the smallest and largest cost values of Pareto front test suites. In a test suite, each test path is created as a random traversal of the state machine graph with a length randomly generated between one and tc . If the overall cost of a generated test suite is strictly greater than tc , then this test suite is discarded and a new attempt is made. This test suite generation procedure continues until the execution time of the random generation is at least equal to the execution of a GA run.

3.7.2.4 Answering RQ 4—How does our GA compare to other test suite construction techniques?

Mutation analysis is a well-known method commonly used for the evaluation of testing strategies [69]. We used *MAJOR* [70], a mutation analysis tool for Java, to seed faults and perform the mutation analysis. In order to transform the state machine into Java code we used a tool called state machine compiler (SMC) [71]. SMC generates code from a state machine using the state pattern. It generated skeleton of classes using this pattern. Then we completed the code by adding the state variables and implementing the actions. *MAJOR* uses an approach called conditional mutation [70]. We used all the mutation operators available with *MAJOR* and created 198 mutants for Cruise Control and 870 mutants for transport protocol. When identifying whether a mutant is killed or alive, we need an oracle for each test case. Our oracle was to check the target state, transition triggered as well as comparing the value of state variables for each transition in the transition sequence defining a test case.

We created other test suites using existing testing strategies, specifically: round trip path [72] because it has been shown to be a good alternative between the transition and transition pair selection criteria [57], a modified version of round trip path that attempts to increase diversity among test paths [56], a test suite manually generated by an engineer, and the approach that creates one test path at a time to achieve transition coverage (recall RQ 1). We compared the generated test suites with test suites created by our multi-objective GA in terms of effectiveness at detecting faults as well as our four objectives (i.e. cost, coverage, similarity and feasibility). The test suites we selected from the Pareto front were the adequate test suites that had the best (i.e. lowest) similarity values.

The manually generated test suite, referred to as T_{man} , was created by an engineer other than us. The engineer was not given any other instructions than to produce an adequate test suite for the all-transitions criterion; we observed that the result is an adequate test suite that minimizes at the same time overall cost and the number of test cases as well as ensures feasibility (in the case of Transport Protocol).

We note that in the two round trip paths test suites (the original one and the modified one) a couple of test cases were unfeasible because of guards and actions. The test cases systematically created by the generation algorithms were slightly modified (appending a transition to two paths) to ensure feasibility of the transition criterion.

3.7.3 Stochastic aspect of our GA

We note that, since a GA is inherently a stochastic process, we would ideally need to run our GA a number of times on each case study and evaluate trends on a large number of resulting adequate test suites. We ran our GA a number of times on each case study and, although we did not study trends formally, we observed results were similar over multiple runs.

3.7.4 Qualitative analysis of some results

Figure 10 plots lack of coverage versus similarity for each test suite belonging to the Pareto front at the end of one (typical) run of our GA for Cruise Control. This figure is a two-dimensional projection of a three-dimensional plot. Numerical values above each point represent the cost of the corresponding test suites (the third fitness function). Test suites (points) right on the y-axis are all-transitions adequate (no lack of coverage).

Figure 10 shows a test suite (point) with values (0, 0, 52) for coverage, similarity, and cost, respectively. This is an all-transitions adequate test suite with ideal (0) similarity value between its test cases. We refer to this test suite as TSga1. The other three adequate test suites in the Pareto front are referred to as TSga2, TSga3, and TSga4, in increasing order of their similarity value (2, 4, 6, respectively): e.g., TSga4 is (0, 6, 46) at the top left corner of the graph. These test suites are included in appendix D. The cost values of the four adequate test suites are 52, 50, 49 and 46. These cost values are higher than for other all-transitions or transition-tree adequate test suites we have created for this case study (section 3.7.8– results for RQ 4 and [26]): the cost values we obtained, though not trying to obtain dissimilar test cases (which increases cost), where 25 for the all-transitions adequate test suites and 38 for the transition-tree adequate test suites.

Allowing coverage to get further from its optimum value (0) decreases cost. For example, keeping similarity constant (for instance at 0), we see that cost decreases when lack of coverage increases (values of cost are 52, 41, 33, 29, 17, 15, ... for lack of coverage 0, 1, 4, 5, 12, 13, ...). Similarly, allowing dissimilarity to get further from its optimum value (0) decreases cost. For example, keeping lack of coverage constant (for instance at 0), we see that cost decreases when dissimilarity increases (values of cost are 52, 50, 49, 46 for dissimilarity 0, 2, 4, 6). Also not covering a single transition reduces cost: e.g., test suite (1, 0, 41), which misses one transition, has zero similarity and its cost is 41. We note that since cost and dissimilarity are objectives to be minimized and an empty test suite has ideal cost and dissimilarity, the Pareto front contains an empty test suite (Figure 10): (28, 0, 0). Future work can look into removing such a solution for instance by penalizing chromosomes (test suites) that do not achieve at least 50% coverage.

Figure 10- Sample Pareto front contents for Cruise Control: the x-axis shows lack of coverage (0 means adequacy), the y-axis shows chromosome (test suite) similarity (0 means very dissimilar test cases in test suite), the value next to each dot (chromosome) is the cost of the chromosome (test suite).

Figure 11- Cruise Control: Pareto front when only optimizing for cost and coverage

When only using two fitness functions, specifically cost and coverage, we obtain a Pareto front as displayed in Figure 11. What is noticeable is that the maximum cost is 39, whereas (Figure 10) it is 52 when also accounting for similarity. This can be considered as a witness for one’s intuition that improving coverage or improving dissimilarity between test cases increases cost.

The last two observations may suggest that we need to specify weights to our fitness functions, to for instance promote coverage. This can be part of future work.

Similar observations about competing objectives can be made for the Transport Protocol case study: see sample data in Table 5: e.g., decreasing coverage (recall we measure lack of coverage) decreases cost. Coverage level is gradually getting further from its optimal value while the total feasibility penalty of test suite is decreasing. This is due to the fact that not covering some transitions (especially the ones that are difficult to cover) increases the likelihood that test cases will be feasible. Also decreasing coverage while maintaining a similar level of similarity decreases cost. We observed the same trade-offs for the Cruise Control case study.

3.7.5 Results for RQ 1—How do whole and incremental test suite constructions compare?

Recall that RQ 1 is to study the benefits and drawbacks of building an entire test suite at once rather than in a stepwise manner, one test case at a time, each test case achieving only one test objective (reaching one target transition): section 3.7.2.1.

Since, as mentioned previously, the Cruise Control EFSM has 28 transitions, T_{trad} is all-transitions adequate with 28 test cases. Its cost equals 70 and its similarity equals 348. When reducing redundancy in T_{trad} , we obtain an all-transitions adequate test suite T_{red} with 23 test cases, with a cost of 58 and a similarity of 270. The manually generated adequate test suite (T_{man}) has two test cases for a cost of 31 and a similarity of zero. These are to be compared with all-transitions adequate test suites generated by our GA (TS_{ga1} to TS_{ga4}): their cost ranges from 46 to 52; their similarity ranges from 0 to 6 (on the y-axis in Figure 10). Without optimizing for similarity (i.e., with only coverage and cost as objective functions), the cost is as low as 39 (on the y-axis in Figure 11).

We first discuss T_{trad} , T_{red} and our GA solutions. A cost of 39 (GA not optimizing for similarity) is almost half of the cost obtained with T_{trad} and much less than T_{red} . Comparing these makes sense since similarity is not accounted for when creating T_{trad} and T_{red} . Even when accounting for test case similarities, which we know increases costs, we obtain lower values with the GA than with the traditional stepwise approach (46 to 52 instead of 58 or 70).

With respect to similarity, we observe that the structure of the EFSM graph demands that longer paths than those in T_{trad} or T_{red} be used to increase dissimilarity: this explains why our GA performs much better in terms of similarity (0 to 6 instead of 348 and 270 for T_{trad} and T_{red}). For instance (recall the EFSM in Figure 8), test paths of length two that cover transitions T8 or T14 all start with T5 and to make them dissimilar one needs to take some loops on state Inactive/Idle before going to state Active/Running through T5. We therefore conclude that creating a test suite one test case at a time (as in T_{trad}) while ensuring we have dissimilar test cases would lead to a test suite that is much more expensive than 70, thereby increasing the difference with our approach. We

therefore conclude that creating an adequate test suite one test case at a time, each test case satisfying one test objective, is sub-optimal.

When comparing T_{man} with our GA solutions, we observe striking differences in terms of cost and similarity. We tried to understand why our GA is not able to find such a good solution by itself and we therefore determined the size of the search space. We limited the study of the search space to solutions similar to T_{man} in terms of number of test cases (two) and lengths of test cases (15 and 16). Given that each state of the EFSM has seven outgoing transitions, there exist 7^{15} (resp. 7^{16}) different traversals of the EFSM of length 15 (resp. 16). There exist therefore $7^{15} \times 7^{16} = 7^{31}$ different test suites with the same number of test cases and the same test case lengths as T_{man} in the search space. Considering that these represent a tiny subset of the search space, it is not entirely surprising that our GA does not find such a good solution. We note however that TS_{gal} is close to T_{man} : (0, 0, 52) instead of (0, 0, 31). We conclude that a human can still do better than our GA, though our GA is very close to a human-generated solution.

We followed the same procedure for Transport Protocol and obtained similar results when comparing our GA solutions to T_{trad} , T_{red} and T_{man} in terms of cost, feasibility and similarity. Because there are 21 transitions in the EFSM of Transport Protocol, T_{trad} had 21 test cases. Its total cost was 55 and similarity within whole test suite was 175 and 20 out of 21 test cases were feasible. T_{red} , had 17 test cases. Its total cost was 48 and similarity within whole test suite was 113 and 16 out of 17 test cases were feasible. T_{man} had only two test cases with a total cost of 26, similarity of zero and both test cases were feasible. If we compare these values with the ones for point 1 in Table 5, which is an adequate test suite with total cost of 37 and similarity value of 37 and test paths that are all feasible, we come to the same conclusion as for Cruise Control. All the test suites mentioned above are included in the Appendix.

Table 5- Selected test suites from the Pareto front for the Transport Protocol case study

Fitness Function	Point1	Point2	Point3	Point4	Point5	Point6
Coverage	0	1	2	3	4	5
Similarity	3	4	4	3	2	0
Cost	37	38	34	24	22	20
Feasibility	270	246	198	156	120	108

3.7.6 Results for RQ 2—Are test suites feasible?

We verified whether a subset of the test suites in the Pareto front for Transport Protocol are indeed feasible: Table 5. We managed to manually find parameter values that make each of the test paths in those test suites feasible. Based on the results in Table 5, we may conclude that a higher value of our feasibility measure does not necessarily mean that the corresponding test suite has less chances of being feasible and we need to take into consideration other factors like the size of the test suite (e.g. point1 vs. point6). Future work should look into studying improved measurements of feasibility.

As an example, referring to the first path of the test suite corresponding to point 1 (Table 5), i.e., Path1: $\langle T_1, T_3, T_{15}, T_8, T_{17}, T_{19}, T_2 \rangle$, we observe that guard conditions, directly or indirectly, depend on parameter and context variable values. Guard conditions and actions of the Transport Protocol EFSM can be found in Appendix B. When triggering T_3 we need to select a value for its first parameter that is smaller than the second parameter of T_1 ($opt_ind < prop_opt$) and set its second parameter to an integer greater than zero ($cr > 0$). This is because when T_1 is executed the value of its second parameter is assigned to the context variable opt which should be greater than the first parameter of T_3 based on the guard condition of T_3 . Restrictions on the second parameter are caused by the guard condition of T_8 ($S_credit > 0$). S_credit is a context variable and the last time it is assigned a value before T_8 is executed in the given path is when T_3 is triggered ($S_credit=cr$). The other conditions are on the parameters of T_{15} . The first parameter should be greater than zero ($XpSsq > 0$) and the sum of the two parameters should be either greater than 143 ($XpSsq + cr > 143$) or less than 128 ($XpSsq + cr < 128$). The last two conditions are caused by the guard condition of T_{15} ($TSsq < XpSsq \ \& \ [cr + XpSsq - TSsq - 128 < 0 \vee cr + XpSsq - TSsq - 128 > 15]$). $TSsq$ is a context variable

and the last time it is assigned a value, before T_{15} in the given path, is when T_3 is executed ($TSsq=0$).

It is possible to satisfy all the conditions above and make Path1 feasible (executable): e.g., $\langle T_1(3,10), T_3(9,4), T_{15}(10,100), T_8(4,6), T_{17}(), T_{19}(), T_2(10,8,2) \rangle$ is a feasible path.

3.7.7 Results for RQ 3—How does our GA compare to a random search?

We compared results of our approach with the random test suite generations described in section 3.7.2.3 in terms of adequacy and feasibility (cost is by design the same as in our GA solutions). We used a typical execution of our GA, which stopped after 151 generations, evaluated 21,350 test suites, and took about 30 minutes to execute.

The first random generation is to randomly generate the same number of test suites, with the same distribution of overall cost (section 3.7.2.3) as the GA run. Among those two series of test suites, the GA generated series and the randomly generated series, we selected the all-transitions adequate ones, and further selected the ones that have a chance of being feasible. In other words, we discarded test suites that are either inadequate or definitely infeasible. The total number of adequate and not absolutely infeasible test suites generated by the GA was 1,495 while the random generation could only generate 44 such test suites.

In the second random generation, where the GA and the random generation were given the same execution time we obtained similar results. We repeated the random generation 20 times. Each random generation created between 16,000 and 17,000 test suites. This is less than the number of test suites evaluated by the GA. This could be due to two reasons. First, this is actually a pseudorandom process where we are controlling cost and discarding test suites which have a cost greater than a threshold: the process may be discarding many test suites thereby consuming time (to create them) but not contributing to any result. Second, the GA is probably doing some optimization in the evaluations: for example it only re-evaluates those chromosomes affected by reproduction operators (i.e., only the ones affected by mutation or crossover) while the random generation creates new test suites each time. However, discarding test suites that

were either inadequate or definitely infeasible only left between six and 11 test suites with an average of 7.8 and standard deviation of almost 1.85 over the 20 random generations of test suites.

We conclude from both random generation strategies that there is a very low chance of achieving our optimization objectives by randomly generating test suites.

Figure 12 plots the number of adequate not absolutely infeasible test suites found by the GA and the first random generation strategy at each iteration of their construction: random and GA generations occurred concurrently. During the first few iterations, the two procedures are close to each other but in later iterations the number of (adequate, not definitely infeasible) test suites generated by the GA is dramatically higher than the ones randomly generated. This is due to the fact that in early generations, the GA considers costly test suites, and a higher overall cost gives more chances to the random generation to obtain adequate, likely feasible test suites.

Figure 12- Number of adequate not absolutely infeasible test paths found at each iteration

3.7.8 Results for RQ 4—How does our GA compare to other test suite construction techniques?

We selected a couple of adequate test suites generated by our GA and compared them with test suites created by using other testing strategies. For Cruise Control we selected TS_{ga1} and TS_{ga4} (section 3.7.4) and compared them with four other transition adequate test suites: T_{man} and T_{red} (section 3.7.5), two versions of the round trip path

strategy. Mutation score is almost the same for all the test suites: Table 3. In addition to test suite similarity values with the Levenshtein distance (used by the GA) we also report on the Gower-Legendre measure of similarity since others found it to be a better measure. Our data show a strong positive relationship between the two: Pearson's r coefficient of .99. Every test suite, except T_{man} , has higher values of similarity and cost compared to ones created by our GA. T_{man} can be considered as the optimum the GA is searching for. Different types of variability that we have in our encoding (variable number of test cases in test suites as well as variable length test paths) have made it very difficult for the GA to reach that point. However, it has reached points that are much closer to this optimum compared to other approaches.

For Transport Protocol we compared a GA generated adequate test suite (point 1 in Table 7) to three other test suites: two versions of the round trip path strategy, and T_{man} . The two measures of similarity are once again highly correlated: Pearson's r coefficient of .99. The GA test suite had the highest mutation score. Although T_{man} has the lowest cost it is killing fewer mutants compared to other test suites. Results of mutation analysis confirmed our intuition about a relationship between diversity in a test suite and its cost-effectiveness (Table 6 and Table 7).

Table 6- Results of mutation analysis (Cruise Control)

	TSga1	TSga4	T_{man}	T_{red}	RTP	RTP (Modified)
Levenshtein	0	6	1	270	296	44
Gower-Legendre	2.74	2.95	0.38	62.85	85.65	10.36
Mutation score	40%	40%	40%	38.38%	38.38%	40%
Cost	52	46	31	58	64	41

Table 7- Results of mutation analysis (Transport Protocol)

	GA (point 1)	RTP	RTP Modified	T_{man}
Gower-Legendre	1.9347	129.6714	5.6314	1.1833
Levenshtein	3	394	36	5
Mutation score	46.48	40.84	43.38	41.83
Cost	37	89	37	25

3.8 Threats to validity

Regarding external validity [73], which relates to the extent to which the results of our study can be generalized to other situations, there are two points to consider. First, as

mentioned in section 3.7.3, a GA is a stochastic process. Typically, during research, a GA is run a number of times and trends in the results are observed. We ran our GA a number of times on each case study and, although we do not report on trends, we observed results were similar over multiple runs. Therefore, for each case study we report on only one representative run.

The other threat to external validity is the size and number of case studies. As mentioned in section 3.7.1 we report on two case studies, which are admittedly small. But, we believe they are good representatives of what needs to be dealt with in state-based testing. First, similar size state machines are used in UML-based development, and in industrial case studies reported in the literature (section 3.7.1). Second, it is very uncommon in practice to model an entire system using a single large state machine. Third, these state machines are test models which usually represent a single test purpose [1]. Last, the second case study is a representative sample of a series of case studies used by others [30].

Internal validity relates to how well we have conducted the studies. The feasibility and similarity measures we have used and to what extent they represent feasibility of a test path or diversity in a test suite can be considered as a threat to internal validity. Regarding feasibility in addition to the fact that we use a surrogate measure of feasibility one may argue that summing up the values is not ideal from a measurement point of view because we have test suites of varying sizes. A test suite with 10 low feasibility test cases may appear to be better than a test suite with one test case with higher feasibility. The same critique can apply to similarity of whole test suite. A test suite with only one pair of very similar test paths (Similarity measure = S) will get the same score as a test suite with n similar test paths each having a similarity measure of S/n . However, others before us have used exactly the same measures [3], [30] and achieved convincing results. Also, as mentioned in section 3.7.6, test paths found by our GA are indeed feasible. We will experiment with different similarity measures in chapter 4. Another potential threat to validity is the way we implemented the random generation of test suites and compared them with our GA. We put a limit on the cost of randomly generated test suites so that they have the same distribution as the ones found by our GA and each time the cost of a

randomly generated test suite exceeded that limit we discarded that. Therefore, the process may not be totally random. But it was as realistic and comparable to the GA generation as possible.

3.9 Summary

In this chapter we presented the main contribution of this thesis, which is a multi-objective GA that generates whole test suites from an EFSM. After providing details on the implementation of our solution we used two different case studies to evaluate our approach and answer four different research questions discussed in section 3.7, showing that it is better (e.g., cost) to construct a test suite at once, as we do, rather than one test case at a time, the test suite our solution generate is made of feasible test cases, our solution performs better than a random search and other test suite construction techniques.

4 Measuring similarity in a test suite

One of the main factors affecting the success of any search based algorithm is (are) the objective function(s). Although the impact of different test case similarity measures on effectiveness at detecting faults has been already studied [3], it is not clear whether those results would apply to a multi-objective test suite construction. As discussed in section 3.8, we are looking for measures of similarity within a test suite other than using sum of pair-wise similarities of its test cases. In this section, we therefore investigate the effect of using different methods of measuring similarity in a test suite on our multi-objective GA for test suite generation from EFSMs.

We start by explaining how existing methods used by Hemmati et al [3] can be applied to our multi-objective GA in section 4.1 and then discuss new methods of computing diversity within a test suite driven from measures used in life sciences in section 4.2. We use the Cruise Control case study to compare, through statistical analysis, different measures in terms of cost and effectiveness of test suites generated by our multi-objective GA in section 4.3.

4.1 Applying existing measures to our Multi-Objective GA

As mentioned in section 3.4, our intuition, as well as others' [3], was that test paths should be as dissimilar as possible to increase effectiveness at fault detection. One of the four objective functions in our GA therefore computes a similarity value for a test suite. To obtain a similarity value for a set of test paths (test suite), similarly to others [3], we first compute the similarity between every unordered pair of test paths (genes) and then aggregate those pair-wise comparisons in a similarity value for a test suite (chromosome). Different pair-wise similarity measures can be used [3]. One of the measures, which is not limited to identical length sequences (we have variable length chromosomes), is the Levenshtein distance [55]. Although not the recommendation of Hemmati et al. (recall section 3.4) this is the best measure of similarity they studied [3] that supports variable length sequences. To change this distance measure into a similarity measure, similar to Hemmati et al. [3], we reward each match between two sequences by

one point and penalize mismatches and gaps by simply ignoring them (i.e., assigning no point).

The Hamming distance is another similarity measure that we used. It is one of the most used distance measures in the literature [3]. The Hamming distance is based on the edit distance. The edit distance between two strings is the minimum number of edit operations (i.e., insertions, deletions and substitutions) required to transform one string into the other. The Hamming distance can be used as a sequence based similarity measure between strings of equal length. However, for variable length strings it is used as a set based measure. To convert variable length strings into strings of equal length, similarly to Hemmati et al. [3], a binary string is constructed that represents which elements from the set of all legal elements of an encoding are present in a given string. In our case legal elements of the encoding are all the transitions of a given EFSM.

The last similarity measure that we considered is the set-based Gower-Legendre (Dice) measure since this is the one suggested by Hemmati et al [3]. As the Dice measure is a set based measure (section 2.3.2), the order of genes (in our case transitions) and the repetition of genes will be lost in computation. The Dice measure of two test paths f and g is based on commonalities and differences between the sets of symbols (in our case, transitions of the EFSM) of the test paths:

$$Dice(f, g) = \frac{|f \cap g|}{|f \cap g| + \frac{1}{2}(|f \cup g| - |f \cap g|)}$$

Similarly to Hemmati et al. [3], we defined the similarity of a test suite (chromosome) as the sum of the similarity measures computed for each unordered pair of test paths (genes) in that chromosome. The following objective function needs to be minimized:

$$similarity_{sum}(S) = \sum_{(f,g) \text{ unordered pairs in } S} similarity(f, g)$$

4.2 Methods from life sciences

As genetic algorithm has its roots in the science of biology and genetics we conjectured that using methods which biologists use to measure similarity from individuals to populations can result in more realistic computations of similarity. We therefore performed a short survey on methods used to measure diversity in the field of biology. The three major parameters utilized by biologists to express variation of a population are: diversity within a population, diversity among populations and distance between two populations. The first and second parameters are referred to as α - and β -diversity respectively and γ -diversity represents total diversity ($\alpha + \beta$) [74].

Kosman [74] divides methods of diversity analysis in plant pathology into five categories:

1. *Genotypic methods*: Genotypic methods were initially developed for measuring species diversity. They use information about relative frequencies of different genotypes in population. In the context of EFSMs, a genotype can be mapped to a transition, pair of transitions, paths, sub-paths or states. Many other mappings can be considered as well. In our GA encoding genotypes can be mapped to transitions in an EFSM.
2. *Gene methods*: These methods analyze gene variation in populations and are based on calculations of frequencies of alleles¹ at individual loci². Transitions can be considered as example of alleles in an EFSM. In this way each position of transitions in a path traversing an EFSM can be considered as loci.
3. *Genetic methods*: After measuring dissimilarity between pairs of individuals using any measure, the diversity of a population is calculated using a matrix of dissimilarities in which each row and each column represents an individual and each element is the pair-wise dissimilarity between two individuals. The

¹ An allele is an alternative form of a gene (one member of a pair) that is located at a specific position on a specific chromosome.

² Locus (plural : loci) is the location of a gene on a chromosome.

matrix can be used to simply compute average of dissimilarities or it can be used in a more sophisticated assignment based approach like Kosman's diversity within population (KW) that we will discuss later in this section.

4. *Functional methods*: Functional methods are mostly used in the study of ecosystems. Each organism (species) is described by a set of functional traits which are believed to be important for the function of an ecosystem. Differences between organisms are assessed based on dissimilarity between their functional profiles.
5. *True diversity methods*: The term true diversity was first coined by Jost [75], [76] and later the idea was complemented by Tuomisto [77]. True diversity is based on the notions of α -, β - and γ -diversity as well as intuitive ecological concept of diversity.

Each of the categories mentioned above is further divided based on the type of diversity parameters used to measure dissimilarity: i.e., diversity within population (α), distance between populations, diversity among populations (β) and total diversity (γ). A complete list of parameters belonging to each category can be found in Kosman's paper [74].

In our multi-objective GA we need to measure diversity within a population, i.e. α -diversity. Therefore, in the following paragraphs we briefly describe methods belonging to each of the five categories that are used to measure α -diversity.

According to Kosman [74] the three most commonly used *genotypic* methods to measure diversity within a population are Shanon Information, $Sh(P)$, Simpson diversity, $Si(P)$, and the Stoddard index, $St(P)$. In population P with q_r as the frequency of genotype r and s as the total number of genotypes these measures are defined as follows:

$$- Sh(P) = - \sum_{r=1}^s q_r \ln q_r \quad (1),$$

$$- Si(P) = 1 - \sum_{r=1}^s q_r^2 \quad (2),$$

$$- St(P) = \frac{1}{\sum_{r=1}^s q_r^2} \quad (3).$$

Nei [78] has defined the most common *gene* method [79] for the assessment of diversity within a population. This method is based on allele frequencies. In a population P with a total number of T loci, A_j alleles at locus L_j ($j = 1, 2, \dots, T$) and q_{ij} as the frequency of the i^{th} allele at locus L_j ($i = 1, 2, \dots, A_j$), Nei's measure is defined as:

$$H(P) = \frac{1}{T} \cdot \sum_{j=1}^T H_j(P) = \frac{1}{T} \cdot \sum_{j=1}^T \left(1 - \sum_{i=1}^{A_j} q_{ij}^2 \right) \quad (4)$$

The average dissimilarity within population (ADW) and Kosman's diversity within population (KW) are two common *genetic* methods [79]. In a population P with n individuals $x_1, x_2, \dots, x_n$ and $\delta(x_i, x_j)$ representing dissimilarity between x_i and x_j , the average dissimilarity within population, using the matrix of dissimilarities defined earlier, is calculated using the following formula:

$$ADW(P) = \frac{1}{n^2} \cdot \sum_{i,j=1}^n \delta(x_i, x_j) \quad (5)$$

Kosman's diversity within population [79] is defined using the idea of optimal overall matching of individuals in a population. In a population of size n , individuals are matched to form $n/2$ pairs in a way that the sum of dissimilarities between the pairs has its maximum value. Finding those pairs is similar to solving the assignment problem [80], which is finding a maximum weight matching in a weighted bipartite graph. Kosman's diversity is calculated by dividing the sum of dissimilarities between matched pairs by the number of matched pairs:

$$KW(P) = \frac{1}{n/2} \cdot ASS_{max}^{\delta}(P, P) \quad (6)$$

Rao has proposed the quadratic entropy as an index of *functional* diversity within a population [81]. In population P with q_i as the frequency of genotype i , t as the total number of genotypes in P and δ_{ij} as the dissimilarity between genotypes i and j , the quadratic entropy with regard to dissimilarity δ is defined as:

$$Q_{\delta}(P) = \sum_{i,j=1}^t \delta_{ij} q_i q_j \quad (7)$$

Two different approaches to generalize Shannon's entropy, originally a genotypic measure, as a functional method to measure diversity within populations have been proposed by Ricotta et al. [82], [83]. In the first approach [82] each genotype is assigned a weight based on its importance in the data set:

$$Sh^w(P) = - \sum_{i=1}^t w_i q_i \ln q_i \quad (8)$$

The weight w_i corresponds to the i^{th} genotype and represents the total dissimilarity between the i^{th} genotype and all other genotypes in the data set. Similarly to the previous measure, q_i is the frequency of genotype i in the set of t genotypes.

In the second approach to extend Shannon's entropy to a functional diversity measure, Ricotta and Szeidl [83] use the following representation of the Shannon index:

$$Sh(P) = - \sum_{i=1}^t q_i \ln(1 - \sum_{j=1, j \neq i}^t q_j) \quad (9)$$

In the above equation, $\ln(1 - \sum_{j=1, j \neq i}^t q_j)$ is the rarity function which represents how difficult it is to find genotype i with relative abundance q_i . A second generalization of the Shannon entropy is defined by integrating the dissimilarity function with the definition of the rarity function. The following equation is used to calculate this second measure of functional diversity based on Shannon's entropy:

$$Sh_p(P) = - \sum_{i=1}^t q_i \ln(1 - \sum_{j=1}^t \delta_{ij} q_j) \quad (10)$$

Jost has proposed a conversion of common diversity measures to *true diversity* [75]. As an example, the true diversity for the Shannon's entropy is defined as its exponent:

$$\exp(Sh) = \exp(- \sum_{r=1}^t q_r \ln q_r) \quad (11)$$

Kosman and Leonard have performed a conceptual analysis of different types of methods mentioned above [79]. They compared those measures on organisms with asexual or mixed mode of reproduction. They used studies of plant pathogenic fungi for their experiments. Based on their study, Kosman's diversity within population (Eqn 6) is

more appropriate and informative than other measures as it includes both genotypic structure of a population and measure of dissimilarity between various genotypes. Another interesting measure to consider based on this study is the average dissimilarity within population (Eqn 5) as a representative of all the measures that are based on allele frequencies (e.g., Nei's measure, Eqn 4).

Therefore we selected Kosman diversity within population (Eqn 6), which we refer to as KW, and average dissimilarity within population (Eqn 5), which we refer to as ADW, to apply to our multi-objective GA. Both methods are based on a function that computes pair-wise dissimilarity between two individuals (δ). These two methods in addition to summing up pairwise similarities (SUM) are the three methods we used to compute dissimilarity within population. We used the three functions mentioned in section 4.1, i.e., Leveneshtein distance, Hamming distance and Dice, as the dissimilarity functions required by SUM, KW and ADW, resulting in nine different combinations to experiment with. Note that Kosman and Leonard have also used the Dice measure in their experiments. Because these two methods are based on pair-wise dissimilarities rather than similarities we used the original version of all these three measures without converting them to similarity measure.

4.3 Comparing different measures using statistical analysis

As mentioned in the two previous sections (sections 4.1 and 4.2) we used three different measures to compute pair-wise similarity (dissimilarity) among test paths and three different methods to compute similarity (diversity) within a whole test suite (chromosome). This resulted in nine different implementations of our multi-objective GA. As our intuition as well as other's [3] was that diversity within a test suites is related to its cost and effectiveness at finding faults, we designed and implemented a statistical analysis procedure to investigate the effect of using different similarity measures on cost and effectiveness of test suites generated by our multi-objective GA. In the following sections we discuss the research questions (section 4.3.1) as well as the statistical analysis procedure we designed to answer our research questions (section 4.3.2). As we used ANOVA (Analysis of Variance) in our statistical evaluation we briefly describe this test in section 4.3.3 and discuss its use in our context in section 4.3.4. We will conclude this

section by discussing results and threats to validity in sections 4.3.5 and 4.3.6 respectively.

4.3.1 Research questions

The main research question we were trying to answer was: what is the effect of using different methods of computing similarity on cost and effectiveness of test suites generated by our multi-objective GA? To be more specific we then broke down this question into the following four research questions:

RQ 1. What is the effect of using different pair-wise dissimilarity measures on cost?

RQ 2. What is the effect of using different pair-wise dissimilarity measures on effectiveness?

RQ 3. What is the effect of using different methods of computing diversity on cost?

RQ 4. What is the effect of using different methods of computing diversity on effectiveness?

4.3.2 Statistical Procedure

We used the Cruise Control case study. As mentioned in section 3.7, all the paths in the corresponding state machine are feasible. Therefore, we could focus on other objectives, which are cost and similarity. We designed the following analysis procedure to answer our research questions:

1. Run each of the nine different versions of the GA 100 times;
2. Randomly select samples of size 20 out of 100 executions and collect the resulting Pareto fronts (i.e., final solutions found by the GA);
3. Select the adequate test suite with the lowest cost from each Pareto front (i.e., 20 different test suites are collected for each of the nine different implementations of the GA). We obtain 180 (20 times nine) different test suites;

4. Compare different versions of the GA in terms of cost using Analysis of Variance (ANOVA);
5. Execute these test suites on mutants and measure mutation scores;
6. Compare different versions of the GA in terms of mutation score using Analysis of Variance (ANOVA).

4.3.3 Analysis of Variance (ANOVA)

The analysis of variance (ANOVA) [84] refers to a group of statistical procedures for the analysis of quantitative responses from experimental units. Single-factor or one-way ANOVA is a type of ANOVA. It is the analysis of data sampled from more than two populations or data collected from experiments in which more than two treatments have been applied. The characteristic that separates different treatments or populations from one another is referred to as the *factor* under study and each treatment or population is called a *level* of the factor [84]. The goal of ANOVA is investigating if difference between different samples is due to chance or caused by different treatments. Assuming there are n different populations or treatments each having mean of μ_i , the null hypothesis (H_0) is that all the mean values are equal, i.e.: $H_0 : \mu_1 = \mu_2 = \dots = \mu_n$; and the alternate hypothesis (H_a) is that at least two of the mean values are different.

Two-way ANOVA is used to compare populations when there are two factors under study. In one-way ANOVA, populations are classified based on a single factor. In the two-way ANOVA, there are two factors, each with its own number of levels.

In our experiment we have two factors under study. The first factor is the pair-wise similarity (dissimilarity) measure, which has three different levels, i.e. dice, hamming distance and leveneshtein distance. The second factor is the method for measuring diversity within a whole test suite, which has two levels: KW and ADW. Each of these factors has three different levels (i.e. three different pair-wise similarity/dissimilarity measures and three different methods to compute similarity/diversity within whole test suites). Therefore, we performed multiple single

factor ANOVAs as well as a Two-way ANOVA for each dependent variable (cost and effectiveness).

As the goal of our experiment is investigating the effect of similarity measure on cost and effectiveness, the null hypothesis will be that changing similarity measure will not change the mean value of cost and effectiveness in a randomly selected sample of solutions found by GA.

Table 8 presents a sample ANOVA table. In addition to well-known statistical measures like sum, average and variance, a typical tool supporting ANOVA provides:

- Sum of squares (SS): Sum of squared deviations from mean;
- Degree of freedom (df): The number of values that are free to vary. For example in our one-way ANOVA, df between groups is two and df within groups is 57;
- Mean square (MS): This is defined as $MS = SS/df$;
- P-value: Quantifies strength of evidence against null hypothesis. Small p-value (less than 0.05) means strong evidence against null hypothesis;
- F and F_{crit} : These values are used in an F-test to reject the null hypothesis.

Table 8-Sample ANOVA table

SUMMARY						
<i>Groups</i>	<i>Count</i>	<i>Sum</i>	<i>Average</i>	<i>Variance</i>		
leveneshtein	20	1184	59.2	44.27368		
hamming	20	1126	56.3	48.22105		
dice	20	959	47.95	27.41842		
ANOVA						
<i>Source of Variation</i>	<i>SS</i>	<i>df</i>	<i>MS</i>	<i>F</i>	<i>P-value</i>	<i>F crit</i>
Between Groups	1364.633	2	682.3167	17.07027	1.55E-06	3.158843
Within Groups	2278.35	57	39.97105			
Total	3642.983	59				

We used Microsoft excel to perform ANOVA. In excel treatments are referred to as groups. Therefore, SS_T is the sum of squares for the total sample, i.e. the sum of the

squared deviations from the grand mean. SS_W is the sum of squares within the groups, i.e. the sum of the squared means across all groups. SS_{Bet} is the sum of the squares between group sample means, i.e. the weighted sum of the squared deviations of the group means from the grand mean. In the results of two-way ANOVA there are rows labeled as sample, column and interaction. The “Sample” stands for the row factor, which in our case, is the method to aggregate pair-wise similarities into a single value representing whole test suites (i.e. Sum, KW, ADW). The “Columns” represents the “pair-wise” similarity factor and "Interaction" is the combination of the two factors. You can refer to appendix E for examples of two-way ANOVA tables generated by excel.

We will also use the Omega test to measure effect size, which presents the strength of a treatment effect and is independent of sample size. Omega, which is defined as an unbiased measure of effect size [85] is given by the following formula.

$$\omega^2 = \frac{SS_{Bet} - df_{Bet}MS_W}{SS_T + MS_W}$$

Elements of the above formula are coming directly from ANOVA tables. SS_{Bet} is the sum of squares between groups and SS_T is the total sum of squares. MS_W is the mean square within groups and df_{Bet} is the degree of freedom between groups. In general, Omega is considered as a more accurate measure of the effect size compared to other measures like Cohen’s d measure. $\omega^2 = .01$ depicts a small effect and $\omega^2 = .06$ and $.14$ depict medium [85] and large effects respectively.

4.3.4 Using ANOVA in our experiments

We used the Cruise Control case study (section 3.7.1) in our experiment and applied the above-mentioned procedure. As mentioned in the previous section we had two different factors under study and each of them had three different levels. Cost and effectiveness were the two dependent variables. Following the statistical procedure described above resulted in three different single factor ANOVA tests with different methods of computing pair-wise similarity as the factor under study and cost of generated test suites as dependent variables. It also resulted in three single factor ANOVA tests with different methods of computing diversity within whole test suite as the factor under

study and cost of generated test suites as dependent variable. Replacing cost with the effectiveness at detecting faults (mutation score) as the dependent variable resulted in six additional single factor ANOVAs (three for pair-wise similarity measure as the ANOVA factor and three with method of computing diversity within whole test suite as the factor under study).

Combining two factors resulted in two different two-way ANOVAs, one of them with cost and the other with effectiveness at detecting faults as the dependent variables.

Results of all 12 different one-way ANOVA tests and the two two-way ANOVA tests are included in Appendix D.

Two assumptions about the data being used in ANOVA are that each sample has a normal distribution and the standard deviations are close enough (i.e. the largest standard deviation is not greater than twice the smaller one). As you will see in Figure 13 and Figure 14 there is no major skewedness in the box plots representing data used in our experiments, which means data have a normal distribution. Also as summarized in tables Table 9 and Table 10 standard deviations of different samples are close enough (the largest standard deviation is not greater than twice the smaller one) to each other. Therefore, we can conclude that the data we used in our experiment satisfies ANOVA assumptions. Note also that ANOVA has proved to be very robust to departures from these assumptions [86].

Table 9- Mean and variance of cost values

	Hamming			Levenshtein			Dice		
	ADW	KW	SUM	ADW	KW	SUM	ADW	KW	SUM
Average	69.1	56.3	51.1	65.95	59.2	48.35	51.85	47.95	44.9
Variance	108.72	48.22	30.09	91.31	44.27	23.82	29.08	27.42	32.73
SD (σ)	10.42	6.94	5.86	9.55	6.65	4.88	5.39	5.24	5.72

4.3.5 Results

In this section we separately study the effect of different configurations of our GA on cost (section 4.3.5.1) and effectiveness at finding faults (section 4.3.5.2).

Figure 13 – Box plot comparing costs

4.3.5.1 Effect on cost

Results of the experiment with nine different configurations of our GA are presented in Table 9. As mentioned earlier these nine different flavors are results of using different pair-wise similarity measure and aggregation method (for the whole test suite). Each column represents one configuration and its effect on cost in terms of average, variance and standard deviation of the 20 test suites selected for each of them.

Figure 13, presents the same result from another point of view. It is a box plot representing the distribution of cost values for each implementation of our GA.

When comparing different samples of data (i.e. the 20 test suites collected for each method after step 3 of the procedure mentioned before) in terms of cost (i.e. cost was the factor under study), the null hypothesis was rejected in all the six one-way ANOVA tests. This means, there is a relation between cost of test suites generated by our GA and the method used to compute diversity/similarity within test suites (Dice vs. Hamming vs. Levenshtein), regardless of the method to aggregate pair-wise comparisons for computing diversity for an entire test suite (ADW, KW, SUM).

We also observed that among the three different measures we used to compute pair-wise similarities/dissimilarities, the Dice (gower-legendre) measure resulted in test

suites of lowest cost (i.e. average cost of test suites was lower than the two other measures) regardless of the method used to compute diversity/similarity within whole test suites. This is in accordance with what Hemmati et al. have reported [3].

When comparing average cost of test suites based on the method used to compute diversity/similarity within whole test suites, summing up all the pair-wise similarities had the lowest cost value regardless of the method used to compute pair-wise similarity/dissimilarity. This means that the way methods from life sciences promote diversity within whole test suites results in test suites of higher cost. Results of this experiment are summarized in the box plot of Figure 13 as well as Table 9.

If we study effect size, which depicts the strength of a treatment effect and is independent of sample size, we come up with the following conclusions.

Following are the effect size for different one way ANOVAs we performed to investigate effect of size:

- Independent variable: pair-wise measure, test suite diversity: ADW $\rightarrow \omega^2 = 0.41$
- Independent variable: pair-wise measure, test suite diversity: KW $\rightarrow \omega^2 = 0.35$
- Independent variable: pair-wise measure, test suite diversity: SUM $\rightarrow \omega^2 = 0.16$
- Independent variable: test suite diversity, pair-wise measure: dice $\rightarrow \omega^2 = 0.20$
- Independent variable: test suite diversity, pair-wise measure: hamming $\rightarrow \omega^2 = 0.48$
- Independent variable: test suite diversity, pair-wise measure: Leveneshtein $\rightarrow \omega^2 = 0.49$

As mentioned earlier we also put the two independent variables (test suite diversity and pair-wise measure) together and performed two-way ANOVA. Results of this analysis can be found in appendix E. For the Two-way ANOVA we had $\omega^2 = 0.30$ for the effect size.

All of these values are greater than 0.14 which means similarity measurement method has a considerable effect on the cost of test suites.

Table 10 – Mean and variance of mutation scores

	Hamming			Leveneshtein			Dice		
	ADW	KW	SUM	ADW	KW	SUM	ADW	KW	SUM
Average	98.65	98.31	97.90	98.95	98.24	97.84	98	98.15	97.77
Variance	1.34	1.31	2.96	1.83	2.51	2.57	2.47	2.93	2.16
SD (σ)	1.16	1.14	1.72	1.35	1.58	1.60	1.57	1.71	1.47

4.3.5.2 Effect on mutation score

Similar to the previous section, we have summarized the results of the experiment in Table 10 and Figure 14. With mutation score as the factor under study the results of one-way ANOVA tests were different. In all the six different scenarios (i.e. three different pair-wise similarity/dissimilarity measures and three different methods to compute similarity/diversity within whole test suites) we were not able to reject the null hypothesis.

Figure 14- Box plot comparing mutation scores

The average mutation scores in each case are very close to each other. In other words changing the method of computing diversity/similarity does not change the

mutation score significantly. Considering the fact that all the test suites selected for our statistical analysis were adequate test suites this is not far from expectations. As all the test suites are adequate their effectiveness at detecting faults is very close to each other. Results of this experiment are summarized in the box-plot of Figure 14 as well as Table 10. As mentioned in section 4.3.3, we used the Omega test to measure effect size. Effect size presents the strength of a treatment effect and is independent of sample size. Therefore, we also used effect size to make sure that the results of ANOVA are valid:

- Independent variable: pair-wise measure, test suite diversity: ADW $\rightarrow \omega^2 = 0.04$
- Independent variable: pair-wise measure, test suite diversity: KW $\rightarrow \omega^2 = 0.01$
- Independent variable: pair-wise measure, test suite diversity: SUM $\rightarrow \omega^2 = 0.03$
- Independent variable: test suite diversity, pair-wise measure: dice $\rightarrow \omega^2 = 0.05$
- Independent variable: test suite diversity, pair-wise measure: hamming $\rightarrow \omega^2 = 0.02$
- Independent variable: test suite diversity, pair-wise measure: Leveneshtein $\rightarrow \omega^2 = 0.05$
- Two-way ANOVA: $\omega^2 = 0.01$

All of the values above are less than 0.06, which means there is a small effect size. Therefore, we can conclude that changing the method of computing similarity does not have a statistically nor a practically significant effect on the mutation score.

4.3.5.3 Qualitative analysis of results

In section 4.3.5.1, we showed there is a relation between cost of test suites and the method used to compute similarity. In this section we characterize this relation.

Figure 15-Relation between cost of test suites and method used to compute similarity

As mentioned earlier Dice, as a pair-wise similarity measure resulted in test suites of lower costs compared to other measures. Also we observed summing up pair-wise similarities is simply the best way of aggregating those pair-wise measures into a single value representing the whole test suite. As Figure 15 suggests in addition to those two observations there is a trend among different methods of computing similarity within whole test suites. As we move from Hamming to Levenshtein and then Dice there is a monolithic decrease in cost of test suites with the exception of KW. Considering the fact that KW measure uses maximum distance to match pairs, this is not surprising. Both dice measure and hamming distance (as implemented by Hemmati and in this thesis) are set based measures while Levenshtein distance is a sequence based measure. Therefore, it results in longer sequences of transitions (test paths) to maximize KW distance which increases cost of generated test suites.

We studied nine different measures of test suite diversity: ADW-H, ADW-L, ADW-D, KW-H, KW-L, KW-D, SUM-H, SUM-L, SUM-D. They are currently used for test suite construction. They could also be used for test suite evaluation. For instance, since SUM-D appears to be the cheapest and as effective as the other ones, one could consider evaluating the ADW-H, ADW-L, ADW-D, KW-H, KW-L, KW-D, SUM-H, SUM-L test suites with SUM-D. This can indicate how good those other measures are in representing diversity in a test suite. For example if between two test suites (TS_1 and TS_2)

one of them (e.g., TS_1) has a better ADW-H it will also have a better SUM-D. In other words, does $SUM-D(TS_1) > SUM-D(TS_2)$ imply $ADW-H(TS_1) > ADW-H(TS_2)$? This will be studied in future work.

In mutation analysis there were mutants that none of the test suites could kill (i.e. equivalent mutants). This could be due to the fact that we used transition coverage and mutation operators (like Literal Value Replacement) that do not change in the sequence of transitions triggered by a test case could not be killed by any of the test suites. This is the reason that all the mutation scores are high. As mentioned there is not much variation in mutation scores because all the test suites we used were adequate and therefore were as effective as each other in detecting faults.

To conclude, as changing the method of computing diversity does not have a significant effect on the mutation score, the method which results in test suites of lower cost is a better one in terms of cost and effectiveness. Therefore, the dice measure is the best measure to compute pair-wise similarities/dissimilarities in terms of cost and effectiveness. Regarding the method of computing diversity within whole test suites summing up all the pair-wise values had better results than the two methods from life sciences (ADW, KW) regarding cost and effectiveness.

4.3.6 Threats to validity

Regarding threats to conclusion validity, there are two points to consider. The first threat is due to the stochastic nature of genetic algorithms. To avoid this threat we executed each different version of our multi-objective GA 100 times before starting the statistical analysis. As mentioned in section 4.3.2 each sample had 20 Pareto front solutions in it. Also to be fair in our comparisons, from each Pareto front we selected the adequate test suite with the lowest cost. This means we picked the best solution in terms of cost and coverage for all different flavors of our GA.

Second threat can be size of case study. As mentioned in section 3.7.1, the size is admittedly small but is representative of what is usually used in state based testing. Similar size state machines are used in model-based development, and in industrial case

studies reported in the literature (section 3.7.1). Also it is very uncommon in practice to model an entire system using a single large state machine.

We tried to minimize construct threats by the way we designed our experiment. We ran our GA 100 times and used ANOVA to perform statistical analysis. As mentioned earlier our data satisfy ANOVA assumptions and ANOVA has proved to be quite robust to deviations from these assumptions.

Threat to internal validity of our experiment is the fact that we used mutation score as a surrogate measure of effectiveness at detecting faults, i.e. we replaced real faults with manual seeded ones. As a test suite's mutation score has proved to be correlated with its real fault detection rate [87] the probability that this threat affects the validity of our results will not be significant.

Regarding external validity, which can limit the extent to which results of our experiments can be applied to similar situations, we need to consider the fact we performed our experiment in the context of our multi-objective GA. As mentioned earlier we consider four different objectives that impact each other. Therefore, they might affect the factors we considered in this study (cost and effectiveness in detecting faults). One studying the relation between diversity and cost and fault detection in the context of another test generation approach may get different results. We tried to minimize this threat by using the Cruise Control EFSM in which all the path are feasible so that feasibility didn't affect results of our experiment.

4.4 Summary

In this chapter we investigated the effect of using different methods of computing similarity on our multi-objective GA. We used three different measures to compute pair-wise similarities between test cases of a test suite and three different methods of aggregating those pair-wise values into a single value that represents similarity within a whole test suite. We used analysis of variance (ANOVA) to analyze the results we got from the Cruise Control case study. We noticed changing the similarity computation method affects cost of generated test suites without having a major effect on the fault detection rate (potentially because all the optimal test suites that we studied were

adequate). Therefore, we came to a final conclusion that a method which results in a test suites with lowest cost is the best one to use in our multi-objective GA.

5 Counter Problem

As discussed in previous chapters, we argue that generating a test suite for an EFSM is a multi-objective optimization problem. Solving this multi-objective problem gets more complicated when there are guard conditions that refer to counter variables, i.e., variables that count how many times a behavior, as simple as a transition or as complex as a sub-path in the EFSM graph, needs to be repeated in order to trigger another behavior (transition). The counter problem has different manifestations. In its simplest form, a single transition should be triggered a particular number of times, in order to trigger another transition, this last condition being guarded by a counter variable. More complex situations involving several counter variables or longer sub-paths to be repeated may also be encountered. A discussion of those manifestations, of different complexity, is one of the contributions of this chapter. To the best of our knowledge, this is the first attempt to discuss a taxonomy of counter problems. We believe this is an important contribution of this chapter since other researchers will then be able to precisely compare techniques, to precisely understand under which conditions test suite construction techniques work (or not) and to complement the taxonomy (we do not argue this is a definite discussion of the counter problem).

Our solution discussed in Chapter 3 does not specifically account for the counter problem, though it may be able to handle some manifestations of the counter problem (for instance the Transport Protocol case study is a manifestation of the counter problem). Others before us have proposed a solution to the counter problem [88]. It is however a solution to be used when constructing one test case at a time, without accounting for conflicting objectives. It therefore, makes sense to combine these two solutions: multi-objective construction of a test suite as discussed in this thesis while accounting for the counter problem as discussed by others. This counter-aware, multi-objective test suite construction is another contribution of this chapter. Recognizing that the existing solution to handle counters [88] has limitations, we suggest a new mechanism to handle counters and integrate it in our multi-objective test suite construction (Chapter 3). This second counter-aware, multi-objective test suite construction is yet another contribution of this chapter. Our last contribution is a comparison of those two counter-aware, multi-

objective test suite construction techniques on three synthetic, but realistic EFSMs exhibiting different manifestations of the counter problem. In doing so, we discuss how the presence of counter variables in an EFSM affects the success of test suite construction at finding feasible (executable) test suites.

To summarize the contributions of this chapter, we provide: (1) a precise definition of the counter problem and a first (to the best of our knowledge) attempt at classifying different types (with corresponding EFSMs) of the counter problem. This should become a benchmark others can use to evaluate alternative solutions; (2) an integration of an existing counter-aware technique with our method of constructing an entire test suite from an EFSM; (3) a new solution to the counter problem which is also integrated into our previous technique to create entire test suites; (4) a comparison of those two counter-aware, multi-objective test suite construction techniques on three synthetic, but realistic EFSMs.

The rest of this chapter is structured as follows. In section 5.1 we discuss a taxonomy of counter problems. Section 5.2 shows that the synthetic examples we use to illustrate different manifestations of the counter problem in our taxonomy can be found in practice. Section 5.3 describes how we account for the counter problem in multi-objective construction of test suites from an EFSM, discussing two different solutions as mentioned earlier. Section 5.4 presents the empirical evaluation we performed to assess those two solutions. Section 5.5 summarizes this Chapter.

5.1 A Taxonomy of counter problems

In this section we precisely define the counter problem and propose a first (to the best of our knowledge) taxonomy of counter problems, discussing several manifestations of the problem of different complexities.

Kalaji and colleagues define a counter variable as a variable that counts how many times another transition (at least) needs to be repeated [88]. This is also the definition Binder assumes when extending the W-method for test suite construction [89]. We believe this definition is a bit restrictive since what gets repeated and counted is not necessarily restricted to a single transition but could involve a sub-path in the entire

EFSM. This has also been confirmed by our own experience at test case construction from an EFSM. To reason about counters and the counter problem, Kalaji and colleagues classify update functions into those that update a variable by using (i) an input parameter, (ii) a variable, or (iii) a constant, and further classify the second type (updating with a variable) into those that (a) increment by a constant value, (b) decrement by a constant value, (c) multiply by a constant value or (d) divide by a constant value. As argued by the authors, this is a simplification of the problem they made to be able to obtain acceptable solutions to the problem of constructing feasible (i.e., executable) transition sequences (i.e., test cases). Indeed, a variable can in theory be updated in a transition by using more complex (linear or not) functions of variables, inputs and constants, though one may argue this would rarely happen. For instance, in the context of UML [90] based software development, actions and guards in a state machine can be specified with any syntactically correct OCL [90] expression (see for instance an attempt at accounting for those expressions during state-based test construction in [90]).

Our own experience with EFSMs that exhibit a counter problem shows that the problem can manifest itself in varying ways, of varying complexities. In other words, solutions such as Kalaji et al's, to find executable test paths from an EFSM while accounting for the counter problem have varying difficulties solving the problem depending on the kind of counters the EFSM has. We therefore, decided to categorize different manifestations of the counter problem. This is, we believe, an important contribution of this work since classifying occurrences of the counter problem should help compare test case generation techniques and understand why, or under which circumstances, some work or not. For instance some EFSM-based test case construction techniques may handle simple manifestations of the counter problem but fail to handle more complex situations; understanding when techniques succeed or fail will help improve test case construction techniques.

In accordance with, and extending Kalaji et al.'s discussion, we consider an EFSM exhibits a counter problem when there exists a sub-path p in the EFSM graph such that repeating p a specific number of times n enables a transition t in the EFSM, i.e., it is necessary to repeat p n times to eventually make the condition guarding t true.

We denote a path p in the EFSM as a sequence of transitions $t_1, t_2, \dots, t_m$, m being the length of p , such that the final state of t_i is the initial state of t_{j+1} , $i=1, \dots, m-1$. The initial (resp. final) state of the path is t_1 's initial state (resp. t_m 's final state).

In general, a path in the EFSM enabling transition t with a counter variable has the form: $q, (r, p, s)^n, o, t$; where q, r, p, s , and o are sub-paths of the EFSM graph such that q 's initial state is the initial state of the EFSM, q 's final state is r 's initial state, r 's final state is p 's initial state, p 's final state is s 's initial state, s 's final state is r 's and o 's initial state, and o 's final state is t 's initial state; sub-path r,p,s is repeated n times. Note that each time p is repeated, its prefix (r) and suffix (s) in the repetition are not necessarily the same during each repetition; What matters is that it is necessary to repeat p n times to eventually enable t .

In its simplest form, which we call Level1, r, s , and o are empty and p 's length is one: p 's final state is t 's initial state. This corresponds to Kalaji et al.'s definition of the counter problem and is illustrated by the EFSM of Figure 16. In this example, the transition guarded (t) is the transition from State1 to State2; its guard depends on variable c , which is incremented by the self-transition on State1 (p); Here, p and t are triggered by the same input, $e1$, though this is not necessarily the case in other instance of a simple counter problem.

Figure 16 – Level1 of the counter problem (synthetic example)

A more complex manifestation of the counter problem is illustrated by the EFSM of Figure 17, which we call Level2. The transition guarded by counter variable $C2$ is $T9$. Enabling $T9$ requires repeating $T8$ but enabling $T8$ requires that $T5$ be repeated. In this case, p 's length is one, like the simplest case, but r is itself a repetition, ending with $T6$. In fact, $T6$ is guarded by counter variable $C1$. In other words, one counter variable's value ($C2$'s), necessary to enable a transition, depends on another counter variable ($C1$).

Figure 17 - Synthetic EFSM with a Level2 counter variable

A synthetic example of a yet more complex manifestation of the counter problem (Level3), with four counter variables (C1, C2, C3 and C4), appears in Figure 18. The path enabling the transitions guarded by C1, C2 and C3 are similar to the first (simplest) example above: r, s, and o are empty and p's length is one (e.g., T5 which updates C1). T6's guard condition requires that T5 be repeated three times before it can be triggered. C4 illustrates a more complex counter problem since p is T21, r is made of three repetitions of T17, followed by T18. T17 updates the value of C3 and it should be repeated three times before T18 can be triggered. In other words C4 depends on C3.

These three EFSMs provide a first attempt at describing a taxonomy of manifestations of the counter problem in EFSMs.

We will use these three examples in section 5.4, where we evaluate our solution to the counter problem and compare it with an existing solution.

Figure 18 - Synthetic EFSM with multiple counters (Level3)

5.2 Real examples matching synthetic ones

In this section we illustrate the two first manifestations of the counter problem discussed above on real examples.

The Level1 of the counter problem of Figure 16 is illustrated by the EFSM of Figure 19, which can typically be obtained when modeling a bounded stack for instance. Another example of this simple counter problem is an ATM that allows a maximum number of erroneous attempts at providing a PIN [88]. One transition guarded by a counter variable (currentIndex) is T6 and repeated sub-path p is reduced to T3 (Figure 19). Here, p and t are triggered by the same input, push(), though this is not necessarily the case in other instance of a simple counter problem. Another transition guarded by a counter variable (again currentIndex) is T5, which requires that sub-path T4 be repeated.

Figure 19 - EFSM of a simple stack

Although Figure 17 is a synthetic example, it is similar to the model of a resizable, bounded data structure [64]: see Figure 20. States Empty, PartiallyFilled and Filled represent the usual states of a fixed size data structure whereby the data structure starts in the Empty state and moves to PartiallyFilled and eventually Filled as elements are added (the data structure also accommodates removals so going back to empty is possible). What is different from a standard data structure is that this one can resize itself: The state can change from Filled to PartiallyFilled with the add() event. The data structure can only resize a maximum number of times, after which the state becomes Overflow. In this example, the variable counting the number of resize times depends on the variable counting the number of elements in the data structure.

Figure 20 – EFSM of a bounded data structure [64]

5.3 The counter problem and test generation from an EFSM

The presence of counter variables can make the process of finding feasible paths in an EFSM complicated. In its simplest form one or more counter variables are independent of each other and the transition(s) affecting counter variables should be repeated a specific number of times to enable guard conditions referring to those counter variables. As we discussed in section 5.1 the counter problem gets more complicated when counter variables depend on each other. This can get even more complicated when there are sequential constraints among transitions as well, i.e., when a specific behavior, triggering a couple of different sub-paths in the EFSM, needs to be triggered in order to eventually enable a specific transition. For example, our last example illustrates this more complex manifestation of the counter problem where there is more than one type of counter variable in a single EFSM. In addition there are sequential constraints among different transitions as well. Using this example we will be able to verify how our solution(s) can handle the EFSM with multiple counter variables that are dependent on each other.

Since the multi-objective GA discussed previously in Chapters 3 and 4 do not specifically account for counter variables, and since Kalaji and colleagues have proposed a solution to account for such variables, we first discuss how we extended our GA with their solution (section 5.3.1). We also discuss a different method we propose to extend the GA to specifically account for counter variables (section 5.3.2). We then discuss the setup of the GAs (section 5.3.3).

5.3.1 Extending our GA using Kalaji et al.'s solution

Feasibility is one of the four fitness functions of our multi-objective GA. Kalaji et al. [88] defined a specific fitness function to account for the counter problem. When there are counter variables in an EFSM in order to make a test path TP, that includes transition t which is guarded by a counter variable, feasible, other transitions that initialize the counter variable (initializer) and update (updater) its value should be included before t in TP. Therefore, Kalaji and colleagues first identify which transitions reference counter variables. They have implemented a function that finds initializers and updaters for the

counter variable in t 's guard as well as the number of times updaters should be included in a test path. For each counter variable this function finds triplets of the form (initializer, updater, updater times) that satisfy the guard condition of t . These triplets are used by another function which checks if the initializer and updater come in the appropriate order in a given test path and if the updater is included in the test path the exact number of times given in the triplet. If the given test path does not satisfy these conditions a very large penalty is added to the corresponding feasibility fitness function value (i.e. the path is marked as infeasible).

Updating the feasibility fitness function we discussed in section 3.4, this way, to account for the counter problem, has several limitations. The GA will not necessarily be guided towards acceptable solutions in an efficient manner because solutions are either marked as feasible or infeasible and the ones that have a higher chance of being feasible are not rewarded by the fitness function. In other words, the fitness function does not help the GA navigate through the solution landscape because the landscape created by the fitness function has high picks and is otherwise relatively flat. Also, creating a list of acceptable triplets of (initializer, updater, updater times) is time consuming, as this is done every time the fitness function is called. Additionally, the way they count updaters does not account for all the possible situations. As an example, consider the case where a guard condition performs a comparison using the “greater than” operator (“>”) and the updater increases the value of the counter variable by one. The list of triplets proposed by Kalaji et al. only includes the minimum number of updater repetitions that is required to meet the guard condition. Any test path in which the updater is repeated more than that minimum number of times is an acceptable solution as well but will not be marked as feasible by their algorithm as the corresponding triplet is not in the list. Their solution only accepts solutions with the exact number of updater repetitions. Also their algorithm only looks for definition-clear paths, w.r.t. the counter variable, from the updaters to the guarded transition, while paths that are not necessarily definition-clear could enable the guarded condition too. Plus, their solution requires that all the occurrences of a specific updater be consecutive. Recall from section 5.1, that we defined a path in the EFSM enabling transition t with a counter variable as: $q, (r, p, s)^n, o, t$. The discussion above about Kalaji et al.'s solution to the counter problem means that they have only considered

the simple case where r and s are empty, p is a single transition and o is definition clear with respect to the counter variable used in t 's guard condition.

We nevertheless extended our original GA by replacing its feasibility fitness function (section 3.4) with the one of Kalaji et al [88]. We will refer to this four-objective GA as KHS-GA in the reminder of this paper.

5.3.2 Extending the GA with a new solution

In order to avoid limitations discussed in the previous section, we propose another solution to the counter problem. In our solution a new fitness function is added to specifically account for the counter problem, making the search a five-objective one. The new fitness function is a penalty value that depends only on the counter variables and the guard conditions referring to them. For each test path that has transitions referring to counter variable(s) we symbolically execute the initializers and updaters of those variables along the test path. Whenever a transition with a guard condition referring to a counter variable is reached the current value of the counter variable is used to evaluate the guard condition. If the guard condition is true, no penalty is added to the path. If the guard condition is not true a penalty value is added to the fitness of the path rather than marking the test path as infeasible (i.e. similar to Kalaji et al [88]). This penalty is proportional to the difference between the current value of the counter variable and the value required to make the guard condition true, similarly to the notion of approximation level in GA-based white box test case construction [35]. The corresponding fitness of a test path, TP , is defined using the following formula, where Cg is the value of the counter variable required by the guard condition referring to it, Cv is the current value (in the symbolic execution of the path) of the counter variable and p is a basic penalty value (we used the value of 100 in our experiments);

$$fitness(TP) = \sum_{\substack{\text{for all counter} \\ \text{variables in } TP}} |Cg - Cv| * p$$

The fitness of a test suite is then defined as the sum of penalty values of its test paths. Our objective is then minimizing this fitness function. We proceeded this way

since the fitness value of a path is no longer binary (as with Kalaji et al.) but should lead the GA towards acceptable (i.e., feasible) solutions. The landscape of the search should show a larger number of hills, which should help the GA towards good solutions. We believe this new solution avoids other limitations of the KHS-GA approach as well. There we will be no need to create all the acceptable triples on initializer, updater and updater times in advance. Our new solution does not need all the paths from updaters to the corresponding guard conditions to be definition clear. Also it does not require all the occurrences of a specific updater to be consecutive as it only considers the final value of a counter variable in a guard condition after all the updates made by different transitions in a given test path. It however has an evaluation cost: the symbolic execution. We will refer to this five-objective GA as New-GA in the reminder of this thesis.

5.3.3 Genetic Algorithm parameters

We mostly (see below) followed the design of the original GA (section 3.6). We selected a population size of 200, which conforms to what has been suggested in the literature [61]: i.e., a value in range [30, 80] multiplied by the number of objective functions (five for New-GA and four for KHS-GA). To keep population sizes identical in both extensions of the GA (KHS-GA and New-GA), so as to not favour one over the other, and given they have different numbers of fitness functions, we used two different values in range [30, 80]: 40 for New-GA and 50 for KHS-GA. Based on results from previous studies [62] we selected a crossover rate of 0.7 and a mutation rate of 0.01. The Pareto fraction parameter controls elitism in a multi-objective GA since it limits the number of individuals in the Pareto set (elite or tradeoff members). Based on a previous study [62], which suggests to set the maximum size of the Pareto set such that the ratio of the Pareto set over the entire population is between $\frac{1}{4}$ and 4, we set the maximum size of the Pareto set to 35% of the entire population. With respect to the stopping criterion and the type of Multi-Objective Evolutionary Algorithm (MOEA), we reuse the one of the Matlab toolbox we relied on, and stopped the GA if the observed average change in any objective function value over 10 generations was less than e^{-4} . The MOEA used by the toolbox is NGS-II [63].

The initial population is generated randomly, i.e., each GA starts with a set of 200 (our population size) randomly generated test suites. Each test suite has a random number of randomly generated valid traversals (of variable length) of the graph representing the EFSM. When generating a test suite, the GA first randomly selects the number of transitions (N) this test suite will have, between one and the maximum number of transitions. Each GA uses the same limit for the maximum size of a test suite to control bloat [53]. Each GA then incrementally creates test paths as random traversals of the state machine graph until the cumulative number of transitions in those test paths reaches N. For the purpose of creating test paths, each GA adds a reset transition from each state to the start state. When creating a test path each GA then iteratively, randomly and uniformly selects an outgoing transition (including the reset) from the current state. Admittedly, this can encourage the generation of short sequences, especially when states have a small number of outgoing transitions. Experiment results show however this does not prevent each GA from producing longer test paths thanks to objective functions which favour diversity for instance.

For our experiment, we modified the original reproduction operators, specifically the mutation operator, to (hopefully) better guide each GA to optimal solutions to the counter problem. In the original multi-objective GA (Chapter 3), six different mutation operators were used, and applied with equal probability, to modify solutions in every generation [91]. To account for the counter problem, in each of New-GA and KHS-GA, we increased the probability of using the operator which adds a new transition to a test path to 50% and made sure all the other operators had the same probability (10%). Since accounting for the counter problem requires that some transitions, or sequences of transitions, be repeated, favouring the addition of transitions should help. Other operators also ensure, though with a smaller probability, that sequences of transitions are added.

5.4 Empirical evaluation

To study the two solutions that account for counter variables and evaluate the impact of various manifestations of the counter problem on results those solutions generate, we experimented with the three example EFSMs we introduced in section 5.1, and both New-GA and KHS-GA. We believe these three examples appropriately present

different manifestations of the counter problem. For the stack EFSM we set the capacity to four. For each case study, we ran both GAs several times and collected the feasible, adequate test suites from the Pareto front. Results are discussed below for each of the three case studies separately.

In section 3.7, we compared our GA to randomly created test suites, making sure the random generation explores the same number of solutions in the search space as the GA. Results showed that random generation was never able to produce feasible test suites. Since the problems we addressed then were simpler, specifically without counters, there is no reason to believe a random generation would perform better at generating feasible test suites than KHS-GA or New-GA in the presence of counters. We therefore do not report on a comparison with a random generation in this chapter.

5.4.1 Counter problem Level1

Each time we ran New-GA, we were able to find feasible, adequate test suites in the Pareto front. A sample of such test suites is included in Appendix F. On the contrary, in each execution of KHS-GA, the Pareto front did not contain a single adequate, feasible test suite. One possible reason could be the fact that in a stack we can push and pop elements alternatively and the path can still be feasible while the solution proposed by Kalaji only accepts consecutive occurrences of a single type of updater, which are push and pop operators in the stack case study. This is one of the limitations of KHS-GA that we mentioned in section 5.3.1 and tried to avoid it in our new solution. The other reason could be that their algorithm does not search the landscape of solutions efficiently, which is not easy since, as already discussed, the landscape has high picks and is otherwise relatively flat.

5.4.2 Counter problem Level2

Contrary to the simple (stack) example, neither New-GA nor KHS-GA were able to produce a single feasible, adequate solution in their respective Pareto front, in each of the many runs we performed. Although the result for KHS-GA is not entirely surprising since KHS-GA was not able to produce satisfactory solutions on the simpler problem (see above), the result of New-GA needs investigation.

We randomly selected test suites from the Pareto front after each run and looked into the test paths they contain to find out why they were all infeasible. The main reason in almost all those cases was either the guard conditions of T_6 or T_9 was not satisfied: T_5 must be executed three times to enable T_6 ; T_8 must be executed at least twice to enable T_9 . In most of those test suites the guard condition of T_9 was not satisfied.

However, New-GA produced solutions that were very close to being adequate and feasible since the guard condition of T_6 , which represents the first type of counter was satisfied. Minor modifications to those test suites (adding or removing a few transitions) can make them feasible and adequate. A sample of those test suites is included in Appendix F

Solutions found by KHS-GA were not even close to those solutions. One reason could be the method used to compute penalty values. After creating the list of possible triplets (initializer, updater, updater times) a test path is only accepted if it has one of these triplets and otherwise it is marked as totally infeasible. This contrasts with New-GA, which computes how close the value of a counter variable is to the required value to make guard conditions referring to them true.

We tried to change some of the GA parameters mentioned in section 5.3.3 in an attempt to increase the chance for the GAs to find a feasible solution. Based on the hypothesis that the population size may not be large enough for the GAs to find the desired solution we increased the population size to 300, 400 and 500 and executed the two GAs with each of these population sizes several times. These modifications did not result in adequate feasible test suites either. We also tried to increase the maximum length of chromosomes so that the GAs could search for longer test paths. We increased the maximum length to 400 and executed the GAs again, without success.

To better understand why no feasible, adequate solution were found, we looked into the structure of the EFSM (section 5.1), and discovered that the chance of finding a feasible solution in which the guard condition of T_9 is satisfied is very, very small. Considering the structure of the EFSM and guard conditions referring to C_1 and C_2 , the shortest feasible path to reach T_{10} has a length of 16: $T_2, T_5, T_5, T_5, T_6, T_8, T_5, T_5, T_5, T_6,$

T₈, T₅, T₅, T₅, T₆, T₉. Based on the topological structure of the EFSM there are $2^3 \cdot 3^2 \cdot 4^{12}$ (1,207,959,552) different paths of length 16 in the EFSM reaching T₉ and only one of them is feasible. Increasing the length of test paths will increase the size of the search space exponentially and decrease the chance of finding feasible solutions dramatically. This is the main reason why the two GAs are not able to find a feasible test suite. The chance for the GAs to find a solution for the EFSMs with the counter problem is very low. Considering all the possible traversals of an EFSM that have a specific length and the fact that only a few of them will be feasible, finding a feasible path is almost impossible for the GA.

Another potential reason could be the fact that New-GA has five objective functions, considering that some GA frameworks handle such a large number of objective functions better than others [92]. Although future work will have to look into this, we are not convinced this is the main cause for the negative results reported above (recall the discussion about the size of the search space).

5.4.3 Coutner problem Level3

We ran New-GA and KHS-GA on the third case study. Neither of them, in neither of the runs, could find feasible, adequate test suites, which is not entirely surprising given the previous results and the fact that the third example exhibits more complex counter problems. However, similarly to the second example, New-GA was able to find partially feasible paths solving the counter problems due to counters C₁, C₂ and C₃, while the transition guarded by C₄ is never enabled. On the contrary, KHS-GA was never able to find such partially feasible paths for any of the counters. We have included a sample of those New-GA generated test suites in Appendix F.

Similarly to the previous case study, this could be caused by a very small number of adequate solutions in a very large search space. For counter variables of type two and three, optimal solutions that our GA is looking for (feasible adequate test suites) are very scarce and the probability of finding them in the given large search space is close to zero. However, our new solution could get very close to those optimums. Another reason could be the other objectives of our GA. For example, increasing diversity in test suites can

prevent the GA from looking for solutions in which a specific sequence of transitions is repeated several times to meet requirements of counter problems of type three (C4 in our example).

5.5 Summary

In this chapter we investigated the effect of the counter problem on our multi-objective genetic algorithm. In an EFSM with counters there are specific transitions that require another transition or sequence of transitions to be repeated more than once in order to be enabled. This can complicate test generation from an EFSM. In this chapter we defined three different manifestations of the counter problem. We integrated an existing counter-aware technique with our method of constructing an entire test suite from an EFSM. We also proposed a new solution to addressing the counter problem during test suite construction which was also integrated into our previous technique. We then compared those two counter-aware, multi-objective test suite construction techniques on three different EFSMs that had counter variables. Our new solution proved to be more successful in handling more complex manifestations of the counter problem and examples that included more than once type of counter problem. There is however still room for improvement.

6 Conclusion and future work

In this chapter we first conclude on the idea of this thesis, our approach and results (section 6.1) and then discuss future works (section 6.2).

6.1 Conclusion

Model-based testing (MBT) has become a common approach to software testing. According to Leagard [2] this is due to reasons like increase in software complexity, high cost of software testing and maturity of model-based testing tools. State-based models are widely used in MBT because of their concision and precision. Extended Finite State Machines (EFSMs) are preferred by software engineers over FSMs as EFSMs reduce the size of the state model model by using data variables.

My thesis was that test generation from an EFSM is a multi-objective optimization problem. A tester aims at more than one goal during test generation from an EFSM. There are competing objectives such as maximizing coverage while keeping all the test paths feasible or increasing diversity among test paths while keeping cost as low as possible. As search-based techniques have proved to be successful in finding solutions to optimization problems with competing objectives we used them in this thesis.

We proposed a search-based technique that generates test suites from an EFSM using a multi-objective genetic algorithm. The goal of our GA was to find test suites that achieve a maximum level of coverage, ideally reaching adequacy (we used the all-transitions selection criterion), have a high chance of being feasible (we use a surrogate measure of feasibility), minimize cost (i.e., cumulative number of triggered transitions in all test cases), and minimize the similarity between the test paths that constitute the test suites since this has been shown to relate to the effectiveness of test cases at finding faults. We provided a detailed description of our algorithm and justified the decisions we made at different steps.

To the best of our knowledge, this is the first time in the state-based testing literature that these objectives are used together. Additionally, our solution creates an entire adequate test suite in one search step instead of creating test cases one after the

other for satisfying test objectives (in our case covering transitions) separately in an incremental manner. We argued, and experimentally confirmed, that proceeding incrementally is a sub-optimal procedure for the task, and that this can be compared to a greedy algorithm. Instead, we rely on a meta-heuristic search, specifically a genetic algorithm.

We used two different models to validate our approach (section 3.7). Results confirmed our intuition that generating an entire test suite rather than doing so incrementally results in improvements in terms of cost and dissimilarity, while not hurting with respect to (expected) feasibility and adequacy. Also, we confirmed that when a test suite is expected to be feasible, using a surrogate measure of feasibility, it is indeed possible to find test inputs such that test paths can actually execute, even in the presence of other optimization goals. Also by using diversity as an objective function we managed to find test suites that are less expensive, without sacrificing effectiveness at detecting faults. Comparing test suites generated by our GA with randomly generated test suites proved the search space is so large or complex that a random search is not adequate and we have to resort to a GA.

We also explored different ways of improving our approach. In Chapter 4 we investigated the effect of using different methods of measuring similarity in a test suite on our multi-objective GA. In addition to the Levenshtein distance, which we used in the initial implementation of our GA (section 3.4), we used the Hamming distance and the Dice measure to compute pair-wise similarities between test paths. To aggregate those pair-wise similarities and compute similarity within an entire test suite we used two methods from the life sciences in addition to summing up which was the method used by Hemmati et al. and we used in the initial implementation of GA. One of them was Kosman diversity within population (KW) [79] because experiments indicated this was the most appropriate measure of diversity (i.e., the most informative one) compared to a series of other measures. It accounts for the genotype structure of a population as well as the dissimilarity between various genotypes. The other measure we used is the average distance within population (ADW), which was shown to be a good representative measure of all the measures that are based on allele (i.e. transitions in an EFSM)

frequencies [79]. This resulted in nine different flavours of our multi-objective GA: three pair-wise measures of test cases and three aggregation strategies to compute a similarity value for a test suite. We compared these nine different implementations, i.e., the generated test suites, based on cost and effectiveness at detecting faults. The Dice measure was the best measure to compute pair-wise similarities/dissimilarities in terms of cost and effectiveness. Regarding the method of computing diversity for a whole test suite, summing up all the pair-wise values had better results than the two methods from life sciences (ADW, KW) regarding cost and effectiveness.

We also looked at the effect of the counter problem on our approach. Generating executable test paths from an EFSM is hindered by the presence of counter variables in guard conditions since they require that specific behaviour, as simple as a single transition but possibly much more complex, be repeated a given number of times (counter) prior to enabling another behaviour (i.e., a guarded transition) in the EFSM.

We first tried to characterize different manifestations of the counter problem in EFSMs, from simple ones to more complex ones. Our taxonomy of counter problems, though simple and definitely incomplete, illustrates three different cases, of varying complexities. Although our discussion is not meant to be a definite one on the subject, we believe it is the first attempt at doing so being reported in the literature, which others can build upon: enriching the taxonomy; comparing technologies on a common benchmark. We then modified, in two different ways, our multi-objective genetic algorithm to handle counters in EFSMs. We selected this optimization technique since it was shown (Chapters 3 and 4) to effectively generate entire test suites from an EFSM that are transition adequate, reduce costs, are made of feasible paths (when there is no counter) and that are as diverse as possible (in an attempt to be effective at finding faults). Two extensions, referred to as KHS-GA and New-GA, were considered. The first one (KHS-GA) integrated existing work by others to account for counters when creating one test case at a time [88]. The second one (New-GA) is our contribution and attempts, by adapting solutions for white-box testing [35], to address limitations of this previous counter-aware solution.

We experimented with these two optimization techniques on three different case studies that are representative of different manifestations of the counter problem. Results show that our solution (New-GA) can handle simple counters when they appear alone in an EFSM (first case study), that is, it can find feasible (i.e., executable) test paths. Unfortunately, our solution is not able to produce feasible test suites for the two other, more complex manifestations of the counter problem. It was only able to find test paths that contain feasible/executable sub-paths, accounting for a subset of the counters of the EFSM, the simple manifestations of the counter problem; the entire test paths are not feasible because of more complex manifestations of the counter problem. On the contrary, the extension based on an existing counter-aware solution, KHS-GA, could not find a single feasible solution for any one of the three case studies, even the one with the simple manifestations of the counter problem. In summary, our new solution can find feasible solutions for simple counters and get very close to feasible solutions for more complex ones, while the existing solution did not succeed at finding feasible solutions, or getting close to them, for any type of counter problems.

Our qualitative analysis of results explained why generated test paths were not feasible (although sub-paths were) in relation to the EFSMs and the manifestations of the counter problem. We showed that for complex manifestations of the counter problem, finding an adequate and feasible test suite for an EFSM is like finding a needle in a haystack: in one example (not the most complex manifestation of the counter problem), enabling a transition guarded by a counter variable with the shortest test path as possible amounts to finding a path in one billion other paths. This is a task that is difficult for a genetic algorithm, unless it is adequately guided by fitness functions and cross-over and mutation operators.

Although our results are not entirely positive, we conclude that we present an improvement over existing work to handle the counter problem in EFSMs but that there is much research work in front of us to find a more general solution. One could think of reducing the size of the search space by looking for test paths of a specific length; the issue is however to be able to find this maximum length in an automated way. Modifying reproduction operators, i.e. mutation and crossover, to better guide the GA towards

feasible solutions can be considered as another extension to our solution. One can also consider revisiting our fitness functions. As we added a new fitness function specific to the counter problem, another way to extend our solution is using other multi-objective evolutionary algorithm, specifically ones that have shown to be more successful than NGS-II in handling multiple objective functions (e.g. IBEA [62]). Another way to guide the GA towards feasible solution could be using symbolic execution of the whole EFSM. Our discussion of counter problems (taxonomy) and our qualitative analysis of results should help others and us in this quest. We defined different manifestations of the counter problem with corresponding EFSMs; These could be considered an initial framework for evaluation of alternative test case/suite constructions techniques in future research. However, all these modifications will affect other objectives in test suite generation from EFSMs that we discussed in different sections of this thesis (e.g. maximizing diversity) and their effects need to be investigated thoroughly.

Last, this thesis resulted in the following publications:

- Asoudeh, N., Labiche, Y., “Multi-objective construction of an entire adequate test suite for an EFSM,” IEEE International Symposium on Software Reliability Engineering , 2014
- Asoudeh, N., Labiche, Y., “A Multi-Objective Genetic Algorithm for Generating Test Suites from Extended Finite State Machines,” Symposium on Search-Based Software Engineering, 2013
- Asoudeh, N., Labiche, Y., “On the Effect of Counters in Guard Conditions When State-based Multi-objective Testing,” IEEE International Conference on Software Quality, Reliability & Security – Workshops (QRS-C), 2015
- Asoudeh, N., Labiche, Y., “Mult-objective test generation from EFSMs and effect of test suite diversity”, Journal paper in preparation

6.2 Future work

We tried two different ways of improving our solution to the problem of test generation from EFSMs (multi-objective GA) in this thesis. There are other possible improvements that can be considered as future work, such as:

- Exploring different ways of improving the GA configuration (e.g., using different probabilities of occurrence of our mutation operators); Currently, we have seven different mutation operators and they are selected with equal probability at mutation time. We have an intuition that assigning different probabilities to different mutation operators based on the priority of different objectives (e.g. more diversity vs. less cost) can increase the speed of converging towards different solutions. For parameters like mutation rate or crossover rate we have used what is recommended in the literature but there might not be the best values. One can consider using a sensitivity analysis to investigate the effect of changing those values and then adjust parameter values based on the results of this analysis.
- Using other measures to compute coverage (e.g., transition pairs) or cost of a test suite; Currently, we use transition coverage. As mentioned previously, this criterion has proven to be a good trade-off between cost and effectiveness at detecting faults. However, there are other criteria like transition pairs or round trip path that are more expensive but more effective in detecting faults. Depending on the use case and the importance of fault detection rate these new measures can be used.
- Considering weights for our objective functions, for instance favouring coverage in order to have (hopefully) a larger number of adequate test suites in the Pareto front. The identification of weights is however a difficult problem, and experiments [49] show that a solution with weights does not necessarily perform better than a real multi-objective solution;
- Trying different strategies for creating the initial population; Currently, the initial population is generated randomly. We believe replacing randomly

generated test suites with the ones that are optimal in terms of at least one of the objectives (e.g., test suites that are adequate) could result in better solutions.

- Using heuristics that are independent of running the GA for a specific EFSM in determining maximum length of chromosomes to control the bloat effect; Currently, we need to run our GA for a specific EFSM to compute the upper limit of chromosome length. A heuristic method that can compute that limit based on the EFSM structure and does not depend on any execution results of the GA is an improvement to our approach.
- Experimenting with a broader set of case studies; We used two different case studies to evaluate our GA and three different case studies to investigate the effect of the counter problem. Experimenting with additional case studies will increase the confidence in the results of this thesis. It can also help in finding new ways to further improve our solution.
- Continuing the investigation of the notion of diversity of test cases in a test suite; We experimented with three different pair-wise similarity measures as well as three different methods of computing diversity within whole test suites, two of which are coming from life sciences. We obtained interesting results. Trying different pair-wise measures as well as aggregation methods from other fields (e.g. statistics) can result in new remarks on the effect of diversity.
- Comparing our multi-objective GA with multiple single objective GAs; Another point to investigate in the future would be to confirm the benefits of using a multi-objective GA. One can consider comparing our multi-objective GA with another approach in which a single objective GA is used to address each objective, with exactly the same fitness functions that we have used.

In section 5 we presented an improvement over existing work to handle the counter problem in EFSMs. Another possible future work to this thesis would be focusing on counter problem and finding a more general solution for it. One could think of reducing the size of the search space by looking for test paths of a specific length; the

issue is however to be able to find this maximum length in an automated way. Modifying reproduction operators, i.e. mutation and crossover, to better guide the GA towards feasible solutions can be considered as another extension to our solution. One can also consider revisiting our fitness functions. As we added a new fitness function specific to the counter problem, another way to extend our solution is using other MOEAs (Multi-Objective Evolutionary Algorithm), specifically ones that have shown to be more successful than NGSA-II in handling multiple objective functions (e.g. IBEA [42]). Another way to guide GA towards feasible solution could be using symbolic execution of the whole EFSM. Our discussion of counter problems (taxonomy) and our qualitative analysis of results should help others and us in this quest. Clarified definition for different manifestations of counter problem accompanied by EFSMs representing each type can be used as an evaluation framework for future research. However, all these modifications will impact other objectives of our approach as they compete with each other. These effects need to be investigated thoroughly.

References

- [1] M. Utting and B. Legeard, Practical Model-based Testing, Morgan Kaufmann.
- [2] B. Legeard, "Model-based Testing: Next Generation Functional Software Testing," in *Practical Software Testing: Tool Automation and Human Factors Seminar*, Dagstuhl, Germany, 2010.
- [3] H. Hemmati, A. Arcuri and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, February 2013.
- [4] P. Inverardi, M. Autili, D. Di Ruscio, P. Pelliccione and M. Tivoli, "Producing software by integration: challenges and research directions," in *ESEC/FSE*, Saint Petersburg, Russia, 2013.
- [5] M. Harman, K. Lakhota, J. Singer, D. R. White and S. Yoo, "Cloud engineering is Search Based Software Engineering too," *Journal of Systems and Software*, vol. 86, no. 9, pp. 2225-2241, 2013.
- [6] B. Beizer, Software Testing Techniques, International Thomson Computer Press, 1990.
- [7] M. Harman, P. McMinn, J. Teixeira de Souza and S. Yoo, "Search based software engineering: techniques, taxonomy, tutorial," *Empirical Software Engineering and Verification*, vol. 7007, pp. 1-59, 2012.
- [8] M. Harman and B. F. Jones, "Search-based software engineering," *Information and software engineering*, pp. 833-839, 2001.
- [9] W. Miller and D. Spooner, "Automatic generation of floating point test data,"

IEEE Transaction on Software Engineering, vol. 2, no. 3, pp. 223-226, 1976.

- [10] P. MacMinn, "Search-based software testing: Past, Present and Future," in *Fourth International Conference on Software Testing, Verification and Validation Workshops*, Berlin, Germany, 2011.
- [11] "Global Optimization Toolbox," Mathworks, [Online]. Available: <http://www.mathworks.com/products/global-optimization/>. [Accessed 25 April 2013].
- [12] A. Neto and G. Travassos, "Evaluation of model-based testing techniques selection approaches: An external replication," in *IEEE International Symposium on Empirical Software Engineering and Measurement*, 2009.
- [13] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines - A survey," *Proceedings of IEEE*, vol. 84, no. 8, pp. 1090-1123, 1996.
- [14] A. Petrenko, S. Boroday and R. Groz, "Confirming Configurations in EFSM Testing," *IEEE TSE*, vol. 39, no. 1, pp. 29-42, 2004.
- [15] D. Goldberg, *Genetic Algorithms in Search, Optimization and machine learning*, Boston: Addison Wesley, 1989.
- [16] A. Konak, D. W. Coit and A. E. Smith, "Multi-objective optimization using genetic algorithms: A tutorial," *Reliability Engineering and System Safety*, vol. 91, no. 9, pp. 992-1007, 2006.
- [17] J. D. Schaffer, "Multiple Objective Optimization with Vector Evaluated Genetic Algorithms," in *1st International Conference on Genetic Algorithms*, 1985.
- [18] D. Goldberg and J. Richardson, "Genetic algorithms with sharing for multimodal function optimization," in *Internation conference of genetic algorithms*, 1987.

- [19] H. Lu and G. Yen, "Rank-density-based multiobjective genetic algorithm and benchmark test function study," *IEEE Transactions on Evolutionary Computing*, vol. 7, no. 4, pp. 325-343, 2003.
- [20] C. Coello Coello, D. Van Veldhuizen and G. Lamont, *Evolutionary Algorithms for Solving MultiObjective Problems*, 2nd ed., Springer, 2007.
- [21] J. Horn, N. Nafpliotis and D. Goldberg, "A Niche Pareto Genetic Algorithm for Multiobjective Optimization," in *First IEEE Conference on Evolutionary Computation*, 1994.
- [22] A. P. Mathur, *Foundations of Software Testing*, Pearson, 2008.
- [23] G. Luo, G. Bochmann and A. Petrenko, "Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method," *IEEE TSE*, vol. 20, pp. 149-162, 1994.
- [24] J. Li and W. Wong, "Automatic test generation from communicating extended finite state machine (CEFSM)-based models.," in *IEEE Int. Symp. on Real-Time Distributed Computing*, 2002.
- [25] X. Jin, G. Ciardo, T.-H. Kim and Y. Zhao, "Symbolic verification and test generation for a network of communicating FSMs," in *International conference on Automated technology for verification and analysis*, 2011.
- [26] "Conformiq Software Ltd.," [Online]. Available: http://www.verifysoft.com/ttcn-3_qtronic_sip.pdf.
- [27] Q. Guo, R. Hierons, M. Harman and K. Derderian, "Computing Unique Input/Output Sequences Using Genetic Algorithms. In: FATES, pp. 164-177 (2004)," in *FATES*, 2004.
- [28] J. Tretmans, "Model based testing with labelled transition systems," in *FATES*, 2008.

- [29] A. Y. Dual and M. U. Uyar, "A method enabling feasible conformance test sequence generation for EFSM models," *IEEE Transactions on Computers*, vol. 53, no. 5, pp. 614-627, 2004.
- [30] A. S. Kalaji, R. M. Hierons and S. Swift, "An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models," *Information and Software Technology*, vol. 53, no. 12, pp. 1297-1318, 2011.
- [31] C. Schwarzl and B. Peischl, "Test Sequence Generation from Communicating UML State Charts: An Industrial Application of Symbolic Transition Systems," in *QSIC*, 2010.
- [32] K. Larsen, M. Mikucionis, B. Nielsen and A. Skou, "Testing real-time embedded software using UPPAAL-TRON: an industrial case study.," in *In: ACM EMSOFT*, 2005.
- [33] W. Hasanain, Y. Labiche and S. Gheorgh, "Automated State-based Online Testing Real-time Embedded Software with RTEdge," in *Modelsward*, 2015.
- [34] S. Dalal, A. Jain and J. Poore, "Workshop on advances in model-based testing," in *International Conference on Software Engineering (ICSE)*, 2005.
- [35] J. Wegener, A. Baresel and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841-854, 2001.
- [36] N. Tracey, J. Clark, K. Mander and J. McDermid, "An automated framework for structural test-data generation," in *IEEE ASE*, Honolulu, HI, 1998.
- [37] R. Lefticaru and F. Ipate, "Automatic State-Based Test Generation Using Genetic Algorithms," in *Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2008.
- [38] M. Harman, Y. Jia and Y. Z. Zhang, "Achievements, open problems and

challenges for search based software testing,” in *IEEE ICST*, Graz, Austria, 2015.

- [39] T. Yano, E. Martins and F. L. Sousa, “MOST: a Multi-Objective Search-Based Testing from EFSM,” in *IEEE ICST Workshops*, Berlin, 2011.
- [40] J. Ferrer, F. Chicano and E. Alba, “Evolutionary algorithms for the multi-objective test data generation problem,” *Software: Practice and Experience*, vol. 42, no. 11, pp. 1331-1362, 2012.
- [41] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm : NSGA-II,” *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182-197, 2002.
- [42] A. Nebro, J. Durillo, F. Luna, B. Dorronsoro and E. Alba, “MOCCell: A cellular genetic algorithm for multiobjective,” *Int. J. Intell. Syst*, vol. 24, no. 7, pp. 726-746, 2009.
- [43] E. Zitzler, M. Laumanns and L. Thiele, “SPEA2: Improving the strength pareto evolutionary algorithm,” Technical Report 103, Gloristrasse 35, CH-8092, Zurich, Switzerland, 2001.
- [44] J. Knowles and D. Corne, “The Pareto archived evolution strategy: a new baseline algorithm for Pareto multiobjective optimisation,” in *Congress on Evolutionary Computation*, Washington, DC, 1999.
- [45] S. Yoo and M. Harman, “Pareto efficient multi-objective test case selection,” in *International Symposium on Software Testing and Analysis (ISSTA)*, London, UK, 2007.
- [46] W. Langdon, M. Harman and Y. Jia, “Efficient multiobjective higher order mutation testing with genetic programming,” *Journal of Systems and Software*, vol. 83, no. 12, pp. 2416-2430, 2010.

- [47] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34-41, 1978.
- [48] B. Marculescu, R. Feldt and R. Torkar, "Objective re-weighting to guide an interactive search based software testing system," in *International conference on machine learning and applications*, Miami, Florida, 2013.
- [49] M. Harman, K. Lakhotia and P. McMinn, "A Multi-Objective Approach To Search-Based Test Data Generation," in *GECCO*, London, England, United Kingdom, 2007.
- [50] C. Michael, G. McGraw and M. A. Schatz, "Generating Software Test Data by Evolution," *IEEE Transactions on Software Engineering* , vol. 27, no. 12, pp. 1085-1110 , 2001.
- [51] Y. Zhang, M. Harman and A. Mansouri, "The SBSE repository: A repository and analysis of authors and research articles on search based software engineering," [Online]. Available: [crestweb.cs.ucl.ac.uk/resources/sbse repository/..](http://crestweb.cs.ucl.ac.uk/resources/sbse/repository/)
- [52] S. Ali, L. C. Briand, H. Hemmati and R. Kaur, "A systematic review of the application and empirical investigation of search-based test-case generation," *IEEE Transactions of Software Engineering*, vol. 36, no. 6, pp. 742-762, 2010.
- [53] S. Silva and E. Costa, "Dynamic Limits for Bloat Control in Genetic Programming and a Review of Past and Current Bloat Theories," *Genetic Programming and Evolvable Machines*,, vol. 10, no. 2, pp. 141-179, 2009.
- [54] G. Fraser and A. Arcuri, " Whole test suite generation," *IEEE Transactionaction on Software Engineering*, vol. 39, no. 2, pp. 276-291, 2013.
- [55] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Sience and Computational Biology*, Cambridge: Cambridge University Press, 1997.

- [56] M. Khalil and Y. Labiche, “ On the Round Trip Path Testing Strategy,” in *IEEE International Symposium on Software Reliability Engineering*, 2010.
- [57] L. Briand, Y. Labiche and Y. Wang, “Using Simulation to Empirically Investigate Test Coverage Criteria,” *IEEE/ACM International Conference on Software Engineering*, 2004.
- [58] M. Hutchins, H. Froster, T. Goradia and T. Ostrand, “Experiments on the Effectiveness of Dataflow and Controlflow-Based Test Adequacy Criteria,” in *IEEE/ACM International Conference on Software Engineering*, 1994.
- [59] E. Weyuker, N. Stewart, S. N. Weiss and D. Hamlet, “Comparison of Program Testing Strategies,” in *ACM International Symposium in Software Testing and Analysis*, 1991.
- [60] P. G. Frankl and S. N. Weiss, “An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing,” *IEEE Transactions on Software Engineering*, vol. 19, no. 8, pp. 774-787, 1993.
- [61] M. Laumanns and E. T. L. Zitzler, “On The Effects of Archiving, Elitism, an Density Based Selection in Evolutionary Multi-objective Optimization,” in *International Conference on Evolutionary Multi-Criterion Optimization*, 2001.
- [62] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*, Wiley, 1998.
- [63] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, “A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II,” *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182-197, 2002.
- [64] S. Mouchawrab, L. Briand, Y. Labiche and M. Di Penta, “ Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments,” *IEEE TSE*, vol. 37, no. 2, pp. 161-187, 2011.
- [65] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications*

with UML, Object Technology, Boston, USA: Addison Wesley, 2000.

- [66] B. Bruegge and A. H. Dutoit, Object-Oriented Software Engineering Using UML, Patterns, and Java, Prentice Hall, 2004.
- [67] P. Chevalley and P. Thévenod-Fosse, “Automated Generation of Statistical Test Cases from UML State Diagrams,” in *International Computer Software and Applications Conference*, 2001.
- [68] N. E. Holt, B. C. D. Anda, K. Asskildt, L. C. Briand, J. Endresen and S. Frøystein, “Experiences with Precise State Modeling in an Industrial Safety Critical System,” in *CSDUML*, 2006.
- [69] J. H. Andrews, L. Briand, Y. Labiche and A. Namin, “- Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria,” *IEEE-TSE*, vol. 32, no. 8, pp. 608-624, 2006.
- [70] R. Just, F. Schweiggert and G. M. Kapfhammer, “MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler,” in *ASE*, Lawrence, KS, USA, 2011.
- [71] "State Machine Compiler," [Online]. Available: <http://smc.sourceforge.net/>.
- [72] R. Binder, Testing Object Oriented Systems, Addison-Wesley, 2000.
- [73] D. Campbell and D. Stanely, Experimental and Quasi-Experimental Designs for Research, Boston, MA: Houghton Mifflin Company, 1963.
- [74] E. Kosman, “Measuring diversity : from individuals to populations,” *Eur J Plant Pathology*, vol. 138, no. Special Issue on wild plant pathosystems, pp. 467-486, 2014.
- [75] L. Jost, “Partitioning diversity into independent alpha and beta components,” *Ecology*, vol. 88, pp. 2427-2439, 2007.

- [76] L. Jost, "GST and its relatives do not measure differentiation," *Molecular Ecology*, vol. 17, pp. 4015 - 4026, 2008.
- [77] H. Tumisto, "An updated consumer's guide to evenness and related indices," *Oikos*, vol. 121, pp. 1203-1218, 2012.
- [78] M. Nei, "Analysis of Gene Diversity in Subdivided Population," in *National Academy of Sciences of the United States of America*, 1973.
- [79] E. Kosman and K. J. Leonard, "Conceptual analysis of methods applied to assessment of diversity within and distance between populations with asexual or mixed mode of reproduction," *New Phytologist*, vol. 174, pp. 683-696, 2007.
- [80] R. Bellman, K. L. Cooke and L. J. A., *Algorithms, graphs and computers*, New York: Academic, 1970.
- [81] C. R. Rao, "Diversity and dissimilarity coefficients - a unified approach," *Theoretical Population Biology*, vol. 21, pp. 24-43, 1982.
- [82] C. Ricotta, "Bridging the gap between ecological diversity indices and measures of biodiversity with Shannon's entropy," *Ecological modeling*, vol. 152, pp. 1-3, 2002.
- [83] C. Ricotta and L. Szeidl, "Towards a unifying approach to diversity measures: bridging the gap between Shannon's entropy and Rao's quadratic index," *Theoretical Population Biology*, vol. 70, pp. 237-243, 2006.
- [84] J. L. Devore, *Probability and statistics for engineering and the sciences*, Pacific Grove, CA: Duxbury, 1999.
- [85] J. Cohen, *Statistical power analysis for the behavioral sciences*, New Jersey: Hillsdale, 1988.
- [86] G. a. N. H. Iversen, *Analysis of Variance*, Sage publications, 1987.

- [87] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes and G. Fraser, "Are mutants a valid substitute for real faults in software testing," in *International Symposium on the Foundations of Software Engineering (FSE)*, Hong Kong, China, 2014.
- [88] A. S. Kalaji, R. M. Hierons and S. Swift, "Generating Feasible Transition Paths from an Extended Finite State Machine (EFSM) with the Counter Problem," in *ICST Workshops*, 2010.
- [89] R. Binder, *Testing Object-Oriented Systems - Models, Patterns, and Tools*, Addison-Wesley, 1999.
- [90] L. C. Briand, Y. Labiche and J. Cui, "Automated Support for Deriving Test Requirements from UML Statecharts," *Journal of Software and Systems Modeling*, vol. 4, no. 4, pp. 399-423, 2005.
- [91] N. Asoudeh and Y. Labiche, "Multi-objective construction of an entire adequate test suite for an EFSM," in *IEEE ISSRE*, Naples, Italy, 2014.
- [92] E. Zitzler and S. Kunzli, "Indicator-Based Selection in Multiobjective Search," in *Parallel Problem Solving from Nature*, Berlin, Germany, 2004.

APPENDICES

A. Actions and guard conditions of Qui-Donc EFSM

The name of each guard condition and action of each transition is the same as its triggering event. However, in order to distinguish between guard conditions and actions the suffix “Guard” is added to guard conditions.

```
public Boolean dialGaurd() {
    return curState == State.Start;
}
public @Action void dial() {
    out.println("dial/" + WELCOME);
    currState = State.star;
    timeouts = 0;
}
/*No guard, always enabled. We call this
wait_ to avoid a clash with
object.wait().*/
Public @Action void wait_() {
    timeouts++;
    if(timeouts >= 3 || currState ==
State.Emerg
    || currState==State.Start){
        if(currState == State.Star)
            out.println("wait/" + NOTALLOW);
        else
            out.println("wait/" + BYE);
        currState = State.Start;
        timeouts = 0;
    }
    else
        out.println("wait/...");
}
public Boolean StarGuard() {
    return currState == State.Star ||
currState == State.Emerg
    || currState == State.Info;
}
public @Action void star() {
    out.println("*/" + ENTER);
    currState = State.Enter;
    timeouts = 0;
}
public Boolean badGaurd() {
    return currState == State.Enter;
}
public @Action void bad() {
    out.println("bad/" + ERROR);
    // state is unchanged.
    timeouts = 0;
}
public Boolean num18Guard() {
    return currState == State.Enter;
}

public @Action void num18() {
    out.println("18/" + FIRE);
    currState = State.Emerg;
    timeouts = 0;
}
public Boolean num1Guard() {
    return currState == State.Enter;
}
public @Action void num1() {
    out.println("num1/" + SORRY);
    //state is unchanged
    timeouts = 0;
}
public Boolean num2Guard() {
    return currState == State.Enter;
}
public @Action void num2() {
    out.println("num2/" + NAME);
    currState = State.Info;
    timeouts = 0;
}
public Boolean key1Guard() {
    return currState == State.Info;
}
public @Action void key1() {
    out.println("1/" + SPELL);
    //state is unchanged
    timeouts = 0;
}
public Boolean key2Guard() {
    return currState == State.Info;
}
public @Action void key2() {
    out.println("2/" + ADDR);
    //state is unchanged
    timeouts = 0;
}
```

B. Actions and guard conditions of Class II protocol state machine

Trans.	Input	Guards	Actions
T1: s1→s2	U?TCONreq (dst_add, prop_opt)	Nil	opt = prop_opt; R_credit=0; N!TrCR
T2: s1→s3	N?TrCR (peer_add, opt_ind, cr)	Nil	opt= opt_ind; S_credit=cr; R_credit=0; U!TCONind
T3: s2→s4	N?TrCC (opt_ind, cr)	opt_ind ≤ opt	TRsq=0; TSsq=0; opt=opt_ind; S_credit=cr; U!TCONconf
T4: s2→s5	N?TrCC (opt_ind, cr)	opt_ind > opt	U!TDISind; N!TrDR
T5: s2→s1	N?TrDR (disc_reason, switch)	Nil	U!TDISind; N!terminated
T6: s3→s4	U?TCONresp(accept_opt)	accept_opt ≤ opt	opt= accept_opt; TRsq=0; TSsq=0; N!TrCC
T7: s3→s6	U?TDISreq ()	Nil	N!TrDR
T8: s4→s4	U?TDATAreq (Udata, E0SDU)	S_credit > 0	S_credit= S_credit -1; TSsq = (TSsq +1)mod128; N!TrDT
T9: s4→s4	N?TrDT (Send_sq,Ndata, E0TSDU)	R_credit != 0 & Send_sq= TRsq	TRsq=(TRsq+1) mod128; R_credit=R_credit -1; U!DATAind; N!TrAK
T10: s4→s4	N?TrDT (Send_sq, Ndata, E0TSDU)	R_credit = 0 V Send_sq != TRsq	U!error; N!error
T11: s4→s4	U?U READY (cr)	Nil	R_credit= R_credit + cr; N!TrAK
T12: s4→s4	N?TrAK (XpSsq, cr)	TSsq > XpSsq & cr + XpSsq - TSsq > 0 & cr + XpSsq - TSsq < 15	S_credit = cr + XpSsq - TSsq
T13: s4→s4	N?TrAK (XpSsq, cr)	TSsq > XpSsq & (cr + XpSsq - TSsq < 0 V cr + XpSsq - TSsq > 0)	U!error; N!error
T14: s4→s4	N?TrAK (XpSsq, cr)	TSsq < XpSsq & cr + XpSsq - TSsq - 128 > 0 & cr + XpSsq - TSsq - 128 < 15	S_credit= cr+ XpSsq -TSsq - 128
T15: s4→s4	N?TrAK (XpSsq, cr)	TSsq < XpSsq & (cr + XpSsq - TSsq - 128 < 0 V cr + XpSsq - TSsq - 128 > 15)	U!error; N!error
T16: s4→s4	N?Ready	S_credit > 0	U!Ready
T17: s4→s5	U?TDISreq	Nil	N!TrDR
T18: s4→s6	N?TrDR (disc_reason, switch)	Nil	U!TDISind; N!TrDC
T19: s6→s1	N?terminated	Nil	U!TDISconf
T20: s5→s1	N?TrDC	Nil	N!terminated; U!TDISconf
T21: s5→s1	N?TrDR (disc_reason, switch)	Nil	N!terminated

C. Guard conditions of ordered set case study

Transition	Guard Condition (in OCL)
T1	N/A
T2	N/A
T3	$\text{self_set} \rightarrow \text{includes}(\text{val}) \text{ and } \text{self_set} \rightarrow \text{size}() = 1$
T4	$(\text{self_set} \rightarrow \text{includes}(\text{val}) \text{ and } \text{self_set} \rightarrow \text{size}() > 1) \text{ or } (\text{not } \text{self_set} \rightarrow \text{includes}(\text{val}))$
T5	$(\text{self_set} \rightarrow \text{includes}(\text{n})) \text{ or } (\text{not } \text{self_set} \rightarrow \text{includes}(\text{n}) \text{ and } \text{self_set} \rightarrow \text{size}() < \text{self_set_size}-1)$
T6	$\text{not } \text{self_set} \rightarrow \text{includes}(\text{n}) \text{ and } \text{self_set} \rightarrow \text{size}() = \text{self_set_size}-1$
T7	$\text{self_set} \rightarrow \text{includes}(\text{val})$
T8	$\text{self_resized_times} < \text{max_accepted_resizes} \text{ and } \text{self_set_size} + \text{min_set_size} \leq \text{max_set_size} \text{ and } \text{not } \text{self_set} \rightarrow \text{includes}(\text{n})$
T9	$\text{self_set} \rightarrow \text{includes}(\text{n})$
T10	$\text{not } \text{self_set} \rightarrow \text{includes}(\text{val})$
T11	$(\text{self_resized_times} \geq \text{max_accepted_resizes}) \text{ or } \text{self_set_size} + \text{min_set_size} > \text{max_set_size} \text{ and } \text{not } \text{self_set} \rightarrow \text{includes}(\text{n})$
T12	N/A
T13	N/A

D. Test suites used in chapter 3

Test suites from Figure 10

TSga1:

Path1: < T₃, T₁, T₆, T₆, T₄, T₅, T₁₂, T₁₃, T₁₄, T₁₉, T₂₃, T₂₇, T₁₇, T₁₈, T₂₀>

Path2: < T₂, T₄, T₅, T₁₄, T₂₁, T₂₅, T₂₆, T₂₄, T₂₈, T₁₅>

Path3: < T₅, T₉, T₁₁, T₁₀, T₉, T₈, T₁, T₇, T₅, T₁₁, T₁₄, T₁₆, T₁₈, T₁₅, T₄, T₅>

Path4: < T₄, T₅, T₁₄, T₂₁, T₂₅, T₂₆, T₂₄, T₂₆, T₂₂>

Path5 :< T₁, T₃>

TSga2:

Path1: < T₆, T₄, T₅, T₁₂, T₁₃, T₁₄, T₁₉, T₂₃, T₂₇, T₁₇, T₁₈, T₂₀>

Path2 :< T₃>

Path3: < T₂, T₄, T₅, T₁₄, T₂₁, T₂₅, T₂₆, T₂₄, T₂₈, T₁₅>

Path4: < T₅, T₉, T₁₁, T₁₀, T₉, T₈, T₁, T₇, T₅, T₁₁, T₁₄, T₁₆, T₁₈, T₁₅, T₄, T₅>

Path5: < T₄, T₅, T₁₄, T₂₁, T₂₅, T₂₆, T₂₄, T₂₆, T₂₂>

Path6 :< T₁, T₃>

TSga3:

Path1 :< T₃>

Path2: < T₆, T₄, T₅, T₁₂, T₁₃, T₁₄, T₁₉, T₂₃, T₂₇, T₁₇, T₁₈, T₂₀>

Path3: < T₂, T₄, T₅, T₁₄, T₂₁, T₂₅, T₂₆, T₂₄, T₂₆, T₂₂>

Path4: < T₅, T₁₀, T₁₃, T₁₂, T₁₄, T₂₁, T₂₈, T₁₅, T₇>

Path5: < T₅, T₉, T₁₁, T₁₀, T₉, T₈, T₁, T₇, T₅, T₁₁, T₁₄, T₁₆, T₁₈, T₁₅, T₄, T₅>

Path6 :< T₇>

TSga4:

Path1: < T₄, T₅, T₁₁, T₁₃, T₁₂, T₁₄, T₂₁, T₂₈, T₁₅, T₇>

Path2: < T₅, T₉, T₈, T₇>

Path3: < T₂, T₄, T₅, T₁₄, T₂₁, T₂₅, T₂₆, T₂₄, T₂₆, T₂₂>

Path4: < T₅, T₁₃, T₁₀, T₁₂, T₁₀, T₁₄, T₁₆, T₁₉>

Path5: < T₅, T₁₃, T₁₄, T₁₉, T₂₃, T₂₇, T₁₇, T₁₈, T₂₀>

Path6 :< T₁, T₃>

Test suites from table 5

Point1: adequate test suite

Path1: < T₁, T₃, T₁₅, T₈, T₁₇, T₁₉, T₂>

Path2: < T₂, T₆, T₁₄, T₁₈, T₂₁, T₁, T₅, T₂, T₇, T₂₁>

Path3: < T₁, T₅, T₂, T₆, T₁₃, T₁₆, T₁₀, T₁₇, T₂₀, T₁, T₄, T₁₉, T₂, T₇, T₂₁>

Path4: < T₂, T₆, T₁₁, T₁₂, T₉>

Point2:T₉ is not covered

Path1:< T₂, T₇, T₂₁, T₁, T₃, T₁₂, T₁₈>

Path2: < T₁, T₃, T₁₁, T₁₅, T₈, T₁₇, T₁₉, T₂, T₇, T₂₁, T₁>

Path3:< T₂, T₆, T₁₄, T₁₈, T₂₁, T₁, T₅>

Path4: < T₁, T₅, T₂, T₆, T₁₃, T₁₆, T₁₀, T₁₇, T₂₀, T₁, T₄, T₁₉>

Path5:< T₁>

Point3: T₉ and T₁₂ are not covered

Path1: < T₁, T₃, T₁₁, T₁₅, T₈, T₁₇, T₁₉, T₂>

Path2:< T₂, T₆, T₁₄, T₁₈, T₂₁, T₁, T₅, T₂, T₇, T₂₁>

Path3 :< T₁>

Path4: <T₁, T₅, T₂, T₆, T₁₃, T₁₆, T₁₀, T₁₇, T₂₀, T₁, T₄, T₁₉, T₂, T₇, T₂₁>

Point4: T₉, T₁₂ and T₁₄ are not covered

Path1:< T₂, T₆, T₁₅, T₁₈, T₂₁, T₁, T₅, T₃, T₈, T₁₁>

Path2: $\langle T_2, T_6, T_{13}, T_{16}, T_{10}, T_{17}, T_{20}, T_1, T_4, T_{19}, T_2, T_7, T_{21} \rangle$

Point5: T_8, T_9, T_{12} and T_{15} are not covered

Path1: $\langle T_1, T_3, T_{18} \rangle$

Path2: $\langle T_1, T_3, T_{14}, T_{11} \rangle$

Path3: $\langle T_1, T_5, T_2, T_6, T_{13}, T_{16}, T_{10}, T_{17}, T_{20}, T_1, T_4, T_{19}, T_2, T_7, T_{21} \rangle$

Point6: T_3, T_8, T_{12}, T_{14} and T_{15} are not covered

Path1: $\langle T_2, T_6, T_{11}, T_9, T_{18} \rangle$

Path2: $\langle T_1, T_5, T_2, T_6, T_{13}, T_{16}, T_{10}, T_{17}, T_{20}, T_1, T_4, T_{19}, T_2, T_7, T_{21} \rangle$

Test suites from Cruise Control case study used to answer RQ1

• T_{trad}

T1	T5-T8	T5-T14-T15	T5-T14-T19-T22
T2	T5-T9	T5-T14-T16	T5-T14-T20-T23
T3	T5-T10	T5-T14-T17	T5-T14-T21-T24
T4	T5-T11	T5-T14-T18	T5-T14-T19-T25
T5	T5-T12	T5-T14-T19	T5-T14-T20-T26
T6	T5-T13	T5-T14-T20	T5-T14-T21-T27
T7	T5-T14	T5-T14-T21	T5-T14-T19-T28

• T_{red}

T1	T5-T10	T5-T14-T19-T22
T2	T5-T11	T5-T14-T20-T23
T3	T5-T12	T5-T14-T21-T24
T4	T5-T13	T5-T14-T19-T25
T6	T5-T14-T15	T5-T14-T20-T26
T7	T5-T14-T16	T5-T14-T21-T27
T5-T8	T5-T14-T17	T5-T14-T19-T28
T5-T9	T5-T14-T18	

• T_{Man}

T1-T2-T3-T4-T6-T7-T5-T8-T5-T9-T10-T11-T12-T13-T14-T15

T5-T14-T16-T17-T18-T19-T23-T28-T20-T26-T27-T21-T25-T24-T22

Test suites from Transport protocol case study used to answer RQ1

- T_{trad}

T1: T1	T8: T2-T6-T8	T15: T2-T6-T15
T2: T2	T9: T2-T6-T9	T16: T2-T6-T16
T3: T1-T3	T10: T2-T6-T10	T17: T2-T6-T17
T4: T1-T4	T11: T2-T6-T11	T18: T2-T6-T18
T5: T1-T5	T12: T2-T6-T12	T19: T1-T4-T19
T6: T2-T6	T13: T2-T6-T13	T20: T1-T4-T20
T7: T2-T7	T14: T2-T6-T14	T21: T2-T6-T18-T21

- T_{red}

T1-T3	T2-T6-T9	T2-T6-T15
T1-T4	T2-T6-T10	T2-T6-T16
T1-T5	T2-T6-T11	T2-T6-T17
T2-T7	T2-T6-T12	T1-T4-T19
T2-T6-T8	T2-T6-T13	T1-T4-T20
	T2-T6-T14	T2-T6-T18-T21

- T_{Man}

Tp1: T1-T5-T1-T4-T19-T1-T4-T20-T1-T3-T17

Tp2: T2-T6-T8-T10-T11-T9-T12-T13-T14-T15-T16-T18-T21-T2-T7

E. Results of statistical analysis for section 4.3

One-way ANOVA:

- dependent variable = **cost**
- independent variable = **pair-wise measure**
- test suite diversity = **ADW**

SUMMARY						
<i>Groups</i>	<i>Count</i>	<i>Sum</i>	<i>Average</i>	<i>Variance</i>		
hamming	20	1382	69.1	108.7263		
dice	20	1037	51.85	29.08158		
leveneshetien	20	1319	65.95	91.31316		
ANOVA						
<i>Source of Variation</i>	<i>SS</i>	<i>df</i>	<i>MS</i>	<i>F</i>	<i>P-value</i>	<i>F crit</i>
Between Groups	3375.3	2	1687.65	22.09727	7.86E-08	3.158843
Within Groups	4353.3	57	76.37368			
Total	7728.6	59				

One-way ANOVA:

- dependent variable = **cost**
- independent variable = **pair-wise measure**
- test suite diversity = **KW**

SUMMARY						
<i>Groups</i>	<i>Count</i>	<i>Sum</i>	<i>Average</i>	<i>Variance</i>		
leveneshetien	20	1184	59.2	44.27368		
hamming	20	1126	56.3	48.22105		
dice	20	959	47.95	27.41842		
ANOVA						
<i>Source of Variation</i>	<i>SS</i>	<i>df</i>	<i>MS</i>	<i>F</i>	<i>P-value</i>	<i>F crit</i>
Between Groups	1364.633	2	682.3167	17.07027	1.55E-06	3.158843
Within Groups	2278.35	57	39.97105			
Total	3642.983	59				

One-way ANOVA:

- dependent variable = **cost**
- independent variable = **pair-wise measure**
- test suite diversity = **Sum**

SUMMARY						
<i>Groups</i>	<i>Count</i>	<i>Sum</i>	<i>Average</i>	<i>Variance</i>		
leveneshetien	20	967	48.35	23.81842		
hamming	20	1022	51.1	30.09474		
dice	20	898	44.9	32.72632		
ANOVA						
<i>Source of Variation</i>	<i>SS</i>	<i>df</i>	<i>MS</i>	<i>F</i>	<i>P-value</i>	<i>F crit</i>
Between Groups	386.0333	2	193.0167	6.683443	0.002469	3.158843
Within Groups	1646.15	57	28.87982			
Total	2032.183	59				

One-way ANOVA:

- dependent variable = **cost**
- independent variable = **test suite diversity**
- pair-wise measure = **Dice**

SUMMARY					
Groups	Count	Sum	Average	Variance	
adw	20	1037	51.85	29.08158	
kw	20	959	47.95	27.41842	
sum	20	898	44.9	32.72632	
ANOVA					
Source of Variation	SS	df	MS	F	P-value
Between Groups	485.4333	2	242.7167	8.160709	0.000764
Within Groups	1695.3	57	29.74211		
Total	2180.733	59			

One-way ANOVA:

- dependent variable = **cost**
- independent variable = **test suite diversity**
- pair-wise measure = **Hamming**

SUMMARY					
Groups	Count	Sum	Average	Variance	
adw	20	1382	69.1	108.7263	
kw	20	1126	56.3	48.22105	
sum	20	1022	51.1	30.09474	
ANOVA					
Source of Variation	SS	df	MS	F	P-value
Between Groups	3432.533	2	1716.267	27.52749	4.3E-09
Within Groups	3553.8	57	62.34737		
Total	6986.333	59			

One-way ANOVA:

- dependent variable = **cost**
- independent variable = **test suite diversity**
- pair-wise measure = **Leveneshtein**

SUMMARY					
Groups	Count	Sum	Average	Variance	
adw	20	1319	65.95	91.31316	
kw	20	1184	59.2	44.27368	
sum	20	967	48.35	23.81842	
ANOVA					
Source of Variation	SS	df	MS	F	P-value
Between Groups	3153.633	2	1576.817	29.67562	1.47E-09
Within Groups	3028.7	57	53.13509		
Total	6182.333	59			

One-way ANOVA:

- dependent variable = **mutation score**
- independent variable = **pair-wise measure**
- test suite diversity = **ADW**

SUMMARY						
<i>Groups</i>	<i>Count</i>	<i>Sum</i>	<i>Average</i>	<i>Variance</i>		
leveneshtein	20	1979	98.95	1.833158		
hamming	20	1973	98.65	1.342895		
dice	20	1960	98	2.474474		
ANOVA						
<i>Source of Variation</i>	<i>SS</i>	<i>df</i>	<i>MS</i>	<i>F</i>	<i>P-value</i>	<i>F crit</i>
Between Groups	9.433333	2	4.716667	2.504192	0.090698	3.158843
Within Groups	107.36	57	1.883509			
Total	116.7933	59				

One-way ANOVA:

- dependent variable = **mutation score**
- independent variable = **pair-wise measure**
- test suite diversity = **KW**

SUMMARY						
<i>Groups</i>	<i>Count</i>	<i>Sum</i>	<i>Average</i>	<i>Variance</i>		
leveneshtein	20	1964.9	98.245	2.513132		
hamming	20	1966.25	98.3125	1.318914		
dice	20	1962.95	98.1475	2.932757		
ANOVA						
<i>Source of Variation</i>	<i>SS</i>	<i>df</i>	<i>MS</i>	<i>F</i>	<i>P-value</i>	<i>F crit</i>
Between Groups	6.27525	2	0.137625	0.061033	0.940854	3.158843
Within Groups	128.5313	57	2.254934			
Total	128.8065	59				

One-way ANOVA:

- dependent variable = **mutation score**
- independent variable = **pair-wise measure**
- test suite diversity = **Sum**

SUMMARY						
<i>Groups</i>	<i>Count</i>	<i>Sum</i>	<i>Average</i>	<i>Variance</i>		
leveneshtein	20	1956.8	97.84	2.570684		
hamming	20	1958.15	97.9075	2.968757		
dice	20	1962.95	97.7725	2.16302		
ANOVA						
<i>Source of Variation</i>	<i>SS</i>	<i>df</i>	<i>MS</i>	<i>F</i>	<i>P-value</i>	<i>F crit</i>
Between Groups	10.4475	2	0.522375	0.184973	0.831624	3.158843
Within Groups	160.9718	57	2.824066			
Total	162.0165	59				

One-way ANOVA:

- dependent variable = **mutation score**
- independent variable = **test suite diversity**
- pair-wise measure = **Dice**

SUMMARY						
Groups	Count	Sum	Average	Variance		
adw	20	1960	98	2.474474		
kw	20	1962.95	98.1475	2.932757		
sum	20	1955.45	97.7725	2.16302		
ANOVA						
Source of Variation	SS	df	MS	F	P-value	F crit
Between Groups	14.27583	2	0.713792	0.282867	0.754672	3.158843
Within Groups	143.8348	57	2.523417			
Total	145.2623	59				

One-way ANOVA:

- dependent variable = **mutation score**
- independent variable = **test suite diversity**
- pair-wise measure = **Hamming**

SUMMARY						
Groups	Count	Sum	Average	Variance		
adw	20	1973	98.65	1.342895		
kw	20	1966.25	98.3125	1.318914		
sum	20	1958.15	97.9075	2.968757		
ANOVA						
Source of Variation	SS	df	MS	F	P-value	F crit
Between Groups	5.52825	2	2.764125	1.472743	0.237889	3.158843
Within Groups	106.9808	57	1.876855			
Total	112.509	59				

One-way ANOVA:

- dependent variable = **mutation score**
- independent variable = **test suite diversity**
- pair-wise measure = **Leveneshtein**

SUMMARY						
Groups	Count	Sum	Average	Variance		
adw	20	1979	98.95	1.833158		
kw	20	1964.9	98.245	2.513132		
sum	20	1956.8	97.84	2.570684		
ANOVA						
Source of Variation	SS	df	MS	F	P-value	F crit
Between Groups	12.621	2	6.3105	2.736963	0.073285	3.158843
Within Groups	131.4225	57	2.305658			
Total	144.0435	59				

Two-way
ANOVA:

• Dependent
variable =
Cost

• Independent
variable 1 =
**Test suite
diversity**

• Independent
variable 2 =
**Pair-wise
measure**

Anova: Two-Factor With Replication							
SUMMARY	hamming	dice	leveneshtein	Total			
<i>adw</i>							
Count	20	20	20	60			
Sum	1382	1037	1319	3738			
Average	69.1	51.85	65.95	62.3			
Variance	108.7263	29.08158	91.31315789	130.9932			
<i>kw</i>							
Count	20	20	20	60			
Sum	1126	959	1184	3269			
Average	56.3	47.95	59.2	54.48333			
Variance	48.22105	27.41842	44.27368421	61.74548			
<i>sum</i>							
Count	20	20	20	60			
Sum	1022	898	967	2887			
Average	51.1	44.9	48.35	48.11667			
Variance	30.09474	32.72632	23.81842105	34.44379			
<i>Total</i>							
Count	60	60	60				
Sum	3530	2894	3470				
Average	58.83333	48.23333	57.83333333				
Variance	118.4124	36.96158	104.7853107				
ANOVA							
<i>Source of Variation</i>	<i>SS</i>	<i>df</i>	<i>MS</i>	<i>F</i>	<i>P-value</i>	<i>F crit</i>	
Sample	6056.033	2	3028.016667	62.55175	4.1E-21	3.048833	
Columns	4110.4	2	2055.2	42.45563	1.07E-15	3.048833	
Interaction	1015.567	4	253.8916667	5.244808	0.000522	2.424502	
Within	8277.8	171	48.40818713				
Total	19459.8	179					

Two-way
ANOVA:

- Dependent variable = **Mutation score**

- Independent variable 1 = **Test suite diversity**

- Independent variable 2 = **Pair-wise measure**

Anova: Two-Factor With Replication						
SUMMARY	leveneshtein	hamming	dice	Total		
<i>adw</i>						
Count	20	20	20	60		
Sum	1979	1973	1960	5912		
Average	98.95	98.65	98	98.53333		
Variance	1.833157895	1.342895	2.474474	1.979548		
<i>kw</i>						
Count	20	20	20	60		
Sum	1964.9	1966.25	1962.95	5894.1		
Average	98.245	98.3125	98.1475	98.235		
Variance	2.513131579	1.318914	2.932757	2.183161		
<i>sum</i>						
Count	20	20	20	60		
Sum	1956.8	1958.15	1955.45	5877.9		
Average	97.84	97.9075	97.7725	97.965		
Variance	2.570684211	2.968757	2.16302	2.746042		
<i>Total</i>						
Count	60	60	60			
Sum	5900.7	5897.4	5885.9			
Average	98.345	98.29	98.09833			
Variance	2.441415254	1.906932	2.690675			
ANOVA						
Source of Variation	SS	df	MS	F	P-value	F crit
Sample	9.698111111	2	4.849056	2.089357	0.126915	3.048833
Columns	2.012111111	2	1.006056	0.433488	0.648954	3.048833
Interaction	8.741222222	4	2.185306	0.941603	0.441296	2.424502
Within	396.863	171	2.320836			
Total	417.3144444	179				

F. Examples of EFSMs having different types of counter problem

Stack (Simplest) Example

Test suite 1 (cost = 12, similarity = 0, adequate and feasible)

TP1 = T2, T5, T2, T3, T3, T4, T3, T3, T6, T7, T8

TP2 = T1

Second Example

Test suite 3 (cost = 26, similarity = 2, adequate, partially feasible)

TP1 = T1, T2, T3, T2, T5, T4, T5, T5, T5, T6, T7

TP2 = T2, T5, T5, T5, T6, T8, T5, T4, T5, T5, T5, T6, T9, T10

Third Example

Test suite 3 (cost = 43, similarity = 4, adequate, partially feasible)

TP1 = T1, T5, T5, T5, T6, T7, T6, T7, T4

TP2 = T2, T8, T2, T9, T11, T9, T12, T11, T10, T13, T9, T11, T10, T13, T10, T14, T15, T8

TP3 = T3, T17, T17, T18, T20, T16, T3, T17

TP4 = T3, T17, T17, T18, T19, T18, T21, T2