SELECTION OF TESTS FOR FINITE STATE MACHINES

MIGUEL BENITO PAREJO

MÁSTER EN MÉTODOS FORMALES EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Máster en Métodos Formales en Ingeniería Informática

Septiembre 2019

Directores:

Mercedes García Merayo
Manuel Núñez García

Convocatoria:

Septiembre 2019

Calificación:

Sobresaliente - 9

Autorización de difusión

Miguel Benito Parejo

Septiembre 2019

El abajo firmante, matriculado en el Máster en Métodos Formales en Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Selection of tests for Finite State Machines", realizado durante el curso académico 2018-2019 bajo la dirección de Mercedes García Merayo y Manuel Núñez García en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen en castellano

Habitualmente no se pueden aplicar todos las posibles pruebas (tests) a una implementación para comprobar su corrección. Por ello, es necesario seleccionar subconjuntos de pruebas relativamente pequeños que permitan detectar el mayor número de errores posible. En este trabajo proponemos diferentes enfoques para seleccionar dichos conjuntos de pruebas. Para determinar la calidad de un conjunto de pruebas, este se aplicará a un grupo de *mutantes*. Un mutante corresponde a una variación de la especificación del sistema bajo prueba que induce un error en la misma. El objetivo de nuestro trabajo es que los algoritmos propuestos generen conjuntos de pruebas que *maten* el mayor número de mutantes posible. Comparamos los enfoques propuestos entre los que se consideran todos los posibles subconjuntos dada una cota en las entradas (inputs), un algoritmo devorador inteligente y distintos algoritmos genéticos. Finalmente, discutimos los resultados obtenidos en los experimentos realizados para determinar su efectividad. Todas las propuestas han sido implementadas y la herramienta desarrollada es totalmente libre y accesible.

Palabras clave

Algoritmos genéticos; testing de máquinas de estados finitos; Mutation testing; Métodos formales.

Abstract

It is unaffordable to apply all the possible tests to an implementation in order to assess its correctness. Therefore, it is necessary to select relatively small subsets of tests that can detect as many faults as possible. In this paper we propose different approaches to select the best subset of tests from the original one: all the possible subsets up to a given number of inputs, an intelligent greedy algorithm and several genetic algorithms. In order to decide how good a test suite is, we apply it to a set of *mutants* that correspond to small variations of the specification of the system to be developed. The goal is that our algorithms generate test suites that *kill* as many mutants as possible. We compare the proposed approaches and discuss the obtained results. The whole framework has been fully implemented and the tool is freely available.

Keywords

Genetic algorithms; Testing Finite State Machines; Mutation testing; Formal methods.

Contents

Index i

Acknowledgements ii

Dedication iii

1 Introduction 1

2 Preliminaries 5
2.1 Mutation Testing for Finite State Machines 5
2.2 Genetic Algorithms 8

3 Related work 11

4 Our proposal for test cases selection 15
4.1 Global search 15
4.2 Greedy algorithm 16
4.3 Genetic algorithm 18

5 The tool 31
5.1 GUI and MVC 31
5.2 Other patterns 37
5.3 Class diagram 38

6 Experiments 41
6.1 Description of the experiments 41
6.2 Evaluation 44

7 Conclusions and future work 51

Bibliography 55

Acknowledgements

I would like to thank my supervisors and Inmaculada Medina-Bulo for their effort, work, help and advice before and during this thesis. I would like to thank these close friends who helped through the toughest moments. Last, but definitely not least, I would like to thank my family for all their unconditional help and support.

Dedication

To my parents, who made me be what I am.

Chapter 1

Introduction

Testing is the main technique to validate the correctness of software systems [1]. It is quite common to find ourselves with a group of properties that should be satisfied by the system under development and we want to reassure that it does. In testing, these properties are encoded as tests and we have to check that the system, usually called System Under Test (SUT), successfully passes them. In practice, this approach is unfeasible because the number of tests may be astronomical. In particular, one property may give rise to many tests. In addition, we may have a bound on the number of tests that we can apply (e.g. due to budget or temporal constraints). Therefore, it is important to *wisely* choose among these tests a subset that is able to detect most faults. Clearly, the method to select these tests should rely on a measure of how good a test is. In this line, *mutation testing* [5, 6, 13] is a useful tool. The idea behind mutation testing is that if a test suite distinguishes the SUT from other faulty versions of the system then it is probably good at discovering faults. The technique introduces small changes in the SUT by applying *mutation operators* to generate a set of *mutants*. Intuitively, good test suites are the ones *killing* most of the mutants.

In this thesis we analyze different strategies to select *good* sets of tests. We assume

that we have a formal representation of the SUT, that is, its specification, and that we are provided with a set of mutants and a set of tests, usually huge, that we might apply to the SUT. The mutants, maybe constructed from the specification, present the representative faults during the development of the systems. This is usually called a *fault model*. Our goal is to select a subset of tests, up to a certain *complexity*, that kills as many mutants as possible. We will measure the complexity of a test suite in terms of the number of inputs included in it. If T is the whole set of tests and n is the bound on the number of inputs, then the obvious solution is to compute all the subsets of T with up to n inputs, apply them to the set of mutants and choose the subset killing more mutants. This result will always be the best subset, since all possibilities are explored. Unfortunately, in this case we have an exponential explosion that disallows us to use this approach for a general problem. A second option, based on previous work [2], considers a greedy algorithm where we select the best tests individually, according to the number of mutants that they kill, until we reach the specified limit of inputs. This technique will generally provide good results, both in cost and in faults detected, but it may not always yield the best result. For instance, there could be a combination of individually worse elements that were able to cover more faults. In order to solve this problem, and this is the main contribution of this work, we have developed a genetic algorithm to find better solutions than the greedy algorithm. The algorithm is versatile and allows users to apply different variants. We have developed a tool that fully implements all the algorithms presented in this work. Finally, we have performed several experiments to compare the different methods. We have analyzed the performance both in time and in goodness of the different variants of the genetic algorithm, and we have compared them with the greedy algorithm and the full search.

The rest of this document is structured as follows. In Chapter 2 we introduce the main concepts used in the thesis and set the background knowledge for the next chapters.

In Chapter 3 we present the state-of-the-art in the field. In Chapter 4 we introduce the different methods that we propose to select the best subsets of tests. In Chapter 5 we describe the tool that we have developed. In Chapter 6 we report on the experiments that we performed. Finally, in Chapter 7 we present our conclusions and some lines for future work.

Chapter 2

Preliminaries

In this chapter we introduce the main concepts used in this work related to mutation testing of Finite State Machines and genetic algorithms.

2.1 Mutation Testing for Finite State Machines

Mutation testing is a software testing technique that consists in inducing faults into a program by generating mutants, that is, faulty versions of the original program. The changes performed to generate the mutants are defined by *mutation operators*. The mutants and the original program are executed against test suites of interest with the goal of determining their efficiency to distinguish the mutants from the original program. Given a test suite, if a test case is able to distinguish a mutant from the original program, then we say that the mutant is *killed*. Similarly, when the mutant is not detected by any test case in the test suite, the mutant is *alive*. If the mutation does not change the behavior of the original program, the mutant is called *equivalent* and, therefore, there is no test case able to kill this mutant. The efficiency of the test suite for detecting the errors injected in the original program is

measured by the *mutation score*. The mutation score is the ratio of killed mutants over the non-equivalent ones. Figure 2.1 graphically represents the behaviour of mutation testing.

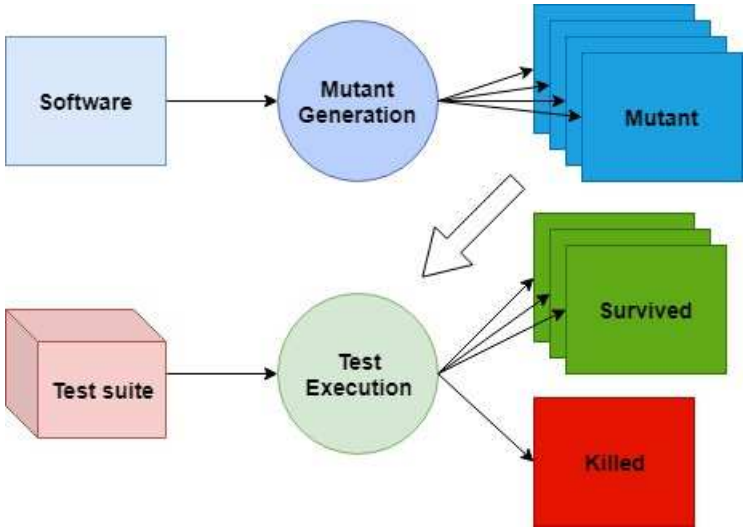


Figure 2.1: Mutation Testing

Finite State Machines are used in this project to represent specifications and mutants. Although mutation testing is often used to change code in programs, in this work we apply the mutation technique to Finite State Machines that represent the specifications of systems. The generated mutants will be modified instances of them that will be used to determine the efficiency of the test cases to differentiate them from the original specification.

Definition 1. *A* Finite State Machine*, in the following* FSM*, is a tuple* $M = (S, I, O, Tr, s_{in})$ *where* S *is a finite set of* states*,* I *is the set of* input actions*,* O *is the set of* output actions*,* Tr *is the set of* transitions *and* $s_{in} \in S$ *is the* initial state*. A transition belonging to* Tr *is a tuple* (s, s', i, o) *where* $s, s' \in S$ *are the initial and final states of the transition,* $i \in I$ *is the input action and* $o \in O$ *is the output action. We say that* M *is* input-enabled *if for each* $s \in S$ *and input* $i \in I$*, there exist* $s' \in S$ *and* $o \in O$ *such that* $(s, s', i, o) \in Tr$*. We say that* M *is* deterministic *if for each* $s \in S$ *and* $i \in I$*, there exists at most one transition* (s, s', i, o)

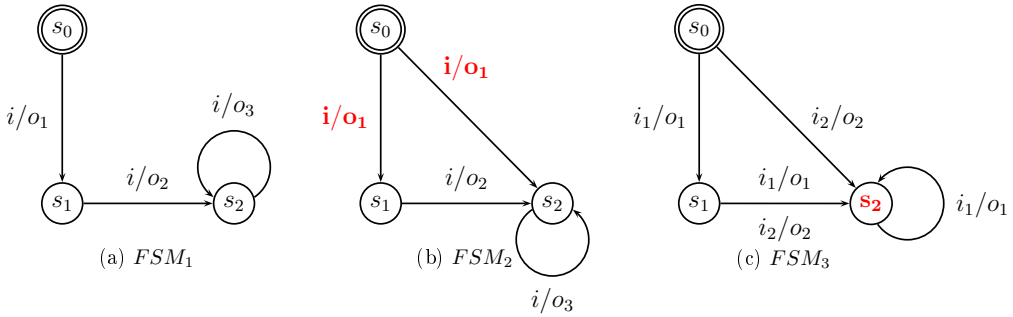


Figure 2.2: Three FSMs with different properties

belonging to Tr.

In this work we will restrict ourselves to input-enabled deterministic FSMs, that is, from each state of the machine, it is possible to perform all the inputs and there will be only one possible evolution. This restriction mimics testing of programs: programs are (usually) deterministic and should react to any possible input.

Example 1. *Figure 2.2a presents an input-enabled deterministic FSM. There exists only one transition outgoing from each state and labelled by the only input action* i. *Figure 2.2b shows a non-deterministic behavior of the machine in state* s_0. *There exist two outgoing transitions labelled by the same input action. Finally, Figure 2.2c depicts a non input-enabled FSM. State* s_2 *has no outgoing transition associated with input* i_2. *In this case, we say that such state is not input-enabled and, as a consequence, the FSM is not input-enabled.*

Next, we introduce the notions of mutant and test that are used in this work. Note that mutants are still deterministic and input-enabled.

Definition 2. *Let* $M = (S, I, O, Tr, s_{in})$ *be an FSM. We say that a FSM* $M' = (S, I, O, Tr', s_{in})$ *is a* mutant *of* M *if* Tr' *differs from* Tr *in only one transition. This* mutation *can be produced either by changing the output of a transition, that is, replacing* $(s, s', i, o) \in Tr$ *by* $(s, s', i, o') \in Tr'$ *with* $o \neq o'$, *or by changing the target state of a transition, that is, replacing* $(s, s', i, o) \in Tr$ *by* $(s, s'', i, o) \in Tr'$, *with* $s' \neq s''$.

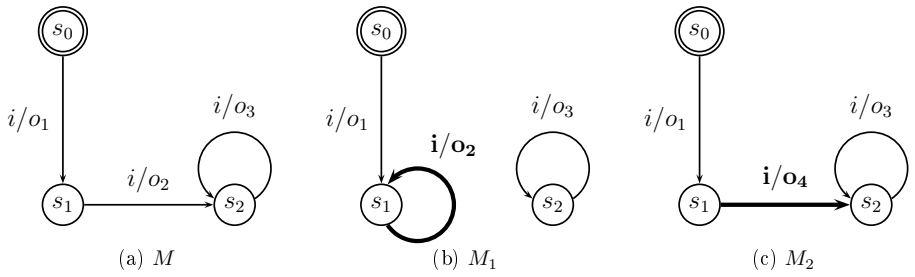


Figure 2.3: An FSM and two of its mutants

Example 2. *Let us consider the FSM given in Figure 2.3a, being s_0 the initial state. Two possible mutants are shown in Figures 2.3b and 2.3c: the first one represents the change of the final state of a transition while the second one represents the modification of an output action.*

Definition 3. *Let $M = (S, I, O, Tr, s_{in})$ be an FSM. A* test *for M is a pair $\sigma = (\sigma_{in}, \sigma_{out})$ where $|\sigma_{in}| = |\sigma_{out}|$, $\sigma_{in} \in I^*$ is a sequence of inputs and $\sigma_{out} \in O^*$ is the sequence of outputs that M produces when applying σ_{in}.*

Let $t = (\sigma_{in}, \sigma_{out})$ be a test for M. We say that a system M' passes t if the application of σ_{in} produces σ_{out}; otherwise, we say that the system M' fails t.

Example 3. *Let us consider again M, M_1 and M_2 given in Figure 2.3. We have that $t_1 = (i, o_1)$, $t_2 = (ii, o_1o_2)$ and $t_3 = (iii, o_1o_2o_3)$ are tests for M. M_1 passes t_1 and t_2 and fails t_3 while M_2 passes t_1 and fails t_2 and t_3.*

2.2 Genetic Algorithms

A *Genetic Algorithm* (GA) [11, 23] is a heuristic optimization technique, which is inspired in a metaphor of the processes of evolution in nature. GAs and other meta-heuristic algorithms have been used in Software Testing [7, 14, 18, 20]. Generally, a GA works with a group of

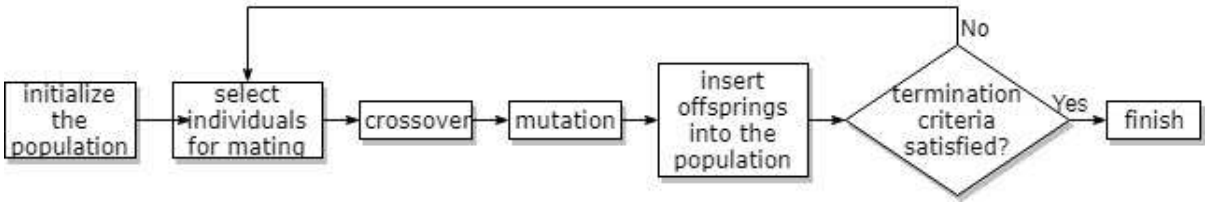


Figure 2.4: Genetic Algorithm flowchart

individuals or *chromosomes*, each representing a potential solution to the problem in hand. In our case, chromosomes will be subsets of tests. The process can usually be divided into five phases. An *initial population* is usually selected at random. Then, a parent *selection* process is used to pick some individuals from the initial population. A new offspring is produced using *crossover*, keeping some of the characteristics of their parents, and *mutation*, which introduces some new genetic material. Crossover exchanges information between two or more individuals. The mutation process randomly modifies individuals of the offspring. The quality of each individual is measured by a *fitness function*, defined for the particular search problem. The population is iteratively combined and mutated to generate successive populations, known as *generations*. When the specified termination criterion is satisfied, the algorithm terminates. The idea behind GAs is that the combination of good elements will generate good elements for a future generation. In this sense, being able to select the best elements of the current generation and properly combining them will generally improve the partial solution obtained at the next generation. Nevertheless, we might want to keep information about individuals of previous generations whenever such individuals have a high quality. In addition, it is important to refresh small parts of the population by introducing slight mutations, in order not to get stuck in a local minimum, after several evolutions. The flowchart for a simple GA is presented in Figure 2.4. In addition, Algorithm 1 shows the pseudo-code of the GA.

```
Generate an initial population;
Evaluate the population;
while termination criterion not fulfilled do
    Select the individuals;
    Perform the crossover of the selected individuals;
    Perform mutations;
    Replace the old generation by the new one;
    Update the fitness value;
end
```

Algorithm 1: Pseudo-code of the genetic algorithm

Chapter 3

Related work

This chapter is devoted to present the state of the art in the field. We focus on those works most related to ours.

Wegener et al. [25] considered GAs to obtain the longest and shortest tests, in terms of time, in a program of 1511 lines of code. They were able to improve the results of random tests. Their tests were likely to find real-time failures when actions were performed either faster than the short test, or slower than the long one.

This approach considers the problem of generating a good test suite. Girgis [9] produces a set of tests from the specification and a list of def-use paths to be covered and uses different parameters for the GA. In a similar way, Jones et al. [19] used a library of GAs to obtain tests able to identify faults in the areas where more mistakes could have been produced.

Following the idea of generating tests using GAs, Berndt and Watkins [4] applied them to long sequence testing. The goal is to observe failures concerning extended periods of operation that are not adequately captured with traditional measures of code coverage. These errors are related to component coordination, system resource consumption or cor-

ruption. The authors combined search and random-like behaviors to generate many variants of specific test cases.

Testing is a vast area that considers more complex structures than procedural programming, such as object-oriented software. This is the case of the work of Gupta and Rohil [12], where the authors used GAs to generate test cases for classes. The approach considered a tree representation of statements in test cases in order to facilitate the automatic generation and the evolution of the GA. In the same way, Wappler and Lammermann [24] focus on generating white-box test cases for object-oriented software where the GAs complete some *structures* obtained from the higher coverage sequences extracted from the code.

Concerning work more similar to ours, Shi et al. [22] compared test suite reduction with other selection of tests techniques. In this case, they experimentally compared test suite reduction with regression test selection and proposed another criterion. In the experiments, they observed a bigger reduction of the number of tests with regression test selection. Our proposal focuses on testing FSMs instead of code. Shi et al. also realized that a loss of fault-detection was linked to the test suite reduction, since part of the tests are no longer considered. In our case, we also observed a loss when we compared the methods we propose to solve our problem with the optimal solution for the same environment.

Gligoric et al. [10] considered regression test selection to fasten regression testing on software. They developed a technique called Ektazi and implemented it for Java and JUnit. Their technique keeps track of the dynamic dependencies of tests on different files of the SUT. Then, only the involved tests have to be executed instead of the whole test suite, reducing the cost of testing.

Domínguez-Jiménez et al. [8] also presented an evolutionary technique to reduce the cost of testing. Their framework involved mutation testing, and a genetic algorithm was the proposed methodology to select *strong mutants*. Similarly, we use a GA to obtain a

good subset of tests. They focused on reducing the number of mutants while having a representative set of faults. These authors were able to significantly reduce the number of mutants without a loss of fault-detection, producing a faster execution of the tests. Our results, despite having a different target, also exploit the power of GAs, getting a fast and efficient solution.

We conclude that there has been research concerning GAs to generate tests as well as a wide study regarding mutation testing. Also, selection testing techniques have been developed to reduce the cost of testing. However, there has barely been successful work on the combination of GAs *and* mutation testing, where we obtain a subset of tests from a test suite too costly to be fully performed. This is where our work is targeted and the objective of our contribution.

Chapter 4

Our proposal for test cases selection

In this chapter we present the proposed approaches to solve the problem of finding good sets of tests. All of them are based on how good a test case is, which is given by the number of mutants that it kills, and its length in terms of the number of inputs that it contains.

4.1 Global search

The global search approach looks through all the possible subsets of the initial set of tests having less inputs than the given bound. This means that a full search has to be performed in order to obtain all the subsets of tests so that every possible solution is considered, including trivially bad choices.

The fact that the worst potential solutions are considered is due to the lack of *intelligence* of this algorithm. This approach always provides the best solution because it explores all the possible subsets. Therefore, it is useful because it helps to compare this solution with the ones produced by other algorithms computing *good enough* solutions. The negative aspect is that for non-trivial systems, it is impossible to apply it because it suffers of a exponential

explosion. In fact, we were only able to compute it for the smallest systems that we have in our experiments.

4.2 Greedy algorithm

Our greedy algorithm is based on a matrix which includes information about tests and mutants.

Definition 4. *Let $M = (S, I, O, Tr, s_{in})$ be an FSM, $T = \{t_i\}_{i=1}^{n}$ be a set of tests for M and $\mathcal{M} = \{M_j\}_{j=1}^{m}$ be a set of mutants of M. We define the* results table *for T and $\mathcal{M}$ as a matrix $(a_{ij})_{i=1,j=1}^{n,m}$, where a_{ij} is the length of the shortest prefix of the test t_i that kills the mutant M_j. In the case that such mutant passes the test, this distance will be equal to infinity.*

Here, the rows of the $n \times m$ matrix represent the tests, while the columns represent the mutants. For instance, the value a_{24} indicates the length of the shortest prefix of the test t_2 that kills the mutant M_4.

Essentially, the algorithm sorts the rows of the matrix by decreasing order of the number of mutants killed by each of the tests. In the case that different rows kill the same number of mutants, they will be ordered increasingly by the number of inputs required to kill the mutants. Then, we include the first test to the test suite that we are constructing. Afterwards, we remove the row of the matrix corresponding to this test case as well as all the columns corresponding to the mutants that it kills. After reducing the matrix, we iterate the process until either all the mutants are killed or the specified bound on the number of inputs is reached.

Example 4. *Let us consider Figure 4.1a. Once the matrix is ordered, see Figure 4.1b, the*

$$\begin{pmatrix} 5 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 9 & \infty & 8 & \infty & \infty \\ 13 & \infty & \infty & 6 & 1 & \infty \\ 4 & 7 & 5 & \infty & 12 & \infty \\ \infty & \infty & \infty & \infty & \infty & 7 \end{pmatrix}$$

(a) Original matrix

$$\begin{pmatrix} 4 & 7 & 5 & \infty & 12 & \infty \\ 13 & \infty & \infty & 6 & 1 & \infty \\ \infty & 9 & \infty & 8 & \infty & \infty \\ 5 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 7 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

(b) Original ordered matrix

$$\begin{pmatrix} 6 & \infty \\ 8 & \infty \\ \infty & \infty \\ \infty & 7 \\ \infty & \infty \end{pmatrix}$$

(c) Reduced matrix

$$\begin{pmatrix} 6 & \infty \\ \infty & 7 \\ 8 & \infty \\ \infty & \infty \\ \infty & \infty \end{pmatrix}$$

(d) Reduced ordered matrix

Figure 4.1: Matrix simplification

first row corresponds to the best test, which is selected to be included in the test suite. This test kills the mutants 1, 2, 3 and 5. Therefore, in the reduced matrix given in Figure 4.1c, one test and four mutants have been removed, that is, the selected test and the mutants that this test kills. The resulting matrix needs to be reordered. We must take into account that rows with the same information do not represent the same test. Thus, none of them can be removed. As we can see in the matrix obtained in Figure 4.1d, the next tests that will be selected for being included in our test suite are the ones corresponding to the first and the second rows of the matrix. The total length of the generated set of tests is 25 *(the length needed to kill all the mutants). This length results from the sum of the* 12 *inputs of the first test included in the test suite that must be applied to kill four mutants plus the* 6 *and* 7 *inputs corresponding to the second and the third test cases, respectively, required to kill the other two mutants.*

This shows that two tests killing the same mutants might not be equally good, as one detecting them sooner will require a smaller number of inputs and therefore it will save

resources.

A good property of this algorithm is that it works in low polynomial order over a space, the matrix, that reduces its size after each iteration. This is the less costly algorithm, in terms of execution time, out of the ones we propose in this work. Our greedy method shows great results and will also help to bound the number of generations and the global cost of our next algorithm.

4.3 Genetic algorithm

GAs excel when we seek for a good approximation of the solution of problems whose optimal solution needs an exponential approach to compute all the potential candidates. This is the case of our problem and its solution, as discussed in Chapter 4.1. Therefore, a GA is a sensible approach to compete with our greedy algorithm, in particular, because our greedy algorithm computes relatively good solutions in a short time.

In Chapter 2.2 we presented the general structure of GAs. We now follow its layout to delve into each section.

Our population is a list of individuals, that might be sorted by fitness depending on the selected methods. In our population, an individual only has one chromosome that represents a subset of the original test suite. Each individual is implemented using an array, where the order of the elements does not matter. The initial population will evolve to generate better subsets. The evolution will continue for a number of iterations specified by the user. Next, we introduce the different elements that define the GA that we propose for the selection of tests cases.

Our approach uses different parameters to configure the main elements of the GA. Among these parameters we have the maximum number of inputs we expect in the solution, the

selection method of the population, the type and rate of the crossover method and the rate of the mutation technique to be applied. However, all the variants of our GA use the same fitness function that we introduce next.

The fitness function

The heuristics that we use to define our fitness function enhances the individuals that improve the efficiency of the generated test suite. Basically, it takes into account how many mutants are killed by the test cases. Specifically, the fitness is calculated by adding the minimum number of inputs required to kill each mutant, considering the subset of tests included in the chromosome at a specific moment. This value does punish the fact that some mutants are not killed by any test in the chromosome. If this is the case, a penalty will be added to the final value on the basis of the number of alive mutants. Therefore, the more mutants a subset kills, the lower the score will be. This value will also be reduced when the number of inputs required to kill a higher number of mutants decreases. This leads us to a minimization problem (a lower value of fitness denotes a better population).

Definition 5. *Let M be a FSM, $T = \{t_i\}_{i=1}^{n}$ be a set of tests for M, $S = \{t'_i\}_{i=1}^{n'}$ be a subset of T, $\mathcal{M} = \{M_i\}_{i=1}^{m}$ be a set of mutants of M, $(a_{ij})_{i=1,j=1}^{n,m}$ be the results table for T and $\mathcal{M}$, and $(b_{ij})_{i=1,j=1}^{n',m}$ be the results table for S and $\mathcal{M}$. We define the fitness function of S for $\mathcal{M}$ as:*

$$f(S, \mathcal{M}) = \sum_{k=1}^{m} \min\left(\alpha(M_k, S), P\right)$$

where $\alpha(M_k, S) = min(b_{ik} : 1 \leq i \leq n')$ and $P = 5 * \max[a_{ij} | 1 \leq i \leq n, 1 \leq j \leq m]$ is the penalty value.

The value 5 used in the penalty value was selected experimentally to provide small differences in solutions with many penalties, but sufficient to be a relevant penalty.

$$\begin{pmatrix} 3 & \infty & \infty & 6 & \infty & \infty \\ 5 & \infty & 7 & 15 & 16 & \infty \\ \infty & 4 & \infty & \infty & 7 & \infty \\ \infty & \infty & 5 & \infty & \infty & 8 \\ \infty & 9 & \infty & \infty & \infty & 21 \end{pmatrix}$$

Figure 4.2: Example matrix

Example 5. *Let us consider Figure 4.2. If we take an individual with tests t_2 and t_5, its fitness would be $5 + 9 + 7 + 15 + 16 + 21 = 73$, which is the result that the greedy algorithm would yield. Also, an individual containing tests t_1 and t_4 would have a fitness of $3+105+5+6+105+8 = 232$ where we find two penalties ($105 = 5*21$). The best result would be obtained by grouping tests t_1, t_3 and t_4 yielding a fitness value of $3+4+5+6+7+8 = 33$.*

In addition, in this work we will use the *score* of an individual in a population to determine the probability of such individual to be chosen in the selection phase. Intuitively, the score is computed as a ratio between the fitness value of the individual and the sum of the fitness values of all the individuals in the population. A constant is required to make the lowest fitness have the highest score.

Definition 6. *Let M be a FSM, T be a set of tests for M, $\mathcal{M}$ be a set of mutants of M and $\mathcal{P}$ be a set of subsets of T. For all $S \in \mathcal{P}$ we define the score of S for $\mathcal{P}$ and $\mathcal{M}$ as:*

$$s(S, \mathcal{P}, \mathcal{M}) = \frac{K(\mathcal{P}, \mathcal{M}) - f(S, \mathcal{M})}{\sum_{S' \in \mathcal{P}} [K(\mathcal{P}, \mathcal{M}) - f(S', \mathcal{M})]}$$

where $K(\mathcal{P}, \mathcal{M}) = 1.05 * \max_{S'' \in \mathcal{P}} f(S'', \mathcal{M})$.

It is important to note that the fitness value is always positive. Therefore, the score is well defined, all its values will be between 0 and 1, and the sum of all the scores in a

population is trivially 1. Also, the value 1.05 used to compute the constant $K(\mathcal{P}, \mathcal{M})$ is important so that the biggest element has a positive score instead of 0.

Example 6. *Let us consider a population consisting of the three individuals from Example 5. The first individual with fitness* 73 *has a score of* $\frac{243.6-73}{392.8} = 0.434$. *The second individual, which is the worst of the three, has a score of* $\frac{243.6-232}{392.8} = 0.030$. *Finally, the best individual has a score of* $\frac{243.6-33}{392.8} = 0.536$.

Initialization method

We have decided to apply an *incremental initialization* for our algorithm. This approach provides a variety of chromosomes, each of them with a different number of tests and inputs, which means more diversity. Such initialization follows the idea of minimizing the number of inputs to apply. As some chromosomes may have too few inputs and others too many, the execution of the algorithm will mix them at some point and improve the final result.

Selection methods

Taking into account that some individuals might be better than others, the transition from one generation to the next one has to ensure that the foremost representatives are selected. The idea is to reward the best ones with more appearances in the selection and the worst ones with even no appearances at all. We allow the user to choose the method to be applied by the algorithm among alternative standard selection models [11, 21]:

- The *tournament* principle is based on the competition of several individuals for a place in the new population. The user must provide the number of participants on the tournament n and the probability of winning for the favorite player p. Then, n chromosomes are randomly selected from the population, where the best individual

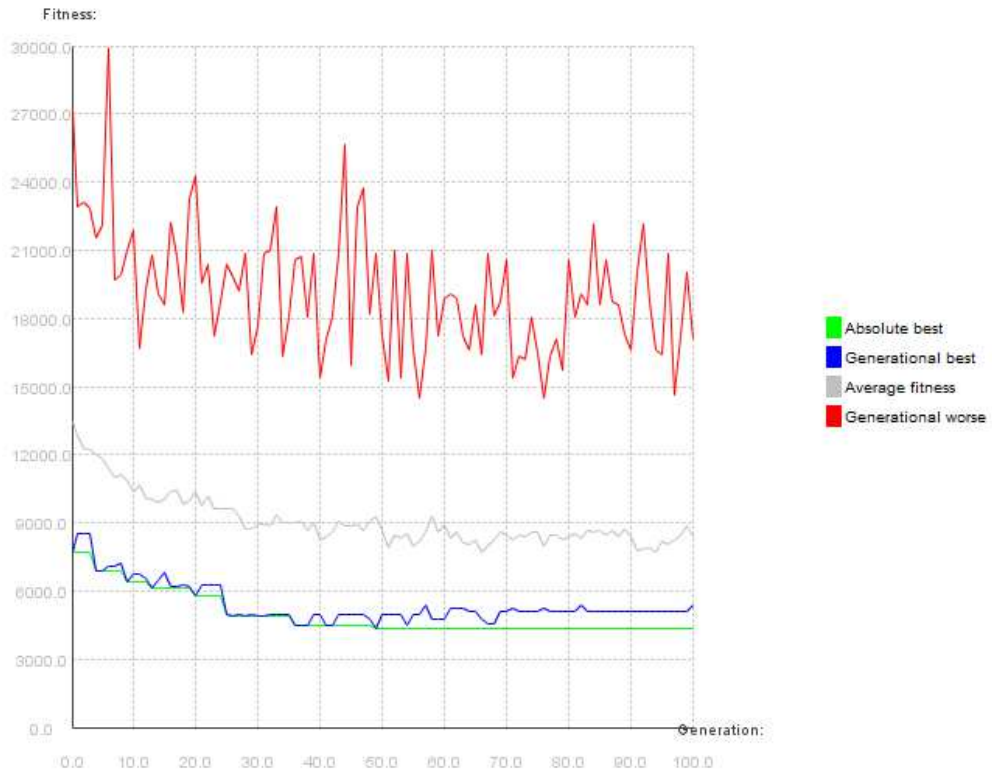


Figure 4.3: Tournament

of this group is chosen for reproduction with probability p. This process is repeated until the new population reaches the specified size. Usually, the chromosome with the highest score will have more chances to be selected, but some diversity is allowed by enabling the underdog to be chosen despite its fitness.

- The *roulette wheel* technique is based on the accumulated probability of choosing an element in a position[1] or any of the previous ones. In our case, given a random number between 0 and 1, the first individual in the population that saturates it by adding its score to a counter will be chosen. This method allows to give more variability to the population, since the selection does not depend on the size of the population.

[1]The population is implemented as an array of individuals. As such, even if the individuals are unsorted, we can use the order of the array.

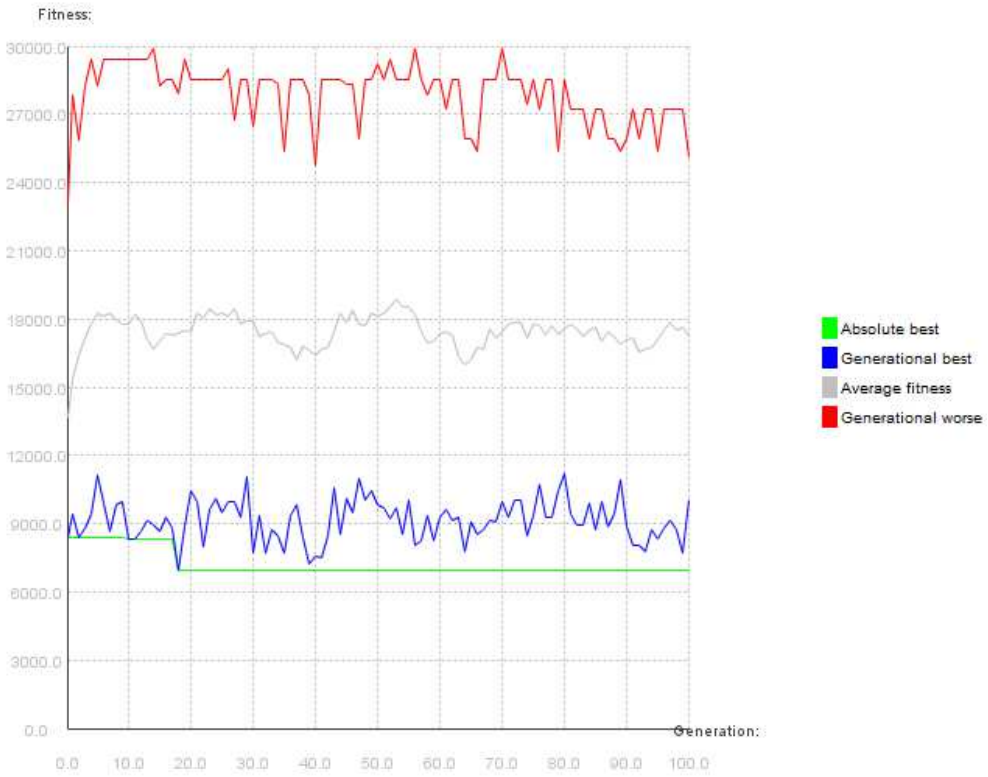


Figure 4.4: Roulette wheel

- The *truncation* method is very restrictive, because it repeatedly selects the individuals with best fitness of the population until the sample is completed with the established size. The positive part of this elitist method is the small likelihood to worsen, as several individuals stay invariant from a generation to the next one.

- The *stochastic universal* selection tries to provide consistency to the sampling, as it evenly distributes the selection of individuals with a single random measure. It has as counterpart that the way the elements of the population are ordered is likely to have an influence on the obtained result.

- The *remains* selection allows a chromosome to be chosen proportionally to its score. More formally, considering an individual S of a population of k chromosomes whose

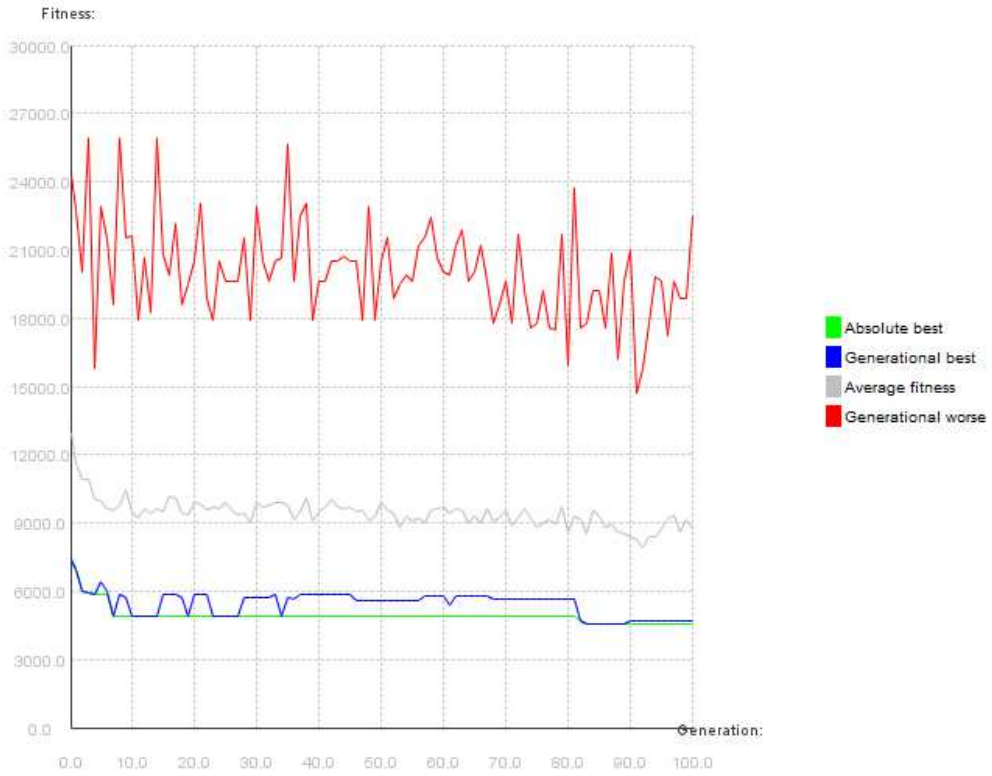


Figure 4.5: Truncation

score is p, S will be selected $p \cdot k$ times. As the resulting value is likely to be a real number, we round it down. Since we want the new generation to have k individuals, the remains method is not able to provide all of them. The remaining chromosomes to be selected are chosen with the roulette wheel method.

We can see in Figures 4.3 to 4.7 the execution of each of the selection methods for the same initial parameters. In general, the fitness tends to improve over the generations. However, the tournament method, see Figure 4.3, exhibits the best behaviour. The other methods do not have such fast improvement, since a local minimum is found, as it is the case of the truncation method, Figure 4.5. In the case of the stochastic universal and the remains methods, Figures 4.6 and 4.7, we find that the best fitness slowly improves in time.

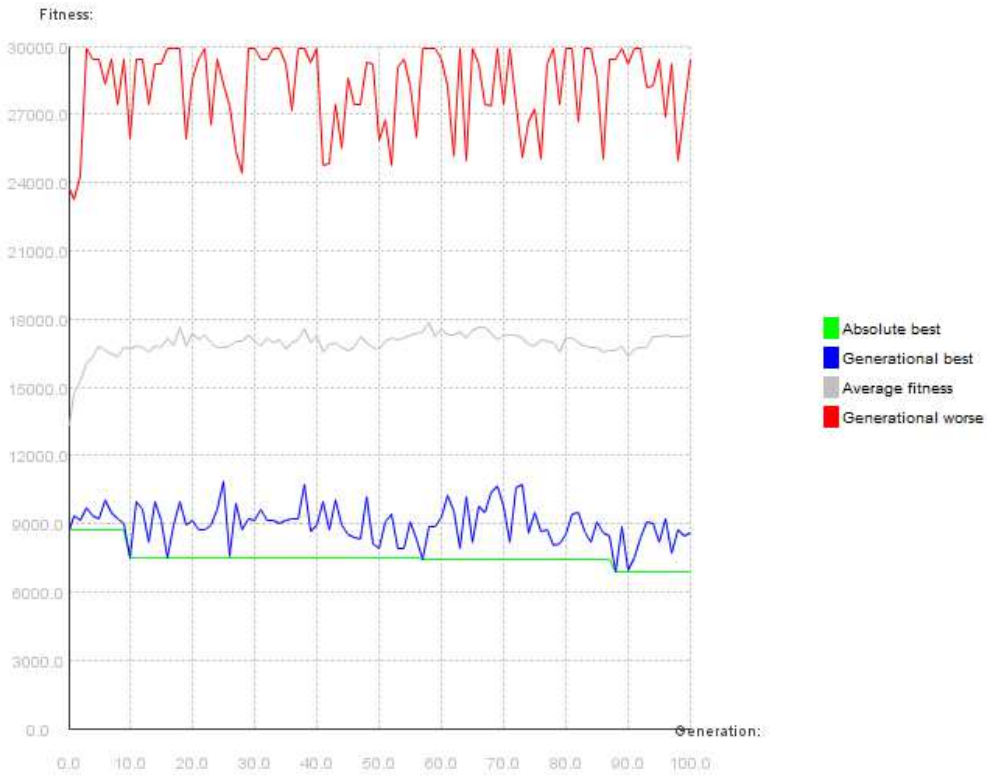


Figure 4.6: Stochastic universal

Nevertheless, there is an important difference between them: the average fitness of the remains method tends to improve whereas the average fitness of the stochastic universal method takes higher values (they represent worse solutions). This effect might occur due to big values on the overall fitness, making it harder to measure the differences between individuals that once combined with this selection technique induce a suboptimal evolution. With the roulette wheel method, see Figure 4.4, we have a rather random behavior. All three main graphics constantly vary within some bounds, although it seems to be a slight improvement at the end. A longer experiment could show further results, but the same number of iterations was considered for all the methods to illustrate their differences.

Concerning the fitness function, in Figures 4.3-4.7 we show the results of several experi-

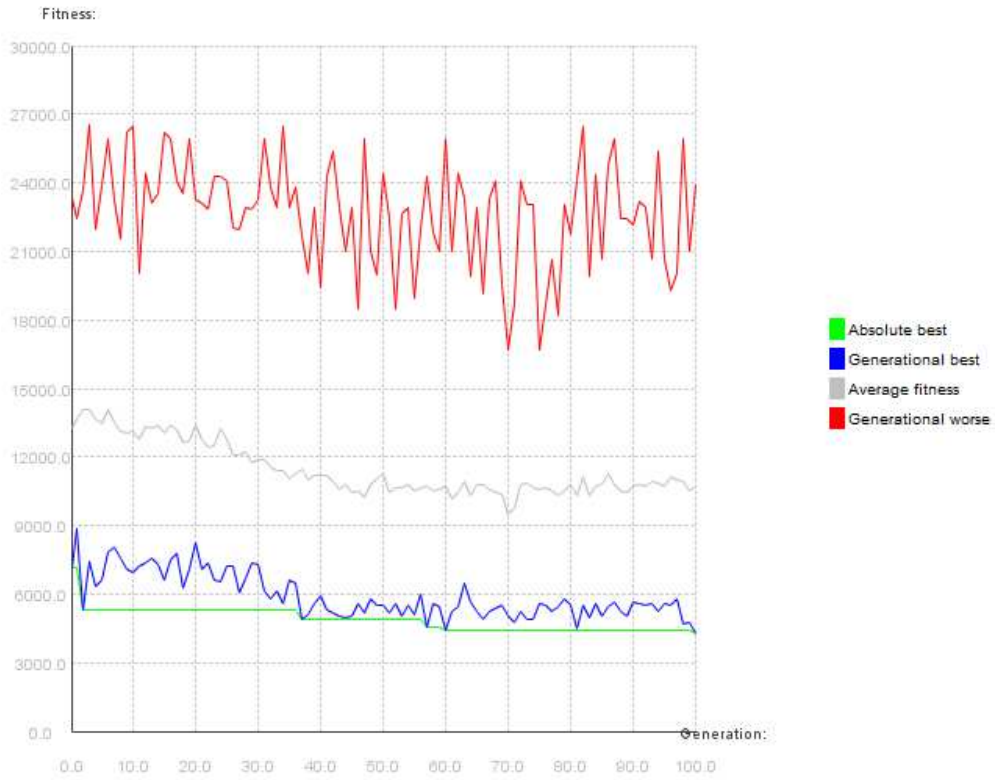


Figure 4.7: Remains

ments indicating how fitness varies along generations. The results are as expected. In short, there is a relatively big variance concerning the *worse individual* of each generation, that is, the highest value of fitness. This variance is smaller for *average fitness*, and the value notoriously improves after only 10 generations. The *generational best* graphic stabilizes, although there are small variations. The *absolute best* quickly converges to a local minimum that is sometimes improved later on. These executions only differ in the selection method, with the goal of illustrating the slight changes among them.

Crossover methods

For the crossover phase, which combines the selected individuals to produce a new generation, we have considered two methods:

- The *standard crossover* involves two chromosomes. It consists in choosing a random point on both individuals. Then, the tests to the right of that point are exchanged. If such modification generates a set of tests with more inputs than the specified bound, then the last tests are discarded until the bound is reached. Figure 4.8 shows the application of this method.

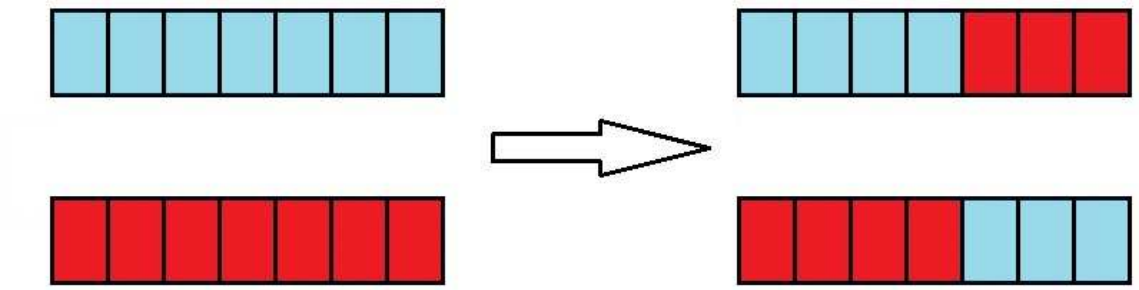


Figure 4.8: Standard crossover

- The *continuous crossover* also involves two chromosomes. In this case, several points are selected and the tests at the corresponding positions are exchanged. This approach is oriented to generate more diversity in the following generations. In this case, we need to pay attention to the total number of inputs of the new individuals. The inclusion of a test case cannot exceed the specified bound. Figure 4.9 represents the application of this method.

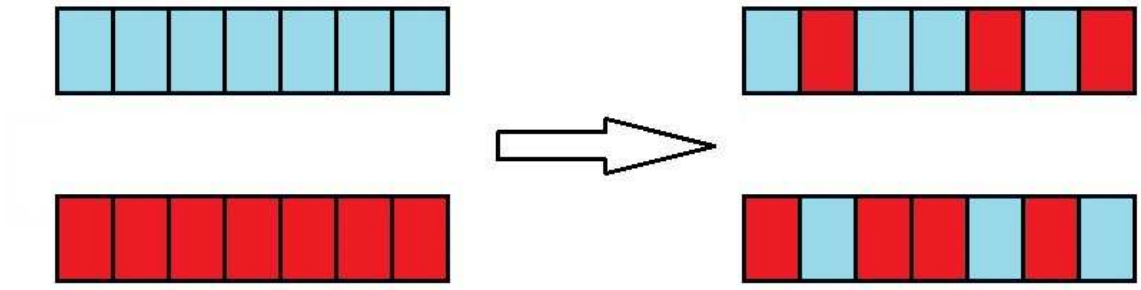


Figure 4.9: Continuous crossover

Mutation methods

As the initial population might not be sufficiently well distributed, it is sensible to refresh the population with some slight changes that could renew some stale state. In our case, we have designed two different techniques:

- *Adding mutation* is oriented to non-complete subsets. In these cases, it is possible to introduce an extra test to an individual without exceeding the bound of inputs. Despite increasing the number of tests on the whole population, due to the application of the crossover methods, it is possible to generate an exchange of tests where some of them have to be discarded. This method complements the possible loss of tests as it reactivates stationary individuals. This approach is illustrated in Figure 4.10.

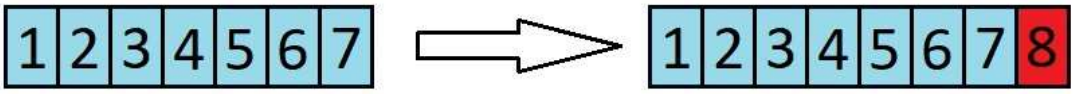


Figure 4.10: Adding mutation

- The *replacing mutation* method allows an individual to change one of its tests by another one from the initial test suite. This technique will include some slight changes to specific individuals that might either increase or decrease the relevance of a subset of tests in the population. This is illustrated in Figure 4.11.

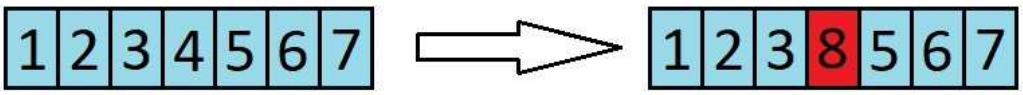


Figure 4.11: Replacing mutation

Replacement methods

Finally, the last step of the process corresponds to the replacement of the population by the new one. Again, we have two possibilities. On the one hand, the trivial option would be to substitute the current population by the new one, even if it could be worse. In this way, less operations are performed at this stage and, as a result, the execution will be faster. On the other hand, we could replace a percentage of the new generation by the best individuals of the current one. This approach will always allow the population to keep the best partial solution until it is improved. As a counterpart, more calculations have to be made and the associated cost might decelerate the execution. In Figure 4.12 we see how this elitist replacement behaves.

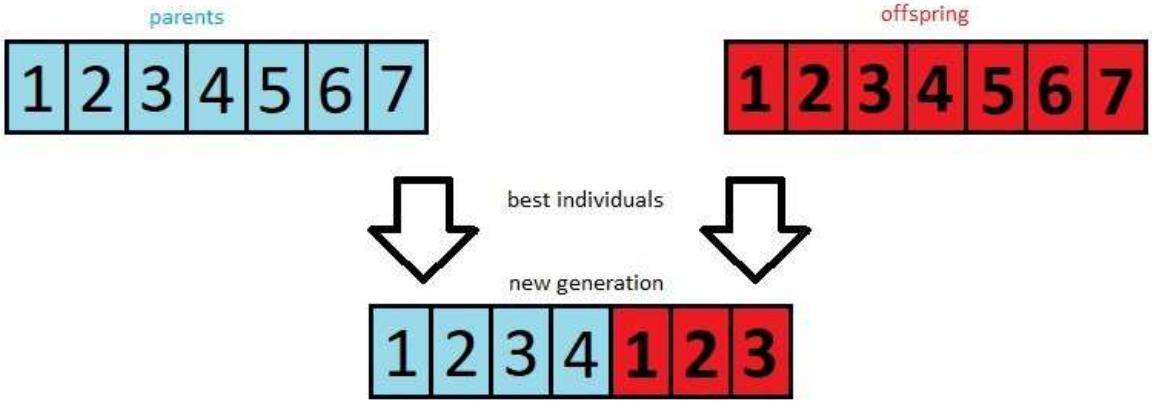


Figure 4.12: Elitist replacement

Chapter 5

The tool

In this chapter we present the tool that we have developed to implement the algorithms that we propose for selection of test cases. We focus on the Software Engineering aspects of the implementation and the design patterns that we have used.

5.1 GUI and MVC

Our tool was created following design patterns, such as Model-View-Controller (MVC), to make easier the interaction of the user and the tool. Figure 5.1 shows the MVC pattern components. The *model* represents the algorithms that we have proposed and the FSMs. Usually, the user interacts with the *controller* in order to perform changes in the model. Finally, the *view* represents the visual environment that the user needs to interact properly. The results and the relevant information are shown at the view.

In figure 5.2 we show the GUI representing the view of the pattern. The control tools appear on the left hand side of the view, where the values associated to the main parameters used in our algorithms can be established. Next, we introduce them.

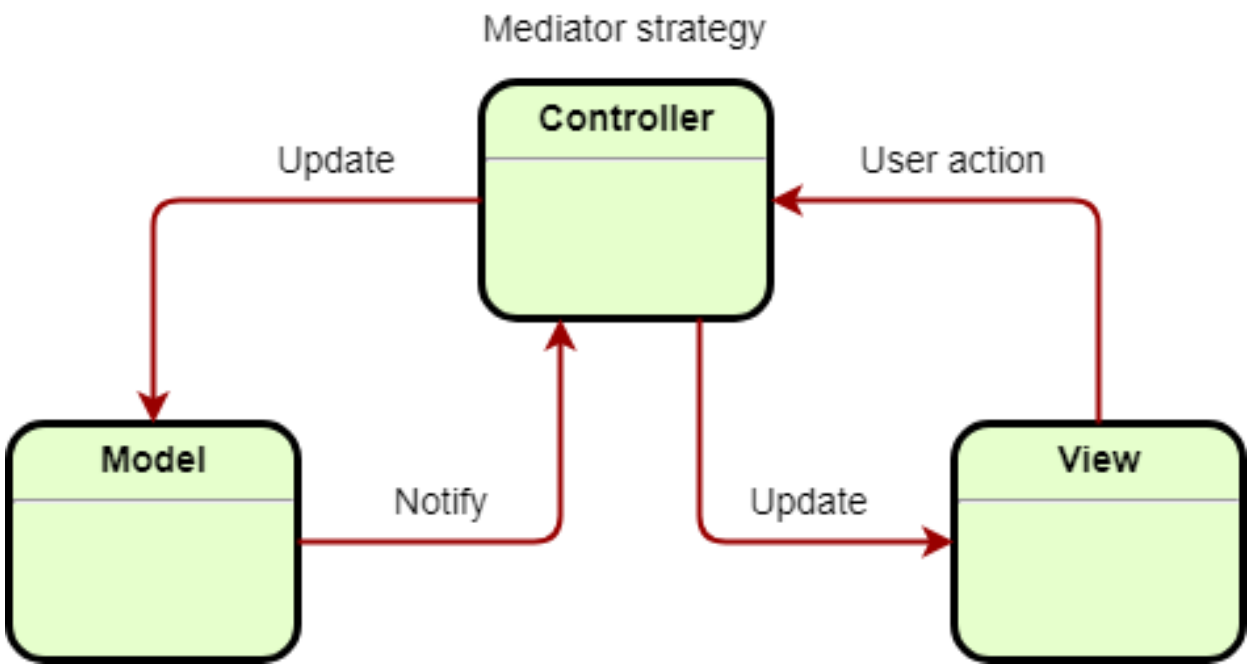


Figure 5.1: MVC pattern

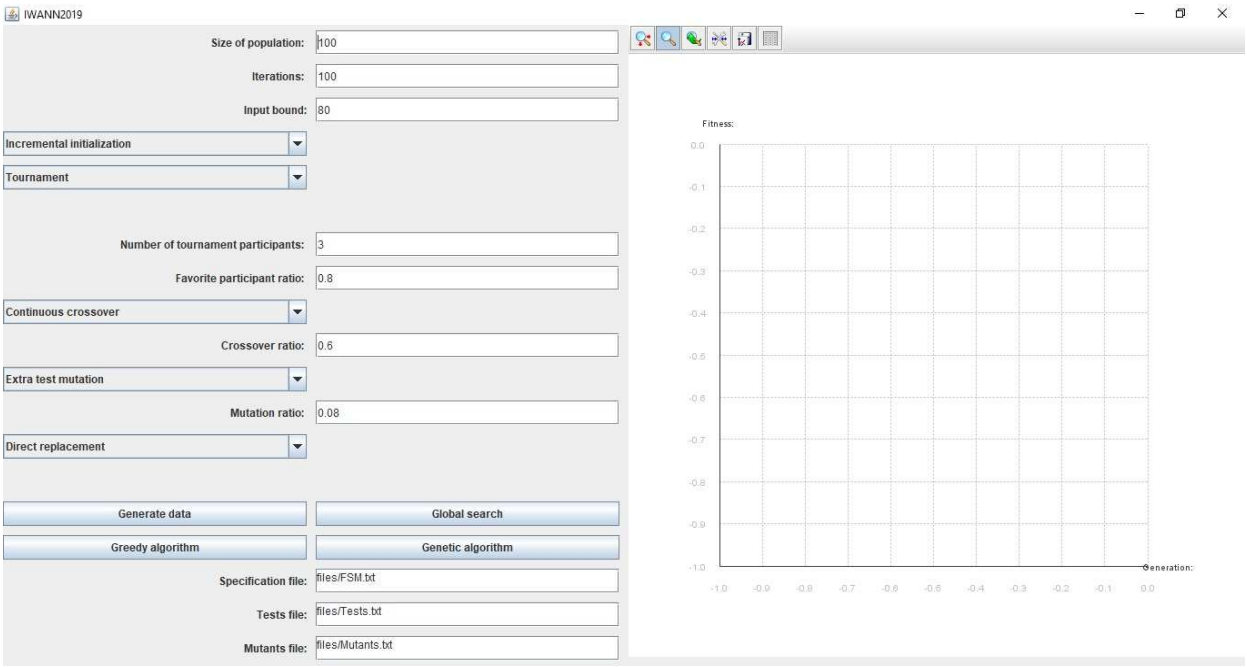


Figure 5.2: GUI of our tool

- *Size of population* indicates the number of individuals corresponding to each generation. This value will condition the evolution of the GAs. For example, a small number of individuals would not allow enough diversity in the breed. On the opposite side, a big number would significantly slow the process.

- *Iterations* corresponds to the upper bound on the number of generations that will be produced. This field provides the termination criterion for the algorithm.

- The *input bound* parameter acts as the budget or time constraint. It restricts the total number of inputs that the user wants to be applied during the testing process. Thereby, the amount of tests of each chromosome is related to the value of this field.

Currently, the type of *initialization* of the chromosomes is only informative because the user cannot select it. Nevertheless, future extensions of the program for other formalisms or different hypothesis (like non-determinism) could give rise to the application of other initialization methods and the users will have the possibility of selecting the most appropriate.

The *selection* method will determine the way the individuals of a generation will be selected to be mingled and evolve towards its offspring. As we indicated in Chapter 4.3, five different methods can be applied. Some of them, like truncation and tournament, require the user to provide values for some specific parameters, although a value is assigned to them by default. Those fields are only visible when the associated selection method has been chosen.

- *Truncation ratio* represents the proportion of the population that will be selected. In order to complete the full generation, these individuals will be selected as many times as required. The selected individuals have the best fitness of the population. This parameter is very sensitive to variations. For instance, a small value would cause a fast

convergence towards a value that does not have to be close to the best solution. Also, a slightly bigger value would tend to ignore the fitness function. It would be ignored in the sense that a big diversity of the population would be selected and, therefore, the distribution of the chosen individuals would be uniform. The consequence of this is that *bad* solutions of the problem would have the same weight as *good* solutions and the evolution of the algorithm would come to a halt. This parameter is only required for the truncation method.

- *Number of tournament participants* reflects how many candidates take part in the tournament. For example, having a single candidate would be equivalent to have a fully randomized selection, as the applicant would immediately win. As those participants are selected at random, a small number of candidates should be selected to hasten the process. This parameter is only required for the tournament method.

- The *favorite participant ratio* indicates the chances that the participant with the best fitness has to be selected. In that way, chromosomes with a low score could win a place in the tournament towards the next generation. This ratio adds some diversity while still allows the individuals with high scores to have a leading role in the evolution of the GA. If more than two participants compete for a place, then they are sorted by fitness; their odds are computed by taking into account the ratio assigned to this parameter. For example, for 3 players and assuming an 80% ratio for the favorite participant, the odds would be 0.8 for the individual with best fitness, 0.16 ($= 0.8 * (1 - 0.8)$) for the individual with the second best fitness, and 0.04 ($= 1 - 0.8 - 0.16$) for the individual with the worst fitness (among the nominees). This field is only required for the tournament method.

Additionally, the user must select the *crossover* method to be used during the execution

of the GA. Since the number of possible changes that may be produced, depending on the method, is significantly different, the crossover method drastically varies the value of the next parameter.

The *crossover ratio* highly depends on the type of crossover selected. As we said previously, one method has a higher spectrum of changes than the other. This value represents the percentage of changes to be performed. In this sense, this number should vary in a higher or smaller scale in order to keep a sensible structure between generations. Usually, the continuous crossover does not need a high value to perform as many changes as the standard crossover does.

The *mutation* method for the specific algorithm that we may want to run must also be selected. The mutation methods have a similar amount of possible changes concerning an individual. As such, the value of the next parameter is not related to the actual choice of the method, but to the diversity during the execution of the GA.

The *mutation ratio* simply decides how many mutations will be performed on a chromosome. We advise not to use a big ratio, as too many changes could be produced. Performing too many mutations could drastically disturb the natural evolution of the GA.

Finally, the user has to choose how to replace the current generation by the new one. The user can select either to replace the population or to keep the best individuals from the previous generation. In the case that the elitist replacement is selected, the *elitist ratio* must be provided. We also recommend keeping this value low. Otherwise, the evolution could be really slow, wasting an unnecessary amount of resources.

We can run the configured algorithm in different ways. The *global search* mode looks for the perfect solution over all the possible combinations. We highly recommend using this option only in the case that the user is dealing with small problems, because in other case the process will not terminate. The second execution mode corresponds to the *greedy*

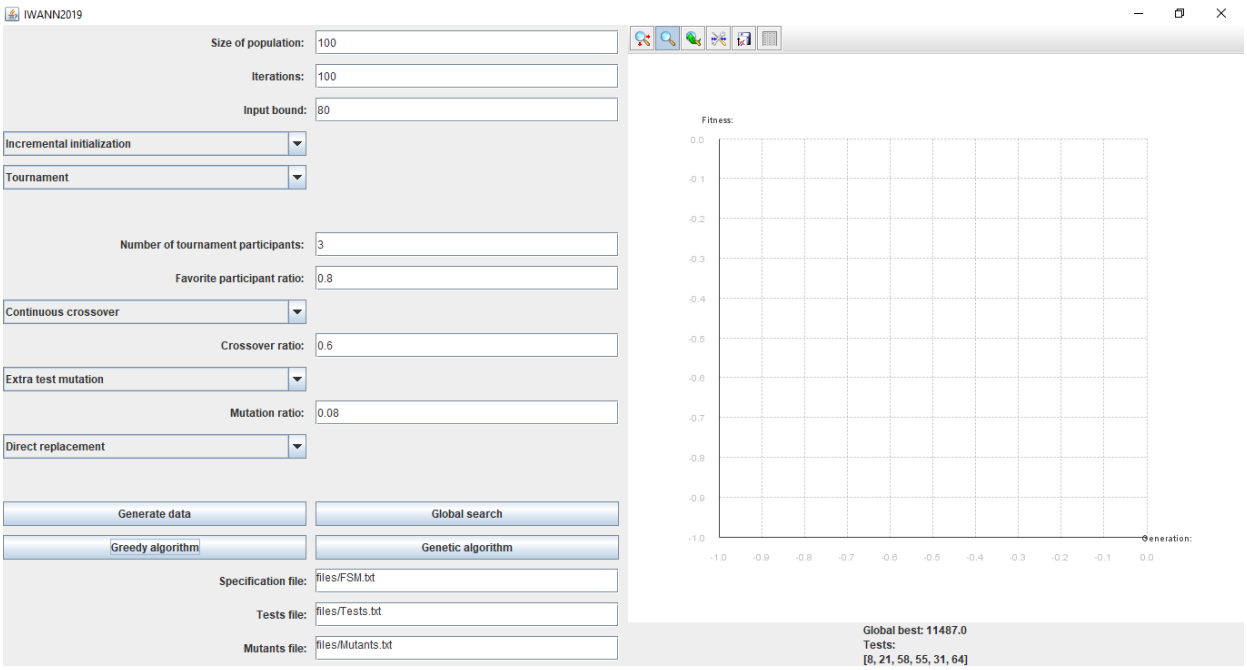


Figure 5.3: GUI with best fitness and tests

algorithm and the last one will apply a *genetic algorithm* that will be configured with the values provided by the user for the different parameters. The tool also provides an option, *generate data*, to generate mutants and tests from a specification. It will generate all the mutants from a specification, and a random set of tests to be used during the execution of the algorithms.

On the right hand side of the GUI, the graphics corresponding to the information related to the execution of the GA is displayed. Due to the fact that the greedy algorithm and the full search are deterministic, no graphics will appear in this area of the GUI. Nevertheless, in all the cases, the fitness of the global best individual is shown below the graphics area. In Figure 5.3 we see the fitness of the best individual and the tests that were used during the execution. In Figure 5.4 we see the results of the execution of a variant of our GA. Therefore, both the value of the best subset of tests found and the graphic of the evolution

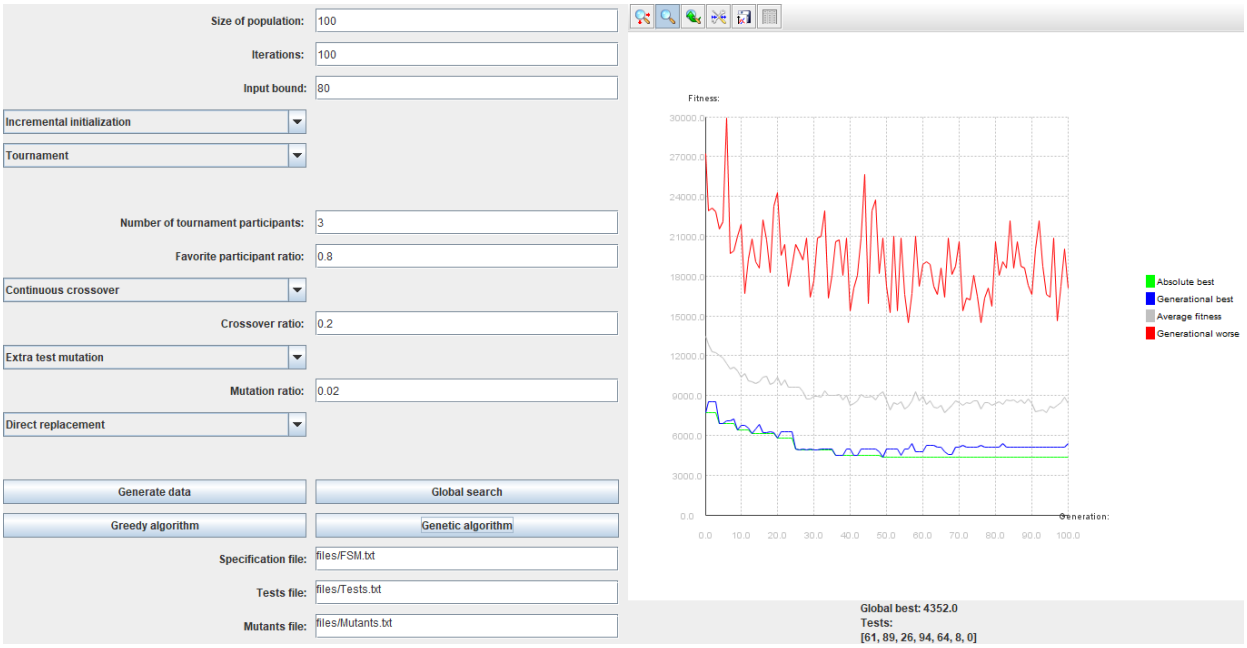
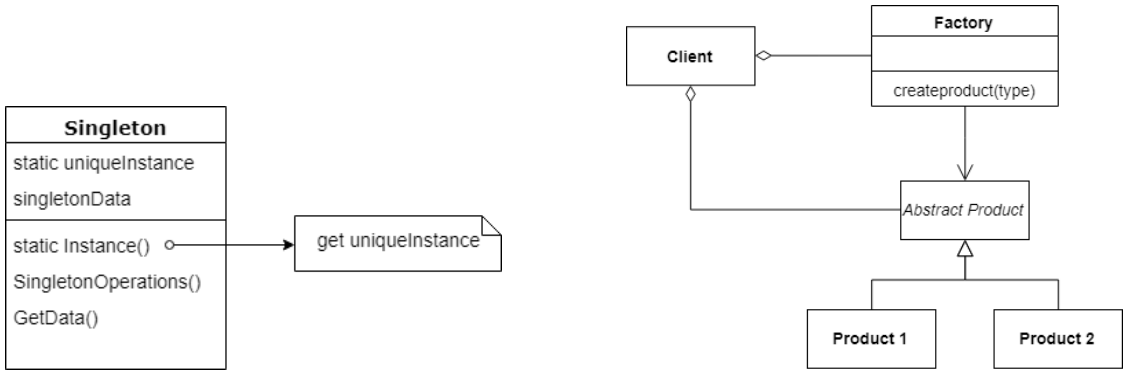


Figure 5.4: GUI with graphics

of each generation appear in the GUI.

5.2 Other patterns

Concerning the controller, we follow the Singleton pattern shown in Figure 5.5a. We use this pattern because we only need one object to manipulate the data from the view towards the model. The controller is in charge of running the algorithms, as well as of interpreting the inputs provided by the parameters in the view. The execution of any of our algorithms will modify the values of the model. Those changes will be reflected in the *right* hand side of the view. In the controller class we can find the critical implementation of the methods described in Chapter 4, although some additional classes help to provide the right elements. These classes correspond to factory classes that generate the appropriate initialization, selection, crossover, mutation and replacement methods. If the user decides to choose them at the



(a) Singleton pattern (b) Factory pattern

Figure 5.5: Design patterns

GUI, then the chosen class is dynamically decided. These classes follow the Factory pattern represented in Figure 5.5b. They provide an actual instance of a class that implements a common interface. This helps to select a method in a dynamic execution with a good structure.

5.3 Class diagram

We will now present a class diagram corresponding to the implementation of the tool. The implementation and some examples are freely available under a GPL-3.0 license at https://github.com/miguelbpsg/IWANN19.

In Figure 5.6 we see that the complexity of our tool developed forces us to present a simplified version of the classes involved in it. As mentioned before, the high number of attributes and methods would complicate the diagram. We also avoided showing the multiplicities of the relationships.

First, we can clearly observe the MVC pattern structured in the packages. The view does not need any additional classes to perform its tasks, so it does not need its own package. The

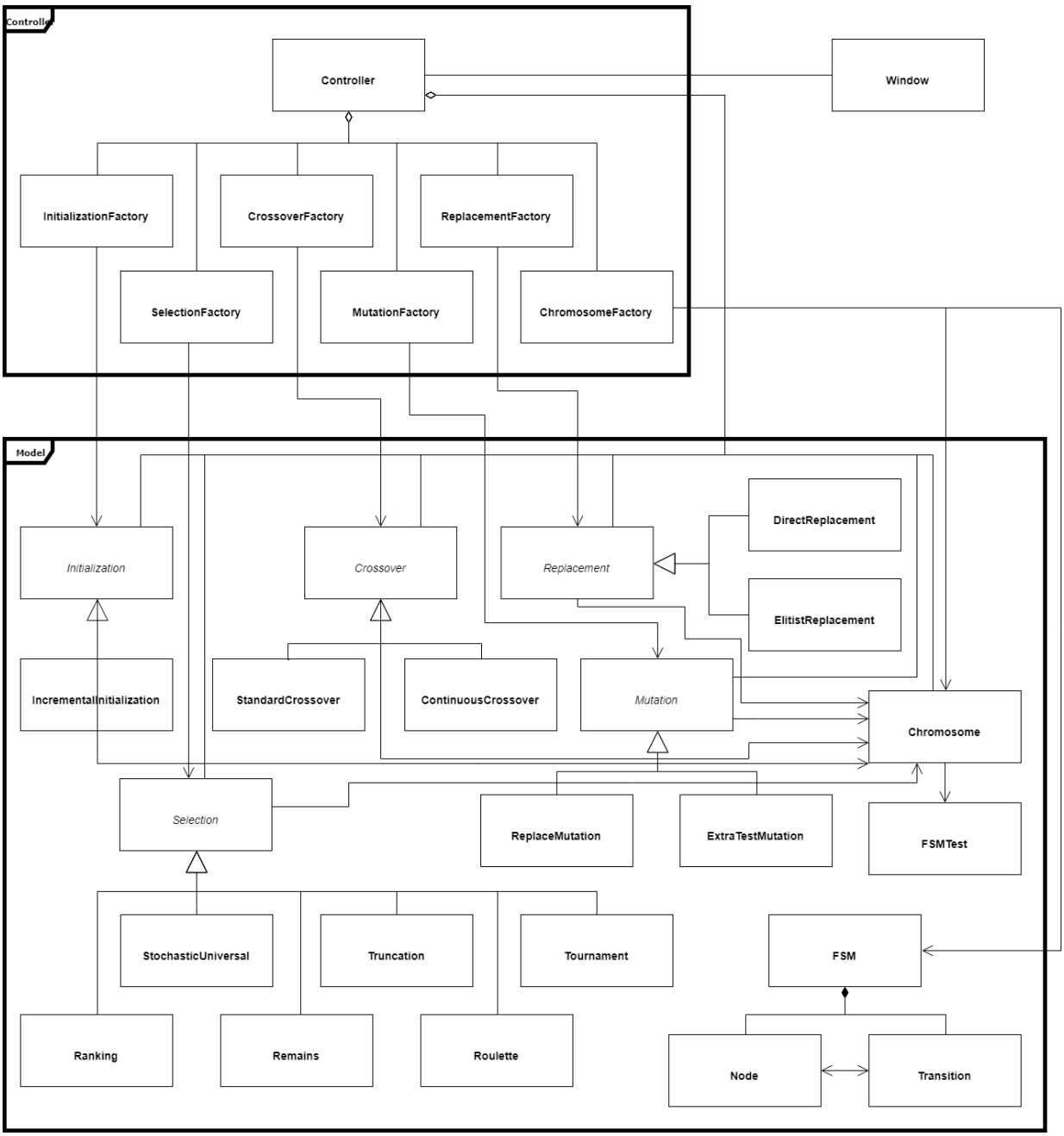


Figure 5.6: Class diagram

controller needs all the factories to adequately choose the methods dynamically selected at the view. Finally, the model stacks most of the classes and the distribution of the program. In the model, several classes are grouped by the methods described in Section 4.3. They are extended into each concrete representation and a representation for FSMs, with nodes and transitions, can be found. We also find the core of our model: the chromosome. The FSM class is the perfect complement to the FSMTest class. The former is able to generate all the relevant information that will be used during the execution of the algorithm. We refer to the generation of a set of tests as well as a ratio of mutants, or even all the possible mutants from a specification, having as a result a complete tool to fulfill our work. The FSMTest will be applied to mutants to generate a matrix as the one introduced in Definition 4.

Chapter 6

Experiments

In this chapter we report on the experiments that we have performed and the obtained results. We discuss the results of applying the different algorithms that we have proposed. We have considered different bounds on the number of inputs that the solution can have. We have also analyzed the time that the different approaches need to compute their solutions.

6.1 Description of the experiments

Our experiments consisted in the execution of the three described algorithms over the same specification, mutants, initial set of tests and maximum number of allowed inputs. We performed the experiments for several combinations of them. Afterwards, we compared the results both in time needed to compute the solution and in the goodness of the solution. Note that the initial set of tests should not be confused with the set of tests produced in the initialization of the GA. The former is provided as a precondition of the problem. This is the set of tests that we should aim to apply but if our resources do not allow us to try all of them, then we should apply a *good* subset of them. Computing this final subset is the

goal of our approaches. The latter is obtained during the first step of the execution of our GA.

It is clear that the smaller the FSM is, the lesser number of mutants will be generated. We have considered a specification with 10 states, 3 inputs and 5 outputs. We have obtained around 300 mutants after applying the mutation operators considered in our framework. We also considered 3 possible bounds on the number of inputs and 2 initial sets of tests, resulting in 6 representative cases. Next, we give the details of each of them.

	Exp. 1	Exp. 2	Exp. 3	Exp. 4	Exp. 5	Exp. 6
Max. Inputs	30	80	150	30	80	150
Tests	99	99	99	957	957	957

Table 6.1: Summary of the experiments

1. The first experiment consisted in allowing a maximum of 30 inputs in the solution and starting with a set of 99 tests. All algorithms yield a very good (if not the best) solution on a reasonable amount of time as there are few possible results. This experiment is a good baseline to show that all the algorithms provide good solutions.

2. Next, we increased the bound on the number of inputs, from 30 to 80, maintaining the initial set of tests. This experiment should show us the evolution of the algorithms depending on the number of inputs.

3. To conclude the experiments with a set of 99 initial test cases, we increased the bound on the number of inputs up to 150 inputs.

4. The next variation considered the smallest bound on the number of inputs, that is, 30 inputs, and a bigger set of tests. We consider a set of 957 tests.

5. The next experiment dealt with the intermediate bound, that is, 80, and the biggest set of tests.

6. Finally, the last experiment considered the biggest bound on the number of inputs and the biggest set of tests, that is, 150 inputs and 957 tests.

These experiments were performed with all the possible configurations of our GA. We also applied the full search and the greedy algorithm. We had two goals. First, we wanted to compare the different versions of the GA. Second, we wanted to compare the execution of the other two algorithms with the GA. We considered the default configuration, that is, tournament selection over 3 participants with a ratio of 80%, continuous crossover with a probability of 0.6, an extra test mutation option with a 0.02 coefficient and a direct replacement. The experiments were carried out on an Intel i5-8250U, with a frequency of 3.4 GHz and 8GB of RAM.

It is worth to mention that most of the configurations of our GAs gave very similar results. It should be also noted that the difference among them only corresponds to the selected methods (selection, mutation, crossover and replacement). The value of the parameters required for the different configurations (mutation ratio, crossover ratio, etc.) were not changed.

We also tested the bounds and initial sets of tests that the full search was able to compute. We were not able to exceed a bound of 60 inputs over an initial set of 99 tests, taking over a week to compute the solution.

6.2 Evaluation

As expected, as soon as the systems were sizable and we had a nontrivial initial set of tests, the generation of all the possible subsets was unfeasible due to the combinatorial explosion on the number of subsets. For example, if we had 40 initial tests with an average of 10 inputs and we could choose tests up to 150 inputs, we would have more than 40 billion possible subsets. Nevertheless, whenever we consider small bounds, it is possible to compute an optimal solution, guaranteeing that we obtain the best subset of tests to be applied to the SUT.

In terms of relative cost, we observed that the greedy algorithm was always the fastest as we can see in Table 6.2. The time needed to compute the solution mainly depended on the size of the given set of tests but it also had a small dependence on the maximum number of allowed inputs. This dependence arises from determining how many times the matrix has to be sorted. Considering its efficiency, our GA was able to provide very good results. In the only experiment where we were able to compute all the combinations, the result was almost equivalent to the optimal one, as Table 6.3 shows. These values are 11.761 of fitness for full search[1] versus 11.887 of fitness on the GA. However, the solution was computed in less time, showing evidence of the usefulness of the approximate technique.

Focusing on Tables 6.2 and 6.3, we have that a higher bound on the number of inputs always increases the execution time. In contrast, the fitness of the obtained solution is improved. Also, comparing all the experiments, we observe that a bigger initial set of tests induces a higher execution time, but better results are obtained for the same bound of inputs.

These experiments show that the GA can adequately compete, depending on the re-

[1]The fitness for full search and greedy algorithm is how we calculate the aggregate value of the solution. In this way, all three methods have the same evaluation function and are easy to compare.

sources, and complement the results of the greedy algorithm. It is true that the GA requires some more time to be evaluated, but considering the obtained results we find it worth to use this extra computing power.

	Time Exp. 1	Time Exp. 2	Time Exp. 3	Time Exp. 4	Time Exp. 5	Time Exp. 6
Genetic	91	190	220	208	296	462
Greedy	22	23	26	147	178	263
Full search	1.355	–	–	–	–	–

Table 6.2: Time results (in milliseconds)

	Fitness Exp. 1	Fitness Exp. 2	Fitness Exp. 3	Fitness Exp. 4	Fitness Exp. 5	Fitness Exp. 6
Genetic	11.887	5.866	2.877	12.225	3.212	1.988
Greedy	16.166	7.687	5.415	22.746	13.543	7.429
Full search	11.716	–	–	–	–	–

Table 6.3: Fitness results (higher values denote worse results)

	Fitness Exp. 1	Fitness Exp. 2	Fitness Exp. 3	Fitness Exp. 4	Fitness Exp. 5	Fitness Exp. 6
S,A,D	11.887	4.966	2.311	12.059	3.795	1.926
S,A,E	11.887	5.575	2.094	11.141	4.520	1.748
S,R,D	11.887	4.508	2.435	12.511	4.502	1.896
S,R,E	**11.683**	4.737	2.649	12.012	4.567	1.957
C,A,D	12.042	3.863	2.116	11.376	2.919	1.729
C,A,E	**11.683**	3.745	**2.034**	**10.376**	2.763	1.675
C,R,D	11.863	3.968	2.063	12.118	3.676	1.764
C,R,E	11.887	3.978	2.137	11.172	**2.625**	1.683

Table 6.4: Fitness results for Tournament

Tables 6.4-6.8 show us that the best results for each experiment are distributed on the different configurations of the GAs, but some important conclusions can be extracted for

	Fitness Exp. 1	Fitness Exp. 2	Fitness Exp. 3	Fitness Exp. 4	Fitness Exp. 5	Fitness Exp. 6
S,A,D	13.209	6.992	2.840	13.196	5.339	3.253
S,A,E	11.847	5.864	2.778	11.227	4.687	2.361
S,R,D	14.483	7.246	3.272	13.449	6.446	3.654
S,R,E	**11.683**	4.966	2.476	11.291	5.274	2.241
C,A,D	13.255	6.626	2.548	12.675	6.564	3.319
C,A,E	12.059	5.339	2.446	11.672	4.758	2.410
C,R,D	13.944	6.540	3.779	13.298	6.727	3.451
C,R,E	12.059	4.720	2.610	12.663	3.852	1.943

Table 6.5: Fitness results for Roulette wheel

	Fitness Exp. 1	Fitness Exp. 2	Fitness Exp. 3	Fitness Exp. 4	Fitness Exp. 5	Fitness Exp. 6
S,A,D	11.863	5.659	2.419	11.731	4.397	2.212
S,A,E	11.887	4.121	2.283	10.619	3.732	2.045
S,R,D	12.412	5.113	2.437	11.367	4.257	1.826
S,R,E	**11.683**	4.808	2.200	10.806	4.304	2.025
C,A,D	11.863	3.745	2.280	11.245	3.596	1.769
C,A,E	**11.683**	3.765	2.196	10.860	2.903	1.799
C,R,D	11.887	4.384	2.274	10.992	3.888	1.857
C,R,E	**11.683**	4.121	2.065	11.434	3.187	**1.626**

Table 6.6: Fitness results for Remains

	Fitness Exp. 1	Fitness Exp. 2	Fitness Exp. 3	Fitness Exp. 4	Fitness Exp. 5	Fitness Exp. 6
S,A,D	12.465	4.171	2.417	13.055	5.383	2.576
S,A,E	12.059	5.159	2.420	12.753	5.323	2.030
S,R,D	11.887	6.425	2.170	13.634	5.742	2.655
S,R,E	13.369	5.201	2.587	12.498	5.383	2.301
C,A,D	12.465	4.117	2.158	12.664	3.587	1.896
C,A,E	12.200	4.172	2.111	11.345	2.972	1.869
C,R,D	12.223	**3.625**	2.211	12.503	4.169	1.808
C,R,E	11.887	4.160	2.183	11.667	3.476	1.729

Table 6.7: Fitness results for Truncation

	Fitness Exp. 1	Fitness Exp. 2	Fitness Exp. 3	Fitness Exp. 4	Fitness Exp. 5	Fitness Exp. 6
S,A,D	12.465	6.366	3.317	12.302	6.510	3.206
S,A,E	12.116	4.822	2.639	11.505	5.327	2.509
S,R,D	13.955	6.717	3.414	13.501	5.049	3.150
S,R,E	12.059	5.937	2.956	11.858	5.011	2.713
C,A,D	12.465	6.545	2.709	12.994	6.023	2.965
C,A,E	11.847	4.996	2.248	11.035	3.669	1.957
C,R,D	12.483	6.612	3.412	13.532	6.444	3.318
C,R,E	11.847	4.447	2.271	12.185	3.788	2.020

Table 6.8: Fitness results for Stochastic universal

the SUT and the tests applied. The best fitness found for each experiment is highlighted in blue and boldface in each of the corresponding tables. We can observe that the tournament selection with elitist replacement had the most appearances among all the configurations, showing more adequacy to this concrete system.

Nevertheless, such selection method was not overwhelmingly dominant over the others. In that sense, we consider that all the algorithms proposed had some good features. The roulette wheel, remains, tournament and truncation methods got at least one minimal solution for a different experiment, on different configurations of crossover, mutation and replacement. Concerning stochastic universal, despite not providing any *excellent* solution, the values obtained are not distant enough to be considered a useless method.

In terms of the time needed to execute the different GAs that appear in Tables 6.9-6.13, the main observation is the big increase of time required when an elitist replacement is considered. In addition, we can also observe that the truncation selection had a bigger cost in terms of time in every experiment. All the delays are due to the need of sorting several times the population considering the order induced by the fitness function.

	Time Exp. 1	Time Exp. 2	Time Exp. 3	Time Exp. 4	Time Exp. 5	Time Exp. 6
S,A,D	70	101	171	171	222	270
S,A,E	406	669	984	476	745	1055
S,R,D	71	101	172	172	202	254
S,R,E	415	586	921	515	791	910
C,A,D	81	102	192	173	215	297
C,A,E	202	316	516	311	403	700
C,R,D	81	121	192	174	222	293
C,R,E	233	303	518	344	436	669

Table 6.9: Time results for Tournament

	Time Exp. 1	Time Exp. 2	Time Exp. 3	Time Exp. 4	Time Exp. 5	Time Exp. 6
S,A,D	73	81	111	164	192	215
S,A,E	152	262	414	263	395	556
S,R,D	68	73	99	160	174	213
S,R,E	162	233	374	281	337	485
C,A,D	70	103	152	174	202	245
C,A,E	162	246	458	253	355	556
C,R,D	79	92	121	172	194	223
C,R,E	171	231	417	243	341	566

Table 6.10: Time results for Roulette wheel

	Time Exp. 1	Time Exp. 2	Time Exp. 3	Time Exp. 4	Time Exp. 5	Time Exp. 6
S,A,D	81	101	164	181	212	281
S,A,E	235	333	536	435	444	667
S,R,D	70	109	164	172	204	263
S,R,E	291	333	558	372	526	677
C,A,D	73	111	185	171	220	293
C,A,E	222	283	523	293	414	645
C,R,D	70	111	182	172	203	292
C,R,E	201	283	508	333	405	669

Table 6.11: Time results for Remains

	Time Exp. 1	Time Exp. 2	Time Exp. 3	Time Exp. 4	Time Exp. 5	Time Exp. 6
S,A,D	366	730	1137	503	820	1343
S,A,E	1559	3405	6170	1711	3630	6448
S,R,D	387	758	1206	465	870	1367
S,R,E	1480	3497	6158	1649	3659	6343
C,A,D	193	343	507	301	425	618
C,A,E	1063	2381	4378	1164	2591	4605
C,R,D	195	334	536	324	426	670
C,R,E	1196	2383	4389	1266	2653	4508

Table 6.12: Time results for Truncation

	Time Exp. 1	Time Exp. 2	Time Exp. 3	Time Exp. 4	Time Exp. 5	Time Exp. 6
S,A,D	70	91	131	171	192	233
S,A,E	162	215	325	261	326	446
S,R,D	71	81	121	170	184	232
S,R,E	151	192	291	251	293	396
C,A,D	80	103	162	181	211	262
C,A,E	161	212	385	233	314	485
C,R,D	71	93	143	172	192	242
C,R,E	160	192	342	234	484	447

Table 6.13: Time results for Stochastic universal

Chapter 7

Conclusions and future work

In this thesis, based on recent conference paper [3], we present different solutions to the problem of obtaining good sets of tests out of big test suites. Ideally, if a tester is provided with a set of tests, then the tester should apply all of them to the SUT. However, the time and resources devoted to testing are usually limited and the tester can apply only a subset of these tests. If we are working within a framework where the tester applies inputs and receive outputs, then this bound is given by the number of inputs that the tester can apply. This is an important problem in testing and in addition to provide a sound theoretical framework, it is a must to develop tools supporting the frameworks. We have developed a tool implementing all the algorithms presented in this work. Our tool is able to, given an initial set of tests and the maximum number of inputs that we can really apply, compute a subset of the initial test suite with any of the proposed algorithms. It can be done with the different variants of the GA and the greedy algorithm discussed in this work. In addition, the tool supports the process of generating mutants from a specification of the SUT.

The results show that our GA usually finds an excellent solution. In general, the GA beats the greedy algorithm, needing a slightly higher amount of time to compute the result.

For smaller experiments, where full search could be effectively computed, the differences between the best solution and the one obtained applying the GA were very small: the fitness of the full search approach was around $1,5\%$ better but it needed 14 times longer to compute it. Therefore, we can be satisfied with the results considering the complexity of the problem.

As future work, we plan to extend the framework to deal with other FSM-like formalisms. A first line of work is to consider probabilistic FSMs, where nondeterminism is probabilistically quantified. We will take as initial step previous work on mutation testing of probabilistic FSMs [15] complemented with recent work on conformance relations for probabilistic systems [17]. An orthogonal line or work that we would like to pursue is to adapt our framework to test in the distributed architecture [16], where several users interact over the same data but cannot observe what the others are doing. We plan to perform experiments on real-world frameworks where bigger systems will be evaluated, presumably clarifying the metrics. Finally, we would like to improve the usability and report features of our GUI so that the whole interaction with the algorithms and its extensions could be followed and such that more complex graphs could be shown and compared.

Bibliography

[1] P. Ammann and J. Offutt. *Introduction to Software Testing.* Cambridge University Press, 2nd edition, 2017.

[2] C. Andrés, M. G. Merayo, and M. Núñez. Formal passive testing of timed systems: Theory and tools. *Software Testing, Verification and Reliability*, 22(6):365–405, 2012.

[3] M. Benito-Parejo, I. Medina-Bulo, M. G. Merayo, and M. Núñez. Using genetic algorithms to generate test suites for FSMs. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 741–752. Springer, 2019.

[4] D. J. Berndt and A. Watkins. High volume software testing using genetic algorithms. In *38th Annual Hawaii Int. Conf. on System Sciences, HICSS'05*, page 318b. IEEE Computer Society, 2005.

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

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Program mutation: A new approach to program testing. In *Infotech State of the Art Report on Software Testing*, pages 107–126, 1979.

[7] K. Derderian, M. G. Merayo, R. M. Hierons, and M. Núñez. A case study on the use of genetic algorithms to generate test cases for temporal systems. In *11th Int. Conf. on Artificial Neural Networks, IWANN'11, LNCS 6692*, pages 396–403. Springer, 2011.

[8] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo. Evolutionary mutation testing. *Information and Software Technology*, 53(10):1108–1123, 2011.

[9] M. R. Girgis. Automatic test data generation for data flow testing using a genetic algorithm. *Journal of Universal Computer Science*, 11(6):898–915, 2005.

[10] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *24th Int. Symposium on Software Testing and Analysis, ISSTA'15*, pages 211–222. ACM Press, 2015.

[11] D.E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning.* Addison-Wesley, 1989.

[12] N. K. Gupta and M. K. Rohil. Using genetic algorithm for unit testing of object oriented software. In *1st Int. Conf. on Emerging Trends in Engineering and Technology, ICETET'08*, pages 308–313. IEEE Computer Society, 2008.

[13] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3:279–290, 1977.

[14] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[15] R. M. Hierons and M. G. Merayo. Mutation testing from probabilistic and stochastic finite state machines. *Journal of Systems and Software*, 82(11):1804–1818, 2009.

[16] R. M. Hierons, M. G. Merayo, and M. Núñez. Bounded reordering in the distributed test architecture. *IEEE Transactions on Reliability*, 67(2):522–537, 2018.

[17] R. M. Hierons and M. Núñez. Implementation relations and probabilistic schedulers in the distributed test architecture. *Journal of Systems and Software*, 132:319–335, 2017.

[18] B. F. Jones, D. E. Eyres, and H.-H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal*, 41(2):98–107, 1998.

[19] B. F. Jones, H.-H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11:299–306, 1996.

[20] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[21] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 3rd, revised and extended edition, 1996.

[22] A. Shi, T. Yung, A. Gyori, and D. Marinov. Comparing and combining test-suite reduction and regression test selection. In *10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'15*, pages 237–247. ACM Press, 2015.

[23] M. Srinivas and L. M. Patnaik. Genetic algorithms: A survey. *IEEE Computer*, 27:17–27, 1994.

[24] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *7th Genetic and Evolutionary Computation Conference, GECCO'05*, pages 1053–1060. ACM Press, 2005.

[25] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.