UNIVERSITY OF BUCHAREST

FACULTY OF
MATHEMATICS AND
COMPUTER SCIENCE

COMPUTER SCIENCE

Bachelor's thesis

SOFTWARE TESTING USING EXTENDED FINITE STATE MACHINES AND GENETIC ALGORITHMS IN HASKELL

Graduate
Carmen-Lorena Acatrinei

Research Supervisor
Lect. Dr. Ana Cristina Iova

Bucharest, June - July 2023

Abstract

Test data generation methods often rely on the use of Finite State Machines (FSMs). To enhance the capabilities of FSMs, an Extended Finite State Machine (EFSM) is introduced. EFSMs incorporate memory (context variables), guards for each transition, and assignment operations, making them more complex than traditional FSMs. When using FSMs, all paths are feasible, but the presence of context variables and guards can result in some paths being infeasible. This can cause issues with feasibility when generating test data using EFSMs. This paper presents an algorithm for generating a test suite for EFSMs exploring the application of Non-dominated Sorting Genetic Algorithm II (NSGA-II). The algorithm's design focuses on creating effective transition paths, also known as test suites, that cover all transitions. This is achieved through the utilization of NSGA-II and two objective functions, with the aim of generating complex test cases for EFSM. The results indicate that the algorithm has great potential and underscore the need for optimization techniques in the testing process. There are many opportunities for further research and development in this area, as the current paper only introduces a novel approach in Haskell. Hence, future work may investigate various optimization techniques and enhancements to build upon these initial findings.

Rezumat

Metodele de generare a datelor de test se bazează deseori pe utilizarea Automatelor de Stări Finite. Pentru a îmbunătăți capacitățile oferite de AF, Automatele de Stări Finite Extinse au fost introduse. Automatele de Stări Finite Extinse sunt mai complexe decât Automatele de Stări Finite obișnuite, prin memorie incorporată (variabile de context), gărzi pentru fiecare tranziție și operații de atribuire. În Automatele de Stări Finite, toate căile sunt fezabile. Prezența variabilelor de context și a gărzilor pe tranziții în Automatele de Stări Finite Extinse conduce la posibilitatea existenței unor căi nefezabile. Aceast lucru poate cauza probleme de fezabilitate îm generarea datelor de test folosind Automatele de Stări Finite Extinse. Această lucrare prezintă un algoritm pentru generarea unui set de teste folosind Automatele de Stări Finite Extinse împreună cu explorarea utilizării NSGA-II (Non-dominated Sorting Genetic Algorithm II). Algoritmul urmărește generarea căilor de tranziții eficiente (seturi de teste) care să acopere toate tranzițiile. Acest scop este îndeplinit prin utilizarea NSGA-II și a două funcții obiectiv, care își propun să genereze seturi de teste complexe pentru Automatele de Stări Finite Extinse. Rezultatele scot în evidență faptul că algoritmul are potențial și subliniază necesitatea tehnicilor de optimizare în procesul de testare. Prin încorporarea acestor tehnici, sistemele controlate software pot fi făcute mai sigure și pot avea o acoperire mai cuprinzătoare. Având în vedere faptul că această lucrare introduce o nouă abordare în Haskell, există numeroase oportunități pentru cercetare și dezvoltare ulterioară. Astfel, direcțiile de cercetare viitoare pot explora diverse tehnici de optimizare adiționale și pot aduce îmbunătățiri pe baza acestor rezultate inițiale.

Contents

Chapter 1

Introduction

Identifying and minimizing potential errors and vulnerabilities is important, and testing plays a vital role in achieving this. To ensure the safety of software-controlled systems, rigorous testing is necessary for secure operations. One commonly used technique in software testing is black-box testing [1], which focuses on verifying the system's functionality without requiring knowledge of its internal structure. Genetic Algorithms and complex systems like Finite State Machines (FSMs) or Extended Finite State Machines (EFSMs) are used in various black-box testing approaches to generate tests.

One effective way to formally represent software systems is using EFSMs, which include states, transitions, actions, and context variables. This provides a thorough and organized representation of the system's dynamics. However, it can be difficult to guarantee the safety and reliability of complex systems as they may generate infeasible paths.

Evolutionary Testing (ET) automatically generates test data based on specific criteria and utilizes evolutionary search algorithms guided by a fitness function, as described in reference [8]. Genetic Algorithms (GA) belong to the evolutionary algorithms and are modeled after the mechanisms found in natural Darwinian evolution.

A highly advanced testing automation method is Model-Based Testing (MBT), which involves automating the test design process. This is achieved by creating abstract models of the system's behavior, using those models to generate test cases, executing the tests, and comparing the actual conduct of the system to the expected behavior defined by the models. One of the benefits of MBT is its ability to link tests directly to the System Under Test (SUT) requirements. This makes tests easier to read, understand, and maintain. In addition, MBT helps ensure that testing is conducted on a repeatable and scientific basis [27].

Non-dominated Sorting Genetic Algorithm II (NSGA-II), a multi-objective evolutionary algorithm, has achieved widespread use in optimization problems over time. Its ability to handle multiple objectives makes it ideal for optimizing the test suite for EFSMs with different coverage criteria. To achieve maximum coverage and expose potential faults in the EFSM, NSGA-II can generate an effective selection of test cases [5].

This paper presents a method for generating test cases (feasible transition paths) for a specific EFSM. This technique uses NSGA-II and considers transition coverage criteria and interdependence among transitions. The goal is to examine the potential of NSGA-II in testing EFSMs and using its optimization features to create complex test suites. Although this paper does not necessarily improve the previous research presented in [19], it does offer a fresh, adapted approach in Haskell that could serve as a solid foundation for future work.

The rest of the paper is structured as follows: Chapter 2 introduces basic concepts about Extended Finite State Machines, data flow dependence, Haskell programming language, genetic algorithms, Genetic Algorithm Library for Haskell [7], and multi-objective evolutionary algorithm. Chapter 3 presents some related work. Chapter 4 presents the proposed approach, Chapter 5 shows the evaluation and results, while Chapter 6 holds the conclusion and future work.

Chapter 2

Preliminaries

2.1 Extended Finite State Machine

A *finite state machine* (FSM) is a transducer with a finite set of states, inputs, and outputs. Every transition has a start state, an end state, an input, and an output.

An *Extended Finite State Machine* (EFSM) is a mathematical concept that comprises elements denoted by $(Q, \Sigma_1, \Sigma_2, I, V, \Lambda)$ [2]. The elements include a finite set of states represented by Q, a finite state of events denoted by Σ_1, a finite set of initial states denoted by Σ_2, a finite set of Inputs denoted by I, a finite set of global variables (that can be accessed from any state) represented by V, and a finite set of transitions known as Λ. A transition will be represented by a starting state, an input (that may have associated input parameters), a condition (logical expression) called guard, a sequential operation (involving a method with assignments and output parameters) called action, and the end state.

A transition has several components involved. These include a starting state, an input (which may have linked input parameters), a condition known as a guard (which is a logical expression), a sequential operation referred to as an action (involving a method with assignments and output parameters), and finally, an end state.

To clarify our proposed method better, we will use the EFSM illustrated in Fig.2.1 throughout this paper. EFSM testing requires input sequences containing parameter values representing individual test procedures. There are three states, six transitions, and three context variables (v_1, v_2, and v_3) in M.

The EFSM's path is a series of connected transitions, denoted by $p = S_1 \xrightarrow{f_1[g_1]} S_2 \xrightarrow{f_2[g_2]} S_3 \xrightarrow{f_3[g_3]} ... \quad S_m \xrightarrow{f_m[g_m]} S_{m+1}$. Each S_i represents the state at that point in the path, while f_j and g_j refer to the method and guard executed on the transition j. According to [20], a path is considered feasible if there are input parameter values that meet all the guards' conditions and trigger all the transitions for that path. If not, it is considered infeasible.

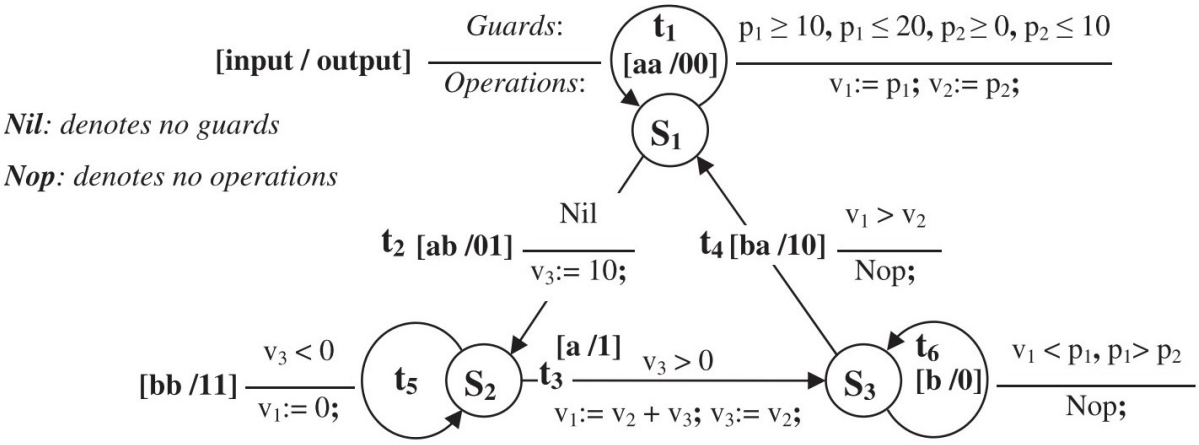


Figure 2.1: An EFSM Example (M) [1]

2.2 Data flow dependence

In this section, the feasibility metric introduced by Kalaji et al. in [1] will be described. This metric serves as a foundation for the first objective function used in this paper, which is explained in detail in Section 4.4.

When working with a program and a variable v, it's important to note that any statement in which the variable appears can be a usage of the variable, an assignment to the variable, or even both. When an *assignment* is made to the variable v, it defines or modifies its value. At that particular statement, it is considered that v has been *defined*. Whenever v is referenced in a predicate, it is known as a predicate use or p-use. Similarly, when v is referenced in a computation that either updates its value or generates it as output, it is known as a computation use or c-use.

According to [17], if there is a path between two statements s_i and s_j where variable v is not defined after s_i and before s_j, then that path is considered a *definition clear path* for variable v. When s_i defines the value of v and s_j uses the value of v, it forms a definition-use (du) pair for v. This creates a dataflow dependence between s_i and s_j [22].

Any transition has guards and operations. A guard is made up of atomic guards that are combined using logical operators AND and OR. A guard is denoted by (e *gop* e'). The expressions e and e' are accompanied by a guard operator *gop*, which can be one of the following: $\{<, \leq, >, \geq, =, \neq\}$. The notations and definitions used in this subsection were introduced by Kalaji et al. in [1].

To indicate the variables present in a specific expression e, the notation $Ref(e)$ is used. Based on the expressions e and e', a transition guard can be classified into the following types [9].

1. g^{pv}: compares a parameter and one or more context variables, where $Ref(e) \cup Ref(e')$ contains a parameter and also context variables;

2. g^{vv}: compares the values of context variables. The comparison involves that every element of $Ref(e) \cup Ref(e')$ is a variable, and neither e nor e' is a constant;

3. g^{vc}: compares a constant and an expression involving context variables. $Ref(e) \cup Ref(e')$ contains at least one context variable; either e or e' is a constant;

4. g^{pc}: compares a constant and an expression involving a parameter. There exists $p \in Ref(e) \cup Ref(e')$ (a parameter), and either e or e' is a constant;

5. g^{pp}: compares expressions involving parameters. There exists $p \in Ref(e) \cup Ref(e')$ (a parameter), and neither e nor e' is a constant.

When dealing with a transition t, an assignment is expressed as $v = e$, with v representing a context variable and e representing an expression. Assignments to context variables can be classified into the following categories:

1. op^{vp}: assigns a value to v based on a parameter, with p being a parameter in $Ref(e)$;

2. op^{vv}: assigns a value to v based on the context variable(s); the expression e is comprised solely of variables and is not a constant;

3. op^{vc}: assigns a constant value to v, with e being a constant.

4. *nop*: there is no assignment.

There are two types of transitions based on guard and assignment classifications: *affecting* and *affected-by* transitions [1].

A transition t_i from a TP $t_1, t_2, ..., t_n$ that has an assignment $op \in \{op^{vp}, op^{vc}, op^{vv}\}$ to v is *affecting* if there exists a guarded transition $t_j \in TP$, where $1 \leq i < j \leq n$, t_j has a guard $g \in \{g^{pv}, g^{vv}, g^{vc}\}$ over v and the path between t_i and t_j is a definition clear with respect to v. t_j is also called an *affected-by* transition.

In some cases, we can immediately determine if a path is infeasible, and so we give that path poor (high-value) fitness. Next, we will provide the rules for when a path is infeasible [1].

An assignment $op \in op^{vc}$ in t_i is *opposed* to the guard $g \in g^{vc}$ of t_j (in a TP $t_1, t_2, ..., t_n$) with $1 \leq i < j \leq n$ if there exists a variable v such that op is an assignment to v, g refrences v, $t_{i+1}, ..., t_j$ is a definition clear path for v and the constants in op are either the same and $gop \in \{>, <, \neq\}$ or they are different and $gop \in \{=\}$.

The guards $g_i, g_j \in g^{vc}$ of t_i and t_j (in a TP $t_1, t_2, ..., t_n$) with $1 \leq i < j \leq n$ are *opposed* if there exists a variable v such that both guards reference v, the path from t_i to t_j is definition clear for v and one of the following is true:

1. The constants in g_i and g_j are identical, and (one $gop \in \{>, <, \neq\}$ and the other $gop \in \{=\}$ or one $gop \in \{>, \geq\}$ and the other $gop \in \{<\}$ or one $gop \in \{<, \leq\}$ and the other $gop \in \{>\}$).

2. The constants differ, and both $gop \in \{=\}$.

A TP $t_1, t_2, ..., t_n$ with length $n > 1$ is *definitely infeasible* if there exists $1 \leq i < j \leq n$ so that one of the following is true [1]:

1. t_i is an affecting transition with operation $op \in op^{vc}$, t_j is an affected-by transition with guard $g \in g^{vc}$ and op opposes g;

2. The guard g_i of t_i and g_j of t_j are of type g^{vc} and g_i opposes g_j.

The penalty values used in the feasibility metric are described in this section. The goal of a penalty value is to estimate how easily a given guard can be satisfied in the TP. Three factors are considered when assigning a penalty to a pair of (*affecting, affected-by*) pair [1]:

1. **Guard Type** - for example, a guard of the type g^{pv} is usually easier to satisfy because the parameter's value can be chosen. At the same time, g^{vc} can be considered the hardest because there is no option to select the values v or c.

2. **Guard Operator** - for example, the operator $\neq$ is usually the easiest to satisfy, while $=$ is the most difficult.

3. **Operation Type** of an *affecting* transition - for example, op^{vc} is the worst because the value of c cannot be selected. In contrast, op^{vp} is favorable because the parameter allows you to try to find a suitable value for v.

When assigning a penalty, in addition to the penalty between the (*affecting, affected-by*) pair, if there is a guard that is not affected by any operation, only the first two factors are considered.

The penalty values used in this work are based on [1], shown in Table 2.1a. INF represents a large positive integer (1×10^5), indicating that the path is definitely infeasible. '-' is used to indicate that the choices op^{vp}, op^{vv}, and op^{vc} are irrelevant for cases where there are no affecting transitions.

One way to provide a guard is by using nested IFs or predicates linked by AND and OR. If OR links guards, the minimum penalty is used, but if they are connected by AND or represented as nested IFs, the penalties are added up [18]. Depending on one or more context variables, the relationship between (*affecting, affected-by*) transitions can occur, and an *affected-by* transition can be affected by one or more transitions in a TP. Consequently, each dependency between two (*affecting, affected-by*) transitions is recorded, along with the context variable where the dependency occurs.

Three types of assignments represented by an integer can be distinguished: **-2** is used to mean an assignment of a constant (op^{vc}), **-1** is used to indicate an assignment of a parameter (op^{vp}). In contrast, **0** means no assignment (nop). A number in $[1..m]$ (with m

Rows ID	Guard and operator	Assignment			
		(nop)	(op^{vp})	(op^{vv})	(op^{vc})
1	$g^{pv}(=)$	4	8	16	24
2	$g^{pv}(<,>)$	3	6	12	18
3	$g^{pv}(\leq,\geq)$	2	4	8	12
4	$g^{pv}(\neq)$	1	2	4	6
5	$g^{vv}(=)$	16	20	40	60
6	$g^{vv}(<,>)$	12	16	32	48
7	$g^{vv}(\leq,\geq)$	8	12	24	36
8	$g^{vv}(\neq)$	4	8	16	24
9	$g^{vc}(=)$	40	30	60	INF if False and 0 otherwise
10	$g^{vc}(<,>)$	32	24	48	INF if False and 0 otherwise
11	$g^{vc}(\leq,\geq)$	24	18	36	INF if False and 0 otherwise
12	$g^{vc}(\neq)$	16	12	24	INF if False and 0 otherwise
13	$g^{pc}(=)$	12	-	-	-
14	$g^{pc}(<,>)$	8	-	-	-
15	$g^{pc}(\leq,\geq)$	4	-	-	-
16	$g^{pc}(\neq)$	1	-	-	-
17	$g^{pp}(=)$	6	-	-	-
18	$g^{pp}(<,>)$	4	-	-	-
19	$g^{pp}(\leq,\geq)$	2	-	-	-
20	$g^{pp}(\neq)$	1	-	-	-
21	g_i opposes g_j	INF	-	-	-

(a) The suggested penalty values where INF is a large positive integer to indicate that a given dependency represents an infeasible case [1].

being the number of context variables in the EFSM) represents the corresponding context variable on the assignment's right-hand side. If the assignment of type op^{vv} refers one or more context variables, one of them is chosen in the calculation to shorten the time required to compute the feasibility metric. The possibility of setting the value of the context variables may not be that important if it is possible to set the value of one of them easily.

For every pair (t_i, t_j), we can represent the dependency as a $(n+2)$-tuple, where n is the number of context variables in EFSM. The first n fields hold the dependency and the penalty for each variable; the $(n+1)^{th}$ field, gp, records the guards' penalties summed that do not contain context variables. The last field is a Boolean that records whether there is a dependency between t_i and t_j. These tuples are stored in a matrix to represent the dependencies and penalties among all transitions in a given path from an EFSM.

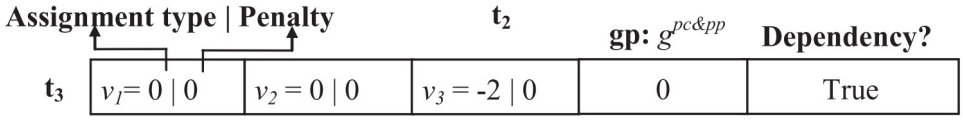


Figure 2.2: Dependencies and Penalties Tuple Example for EFSM showed in Fig.2.1 [1].

The feasibility metric algorithm requires two inputs: the specified path $(t_1, ..., t_n)$ and

the dependencies matrix [1]. The algorithm returns low values for paths that have easily identifiable input sequences. The algorithm identifies the last transition as a possible *affected-by* transition and analyzes which previous transitions are *affecting*. Once the algorithm identifies a pair of $(affecting, affected-by)$, it enters a loop to determine which context variable carries a dependency or penalty. Two cases are distinguished:

1. dependency type is in $[-2, 0]$: the corresponding penalty is added, and the related variable is set as visited;

2. dependency type is > 0: the dependency may continue by an assignment referring context variables. The variable is set as visited, and the dependency continues if the corresponding penalty is > 0. The penalty is added, and all the previous assignments propagated to the current context variable are detected using the method *check*.

The *check* method is a recursive function that analyzes data dependencies. It starts with a context variable and the affecting transition passed to the call, then traces backward to identify all the previous transitions that could affect the value of the context variable. If a previous transition affects the context variable, the subroutine will find the type of the assignment. When the assignment type is either -2 or -1, it indicates that the context variable is assigned with a constant or parameter value (op^{vc} or op^{vp}). In such cases, the subroutine penalizes referencing a constant with 60 and a parameter with 20. After that, the subroutine stops because it implies that no other previous transition affects this assignment. If the assignment type is greater than 0, the context variable is assigned a different context variable called v'. The subroutine penalizes this referencing by 40 and repeats the process by calling *check* with the current transition and v'. If the dependency is *open-ended*, meaning it depends on an undefined initial value of a context variable, 60 is added. After the subroutine stops, it provides the total sum of the incurred penalties. Once the current pair of transitions has been scanned, a new cycle begins to detect any potential relation and penalty between the following pairs, and so on [21]. A high-level description of the algorithm can be found in paper [1].

2.3 Haskell Programming Language

Haskell is a programming language that is purely functional and statically typed. It is known for having a solid type system, lazy evaluation, and emphasizing pure functions and referential transparency [13]. It approaches the principles of functional programming, which involves using immutable values and functions to modify them.

The type system in Haskell is designed to ensure safety, catching many potential errors during the compilation process [13]. It uses a solid type inference system that

can determine a lot about each piece of code. Therefore, explicit labeling is not always necessary. With its support for pattern matching, algebraic data types, and type classes, the language offers a range of expressive options for defining data structures and generic functions.

Monads are a way to organize and control actions that have side effects [13]. They allow impure actions to be done within a pure framework. Using monads, impure actions are put into a particular type that shows a context or effect. This helps manage side effects and shows their importance.

With Haskell's lazy evaluation, functions and calculations are not executed until a result is required or specifically requested. This approach helps optimize performance and timing. Haskell programs are shorter than their imperative counterparts because they use high-level concepts. This makes them more concise and easier to maintain, with fewer bugs [13].

2.4 Genetic Algorithms

Genetic Algorithms (GAs) are metaheuristic search techniques applied in optimization problems [1]. The solution consists of genes and is represented by chromosomes. According to [14], GAs consist of several components, including a population of chromosomes, selection based on fitness, crossover to create new offspring, and random mutation of the resulting offspring.

GAs involve simultaneously evolving multiple chromosomes to discover the best possible solution. According to Lefticariu and Ipate's approach (as per [12]), where a positive fitness function is used, an individual achieving a fitness score of 0 is regarded as the solution. The fitness score is determined by the proximity of the chromosome to the optimal solution.

Algorithm 1 Genetic Algorithm [14]

1: randomly generate a population of n chromosomes
2: compute the fitness function $f(x)$ for each chromosome x from the population
3: **repeat**
4: select a pair of parent chromosomes from the current population with some probability
5: apply crossover with some probability (p_c) to create two offspring; in the case of no crossover, create two offspring that are identical copies of their parents
6: apply mutation to the two offspring with some probability (p_m) and add the resulting chromosomes in the new population
7: **until** n offsprings are created
8: replace the current population with the new one
9: go to step 2

A GA works as shown in Algorithm 1. A single cycle of this procedure is called a

generation. For every problem, the number of generations to be iterated is specified. The entire set of generations is called a *run*. After a running session, finding one or more highly fit chromosomes in the population is expected. The role of randomness is significant in every run, which means that running the program with different random-number seeds will result in distinct behaviors. Typically, the statistics provided are obtained by averaging the results of multiple runs of the GA on the same problem. These statistics may include the best fitness achieved in a run and the generation at which the individual with that best fitness was identified [14].

2.5 Genetic Algorithm Library for Haskell

Genetic Algorithm Library for Haskell [7] offers a range of functions and types to make it easier to implement and execute genetic algorithms in Haskell. The "Moo.GeneticAlgorithm" package is part of the "moo" collection, which stands for "Monads for Multi-Objective Optimization". Its main goal is to provide tools for solving problems related to multi-objective optimization through monadic programming in Haskell.

Users can design custom types and fitness functions with the library to represent individuals and the problem space. The library also includes a range of genetic operators, including selection strategies like tournament selection, crossover methods such as one-point crossover, and mutation operators like Gaussian mutation.

2.6 Multi-objective Evolutionary Algorithm

Many factors are usually involved when dealing with *Multi-Objective* problems that have multiple potential solutions. To address these types of problems, numerous GAs have been adapted [4]. Solutions closer to the ideal outcome provide various tradeoffs among objectives, are not dominated by any other possible solutions, and form what is known as the Pareto front [3].

Non-dominated Sorting Genetic Algorithm II, which is also referred to as NSGA-II [5], is a genetic algorithm that employs a powerful approach. After every round of evolution, the algorithm arranges the parent and offspring populations into various fronts based on the non-dominance factor among individuals. The crowding distance technique maintains the diversity of solutions within each front.

Algorithm 2 Main Loop of the NSGA-II Algorithm [5]

1: combine parent and offspring population
2: $F = (F_1, F_2, ...)$, all non-dominated fronts of R_t
3: until the parent population is not filled
4: calculate crowding distance in F_i
5: include i^{th} non-dominated front in the parent population
6: sort in descending order using the partial order $\preceq_n$
7: choose the first $(N - |P_{t+1}|)$ elements of F_i
8: use selection, crossover, and mutation to create a new population Q_{t+1}
9: increment the generation counter

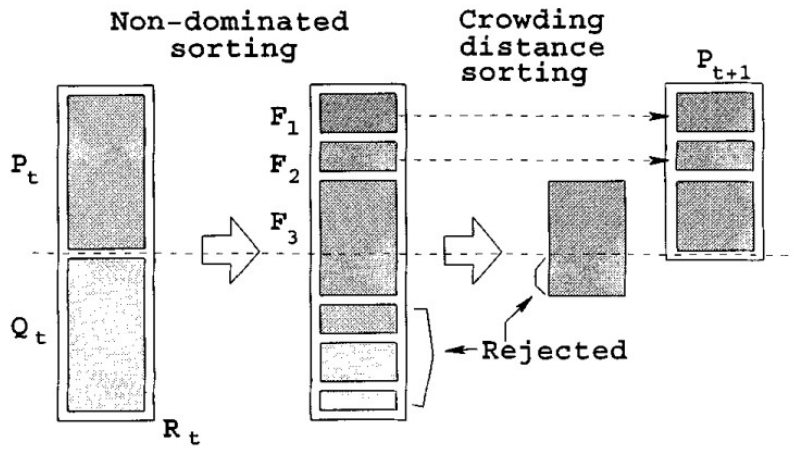


Figure 2.3: NSGA-II Procedure [5]

Chapter 3

Related Work

Generating test data from an extended finite state machine (EFSM) is a complex issue, despite the availability of several ET-based techniques for automating this process from code. This chapter will briefly overview past approaches that have used evolutionary algorithms for path generation in EFSMs.

In their research paper [1], Kalaji et al. approached a search-based method consisting of two phases. The first phase involves utilizing a Genetic Algorithm to create a feasible TP (FTP), with the dataflow dependence metric used as the fitness function. The second phase employs another Genetic Algorithm to find an input sequence capable of triggering the TP, combining the branch distance function and approach level as the fitness function.

In the paper [6], a technique is described for identifying a test suite (a series of input sequences) that corresponds to a particular EFSM. The method involves searching for specific paths that meet certain criteria. The authors use a fitness function to estimate the ease of discovering an input sequence that can trigger a given path through an EFSM. If the algorithm fails to find an input sequence that triggers the desired path, it will repeat searching for an alternative path with good fitness.

A method that generates a set of feasible TPs that satisfy the test criterion is proposed in [10]. This approach uses both control and data analysis that expands upon the TP fitness metric from [16] to identify if a given TP contains a transition that refers to a counter variable in its guard, which additional transitions are necessary for this TP, and how many times they should be called.

The paper referenced as [3] introduces a technique and a software tool that can create test suites from EFSMs while considering multiple objectives. The authors aim to maximize a test suite's coverage and feasibility while reducing the similarities between its test cases and minimizing the total cost. To accomplish this, a multi-objective genetic algorithm has been suggested. It seeks out the best test suites using four objective functions.

In the paper [23], a path-oriented technique for producing test cases for EFSM is suggested. The approach tackles two issues: test adequacy criterion and path searching.

The former is addressed by utilizing basis path coverage. At the same time, the latter involves introducing a new evaluation metric method and implementing techniques to reduce the number of feasibility checks to enhance efficiency.

In [24], a new multi-objective version of the generalized extremal optimization algorithm is introduced. The main use of this tool is to generate tests that discover transition paths from EFSMs. It considers both transition coverage and test length minimization. The paths and associated data are created through the evolution process, which includes the generation of parameter values. To avoid the issue of generating infeasible paths, which this paper does, it is better to obtain the path dynamically rather than solely analyzing the model's structure as is typically done.

The paper [15] represents the test data generation problem as an optimization problem. To generate test cases, heuristics are employed, including genetic algorithms (GAs) and random search. These methods are used to create test data and assess the approach. GAs have been found to outperform random search and seem to exhibit well scalability as the problem size increases. A straightforward fitness function that can be effortlessly adapted for other evolutionary search techniques is employed.

The paper referenced as [25] outlines an evolutionary method for generating test sequences from EFSM. The problem of producing infeasible paths is avoided using an executable model that dynamically obtains feasible paths. An evolutionary algorithm is employed to identify the desired transition that meets a specific test purpose. To determine the parts of the model that impact the test objective, slicing information is obtained using the target transition as a criterion. Furthermore, a multi-objective search is conducted, prioritizing test purpose coverage and sequence size minimization, as longer sequences require more time and resources.

The paper [26] introduces an approach called MOST for generating test cases from extended finite state machines (EFSM) using a multi-objective evolutionary method. The method implemented by the authors to prevent the generation of infeasible paths involves dynamically obtaining viable paths while considering two objectives - the coverage criterion and the length of the solution. MOST generate a set of optimal solutions called Pareto set approximation. This set allows the test team to choose solutions that balance the two objectives well.

Chapter 4

Proposed Approach

This paper presents a new approach to the method from [19]. It does not necessarily improve the past method. Still, it makes a new departure by implementing the algorithm in Haskell and using NSGA-II (adapted from [7] along with other GA methods) with two custom objective functions. The approach was executed in a different programming language, so creating every function and data type from scratch was necessary. The upcoming sections will outline the implementation approach and the methods employed while including relevant examples.

Algorithm 3 The key stages of the Algorithm

1: parse the input EFSM
2: run the NSGA-II algorithm starting with a randomly generated initial population
3: **repeat**
4: select the pairs of parent chromosomes from the current population using Binary Tournament Selection
5: apply crossover between the pairs of parents
6: apply mutation to the two offspring with some probability
7: compute objective functions for each chromosome to evaluate the offspring population
8: apply the replacement operation on both the parent and offspring populations to compute the new population
9: **until** the number of total evolutions is reached **or** the length of the first front stayed the same for ten generations.

4.1 EFSM Encoding

As previously stated, the proposed approach was developed using Haskell, requiring everything to be built from the ground up. To use each type, it was necessary to define new data types. Specifically, for an EFSM, the following `newtypes` were defined: `State`, `VarMem`, and `Input`. Each has the type `String` and its constructor and represents a state,

```
data Transition =
    Transition {
                name :: String,
                s1 :: State,
                s2 :: State,
                condition :: Condition,
                input :: [Input],
                operations :: [Operations]
                }

data EFSM =
    EFSM {
        states :: [State],
        transitions :: [Transition],
        vars :: [VarMem],
        start :: State
        }
```

Figure 4.1: Transition and EFSM representation in the proposed approach with the new data types defined explained in 4.1

a context variable, and the input value. Additionally, `Cint` means a constant with its constructor and the type `Int`.

For a transition, it was necessary to have a guard and an assignment, represented by new `data` types, as follows:

- `ExpAr` is a recursive type and can be represented by a `Integer`, a variable or a parameter, or by applying an arithmetic operation (addition `:+:`, subtraction `:-:`, multiplication `:*:`, division `:/:`, or modulo `:%:`) between two expressions (e.g., `ExpAr :+: ExpAr`);
- `Condition` is a recursive type and can be represented by `Nil` (no condition), `T` (True), `F` (False), boolean operation between two arithmetic operations (`Eq`, `Dif`, `Lt`, `Lte`, `Gt`, `Gte`, denoting the comparison operators $=, \neq, <, \leq, >, \geq$), or by applying logical operators (AND `:&:` or OR `:|:`) between two conditions (e.g., `Condition :&: Condition`, `Lt ExpAr ExpAr`);
- `Operations`, which represents the assignment of a variable (`Atrib VarMem ExpAr`).

The definitions for what is required for a transition and EFSM were previously established. The actual data type for a transition and EFSM is defined in Figure 4.1. Each `Transition` has a name (e.g. $t0, t1$), a start state (`s1`), a destination state (`s2`), a guard (`condition`), a list of inputs (`input`) and a list of assignment operations (`operations`). An EFSM consists of a list of states (`states`), a list of transitions (`transitions`), a list of context variables (`vars`), and a starting state (`start`).

```
transition = [tran1, tran2, tran3, tran4, tran5, tran6]

tran5 = Transition "t5" (S "s2") (S "s2") (Lt (Var (V "v3")) (Const
   0)) [I "bb"] [Atrib (V "v1") (Const 0)]
tran6 = Transition "t6" (S "s3") (S "s3") (Lt (Var (V "v1")) (Param
   "p1") :&: Gt (Param "p1") (Param "p2")) [I "b"] []

efsm =
    EFSM {states = [S "s1", S "s2", S "s3"], transitions =
       transition, vars = [V "v1", V "v2", V "v3"], start = S "s1"}
```

Figure 4.2: An input example for the EFSM presented in Fig. 2.1, where `transition` is the list of transitions, `tran5` and `tran6` and `efsm` are examples of how a transition and the input EFSM are stored.

For an example of the representation of transitions and EFSM, the EFSM M shown in Fig. 2.1 is considered. Its transitions are stored as a list of `Transition` presented in Fig. 4.2, along with an example of two transitions and the EFSM.

Once the new data types had been declared, verifying the validity of a given path became necessary. The functions depicted in Figure 4.3 were employed for this purpose. The first function, called `isValid1`, verifies if all the transitions in the provided path are present in the input EFSM. The second function, named `isValid2`, verifies whether the initial state of the path matches the initial state of the provided EFSM input. The third function, called `isValid3`, confirms whether two transitions from the path (t_i, t_j) with t_i coming before t_j in the path have corresponding destination and starting states, respectively. In the end, the function `isValid` verifies that the three previously described functions are true and ensures that the path list is not empty.

4.2 Chromosome Encoding

A chromosome is represented by a *set of paths* with variable length. A limit on the total number of genes is set to control the bloat phenomenon caused by varying lengths of chromosomes [3]. There can only be one path for each transition to achieve transition coverage. Therefore, the limit is set to be equal to the total number of transitions.

In paper [1], the encoding used represents paths as lists of integers. That ensures that every gene defines a valid path. This approach will use the same method, and there is no need to filter the paths or eliminate invalid ones. The rest of the subsection presents chromosome encoding, using notations and definitions introduced by the authors from [1].

For a given EFSM, the following are defined:

- k - the number of states;
- $n_1, n_2, ..., n_k$ - the number of transitions leaving each state;

```
isValid1 :: Path -> Bool
isValid1 (P[]) = True
isValid1 (P(p:paths)) = elem p (transitions efsm) && isValid1 (P
    paths)

isValid2 :: Path -> Bool
isValid2 (P[]) = False
isValid2 (P paths) = s1 t0 == start efsm
                        where t0 = head paths

isValid3 :: Path -> Bool
isValid3 (P paths) = and [s2 t1 == s1 t2 |(t1, t2) <- zip paths
    (tail paths) ]

isValid :: Path -> Bool
isValid (P paths) = isValid1 (P paths) && isValid2 (P paths) &&
    isValid3 (P paths) && paths /= []
```

Figure 4.3: Functions used for checking the validity of a given path in the input EFSM, with `Path` being a list of transitions with a specific constructor `P`.

- LCM - the lowest common multiple of $n_1, n_2, ..., n_k$;
- $r_1, r_2, ..., r_k$ - ranges having $r_i = LCM/n_i$

A chromosome $C = \{i_1, i_2, ..., i_n\}$, with $i_j \in [1, ..., LCM]$ is transformed into a transition path $t_1, t_2, ..., t_n$ computing each t_j as follows:

- s_c - the current state;
- r_c - the range that corresponds to the current state s_c;
- n_c - the number of transitions leaving the current state s_c;
- find m such that $i_j/r_c \in [m, m+1], m < n_c, n \in \mathbf{N}$;
- t_j is the m-th transition leaving the current state s_c.

For this specific algorithm, the `Gene` type is defined as `Genome Int`, with `Genome` derived from the reference [7] and indicating a list of the designated type. A `Chromosome` is essentially a list of genes and is therefore represented as `Genome Gene`.

Once a valid path was established, the data flow dependencies and assigned penalties from section 2.2 were integrated. The Chromosome Encoding process was initiated after determining the penalty for each path. This involved transforming each path into a gene for a chromosome, using the function outlined in Fig. 4.4.

```
geneToPath :: State -> Gene -> Path
geneToPath (S st) [] =  P []
geneToPath (S st) (g:gene) = P (t : x)
                      where
                          P x = geneToPath st2 gene
                          m = g `div` getRangeForState (S st)
                          t = leavingStateS (S st) !! m
                          st2 = s2 (getmThTransLeavingStateS (S st) m)
```

Figure 4.4: Function used to transform a path into a chromosome gene. `getRangeForState` computes the range for each state as described in the current section, `leavingStateS` computes the list of transitions leaving the given state, and `getmThTransLeavingStateS` gives the m^{th} transition that leaves the given state.

4.3 Genetic operators

For the chromosomes to cross during each step, the *Selection Operator* chooses a pair of two from the population. As outlined in Algorithm 3, the Binary Tournament Selection method is used to make this selection.

The *Crossover Operator* creates two new chromosomes from two existing parents chromosomes. In this paper, the crossover is applied with a probability of 0.5 using one point crossover (selects a random point in two genes and swaps them beyond this point) provided by the Genetic Algorithm Library for Haskell [7].

The *Mutation Operator* can change a chromosome with a specific probability (in this paper, the probability is 0.75). A custom mutation operator was implemented, and it functions in the following manner: it selects each gene in a chromosome with the given probability and replaces it with a new, randomly generated one. The returned type `MutationOp` belongs to [13], and the implementation of the custom mutation operator can be found in Fig. 4.5.

```
myMutation :: Double -> MutationOp Gene
myMutation p vars =  mapM mutate vars
  where
    mutate = withProbability p f
    f v = sequence [getRandomR (0, upperBound) | i <- [1..randomInt2
       1 pathSize]]
```

Figure 4.5: The Mutation Operator implementation. `MutationOp`, `withProbability` and `getRandomR` are a type and method from [7]. `randomInt2` generates a random value within the range (1, `pathSize`), with `pathSize` being half the value of the actual size of a path. `upperBound` represents the lowest common multiple of the number of transitions leaving each state from which the value 1 is substracted

Every step of the algorithm was implemented using `stepNSGA2bt` from Genetic Algorithm Library [7], along with the two objective functions described in Section 4.4, the

crossover operator and the mutation operator described above with their given probability. The `runGA` function found in [7] was used to execute the algorithm. The initial population, a stopping condition, and the step described above were provided for the function. The `stop` condition used methods and data types from [7] and would be triggered if either the maximum number of generations was reached or if the length of the first front remained the same for ten evolutions. If the length of the best solutions from the first front remained unchanged, it suggests that the population has stabilized. Only solutions meeting specific criteria were considered. These criteria required the first metric to be lower than INF (more likely to be feasible; in this paper, INF = 10^5) and the second metric to be equal to or less than one (covers all transitions).

4.4 Objective Functions

This approach uses two objective functions based on path feasibility and transition coverage. The upcoming subsections will provide a detailed explanation of the objective functions.

4.4.1 Path Feasibility - the first objective function

The first objective function is built upon the feasibility metric presented in [1]. This metric considers the dataflow dependencies among transitions and guides the search toward possible feasible transitions (transitions for which associated input sequences exist). The algorithm assigns penalties to each transition based on their potential impact on feasibility, estimating the ease with which the guard can be satisfied. Rather than seeking paths that decrease this metric, this approach aims to discover complex yet feasible paths. The feasibility metric is calculated for an individual (a set of paths) by summing the feasibility values for each gene (path).

When checking for feasibility, the metric will give a value (the computed feasibility value) greater than INF (in this paper, INF= 10^5) if a path is not feasible. If the path is feasible and the computed feasibility value (val) is less than INF, the function will return $1 - (val/INF)$. The algorithm cannot find a solution with a fitness score of 0, but it will halt after reaching the stop criteria (the total number of evolutions is achieved or the length of the first front stays the same for ten evolutions). The closest to optimal chromosome in the population is expected to have a fitness value close to zero, indicating a feasible path that is more complex in terms of transition dependencies. The Path Feasibility implementation can be found in Fig. 4.6.

```
fitness1 :: Chromosome -> Double
fitness1 chr = roundDec $ sum (complexity chr)

complexity chr =  [if val > inf then fromIntegral val else (1 -
   ((fromIntegral val :: Double) / fromIntegral inf)) | gene <-
   chrGenes chr, let val = compute (geneToPath (start efsm) gene)]
```

Figure 4.6: The first objective function, the Feasibility Metric computation. `roundDec` is an auxiliary method that truncates the result to 4 decimals, `inf` is 10^5, `chrGenes` converts from `Chromosome` to `[Gene]`, and `compute` is a method that computes the penalties for a gene (a more detailed description for this method can be found in [1]).

4.4.2 Transition Coverage - the second objective function

The second objective function is centered on transition coverage, and its computation is as follows: if all transitions have been covered, the total number of transitions used by all paths is counted, and that number is returned. If not all transitions are covered, we return INF. This function aims to cover all transitions while minimizing their usage frequency and preventing transition cycles. The count of occurrences for all covered transitions is normalized in $[0, 1]$. If there are transitions that are still not covered, a penalty is imposed, which increases the total objective value beyond 1. The Transition Coverage implementation can be found in Fig. 4.7.

```
fitness2 :: Chromosome -> Double
fitness2 chr = (roundDec resultFinal) + fromIntegral zeros
 where
    result1 = countRes chr
    penalties = computePenalties chr penaltiesAux
    zeros = length [p | p <- penalties, p == 0]
    resultFinal = norm (fromIntegral result1)

    countRes [] = 0
    countRes (gene:chr) = res + countRes chr
            where
                tp = geneToPath (start efsm) gene
                res = if countResAux tp == 0
                        then 10000
                        else countResAux tp

    countResAux :: Path -> Int
    countResAux (P []) = 0
    countResAux (P tp) = sum [1 | t <- tp]

    tran = transition
    penaltiesAux = [0 | i <- [1..length transition]]
    computePenalties [] p = p
    computePenalties (gene:chr) pen1 = computePenalties chr pen
        where
            tp = geneToPath (start efsm) gene
            pen = computePenaltiesAux tp pen1
            computePenaltiesAux (P []) pc = pc
            computePenaltiesAux (P(x:xs)) pc = computePenaltiesAux
               (P xs) (func tran pc x )
            func [] [] x = []
            func (x:xs) (y:ys) z = [if z == x then y + 1 else y] ++
               func xs ys z
```

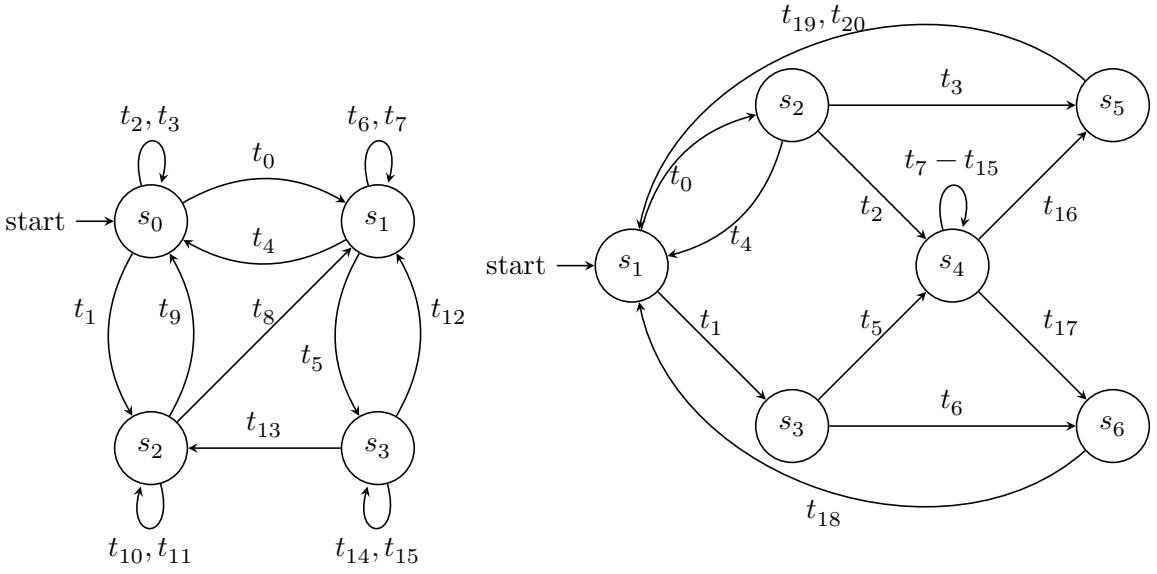
Figure 4.7: The second objective function, the Transition Coverage computation. If a path covers all the transitions, the result will be ≤ 1. `roundDec` is an auxiliary method that truncates the result to 4 decimals, and `transition` represents the list of transitions for the input EFSM.

Chapter 5

Evaluation

The algorithm presented in Chapter 5 was implemented in Haskell. To execute the multi-objective algorithm, NSGA-II was used. The implementation was adapted from the Genetic Algorithm Library [7] as described in Section 4.3. The adapted implementation of the NSGA-II included the two objective functions described in Section 4.4, the Feasibility Metric, and the Transition Coverage.

In the algorithm, a chromosome is considered a solution only if it includes feasible paths and covers all transitions. Any chromosomes with incomplete transitions or infeasible paths are disregarded. A successful end of the algorithm can be marked by finding at least of solution that consists of feasible paths and covers all transitions.



(a) Book EFSM Model

(b) Class 2 Transport Protocol EFSM Model

Figure 5.1

As an input, three EFSMs were considered: Book (Fig. 5.1a) [20], Class II Transport Protocol (Fig. 5.1b) [1] and Lift System (Fig 5.2) [1].

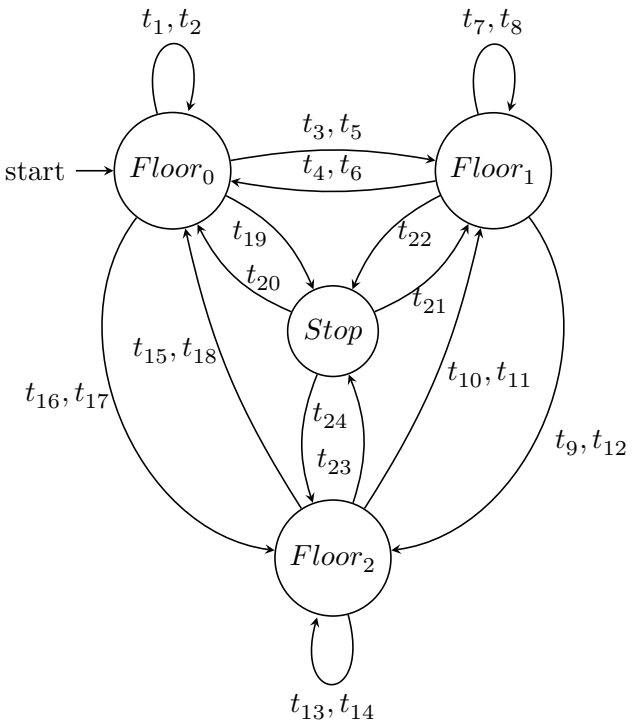


Figure 5.2: Lift System EFSM Model

Table 5.1a describes the first EFSM, which depicts a library book and includes four states and sixteen transitions [12].

In Table 5.2a, information about the second EFSM can be found. This EFSM is modeled after the AP module and serves as a simplified version of a class 2 transfer protocol. It includes six states and twenty-one transitions (a more detailed presentation can be found in [1]).

In Table 5.3a, you can find the description of a synthesized lift system for a three-floor building. This EFSM has four states and 24 transitions. Please refer to [1] for a more detailed presentation.

t	$s_s \to s_e$	Input	Guards	Operations
t_0	$s_0 \to s_1$	x	$x > 0$	$bId := x;$
t_1	$s_0 \to s_2$	x	$x > 0$	$rId := x;$
t_2	$s_0 \to s_0$	x	$x \leq 0$	Nil
t_3	$s_0 \to s_0$	x	$x \leq 0$	Nil
t_4	$s_1 \to s_0$	x	$x = bId$	$bId := 0;$
t_5	$s_1 \to s_3$	x	$x > 0 \wedge x \neq bId$	$rId := x;$
t_6	$s_1 \to s_1$	x	$x \neq bId$	Nil
t_7	$s_1 \to s_1$	x	$x \leq 0 \vee x = bId$	Nil
t_8	$s_2 \to s_1$	x	$x = rId$	$bId := x; rId := 0;$
t_9	$s_2 \to s_0$	x	$x = rId$	$rId := 0;$
t_{10}	$s_2 \to s_2$	x	$x \neq rId$	Nil
t_{11}	$s_2 \to s_2$	x	$x \neq rId$	Nil
t_{12}	$s_3 \to s_1$	x	$x = rId$	$rId := 0;$
t_{13}	$s_3 \to s_2$	x	$x = bId$	$bId := 0;$
t_{14}	$s_3 \to s_3$	x	$x \neq rId$	Nil
t_{15}	$s_3 \to s_3$	x	$x \neq bId$	Nil

(a) The main transition of the Book EFSM 5.1a [11]

t	$s_s \to s_e$	Input	Guards	Operations
t_0	$s_1 \to s_2$	**prop_out**	Nil	$opt := prop\_out; R\_credit := 0;$
t_1	$s_1 \to s_3$	**opt_ind, cr**	Nil	$opt := opt\_ind; S\_credit := cr; Rcredit := 0;$
t_2	$s_2 \to s_4$	**opt_ind, cr**	$opt\_ind < opt$	$TRsq := 0; TSsq := 0; opt := opt\_ind; S\_credit := cr;$
t_3	$s_2 \to s_5$	**opt_ind, cr**	$opt\_ind > opt$	Nil
t_5	$s_3 \to s_4$	**accpt_opt**	$accpt\_opt < opt$	$opt := accpt\_opt; TRsq := 0; TSsq := 0;$
t_7	$s_4 \to s_4$	**Udata**	$S\_credit > 0$	$S\_credit := S\_credit - 1; TSsq := (TSsq + 1) mod 128;$
t_8	$s_4 \to s_4$	**Send_sq**	$R\_credit \neq 0 \wedge Send\_sq = TRsq$	$TRsq := (TRsq + 1) mod 128; R\_credit := R\_credit - 1;$
t_9	$s_4 \to s_4$	**Send_sq**	$R\_credit = 0 \vee Send\_sq \neq TRsq$	Nil
t_{10}	$s_4 \to s_4$	**cr**	Nil	$R\_credit := R\_credit + cr;$
t_{11}	$s_4 \to s_4$	**XpSsq, cr**	$TSsq \geq XpSsq \wedge cr + XpSsq - TSsq \geq 0 \wedge cr + XpSsq - TSsq \leq 15$	$S\_credit := cr + XpSsq - TSsq;$
t_{12}	$s_4 \to s_4$	**XpSsq, cr**	$TSsq \geq XpSsq \wedge (cr + XpSsq - TSsq < 0 \vee cr + XpSsq - TSsq > 15)$	Nil
t_{13}	$s_4 \to s_4$	**XpSsq, cr**	$TSsq < XpSsq \wedge cr + XpSsq - TSsq - 128 \geq 0 \wedge cr + XpSsq - TSsq - 128 \leq 15$	$S\_credit := cr + XpSsq - TSsq - 128;$
t_{14}	$s_4 \to s_4$	**XpSsq, cr**	$TSsq < XpSsq \wedge (cr + XpSsq - TSsq - 128 < 0 \vee cr + XpSsq - TSsq - 128 > 15$	Nil
t_{15}	$s_4 \to s_4$	**-**	$S\_credit > 0$	Nil

(a) The main transition of the Class 2 Transport Protocol EFSM 5.1b (transitions t_4, t_6, $t_{16}-t_{20}$ are omitted because they have no guards nor assignment operations)

t	$s_s \rightarrow s_e$	Input	Guards	Operations
t_0	$\rightarrow s_0$	$reset$	–	$Floor := 0; DrSt := 0; w := 0;$
t_1	$s_0 \rightarrow s_0$	Pos	$DrSt = 0 \wedge Pos \geq 0 \wedge Pos \leq 15$	$DrSt := 1;$
t_2	$s_0 \rightarrow s_0$	Pos, Pw	$DrSt = 1 \wedge Pos \geq 0 \wedge Pos \leq 15$	$DrSt := 0;$
t_3	$s_0 \rightarrow s_1$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 1 \wedge w \geq 15 \wedge w \leq 250 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 1;$
t_4	$s_1 \rightarrow s_0$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 0 \wedge w \geq 15 \wedge w \leq 250 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 0;$
t_5	$s_0 \rightarrow s_1$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 1 \wedge w = 0 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 1;$
t_6	$s_1 \rightarrow s_0$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 0 \wedge w = 0 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 0;$
t_7	$s_1 \rightarrow s_1$	Pos	$DrSt = 0 \wedge Pos \geq 0 \wedge Pos \leq 15$	$DrSt := 1;$
t_8	$s_1 \rightarrow s_1$	Pos, Pw	$DrSt = 1 \wedge Pos \geq 0 \wedge Pos \leq 15$	$DrSt := 0; w = Pw;$
t_9	$s_1 \rightarrow s_2$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 2 \wedge w \geq 15 \wedge w \leq 250 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 2;$
t_{10}	$s_2 \rightarrow s_1$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 1 \wedge w \geq 15 \wedge w \leq 250 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 1;$
t_{11}	$s_2 \rightarrow s_1$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 1 \wedge w = 0 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 1;$
t_{12}	$s_1 \rightarrow s_2$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 2 \wedge w = 0 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 2;$
t_{13}	$s_2 \rightarrow s_2$	Pos	$DrSt = 0 \wedge Pos \geq 0 \wedge Pos \leq 15$	$DrSt := 1;$
t_{14}	$s_2 \rightarrow s_2$	Pos, Pw	$DrSt = 1 \wedge Pos \geq 0 \wedge Pos \leq 15$	$DrSt := 0; w = Pw;$
t_{15}	$s_2 \rightarrow s_0$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 0 \wedge w \geq 15 \wedge w \leq 250 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 0;$
t_{16}	$s_0 \rightarrow s_2$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 2 \wedge w \geq 15 \wedge w \leq 250 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 2;$
t_{17}	$s_0 \rightarrow s_2$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 2 \wedge w = 0 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 2;$
t_{18}	$s_2 \rightarrow s_0$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 0 \wedge w = 0 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 0;$
t_{19}	$s_0 \rightarrow s_s$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 100 \wedge w \geq 15 \wedge w \leq 250 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 100;$
t_{20}	$s_s \rightarrow s_0$	Pf	$DrSt = 0 \wedge Pf = 0$	$Floor := 0;$
t_{21}	$s_s \rightarrow s_1$	Pf	$DrSt = 0 \wedge Pf = 1$	$Floor := 1;$
t_{22}	$s_1 \rightarrow s_s$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 100 \wedge w \geq 15 \wedge w \leq 250 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 100;$
t_{23}	$s_2 \rightarrow s_s$	Pf, Ph, Ps	$DrSt = 0 \wedge Pf = 100 \wedge w \geq 15 \wedge w \leq 250 \wedge Ph \geq 10 \wedge Ph \leq 35 \wedge Ps \geq 0 \wedge Ps \leq 25$	$Floor := 100;$
t_{24}	$s_s \rightarrow s_2$	Pf	$DrSt = 0 \wedge Pf = 2$	$Floor := 2;$

(a) The Lift system EFSM 5.2 transitions specifications [1]

5.1 Results

Several experiments with different configurations of the algorithm were conducted. Each EFSM was subjected to the algorithm, with 100, 250, 500, and 1000 generations with variable population sizes. During the initial test, the algorithm used these configurations. The sole stopping criteria was based on the number of generations, resulting in the algorithm stopping solely when the generation limit was reached. Based on the findings, the algorithm consistently provided solutions; however, the execution time was notably lengthy. The time needed to execute the algorithm on each EFSM with these configurations can be found in Table 5.4a.

	100 gen.	250 gen.	500 gen.	1000 gen.
Book EFSM Model	26556	39683	83740	133908
Class II Transport Protocol EFSM Model	30716	50013	182970	256431
Lift System EFSM Model	29578	65470	129461	247699

(a) The time (measured in milliseconds) the algorithm took to execute for a given number of generations with variable population sizes on each EFSM, recorded **before** implementing the second stop criteria. This criterion halts the algorithm after discovering over ten solutions in the first front that remains unchanged.

In Table 5.5a, you can see the average results for EFSMs presented in Tables 5.1a and 5.2a after implementing the stop criteria described in Section 4.3. Now, not only the number of generations is taken into account but also the first ten unchanged solutions on the first front are considered. The program has been run 100 times, each consisting of 500 generations and a population size of 50. The results indicate a significant improvement in execution time, with a noticeable reduction in the average time required. This is because the algorithm stopped after discovering the solutions without reaching the total number of generations specified. Analyzing the algorithm's performance, the number of solutions found, the time it took to find them, and the subsequent generations after the algorithm had stopped were considered. To track the progress of the algorithm applied to the Class II Transport Protocol EFSM Model (shown in Figure 5.1b and discussed in Table 5.2a), a visualization has been created. This visualization displays the number of generations required, the time needed, and the solutions discovered for each of the 100 executions. The visualizations are showed in in Figures 5.3, 5.4, and 5.5.

	Generations	Time (ms)	Solutions
Book EFSM Model	74.27	55287.45	48.24
Class II Transport Protocol EFSM Model	60.67	133862.19	39.42

(a) The average results for the EFSMs presented in 5.1a and 5.2a after 100 iterations of the algorithm. The first column displays the average number of generations required to find a solution, the second column represents the average time taken, and the last column shows the average number of solutions found. In this experiment, 500 generations with a population size of 50 were executed.

In the tables labeled as tables 5.6a, 5.7a, and 5.8a, an example solution for each EFSM presented earlier can be found. The *Complexity Value* displayed in the tables represents the feasibility metric for each path, and the sum of those complexities is the *Total Complexity*. Additionally, the *Coverage Measure* in the tables serves as the second objective function.

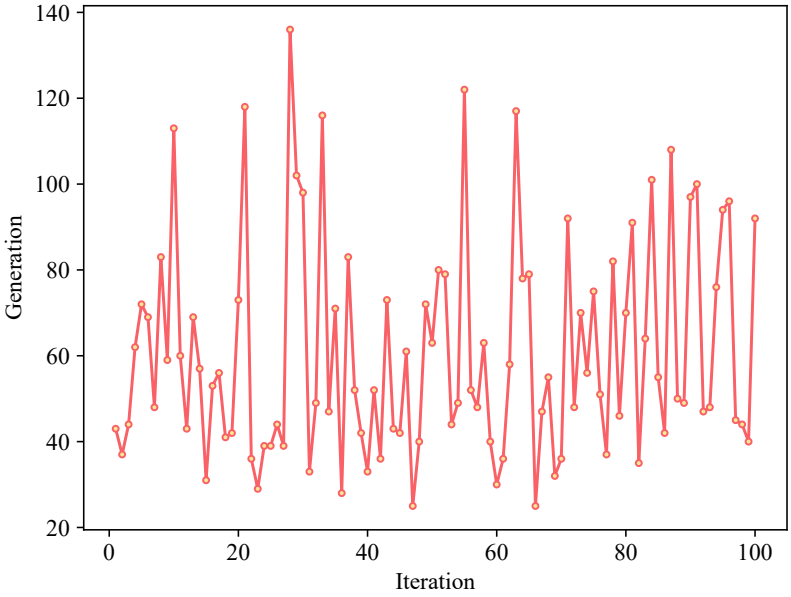


Figure 5.3: The number of generations needed to find solutions executing the algorithm for 500 generations and a population size of 50. The EFSM parsed as input was Class II Transport Protocol 5.1b. This process was repeated 100 times to ensure accuracy. The solution did not need 500 generations to be discovered, as the average is 60.67.

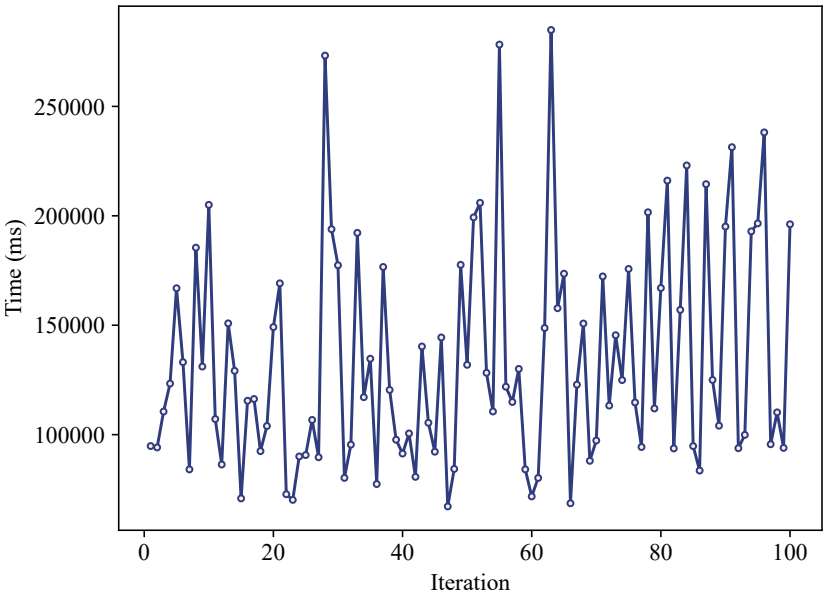


Figure 5.4: The duration needed in milliseconds to find solutions executing the algorithm for 500 generations and a population size of 50. The EFSM parsed as input was Class II Transport Protocol 5.1b, and the execution was repeated 100 times to ensure accuracy. The time required for the current algorithm shows significant improvement compared to the previous version, which did not stop after finding a solution and had variable population sizes. The average time for the current algorithm is 133862.19 milliseconds.

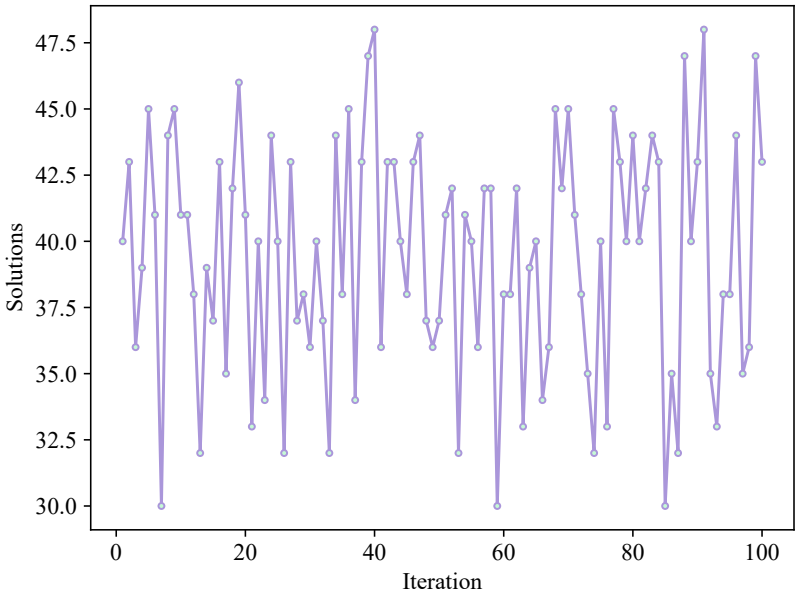


Figure 5.5: The number of solutions found after each execution of the algorithm for 500 generations and a population size of 50. The EFSM parsed as input was Class II Transport Protocol 5.1b, and the execution was repeated 100 times to ensure accuracy. On average, 39.42 solutions are found by the algorithm, and it consistently discovers a similar number of solutions.

Path	**Complexity**
$t_2 \rightarrow t_1 \rightarrow t_1 1 \rightarrow t_8 \rightarrow t_7 \rightarrow t_5 \rightarrow t_{14} \rightarrow t_{15} \rightarrow t_{13} \rightarrow t_9$	0.99952
$t_2 \rightarrow t_1 \rightarrow t_8 \rightarrow t_5$	0.99974
$t_3 \rightarrow t_1 \rightarrow t_{10}$	0.9999
$t_2 \rightarrow t_3 \rightarrow t_1 \rightarrow t_8 \rightarrow t_6 \rightarrow t_5$	0.99968
$t_3 \rightarrow t_1 \rightarrow t_8 \rightarrow t_4 \rightarrow t_0 \rightarrow t_6 \rightarrow t_4 \rightarrow t_0 \rightarrow t_5 \rightarrow t_{12}$	0.99932
$t_2 \rightarrow t_1 \rightarrow t_8$	0.99984
t_1	1.0
$t_2 \rightarrow t_1 \rightarrow t_{11} \rightarrow t_9$	0.99982
Total Complexity	7.99782
Coverage Measure	0.04015

(a) An example of a solution generated for Book EFSM 5.1a

Path	Complexity
t_1	1.0
$t_1 \rightarrow t_5 \rightarrow t_9 \rightarrow t_8$	0.99976
$t_0 \rightarrow t_2 \rightarrow t_9 \rightarrow t_8 \rightarrow t_8 \rightarrow t_7 \rightarrow t_{13}$	0.99592
$t_1 \rightarrow t_5 \rightarrow t_7 \rightarrow t_{12} \rightarrow t_{14} \rightarrow t_{11}$	0.99648
$t_0 \rightarrow t_2 \rightarrow t_{13} \rightarrow t_{10} \rightarrow t_{15} \rightarrow t_{16} \rightarrow t_{19}$	0.99974
$t_1 \rightarrow t_5 \rightarrow t_{11}$	0.9998
$t_1 \rightarrow t_6 \rightarrow t_{18} \rightarrow t_1 \rightarrow t_5 \rightarrow t_{15}$	0.99994
$t_0 \rightarrow t_4 \rightarrow t_1 \rightarrow t_5 \rightarrow t_{11} \rightarrow t_{14}$	0.99948
$t_0 \rightarrow t_2 \rightarrow t_9 \rightarrow t_17 \rightarrow t_{18} \rightarrow t_0 \rightarrow t_3$	0.99994
$t_0 \rightarrow t_3 \rightarrow t_{20} \rightarrow t_1$	1.0
Total Complexity	9.99106
Coverage Measure	0.0497

(a) An example of a solution generated for Class II Transport Protocol EFSM Model 5.1b

Path	Complexity
$t_{17} \rightarrow t_{10} \rightarrow t_8$	0.99964
$t_{17} \rightarrow t_{11} \rightarrow t_6$	0.99944
t_3	1.0
$t_{16} \rightarrow t_{11} \rightarrow t_{12}$	0.99944
$t_1 \rightarrow t_5 \rightarrow t_9 \rightarrow t_{14} \rightarrow t_{13} \rightarrow t_{14} \rightarrow t_{15} \rightarrow t_{19}$	0.99792
$t_{17} \rightarrow t_{18} \rightarrow t_{16} \rightarrow t_{10}$	0.99916
$t_{19} \rightarrow t_{20} \rightarrow t_5 \rightarrow t_4 \rightarrow t_{19} \rightarrow t_{20} \rightarrow t_3$	0.99864
$t_2 \rightarrow t_5 \rightarrow t_7$	0.99964
$t_{17} \rightarrow t_{13}$	0.99992
$t_2 \rightarrow t_{19} \rightarrow t_{20}$	0.9996
Total Complexity	9.9934
Coverage Measure	0.03631

(a) An example of a solution generated for Lift System EFSM Model 5.2

The algorithm exhibited positive outcomes for all the EFSMs that were given as input. While it is impossible to compare these results to the experiments presented in [19] due to the use of different algorithms, they still show promise. There is ample scope for further improvements, particularly in the genetic algorithm and the parsing of EFSMs, among other aspects, which boosts motivation to continue the research in the future.

5.2 Challenges

Initially, the thought of tackling this topic was quite daunting. Right from the start, it appeared to be beyond my capabilities. However, with a combination of hard work and determination and help from an experienced person, anything is achievable.

Dealing with this task is certainly not an everyday occurrence. It demands meticulous documentation, focused attention, and a desire to learn. Each configuration had to

be carefully considered to ensure compatibility with future developments. Despite this, some revisions were necessary with each step forward, requiring modifications to previous work. Ultimately, the project reached a promising conclusion, with room for further improvement.

Another challenge I faced was having to write my approach in a programming language that was not very familiar to me. However, I was immediately drawn to it and found it quite challenging. Each line of code that I completed successfully gave me a boost of confidence and kept me motivated. Unlike other object-oriented programming languages, Haskell does not offer as much information on how to get things done. The Genetic Algorithm Library [7] was a great resource for my paper, but I encountered some difficulties. The explanations were vague, and the examples didn't address my needs. Therefore, I had to modify certain aspects of the library to adapt it to my algorithm.

Chapter 6

Conclusions

Ensuring software applications' reliability, functionality, and overall quality is crucial, and software testing plays a significant role in achieving this. Two effective approaches in this domain are using EFSMs (Extended Finite State Machines) and NSGA-II (Non-dominated Sorting Genetic Algorithm II).

This paper suggested a modified NSGA-II algorithm (from the Genetic Algorithm Library found in reference [7]) to produce a set of paths that cover all transitions in a provided EFSM. While this approach is inspired by the work presented in [19], it differs in using Haskell. All data types and methods needed to be constructed from the ground up since the algorithm had not previously been written in Haskell, as far as my knowledge goes.

Two objective functions were evaluated: Path Feasibility and Transition Coverage. The former aims to identify feasible yet complex paths, while the latter aims to cover all transitions while minimizing their usage frequency and preventing transition cycles. Section 4.4 comprehensively explains the objective functions.

In this method, the One-point crossover genetic operator from the Genetic Algorithm Library [7] alongside a customized mutation operator were considered. The crossover operator swaps two genomes beyond a randomly chosen point with a probability of 0.5. On the other hand, the mutation operator has a probability of 0.75 and selects each gene (path) from a chromosome (set of paths), replacing it with a newly generated one. For a thorough description of the genetic operators, please refer to Section 4.3.

The paper does not necessarily enhance the approach presented in [19] but offers a fresh adaptation. This approach, implemented using Haskell, features new data types and methods that lay a strong foundation for future research. While the results cannot be directly compared to those of [19] due to different algorithms used, it is reasonable to expect better performance in terms of time from using Haskell. Based on the results, solutions were consistently generated for the provided input EFSM. Section 5.1 provides a comprehensive description of the experiments and outcomes.

As previously stated, this paper provides a basis for future progress and advancements.

One potential area for improvement is enhancing the EFSMs by developing an alternative parsing method that allows for more efficient coding. Additionally, it is possible to introduce new objective functions. It would be beneficial to make some improvements to the genetic algorithm. One area that could use attention is the customization of the crossover operator to cover a broader range of scenarios. Moreover, there are several ways to improve the mutation operator, such as removing redundant paths and path transitions, minimizing the number of paths required to cover the transitions, and substituting a single transition within a specific path. These are merely a few instances of improvements that could potentially be executed in future work.

For future work, this study will undergo further expansion through additional experiments and algorithm improvements. The resulting data will then be compared to the current paper's findings and alternative evolutionary methods.

Bibliography

[1] R. M. Hierons A. S. Kalaji and S. Swift. "An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models." In: 53.12 (2011), pp. 1297–1318. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2011.06.004. URL: https://doi.org/10.1016/j.infsof.2011.06.004.

[2] V. Alagar and K. Periyasamy. "Extended Finite State Machine." In: (Jan. 2011). DOI: 10.1007/978-0-85729-277-3_7.

[3] Nesa Asoudeh and Yvan Labiche. "Multi-objective Construction of an Entire Adequate Test Suite for an EFSM." In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 2014, pp. 288–299. DOI: 10.1109/ISSRE.2014.14.

[4] Wesley Klewerton Guez Assunção, Thelma Elita Colanzi, Silvia Regina Vergilio, and Aurora Pozo. "A multi-objective optimization approach for the integration and test order problem." In: *Information Sciences* 267 (2014), pp. 119–139. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2013.12.040. URL: https://www.sciencedirect.com/science/article/pii/S0020025513008967.

[5] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. "A fast and elitist multiobjective genetic algorithm: NSGA-II." In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017.

[6] Karnig Derderian, Robert Hierons, Mark Harman, and Qiang Guo. "Estimating the feasibility of transition paths in extended finite state machines." In: *Autom. Softw. Eng.* 17 (Mar. 2010), pp. 33–56. DOI: 10.1007/s10515-009-0057-9.

[7] *Genetic Algorithm Library*. URL: https://hackage.haskell.org/package/moo.

[8] Mark Harman and Phil McMinn. "A Theoretical & Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation." In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 73–83. ISBN: 9781595937346. DOI: 10.1145/1273463.1273475. URL: https://doi.org/10.1145/1273463.1273475.

[9] A.S. Kalaji. "Search-Based Software Engineering: A Search-Based Approach for Testing from Extended Finite State Machine (EFSM) Models." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 66–79.

[10] Abdul Salam Kalaji, Robert Mark Hierons, and Stephen Swift. "Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM) with the Counter Problem." In: *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 2010, pp. 232–235. DOI: 10.1109/ICSTW.2010.25.

[11] Raluca Lefticaru and Florentin Ipate. "An Improved Test Generation Approach from Extended Finite State Machines Using Genetic Algorithms." In: Oct. 2012, pp. 293–307. ISBN: 978-3-642-33825-0. DOI: 10.1007/978-3-642-33826-7_20.

[12] Raluca Lefticaru and Florentin Ipate. "Automatic State-Based Test Generation Using Genetic Algorithms." In: *Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007)*. 2007, pp. 188–195. DOI: 10.1109/SYNASC.2007.47.

[13] Miran Lipovaca. *Learn You a Haskell for Great Good! A Beginner's Guide*. 1st. USA: No Starch Press, 2011. ISBN: 1593272839.

[14] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998.

[15] Alberto Núñez, Mercedes Merayo, Robert Hierons, and Manuel Núñez. "Using genetic algorithms to generate test sequences for complex timed systems." In: *Journal of Soft Computing (in press) DOI: 10.1007/s00500-012-0894-5* (Feb. 2012). DOI: 10.1007/s00500-012-0894-5.

[16] Abdul Salam, Robert Hierons, and Stephen Swift. "Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM)." In: *Proceedings - 2nd International Conference on Software Testing, Verification, and Validation, ICST 2009* (Apr. 2009). DOI: 10.1109/ICST.2009.29.

[17] Kuo-chung Tai. "A program complexity metric based on data flow information in control graphs." In: *International Conference on Software Engineering*. 1984.

[18] N. Tracey, J. Clark, K. Mander, and J. McDermid. "An automated framework for structural test-data generation." In: *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No.98EX239)*. 1998, pp. 285–288. DOI: 10.1109/ASE.1998.732680.

[19] Ana Ţurlea. "Testing Extended Finite State Machines Using NSGA-III." In: *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1–7. ISBN: 9781450368506. DOI: 10.1145/3340433.3342820. URL: https://doi.org/10.1145/3340433.3342820.

[20] Ana Ţurlea, Florentin Ipate, and Raluca Lefticaru. "A Hybrid Test Generation Approach Based on Extended Finite State Machines." In: *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 2016, pp. 173–180. DOI: 10.1109/SYNASC.2016.037.

[21] Ana Ţurlea, Florentin Ipate, and Raluca Lefticaru. "Generating Complex Paths for Testing from an EFSM." In: *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 2018, pp. 242–249. DOI: 10.1109/QRS-C.2018.00052.

[22] Mark Weiser. "Program Slicing." In: *IEEE Transactions on Software Engineering* SE-10.4 (1984), pp. 352–357. DOI: 10.1109/TSE.1984.5010248.

[23] Tianyong Wu, Jun Yan, and Jian Zhang. "A Path-oriented Approach to Generating Executable Test Sequences for Extended Finite State Machines." In: *2012 Sixth International Symposium on Theoretical Aspects of Software Engineering*. 2012, pp. 267–270. DOI: 10.1109/TASE.2012.38.

[24] Thaise Yano, Eliane Martins, and Fabiano Sousa. "A multi-objective evolutionary algorithm to obtain test cases with variable lengths." In: July 2011, pp. 1875–1882. DOI: 10.1145/2001576.2001828.

[25] Thaise Yano, Eliane Martins, and Fabiano L. de Sousa. "Generating Feasible Test Paths from an Executable Model Using a Multi-objective Approach." In: *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 2010, pp. 236–239. DOI: 10.1109/ICSTW.2010.52.

[26] Thaise Yano, Eliane Martins, and Fabiano L. de Sousa. "MOST: A Multi-objective Search-Based Testing from EFSM." In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 2011, pp. 164–173. DOI: 10.1109/ICSTW.2011.37.

[27] Justyna Zander, Ina Schieferdecker, and Pieter Mosterman. *Model-Based Testing for Embedded Systems*. Sept. 2011. ISBN: 9781439818459.