Mutation testing from Finite State Machines

Curso Académico 2019-2020

Juan Bosque Romero
Jose María López Morales
Dirigido por Manuel Núñez

TRABAJO DE FIN DE GRADO
GRADO DE INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Contents

1 Introduction **1**

1.1 Finite state machines, mutants and tests 1

1.2 Previous Work 2

1.3 Objectives 4

1.4 Project structure 5

1.4.1 Finite state machine functionality 5

1.4.2 Test functionality 6

1.5 Interface functionality 6

2 Formal definitions **9**

3 Development of the system **13**

3.1 Mealy Machine Package 13

3.1.1 MealyMachine.java 13

3.1.2 Transition.java 15

3.1.3 State.java 16

3.1.4 TransitionFunction.java 16

3.2 Test Suite Package 17

3.2.1 Test.java 17

3.2.2 TestComparer.java 18

3.3 Adaptation of tests for non-determinism 19

3.3.1 Tree.java 19

3.3.2 NonDeterministicTest.java 20

3.3.3 NonDeterministicTestComparer.java 20

3.4 Visual Interface Package 21

3.4.1 FirstScreen.java 21

3.4.2 CreateManualAutomaton.java 22

3.4.3 TestingScreen.java 23

4 Experiments **27**
4.1 Use case 1 27
4.2 Use case 2 29
4.3 Use case 3 29
4.4 Use case 4 31
4.5 Use case 5 33

5 Contributions **37**
5.1 Contribution of Jose María 37
5.2 Contribution of Juan 38

6 Conclusions and future work **41**

A Implementation of core functions **45**
A.1 Mutation functions 45
A.2 TreeAux attributes and children generation 46
A.3 setStatesI implementation 47
A.4 First implementation of mutation testing applied to deterministic FSMs . . 47
A.5 Core functions of Tree.java 48
A.6 NonDeterministicTest.java 49
A.7 NonDeterministicTestComparer.java 49

List of Figures

1.1 State diagram for a turnstile 2
1.2 Moore machine 3
1.3 Mealy machine 3

2.1 Example of an FSM 9
2.2 Mutation *Mu* of previous FSM 10
2.3 Tree test 11

3.1 FirstScreen 22
3.2 CreateManualAutomaton 23
3.3 TestScreen 24
3.5 Comparison matrix and corresponding outputs 26

4.1 FirstScreen after alphabets are introduced 27
4.2 Transitions introduced in CreateManualAutomaton screen 28
4.3 FirstScreen after the transitions have been created 28
4.4 TestScreen after creating the test suite and running the tests 29
4.5 Creating a random FSM with given alphabets and states 30
4.6 FSM Randomly generated as text 30
4.7 FSM Randomly generated as diagram 30
4.8 TestSuite and comparison matrix 31
4.9 Outputs generated by FSM when applied inputs from test case 31
4.10 FSM that we will manually generate 32
4.11 Transitions introduced in CreateManualAutomaton screen 32
4.12 TestScreen after all tests of length 9 have been created 33
4.13 TestSuite and comparison matrix 33
4.14 Loading the $coffeemachine$ implementation 34
4.15 Saving the most resilient mutants 34
4.16 Results of the second comparison 35
4.17 Results of the experiment 35

Resumen

Una máquina de estados finita (FSM) es un modelo matemático de computación definido por una lista finita de estados, datos de entrada y datos de salida, en el cual los datos de salida no están determinados únivocamente por el último dato de entrada recibido sino también por el estado actual y, por ello, por los datos de entrada previos. Un mutante de una máquina de estados finita es otra máquina de estados finita obtenida mutando la primera. La mutación puede consistir en alterar la respuesta a un dato de entrada, tanto modificando el estado al que la máquina se desplaza como el dato de salida que genera, o en crear un nuevo estado y sus transiciones correspondientes. Un test es una secuencia de datos de entrada con sus correspondientes datos de salida.

El objetivo principal de este proyecto es desarrollar un sistema que genere mutantes de una FSM y les aplique una serie de tests para evaluar su efectividad a la hora de distinguir la FSM original de sus mutantes.

Palabras clave: Máquinas de estados finitas, prueba de mutaciones, desarrollo de herramientas, conformidad.

Abstract

A finite state machine (FSM) is a mathematical model of computation defined by a finite list of states, inputs and outputs, in which outputs are not only determined by the last input but also by the current state, and so by past inputs. A mutant of a finite state machine is another finite state machine obtained by mutating the first machine. This mutation can consist in changing response of the machine to an input, be it the output or the state to which the machine transitions, or in adding a new state and its corresponding transitions. A test is a sequence of inputs with its corresponding outputs.

The main goal of this project is to develop a system that generates mutations of an FSM and applies a series of tests to evaluate their effectiveness at distinguishing the original FSM from its mutants.

Keywords: Finite state machines, mutation testing, development of tools, conformance.

Chapter 1

Introduction

In this chapter we will give an informal presentation of the core concepts referenced throughout this document and we will review previous literature about said core concepts. In addition, we will go over the objectives of this project and how we intended to achieve them.

1.1 Finite state machines, mutants and tests

Finite state machines, in the following FSM, are an automata-like formalism with a finite amount of states. We can move from one state to another by providing an input to the machine. The machine will possibly change its state and produce an output process.

A *mutant* is an FSM obtained by modifying another FSM. This modification can take many forms, be it the addition of a new state or changing how the machine reacts to an input when in a specific state. The main use of mutants is to be the basis of *mutation testing*, as mutants serve to be an approximation of the errors made by programmers.

A *test case* is a sequence of inputs, to be provided to a system, and information about expected/unexpected outputs after the application of each input. Test cases allow us to feed the same inputs to a mutant and to the original machine so that we can conclude that both systems are not equivalent if the outputs differ.

The main objective of our project consists in developing a system that serves as a tool for a generalized study of mutation testing in FSMs. Our system will give users the ability to apply mutation testing to either their own FSMs and test cases or ones created by the system itself.

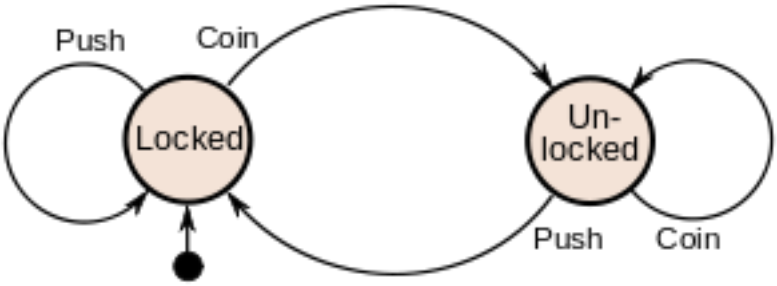


Figure 1.1: State diagram for a turnstile

Source: https://en.wikipedia.org/wiki/Finite-state_machine

1.2 Previous Work

Automata Theory established itself as a theoretical branch of Computer Science during the 20th Century when mathematicians began developing machines which completed calculations more quickly and reliably. The name itself denotes automatic processes carrying out specific tasks. Thanks to the use of automata, computer scientists are able to better understand how machines compute functions, solve problems and what it means for a function to be computable or for a question to be decidable [1].

The first researchers to give birth to a finite-sate machine include a team of biologists, psychologists, mathematicians, engineers and some of the first computer scientists. Two neurophysiologists, Warren McCulloch and Walter Pitts, were the first to present a description of a finite automata in 1943 in their paper "A Logical Calculus Immanent in Nervous Activity" [2]. In later years, two computer scientists, G.H. Mealy and E.F. Moore, generalized the theory to new machines, the Mealy machine and the Moore machine. The Mealy machine determines its outputs through its current state and the input; a Moore machine determines its output through the current state alone [3, 4].

Work on automata and automata learning is still being developed. An automata wiki has been made to collect benchmarks and frameworks from different automata. We have obtained the text representation and framework of the FSMs we will develop in this project from said wiki [5].

Mutation testing originated in the 1970s and initially considered imperative programming languages like FORTRAN [6, 7]. The first mutation testing tool was Mothra, which operated on FORTRAN programs [8, 9]. This led to Mothra and its mutation operators acting as a blueprint for most work on mutation testing. More recent work has investigated

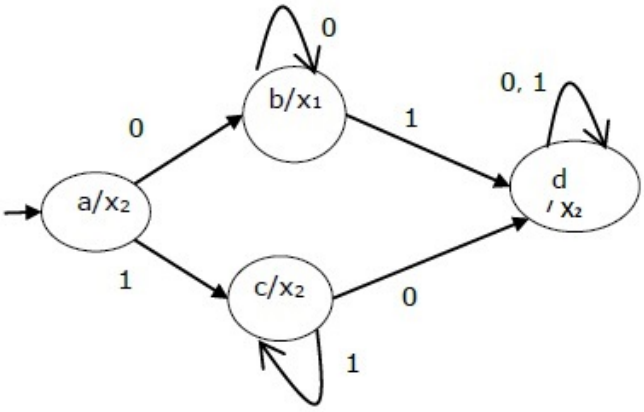


Figure 1.2: Moore machine

Source: https://www.tutorialspoint.com/automata_theory/moore_and_mealy_machines.htm

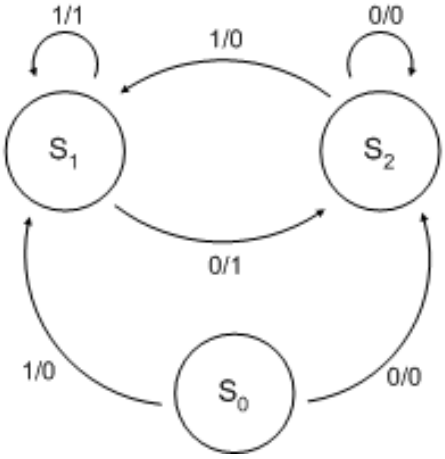


Figure 1.3: Mealy machine

Source: https://es.wikipedia.org/wiki/Máquina_de_Mealy

object-oriented programming languages [10, 11]. There has also been interest shown in mutating a model of the system under test rather than the source code [12].

The logic behind the use of mutation testing is based on the assumption that, because mutants are small alterations of the original system, a test suite that kills mutants is likely to also find real faults. The reason why mutation operators make small changes is the hypothesis that states that programmers usually make small mistakes [6]. Mutation testing is still being expanded on and the topic itself is of increasing interest, as the results of

several development trend analyses clearly show [13].

1.3 Objectives

The main objective of this project was to produce a tool [1]. with which to apply test suites to sets of mutants of an FSM to find out the efficiency of different tests at killing the mutants and presenting this data in a readable manner. The system should be able to generate test suites, FSMs and mutations of said FSMs in case those were not provided by the user.

Our starting point was the Automata Wiki [2] from where we obtained the text format for our FSMs. With this our system had to be able to parse said text format into the corresponding FSM. This way we would be able to read from a text file containing an already existing FSM and create it within the system making it so FSMs with numerous states and transitions would not have to be made at the moment of executing the system. This also meant that we had to parse the machines created within an execution of the system into their corresponding text form.

In order to modify and work with FSMs in a practical manner, the system divides the machine into its components, these being its states and transitions. This way we can more easily work with each of the individual parts that make up the machine and their functionalities instead of having to make broad changes affecting the whole structure of the FSM.

The ability of the tool to work with the states and transitions of the machine in an individual level allows it to modify them while keeping the functionality of the rest of the FSM intact. This way it creates mutants by modifying a transition of the FSM or by adding a new state with its corresponding transitions connecting it to the rest of the FSM. Doing so does not modify the initial FSM as the tool mantains the multiple machines created throughout its runtime. It is this which lets it compare the various mutants to the initial FSM through mutation testing.

The system will use test cases to compare mutants and the FSM they were created from. It keeps a list of test cases that can be either created by the user or created randomly by the system following a set of guidelines set by the user. It applies these tests to both the FSM and its mutants comparing the outputs obtained and generating a matrix during the execution which shows the results of the comparisons. With this matrix the user can know at first glance which tests were better at killing the mutants and which mutants are alive.

An important feature that was not originally planned but that we have included during the development of the system is that we can still apply our mutation testing framework

[1] This tool can be found at *https : //github.com/superspike/AutomatasTesting*, instructions about how to instale it can be found at the same site

[2] https://automata.cs.ru.nl/

when working with non-deterministic FSMs. In particular, this leads to a more complicated representation of test: they cannot be sequences of inputs and outputs but must have a tree-like structure.

Finally, the user of the system has access to all the developed functionalities through an easy to use and intuitive interface.

In order to accomplish this main objective, a series of smaller objectives had to be met first:

- Flexible implementation of an FSM. In the beginning of the project we were aware of the fact that additional functionalities were likely to be added to the system. Therefore, it was a necessity that the implementation of the FSM was not overly complex and allowed for easy modifications and addition of new features.
- Implementation of test cases that facilitates measurement and comparison of the quality of a given test suite and works regardless of the type of both input and output data.
- Finding a way to represent in an easily understandable and readable manner the results obtained when comparing the quality of the tests composing a test suite to various mutants of a given FSM.
- Giving users an interface that allows them to easily understand and use the system. Different amounts of information can be provided by the user and the system will have to generate the rest of the information needed to function while following guidelines that can be set by the user.

1.4 Project structure

The plan of the project was to divide the development of the system into three main blocks. The first one corresponds to the functionality of finite state machines and its mutants, the second one considers everything related to definition and application of tests and the last one implements the representation and interface of the tool.

1.4.1 Finite state machine functionality

As stated in the forthcoming Definition 1, a Finite State Machine is defined by a finite set of states, one of them marked as the initial state, finite input and output alphabets and a set of transitions, where each transition is defined by an origin state, a destination state, the applied input and the obtained output. Because of this structure, the attributes of the machine consist of an initial state, a list of states, an input alphabet and an output alphabet. With transitions being the most complex part of the project we created multiple

classes to manage their attributes and functionalities. These classes will allow us to better work with the different states and the transitions between them.

Each *state* is univocally distinguished by an integer number, with the number corresponding to the initial state being 0. While this is enough to differentiate each state from each other it is not enough to manage its transitions. Because of this each state will also have an instance of the *TransitionFunction* class which will manage all the transitions of which the state is origin of. The attributes of each transition will be the ones stated previously, these being origin and destination states, an input and an output.

TransitionFunction is the class used to manage the transitions for each state. It consists of an integer *size* that equals the number of transitions the state is the origin of and a HashMap *map* that maps together input strings and the list of transitions for said input for that particular state.

1.4.2 Test functionality

During the early stages of the system the attributes of a test case were a sequence of inputs and its corresponding list of outputs, *inputs* and *outputs* respectively, and the FSM M on which the test would be run. These let us compare the outputs obtained by different test cases when applied to different mutants of an FSM.

During development the ability to apply mutation testing to non-deterministic FSMs was added to the system. With this change one input could have more than one possible output corresponding to it. As a direct consequence the existing structure of a test case was no longer valid and a more complex one was needed. To address this each input now had a corresponding list of outputs instead of only one output. This change allowed us to process all the possible outcomes caused by an input and gave tests a tree structure where every input generates a branch for each of the outcomes it can trigger.

To make the comparisons between test cases easier and for a more coherent code we implemented the *TestComparer* class which contains the attributes and functions that allow us to compare the efficiency with which different tests kill different mutants of the machine M.

1.5 Interface functionality

Users should be able to use the system with as little necessary existing data as possible but also not be limited to the amount of data they can feed the system as to not limit its functionality. This breeds the necessity of an interface that allows users to input any existing data they may have into the system but also allows them to request the system to generate the rest of the data necessary for the execution.

The interface will give users the ability to determine the different characteristics of the FSM and its mutations or let the tool generate them given some guidelines. The creation

of tests follows the same path where users can let the tool generate random tests following limitations given by the user or they can give the tool specific tests to use.

Chapter 2

Formal definitions

In this section we will introduce the formalisms to specify finite state machines, tests and mutants, which will all be fundamental to develop our system as they are our main objects of interest.

Definition 1. A *Finite State Machine*, FSM, is a tuple $M = (S, s_0, I, O, \mathcal{T})$, where S is a finite set of states, $s_0 \in S$ is the initial state, I is the finite input alphabet, O is the finite output alphabet and $\mathcal{T} \subseteq S \times S \times I \times O$ is the set of transitions.

We say that an FSM M is *deterministic* if there do not exist two different transitions $(s, s_1, i, o), (s, s_2, i', o') \in \mathcal{T}$ such that $i = i'$. □

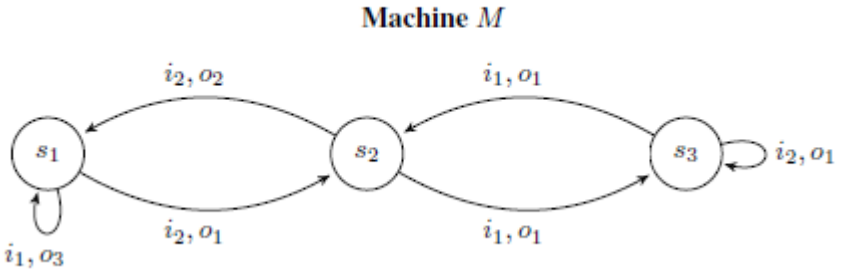


Figure 2.1: Example of an FSM

In Figure 2.1 we show a graphical representation of an FSM. For example, the tuple (s_1, s_2, i_2, o_1) is one of its transitions.

We say that an implementation *Imp conforms* to an FSM M if for all possible evolution of M the outputs that the implementation Imp may perform after a given input are a subset of those of M.

Definition 2. A *first-order mutant* Mu of an FSM M is a variation of M obtained by modifying either the output or the destination state of a transition $trans = (s, s', i, o) \in \mathcal{T}$, or by adding a new state s'_2 to S and a new set of transitions corresponding to s'_2. If Mu is functionally equivalent to M then we say that Mu is an *equivalent mutant.* □

First-order mutants imitate the effects that an error would have over M. Going back to the FSM M showed in Figure 2.1 and its transition (s_1, s_2, i_2, o_1), we obtain a *mutant* Mu by mutating the output of the transition to (s_1, s_2, i_2, o_2) (see Figure 2.2).

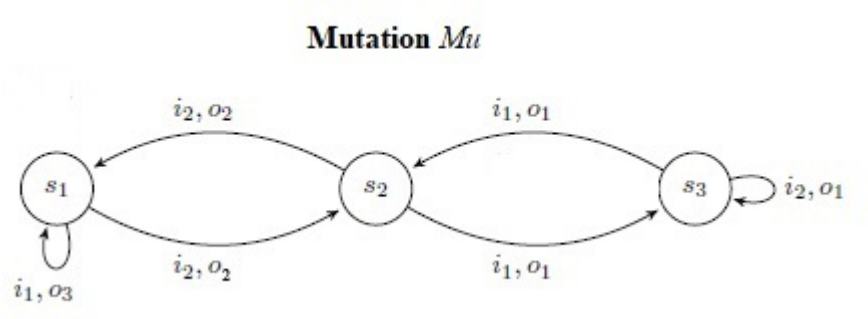


Figure 2.2: Mutation Mu of previous FSM

Definition 3. A *test case* is a tuple $t = (In, \mathcal{O}, M)$, where $M = (S, s_0, I, O, \mathcal{T})$ is an FSM, $In \in I^*$ is the list of inputs that we will apply and $\mathcal{O} \in \mathcal{P}(O)^*$ is the list of sets of outputs that we will analyze after the application of each input. Let Out_n be the nth set of $\mathcal{O}$ and i_n the nth input of In. Then the elements $o_i \in Out_n$ are the outputs corresponding to the transitions $trans \in \mathcal{T}$ where $trans = (s, s', i_n, o_i)$. A set of tests $T = \{t_1, ..., t_n\}$ is a *test suite.*

Let $\mathcal{M}$ be a set of mutants of M. We say that a test case $t = (In, \mathcal{O}, M) \in T$ *kills* the mutant $m \in \mathcal{M}$ if m and M produce different sets of outputs when receiving In as input. In this case we also say that T kills m. If a mutant has not been killed then it is *alive*

Let $\mathcal{M}$ be a set of mutants of M and $\mathcal{T}$ be a set of test cases. The *mutation score* of a test case is the percentage of mutants belonging to $\mathcal{M}$ that it kills. Analogously, the *resilience* of a mutant belonging to $\mathcal{M}$ is given by the percentage of tests of $\mathcal{T}$ that it survives. □

As an example, let us consider the sequence of inputs $i = i_1, i_2, i_2$. When the FSM M depicted in Figure 2.1 is given i as input, it generates the sequence of sets outputs $Out = \{o_3\}, \{o_1\}, \{o_2\}$. The test case $t = (i, Out)$ *kills* the mutant Mu of M given in Figure 2.2 since the sequence the mutant generates is $Out_m = \{o_3\}, \{o_2\}, \{o_2\}$, which is different to the output set sequence Out.

Due to non-determinism given a set of inputs In and an FSM M we can have multiple transitions for each input $i \in In$ and consequently multiple outputs o and multiple destination states s'. Therefore, after applying the first input i_0 from In we might have to apply the second input i_1 to more than one state s. To follow this trace of outputs and states we use tests with a tree structure as we can see in Figure 2.3. This lets us "go down the tree" alternating inputs received and outputs obtained getting a representation of the outputs obtained when applying a given test case t to an FSM M. We can then compare it to the one obtained from applying the same test case to a mutant mu of M killing it if there are differences in the outputs obtained after each input.

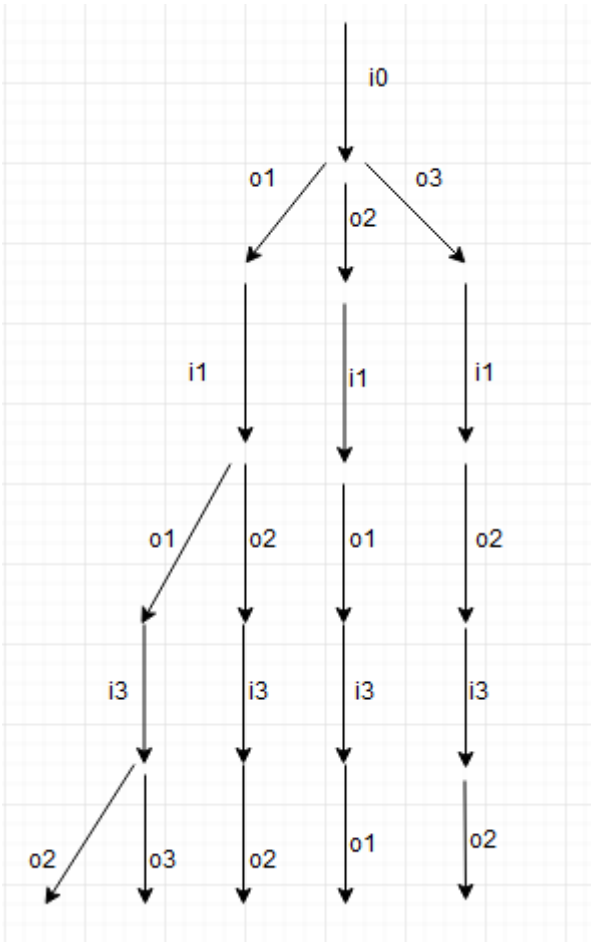


Figure 2.3: Tree test

Chapter 3

Development of the system

In this chapter we present the main ideas, concepts and decisions related to the development of the system. Throughout the development process, different classes were implemented and modified, with new functionalities added to them and their structures and attributes changed frequently. We will review the classes that make up the system and their main functions, along with any important changes made to them during development together with the reasoning behind said design decisions. The code for some of the most important and interesting functions can be found in the appendix A.

3.1 Mealy Machine Package

In this package we have all the classes related to the construction and management of FSMs, their structure and their functionality.

3.1.1 MealyMachine.java

The *MealyMachine* class is the base of the system as it is the class that will construct the FSMs and their mutants. As such, the class has the attributes needed to store the necessary information to define an FSM. As presented in Definition 1 these are an initial state *initialState*, which is 0 for all the FSMs, a list of all the states of the FSM *stateList*, an input alphabet *inAlph*, an output alphabet *outAlph* and a set of transitions $\mathcal{T}$. This last attribute will be shown in the upcoming Section 3.1.3 as it is implemented in the *State* class.

Due to the different ways in which a user can provide the required information to the system, we had to implement various constructors of the *MealyMachine* class since the early phases of the project. The first ones were the more simple constructors, which allowed us to execute the system and check for possible errors in it. One of these was a constructor

such that given the number of states and the length *AlphLength* of the alphabet, it would generate an FSM with one transition for each input in every state with a random destination state and output (that is, initially, only *complete* FSMs were produced). The other one was a constructor such that given the number of states and *AlphLength*, it asked the user to input through text the transitions of the FSM. This allowed us to create FSMs with specific characteristics and check whether the system was working properly or not.

Later into development of the system we extended its features so that randomly generated FSMs did not have to be complete anymore, that is, every state no longer needed a transition for each possible input. We did this by adding a new argument to the constructor where we could specify the chance that a state would have a transition for each input. The most important constructor is the one that generates a text representation of the FSM following the format mentioned in Section 1.3.

At this point there was no distinction between input and output alphabets as both were the same sequence of integer numbers ranging from 0 to *AlphLength-1*. We recognized that users would have FSMs where inputs or outputs were not integer numbers. Our first attempt to address this was to implement the input and output alphabets as two separated HashMaps, which would map the inputs or outputs as strings together with a unique integer number. This allowed the system to still treat alphabets as integer numbers while allowing users to have inputs and outputs of any kind. This decision, however, generated a new problem. In different situations we would want to search for a specific integer, which were the *keys* of the maps, while sometimes we wanted to search for the specific string inputted by the user. This was not possible due to limitations of the HashMap data structure. To fix this we implemented a *Translator* class which kept two HashMaps for each alphabet: one where the strings were the *keys* of the map and one where they were the *values*. In addition, it did the necessary operations with the maps to obtain the inputs and outputs in the desired format for each situation. Finally, we implemented alphabets as lists of strings as the search of a specific element within a list is more efficient than the same operation in a HashMap.

For the creation of mutants we needed two main functions. One to clone an FSM, as we still needed the initial FSM in order to create more mutants and to later apply mutation testing, and one to mutate the cloned machine. This last function turned into three as we had to create three different types of mutants. For two of them we needed to mutate a transition of the machine, be it its output or destination state. To do this we selected at random one state of the FSM followed by selecting a random transition of said state. After this, we randomly chose an output from the output alphabet or a state from the list of states of the FSM and changed the parameters of the selected transition to reflect its new output or destination state. For the third type of mutation we had to add a new state to the FSM and create its transitions connecting it to the rest of the FSM with a chance

determined by the user for the state to have a transition tied to any particular input. To avoid a situation in which for every input there was no transition generated making it so the new state was not connected to the rest of the FSM we first select a random state from the machine and create a transition between the selected state and the randomly generated one. Let us emphasize that this last type of mutation does not appear in the previous work on mutation of FSMs reviewed to prepare this project. The code to apply these mutations can be seen in Appendix A.1

Further down into development we decided to add to our system the ability to manage non-deterministic FSMs. To determine if an FSM is non-deterministic we added two new attributes to *Mealymachine*: an integer number *noDet* which determines the maximum amount of transitions a state can have for any particular input, and a boolean *det* which is false if *noDet*> 1. Since the *MealyMachine* class stores the list of states of the FSM and its input and output alphabets, we decided to implement most of the necessary functions within it, as we can access the most information from this class. The main challenge that arose with non-determinism was that after an input, our FSM could now be at multiple different states, and so we would have to apply the next input to all of those states. To deal with this problem we implemented *TreeAux.java*. *TreeAux* stores the state s and input i which started the transition with s as destination state of one node of the tree and a list of all its children. To generate its children, an instance of *TreeAux* calls a function from *MealyMachine* which returns a HashSet of pairs of the destination states reachable from s and the inputs which start the corresponding transitions. The *TreeAux* attributes and function to create its children can be found in Appendix A.2

The *TreeAux* class allows us to easily store the sets of outputs and destination states generated when applying test cases to a non-deterministic FSM.

3.1.2 Transition.java

Before going over the implementation of the states of the FSM and how each state manages its transitions we will show the implementation of transitions themselves. This is done in the *Transition.java* class.

A state transition has four main elements: the state in which the FSM is before the transition happens *orig*, the input that triggers the transition *input*, the state in which the FSM is after the transition *dest* and the output that results of it *output*. This lead us to implement transitions as objects with those four attributes.

Transitions are objects that provide necessary information for the rest of the system to properly function. As such, they do not have many functions themselves besides the necessary ones to construct them and modify their attributes. The other functions implemented in *Transition* allow us to clone a transition, to check whether two transitions are equal by comparing each one of their attributes, and to parse a transition from its text format into

its four attributes or vice versa. This last one is particularly useful as it allows for the parsing of every transition of a state to be done by recursively calling this function for each of the transitions of a state.

3.1.3 State.java

In a previous section we showed the implementation of all the components of an FSM, as we described them in Definition 1, except for its transitions. Transitions are the way an FSM changes its state to another one. It proved easier and more organized to implement the functionalities regarding transitions in the *State* class and have each state contain the information of the transitions of which it is the origin state. Once we implemented transitions and considering how states of the FSM react to receiving inputs, instances of the *State* class needed to be univocally distinguishable and store all the transitions of which they were the origin state. These two elements were implemented as an Integer acting as the state ID *id* and an ArrayList of instances of the *Transition* class.

Initially, most of the functions in *State* were similar to those in *Transition*, as states were mostly instances of a class which provided a way to structure information about the FSM and did not have many functionalities of their own. Most of the functions were constructors of the class or functions to alter or obtain the attributes of a state. There was, however, one main exception which allowed for the whole system to work. Given the fact that transitions are stored within each state, we implemented the *step* function in the *State* class. Originally, *step* received an input as an argument and returned the transition which occurred when that instance of *State* received said input. This allowed the system to apply the first input to the starting state and obtain the corresponding output and the state to which apply the next input and to follow this process until all inputs were applied.

As previously mentioned, during development we first made it so that FSMs did not have to be complete and later on added non-deterministic functionality to them. Because of this we had to implement additional functions within the *State* class. Non-completeness meant that states no longer needed to have a transition for each input, as such we added functions to check which inputs were accepted by a state. When we implemented non-determinism, a state could have more than one transition with the same input. This meant that while we could still store the transitions of a state in a list of transitions, it was no longer practical to do so since it was in our best interest to be able to easily access all the transitions tied to the same input. As a solution, we implemented a new class *TransitionFunction*.

3.1.4 TransitionFunction.java

With the implementation of *TransitionFunction* we had a class fully devoted to managing the transitions of a state.

In this situation we do not need a list of transitions for each state but for each input that a state could receive. Therefore, we added a HashMap *mapa* as an attribute to *TransitionFunction*. This way we could map together the inputs that the state could receive and the list of transitions triggered by said input. Thanks to the existing functions of a HashMap we could also easily access and modify the map. Another attribute of the class was an integer *nodet*, which represents the maximum amount of transitions that can have the same input as the input.

Through *TransitionFunction* we could now receive a list of all the possible outputs that we could obtain after applying an input to a state. In the same way, we could generate a list with all the possible states the FSM could be at after receiving an input. Thanks to this possibility we could implement the *setStatesI* function covered in Appendix A.3. This function takes an input i as an argument and returns a set of pairs with the destination states and outputs of all the transitions in the state with i as the input. This function allows the system to keep track of the multiple branches a non-deterministic FSM can take and the outputs in can generate when applying a set of inputs to it. This lets us compare the behaviour of different non-deterministic FSMs to the same set of inputs and, in particular, to apply mutation testing by comparing an FSM and its mutants.

3.2 Test Suite Package

Having already implemented FSMs and their ability to receive inputs and generate outputs, the next step was to implement tests as an object we could work with, instead of treating them as non-connected sequences of inputs and outputs, an approach that only works if we restrict ourselves to test that will be applied to deterministic FSMs.

3.2.1 Test.java

A test case is, at its base, a sequence of inputs which will be applied to an FSM and will encapsulate information to decide whether the observed outputs are expected or not. At each stage, the observed output will also guide the next step of the application of the test. Because of this, the three attributes we initially decided to use in order to define a test were two ArrayLists of Strings, one for inputs and one for outputs, and the FSM in which the test would be run unless specified otherwise. While this allowed for mutation testing with deterministic FSMs it would prove unable to properly function with non-deterministic FSMs. The impact the addition of non-deterministic functionality to the tool had on test implementation will be covered in depth in the forthcoming Section 3.3.

Given that a user could either specify which inputs they wanted for their test case or they could specify a set of guidelines for the system to follow and create a random test case within them we implemented various constructors for *Test*. This way the system

could generate tests by taking the input directly from the user or generate sequences of a determined length of random inputs taken from the sequences that can be performed by the FSM.

Most functions implemented in this class serve to access and modify the various elements of the test case and to present its data in a more readable manner. The two exceptions being the *runTest* and the *runTestMachined* functions, which apply the sequence of inputs from the test to its machine in the case of the former, or another specified FSM in case of the latter, and obtain the sequence of outputs generated by the transitions of the FSM.

With test cases implemented and able to be run and generate the sequences of outputs we wanted to compare we now needed a class to manage said comparisons.

3.2.2 TestComparer.java

As stated at the beginning of this document, the objective of the project was to create a tool that would facilitate all the activities involved in the process of mutation testing from FSMs. For this, we still had to implement a way to compare the results obtained from applying a test to an FSM and its mutants, evaluate and present the information obtained in an understandable and logical manner.

With this in mind, we implemented the *TestComparer* class. Every instance of the class has an FSM M, the type of mutation the FSM will be subject to *mutType*, the amount of mutants of the FSM we will generate *numMuts*, a list to store said mutants *mutList*, a list of the tests that we will run on the FSM and its mutants *testList* and an ArrayList of ArrayLists of integer numbers which acts as a matrix to store the information obtained when running the tests on the FSMs *compMatrix*. When constructing an instance of *TestComparer* we take as inputs the initial FSM, the type and number of mutants that will be generated and the probability for each input to trigger a transition for the new state generated in case the mutants are generated by creating a new state and not by altering an already existing one. With these attributes set, we create the mutants and store them in *mutList*. A function *addTest* was implemented to add tests which we will later apply to the FSM and its mutants.

In order to apply mutation testing we implemented two main functions. The first function, *compare*, takes two FSMs and a test as input and applies the test to both machines obtaining the outputs generated by each of them. As a following step, it compares the sequences of outputs generated and returns how many inputs it took for the FSMs to generate a different output or 0 if no different output was generated. This allows us to not only know if a test kills the mutant but also tells us how many inputs it had to go through in order to do so. This serves to see which tests are more efficient at killing the mutants. The second function, *runAll*, iterates through all the test cases in *testList* and through the mutants in *mutList*, calling the *compare* function on each iteration using as arguments the

initial FSM, the mutant and the test. This way we obtain for each test the comparison of applying it to the initial FSM and to each of the mutants, generating a list of the numbers returned by the *compare* function. After iterating through all the tests, we obtain a matrix where each row represents a test and each column represents a mutant, so that the value in the position $[i, j]$ of the matrix is the amount of inputs it took for test i to kill mutant j, or 0 if mutant j is alive. This first implementation of mutation testing can be found in Appendix A.4.

3.3 Adaptation of tests for non-determinism

The addition of non-determinism to FSMs meant that new functions and classes needed to be implemented in the system. This was no different in the case of tests. An FSM could now generate more than one output when receiving a single input. This meant that our implementation of tests was not valid to cope with non-determinism and neither was our implementation of *TestComparer*. Before going over how we implemented non-determinism, we will first describe a class that we had to implement for it to happen.

3.3.1 Tree.java

In order to adapt tests to the situation where FSMs may have more than one transition for each input, we have to consider that each input needs to take into account a list of outputs instead of only one output. This also means that we cannot precisely know at which state the FSM is after a sequence of inputs: we will have to manage a list of states at which it can be. The easiest way to track this, and to allow for the implementations of non-determinism to work, is the use of a *tree* structure. The core functions of *Tree* explained in this section will be found in Appendix A.5. Each *Tree* instance has an integer *state* univocally identifying a state of the FSM, a string *input* which determines the input of the transition that had *state* as its destination state, and a list of more instances of *Tree*, which are the children of the initial *Tree hijos* paired with the outputs obtained when going to each of them through their corresponding transitions.

In order to generate the children of an instance of *Tree*, we take an FSM and an input as arguments and through the use of the function *setStatesI*, introduced in Section 3.1.4, we generate a set of pairs of the destination states and corresponding outputs of the transitions that have as origin state the attribute *state* of our tree and as input the input received as an argument. Iterating through this set, we can generate the children of our tree by creating pairs, taking each of the destination states in the set returned by *setStatesI* and the input taken as an argument and adding them to *hijos* together with the outputs returned by *setStatesI*.

To apply mutation testing we have to obtain the outputs of a test. For this we implemented the *outputs* function. This function recursively goes down the tree obtaining the possible outputs generated for each input and accessing the information stored in each of the instances of *Tree*. Finally, it returns an ArrayList of ArrayLists of outputs where the *nth* ArrayList of outputs represents the set of outputs corresponding to the *nth* input of the test.

With this structure, and its functions, we can now apply a sequence of inputs to non-deterministic FSMs and construct the different branches the execution can take. This approach allows us to store the list of various possible outputs after each input and compare them between different FSMs.

3.3.2 NonDeterministicTest.java

The implementation of *NonDeterministicTest* ended up being mostly identical to its corresponding implementation of *Test*. One of the main differences was the outputs being an ArrayList of ArrayLists of strings instead of an ArrayList of strings, since processing the response to an input might now involve various outputs and therefore we had to have a list of outputs for each corresponding input. The other one was the function *runTest*. Due to non-determinism, we could no longer just apply the first input of the test to the starting state of the FSM and obtain a second state to which to apply the second input and by repeating this process applying all the inputs of the test case obtaining all the corresponding outputs. We now have to take into account all the possible transitions given an input. In order to implement this idea, we used the the *Tree* structure introduced in the previous section. Thus, we are able to construct the tree representing the different branches of the execution and to obtain the corresponding list of sets of outputs that must be considered in the test.

3.3.3 NonDeterministicTestComparer.java

Similarly to *NonDeterministicTest*, *NonDeterministicTestComparer* was not changed much from its deterministic version. Its attributes stayed the same except for changing the list of tests to a list of *non-deterministic* tests. The most important difference was the manner in which we compared the results obtained when applying a test to the initial FSM and to one of its mutants. Now each mutant of the FSM and the FMS itself generated a list of outputs for each input of the test. We can say that a mutant *mu* exhibited a different behaviour from the FSM M if any of the lists of outputs *mu* generated is not generated by M. In other words, a mutant is now killed if it generates a list of outputs not generated by the initial FSM. As lists can store different instances of the same output we can observe differences in both the content and the size of the lists. This is useful in case a mutation

of the FSM would make it so after a sequence of inputs the mutant could end in a state with no transition for the upcoming input. In this case no transition would happen for that specific branch of the mutant execution and in consequence no output would be generated either. This leads to the list of outputs generated for the input to have one less output which lets the system know there was a difference in the outputs generated by the mutant and the FSM.

Representation of the effectiveness of each test at killing mutants was also modified to where now the values of the comparison matrix are either 1 if the mutant is killed by the test or 0 if not.

These changes to the implementation due to the addition of non-determinism can be found in Appendix A.6 and Appendix A.7.

3.4 Visual Interface Package

The interface had to give the user the ability to provide the system with different amounts of information and ask the system to generate results, following a set of guidelines set by the user. In order to achieve this goal and to avoid visual clutter, we decided to separate the different functionalities in three different screens: *FirstScreen*, *CreateManualAutomatonScreen* and *TestingScreen*.

First, we needed a class to include and manage said screens. We implemented the *Container* class. Its attributes are the three screens that we previously mentioned, *FirstScreen* fs, *CreateManualAutomatonScreen cma* and *TestingScreen* ts, and the *MealyMachine mm* and *NonDeterministicTestComparer tc* that will carry on the functionalities of the system explained in previous sections.

3.4.1 FirstScreen.java

FirstScreen is the first screen the users see when using our system. It allows them to set the number of states of the FSM, the approximate amount of transitions the FSM will have and both the input and output alphabets. Once these values are set, the user can either let the system randomly generate the transitions of the FSM or be taken to a second screen and manually construct them. These two options can be chosen by pressing the buttons *Create Random automaton* and *Create Manual automaton*, respectively. Once the FSM is created, it will be represented in text form in the *Dot Automata* text section. The *Begin test* button will take the user to a new screen where they can construct and set the parameters for the mutants and the test suite.

We implemented the *AlphList* class to receive the information provided by the user regarding the input and output alphabets and transform it into a format that the system can work with. In this class we also take into account that we have to process two alphabets

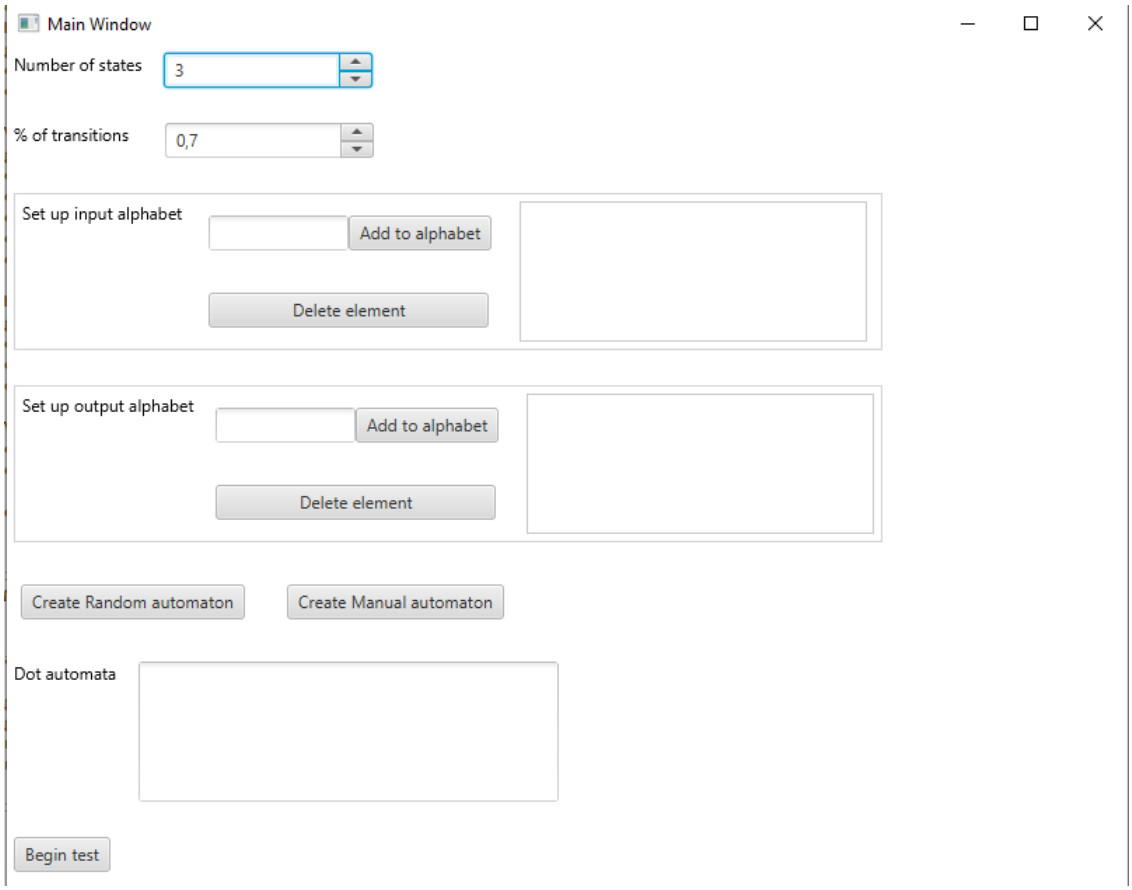


Figure 3.1: FirstScreen

and were particularly careful to avoid unnecessary and redundant code. This decision lets us to store either the input or output alphabet into a list of strings, the format in which the rest of the system interprets both alphabets, and return it to other functions.

3.4.2 CreateManualAutomaton.java

As mentioned in Section 3.4.1, once the user has set the values for the input and output alphabets, they can choose to manually introduce the transitions of the FSM by pressing the *Create Manual automaton* button and being taken to the corresponding screen.

A transition has four parameters that the user can give value to: origin state, destination state, input that triggers the transition and output generated by the transition. The *CreateManualAutomaton* screen had to give the user the ability to set these values, see what transitions had been generated and delete generated ones in case a mistake was made. Once all transitions are available, the user can press a button to generate the FSM.

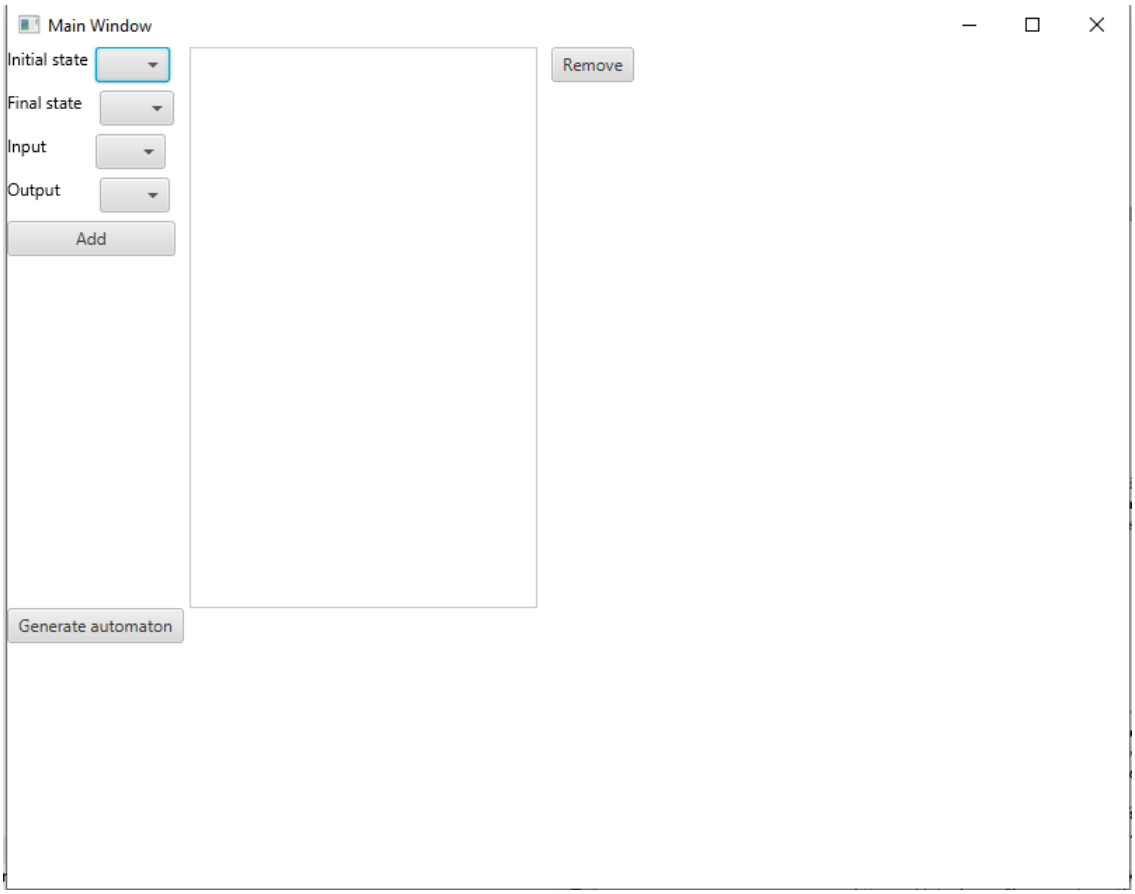


Figure 3.2: CreateManualAutomaton

The *Add* button will check the values provided for these four parameters and add a transition made of said values. *Remove* will remove the selected transition from the list or do nothing if no transition is selected. *Generate automaton* creates an FSM using as information the data input in the previous *FirstScreen* screen and the transitions created in this one. After that, through the *Container* class, it will return to *FirstScreen* with the created FSM represented in the *Dot automata* section.

3.4.3 TestingScreen.java

TestingScreen (see Figure 3.3) allows the user to provide the necessary data to construct the mutants of the FSM and the test suite that will be applied to the system and compare the efficiency of different tests applied to the various mutants of the FSM.

In the top part of this screen, users can choose with which mutants they want to work. We give them the option to choose the type of mutation and the amount of mutants. In

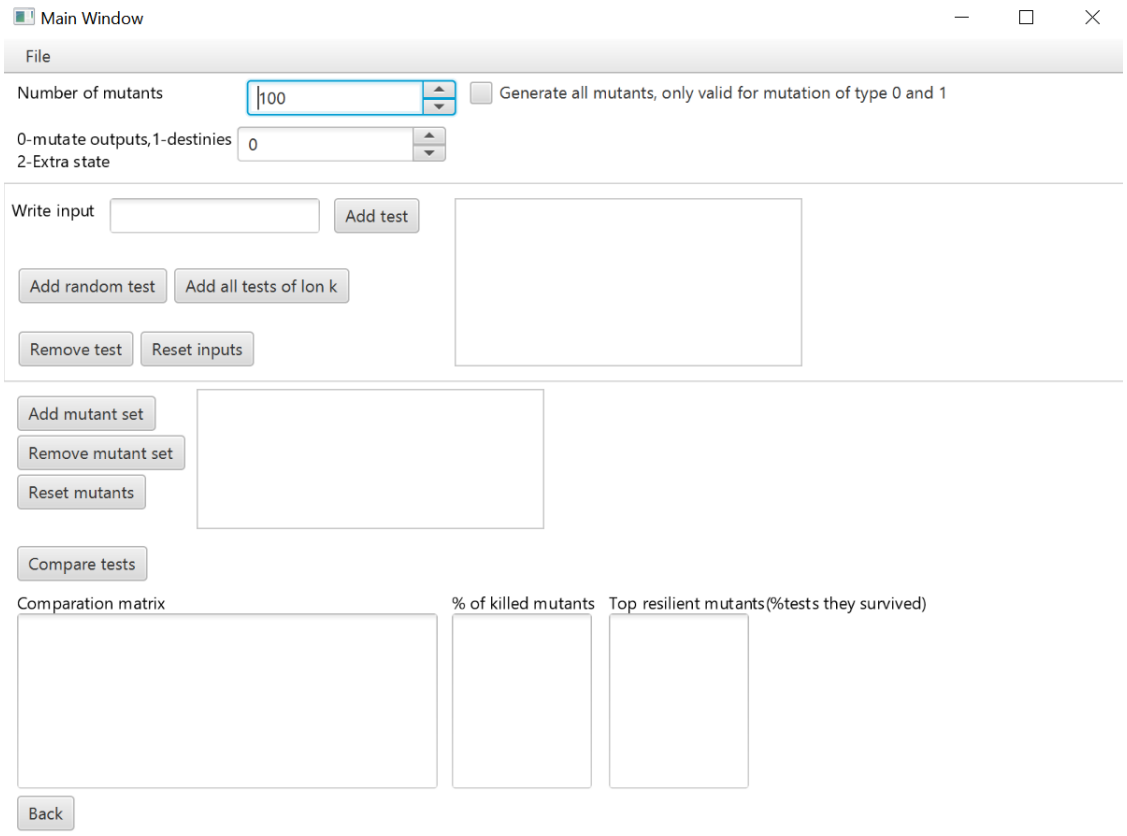
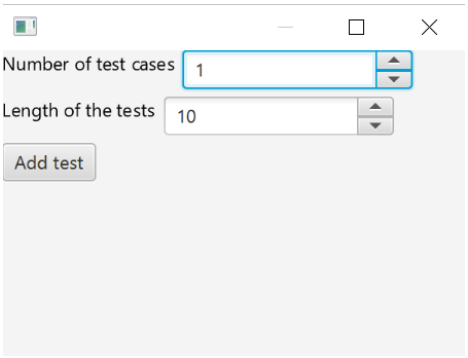


Figure 3.3: TestScreen

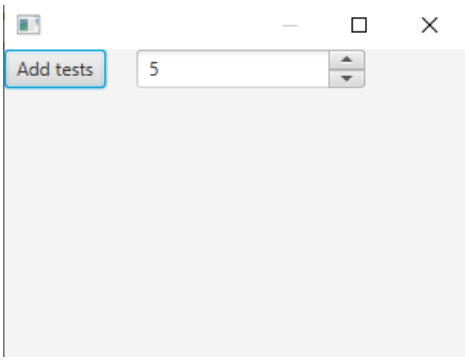
addition, we added a checkbox to give them the opportunity of generating all the mutants of a certain type. If this option is marked, then the tool will ignore the indicated amount of mutants as to avoid the generation of duplicated mutants. If the user selects to generate mutants of type 2, then we ignore this checkbox because of the high cost in memory and time of generating all the mutants of this type.

We decided, during development, that having the user to create each test case by adding each of the inputs, one by one and choosing them from the possible inputs of the alphabet, would be slow and tedious. In order to prevent this undesirable situation, we give the user the ability to write the test case as a list of inputs separated by a space, in the *Write input* text box, and to add them to the test suite via the *Add test* button. While this can lead to input errors, we trust the user will use the *Remove test* button to remove any test mistakenly constructed. The user can also click the *Add random test* button and a new screen will appear and prompt them to choose the length of the test and the amount to be

generated.



(a) Add random test



(b) Add all tests of length K

Clicking on the *Add all tests of lon k* button will create a new screen equal to the one generated through the *Add random test* button but with a different functionality. As the name of the button indicates, this screen allows the user to choose an integer number. Then, all possible combinations of inputs of the specified length that the machine can process will be added to the tests. This means that no sequence of inputs the FSM cannot follow will be generated and added to the tests. For this feature, we implemented *TreeAux.java*, a class very similar to the *Tree* class shown in Section 3.3. Let us emphasize that users have to be careful with the use of this function, because the time it takes to generate all the inputs can be very high if the ramification factor of each state is large enough.

Let us note that our tool also lets the user to select mutants from files, whose format will be specified later on. This is an important feature in order to reuse mutants generated either by other researchers or by previous uses of our tool.

Lastly, when the *Compare* button is clicked, we create an instance of the *NonDeterministicTestComparer* class with the data that the user has provided in this screen. After that, the *NonDeterministicTestComparer* generates the specified amount of mutants of the FSM, runs all the test cases on the FSM generating the corresponding outputs to then run them on the mutants and returning the comparison matrix. Once the comparison has been done, we can see the comparison matrix and if we double click on any of the test cases we can see its corresponding list of sets of outputs.

We have included a feature to show the mutation score of each test case next to them. We also show another column including information about the most resilient mutants, that is, those mutants that survive the application of the largest amount of tests.

Finally, in order to let researchers share their results on an specific machine, we added the possibility of saving and loading inputs and mutant files. This is done by clicking in the menu bar of the upper left corner and selecting the appropriate option. Next, we briefly explain the format of these files.

- Input files. Each string of inputs in a line separated by a space.
- Mutant files. We have two different formats, depending of the type of applied mutation. For mutations of type 0 and 1, each mutant is in a line containing the type of mutation, the origin of the transition that will be mutated, its destination state and the value we are changing. This last value is the new output, in the case of mutation of type 0, and the new destination state for mutations of type 1. Besides, for mutants of type 2, the file will contain the *.dot* of each mutant separated by the string ### in a different line.

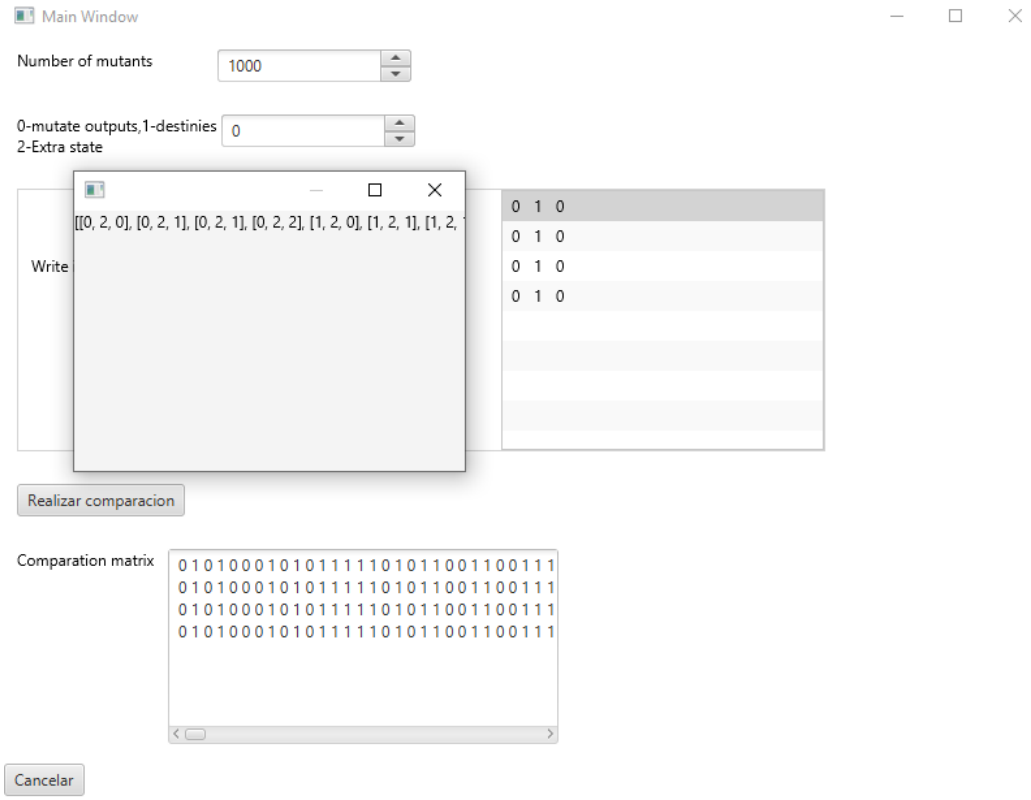


Figure 3.5: Comparison matrix and corresponding outputs

Chapter 4

Experiments

In this chapter we will show different results obtained through usage of our tool. We will consider different use cases representing the most typical applications of our tool.

4.1 Use case 1

In this example we will manually create the FSM shown in Figure 2.1, create 10 mutants mutating a random output, and 8 random tests of varying lengths.

First, we introduce the input and output alphabets and set the number of states. The percentage of transitions is not relevant in this case since we are going to be inputting the transitions manually. We remind the reader that states are always treated as integer numbers that start at 0 and count up, state 0 being the starting state (see Figure 4.1).

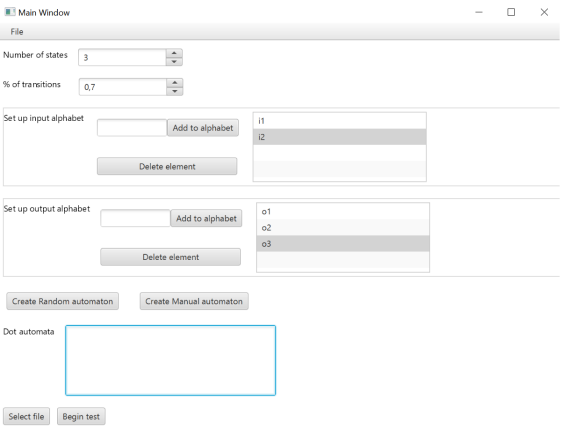


Figure 4.1: FirstScreen after alphabets are introduced

After this has been done, we click on the *CreateManualAutomaton* button and proceed

to create the transitions. Because of the number of states and alphabets input in the previous screen, our options when creating the transitions are limited to states 0, 1, and 2, inputs to $i1$ and $i2$, and outputs to $o1$, $o2$ and $o3$. This way we cannot introduce transitions that are not possible in our FSM (see Figure 4.2).

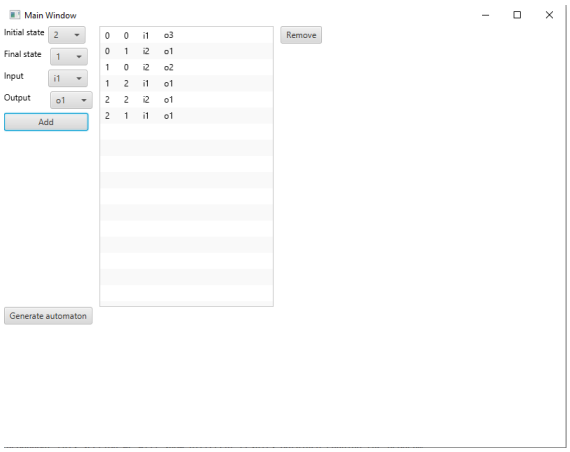


Figure 4.2: Transitions introduced in CreateManualAutomaton screen

Clicking the *Generate automaton* button will take us to the previous screen with our FSM in text form in the *Dot automata* text box (see Figure 4.3).

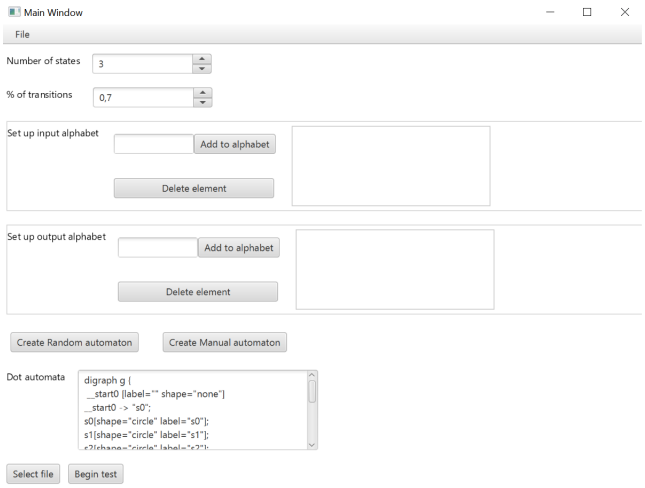


Figure 4.3: FirstScreen after the transitions have been created

With the FSM completely created we only have to create the test suite before we run the test cases on the FSM and its mutants to obtain the comparison matrix. For that, we click on the *Begin test* button, and in *TestScreen* we choose the number and type of

mutants and create 2 random tests for each length 6, 7, 8 and 9. After that, we apply the tests and obtain the comparison matrix.

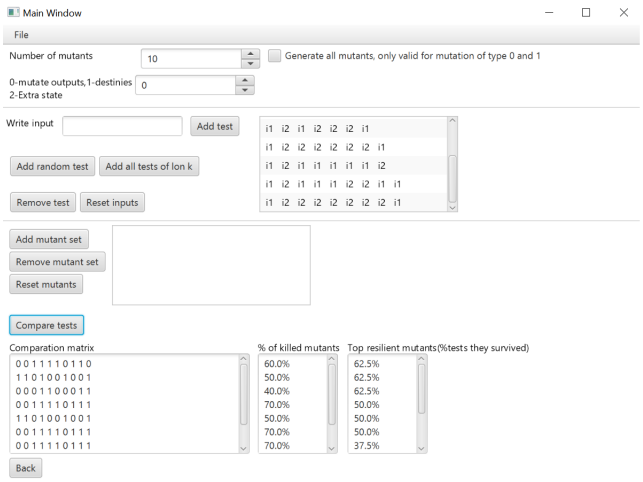


Figure 4.4: TestScreen after creating the test suite and running the tests

As was said in Sections 3.2.2 and 3.3.3, each column represents one of the mutants, each row a test case from the test suite, and the number represents whether the mutant was killed or not, with 0 indicating the latter and 1 the former (see Figure 4.4).

4.2 Use case 2

We will create a random FSM with 3 states, a 70% of transitions, 4 possible inputs and 3 possible outputs (see Figure 4.5). We can see the FSM both in text (see Figure 4.6) and diagram form (see Figure 4.7).

We create 10 mutants and follow the same process to create the test suite as we did in the previous use case, but instead of mutating the outputs, we will mutate the destination states. The final result can be seen in Figure 4.8. We can see that no mutants were killed. This makes sense as not many tests started with 1 as an input, which is the only input accepted by the FSM on its starting state. We can also see that only the tests starting with 1 were able to generate outputs when applied to the initial FSM (see Figure 4.9).

4.3 Use case 3

We will now use a modified version of the FSM that we used in our first use case (see Figure 4.10), generate all possible tests of length 9 and create 10 mutants of type 2 with a

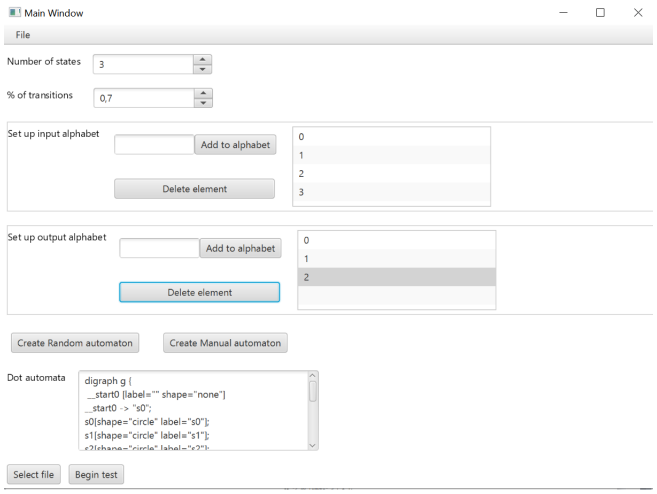


Figure 4.5: Creating a random FSM with given alphabets and states

```
 __start0 [label="" shape="none"
__start0 -> "s0";
s0[shape="circle" label="s0"];
s1[shape="circle" label="s1"];
s2[shape="circle" label="s2"];
s0 -> s1 [label="1 / 2"];
s1 -> s1 [label="1 / 1"];
s1 -> s1 [label="2 / 2"];
s1 -> s0 [label="3 / 0"];
s2 -> s2 [label="1 / 2"];
s2 -> s1 [label="2 / 0"];
s2 -> s0 [label="3 / 2"];
}
```

Figure 4.6: FSM Randomly generated as text

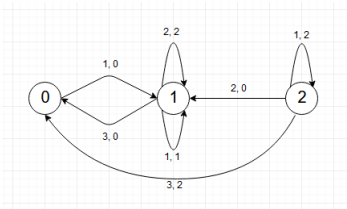


Figure 4.7: FSM Randomly generated as diagram

value of 0.7 for the percentage of transitions. As said in Section 3.1.1, this type of mutant will have an extra state and its corresponding transitions will be created.

We first set the input and output alphabets and manually create the transitions of the FSM (see Figure 4.11).

With the FSM created, we set the number and type of mutants and use the *Add all tests of lon k* button to add all possible sequences of inputs of length 9 that our FSM can process (see Figure 4.12).

Finally, we apply mutation testing and obtain the comparison matrix where we can see

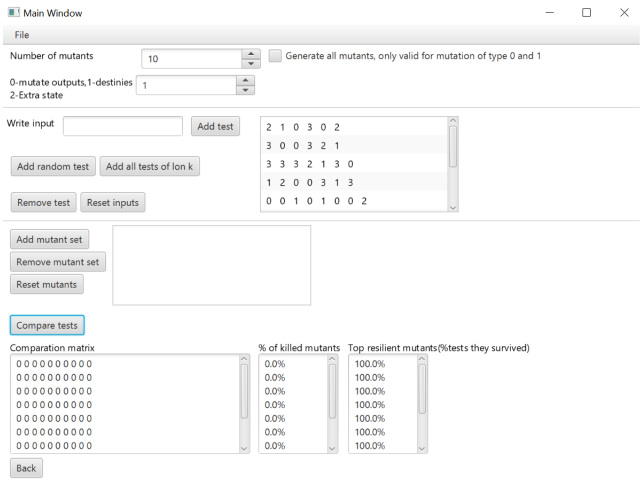


Figure 4.8: TestSuite and comparison matrix

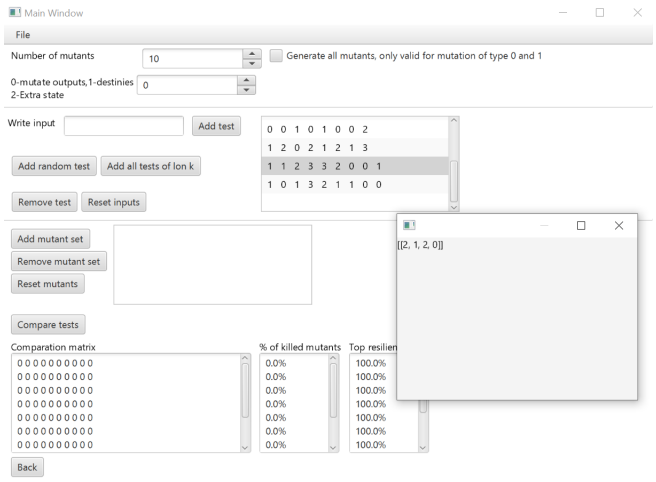


Figure 4.9: Outputs generated by FSM when applied inputs from test case

in Figure 4.13, looking only at the results obtained for the first 8 tests, that all mutants were killed.

4.4 Use case 4

Now we are going to work with a *real* system: the $coffeemachine$ from a well-known and recent benchmark [5]. Note that out tool includes, by default most of the FSMs belonging to the benchmark. In order to import that machine, we click on $File$ in $FirsScreen$ and

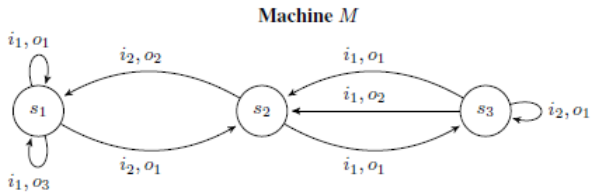


Figure 4.10: FSM that we will manually generate

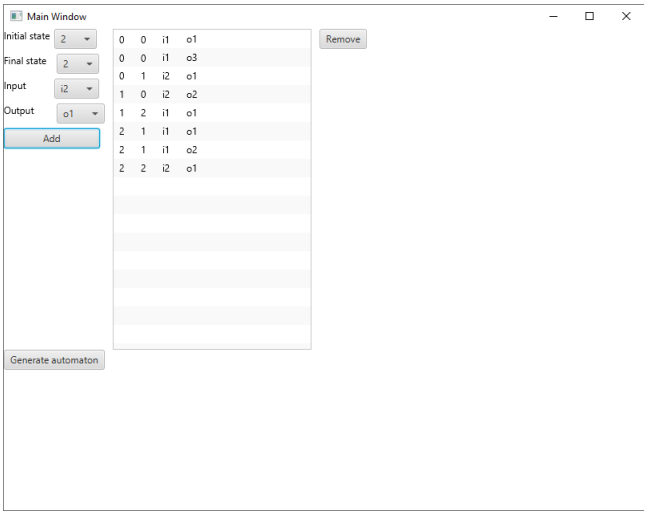


Figure 4.11: Transitions introduced in CreateManualAutomaton screen

select the desired implementation (see Figure 4.14).

Our goal will be to find test cases that kill the most resilient mutants. In order to carry out this task, we first generate all mutants of the first type. After that we generate all inputs of length 5 and compare them. We proceed to save the 20 most resilient mutants (see Figure 4.15).

Now, we add this set of mutants to the benchmark and uncheck the option of generating all mutants. Finally we proceed to compare the same tests with this subset of the mutants and save the the best 20 test cases (see Figure 4.16). We can see that neither a mutant survives all tests nor a test case kills these subsets of resilient mutants. We could find one better test if we explore the machine at a higher depth, with larger inputs.

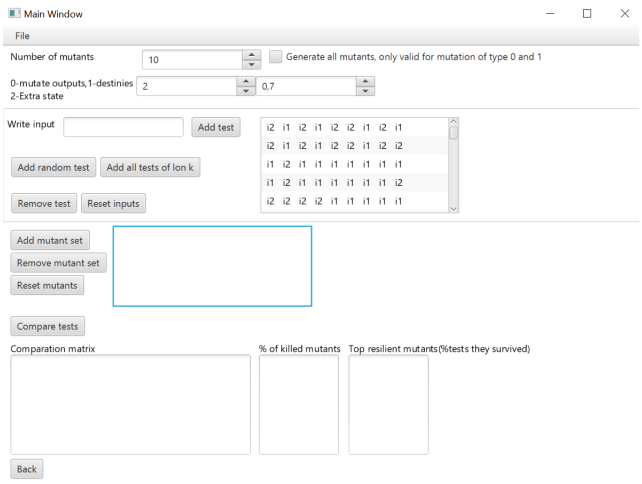


Figure 4.12: TestScreen after all tests of length 9 have been created

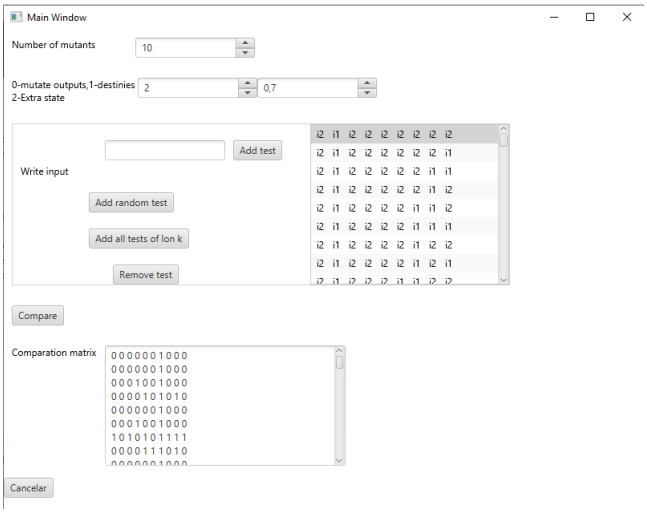


Figure 4.13: TestSuite and comparison matrix

4.5 Use case 5

We will work again with a machine of the benchmark. In this case we will choose the machine $m106$ of the $ASML$ benchmark. This is a specially large machine, with 25 states and more than 1000 transitions. In this case, our goal will be to find test cases that kill different mutants. In order to do this task we would need large inputs, so we cannot generate all the inputs of a fixed length because this function takes a lot of time if the ramification factor of each state is high enough, which is the case for this machine.

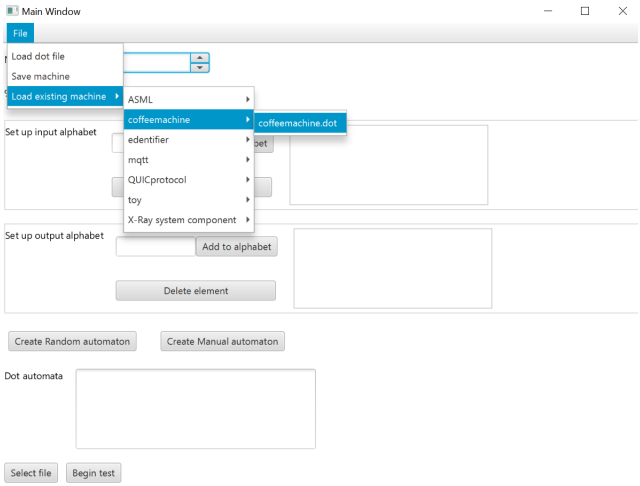


Figure 4.14: Loading the *coffeemachine* implementation

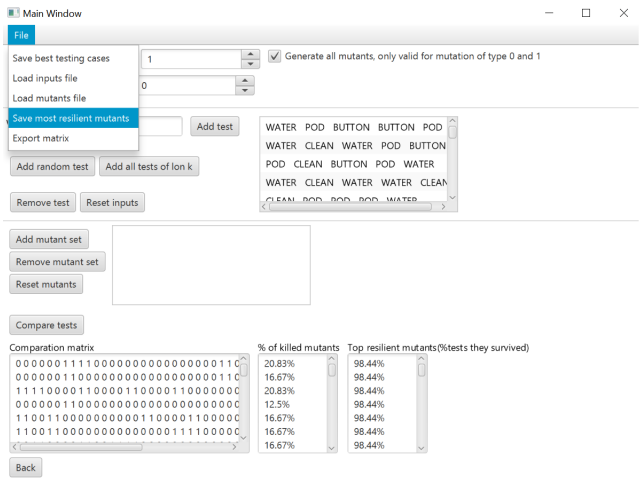


Figure 4.15: Saving the most resilient mutants

We will start, as in the previous example, importing the implementation. As we have already said, given the fact that the machine has a lot of states, we will certainly need large inputs in order to kill some mutants. but we will not be able to generate all the inputs of a certain *long* length. We will generate 1000 random input sequences of length 50, apply them to 100 mutants of the second type and compare the tests.

The obtained result is not very good, as can be seen in Figure 4.17. Our best test does not kill an important amount of mutants. Actually there are a lot of mutants that survive all tests. In order to improve this situation, we could apply to the subset of alive

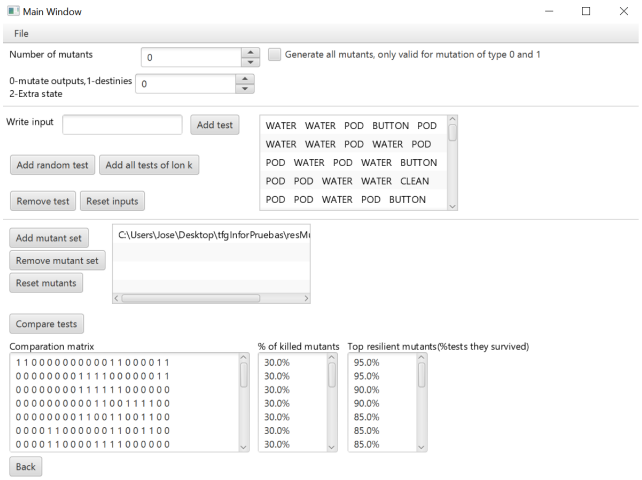


Figure 4.16: Results of the second comparison

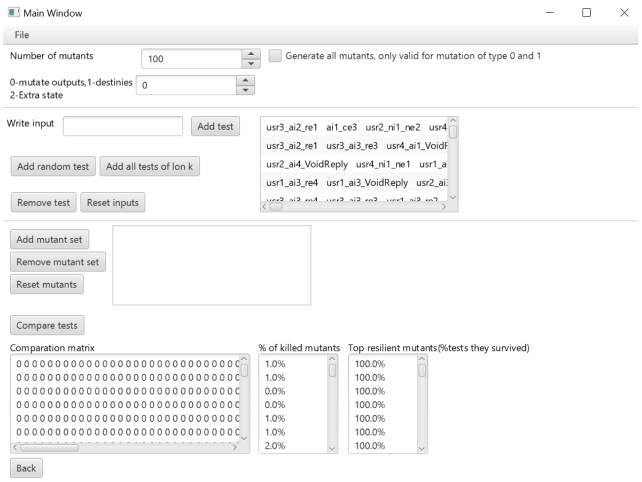


Figure 4.17: Results of the experiment

mutants this procedure again, generating other random tests and saving the ones that kill most mutants. Iterating this process we can get a set of tests that eventually will kill the initial set of mutants.

Chapter 5

Contributions

In this chapter we indicate the specific contributions of each author to the project.

5.1 Contribution of Jose María

The first thing I did was reading the papers we were given by our tutor in our first meeting. After that, we decided to use Java as there were tools developed in this language that could be useful or serve as inspiration. Then I started creating the model we would use in order to simulate a MealyMachine. This includes the clases $MealyMachine$, $Transition$ and $State$. The first version of the model was restricted because the user could only introduce integers as inputs and outputs. Therefore, changes were made in order to make it be able to use any string. This was made with use of the class $Translator$, which we used in order to map the old integers to new strings. We needed to keep the integers because they allowed us to access a random element in constant time.

After the model was finished, I started developing the structures of the tests based on the ideas our tutor gave us and the rsults presented in [12]. I started implementing the most simple tests, which were the ones using an input and an implementation using the latter as an oracle to generate the outputs. Then, with the help of Juan, I created the structure of the testing tool. This includes the way we would generate mutants, how we would compare them to the initial FSM and the way we would create the matrix used to see if a mutant was killed or not.

The next task was implementing the GUI. This was done using $javafx$. In order to use this technology, I had to install some packages in eclipse. I decided that I would divide the interface in 3 parts:

- The main screen.
- The Mealy Machine editor.

- The testing tool.

After this, we were told by our tutor that we could expand the tool in order to let the user work with non-deterministic FSMs. Fortunately, not a lot of changes were needed. We just needed to take into account that an input would not only generate a single output, but a set of them. So, I had to group them into an Array and work with it.

Another thing we needed to take into account was the way we compared the application of tests because now we were not comparing 2 single elements, but 2 sets of outputs. So, it was not so easy to give a non-binary value to a test result. After talking with our tutor, we decided to give a binary number as a result; 1 if the mutant is killed by the test or 0 otherwise.

We also implemented the suggestion of testing with all the inputs of a fixed length, which was done by an BFS (Breadth First Search) of the machine.

In our next meeting our tutor told us to add the Mealy Machines from a well-known benchmark [5] to the distribution of the tool. So, I added them to the system. I also added to the interface all the buttons dealing with saving and loading mutants and inputs, and getting statistics about them such as the mutant score for each test case and the resilience of the mutants.

I contributed to the writing and proof-reading of this document. In particular, I wrote the part of the document corresponding to use cases 4 and 5 and this section of the document where I just enumerated the main tasks that I did for the project.

5.2 Contribution of Juan

The first step I took in regards to this project was reading the papers given to us by our tutor so that I could have a better understanding of the concepts we would be using throughout the implementation of the system.

Jose María implemented the first working iteration of the tool, which we presented to our tutor in one of our first meetings. Most of the functionalities were already implemented but would be iterated upon to enhance them. This first iteration only had one alphabet made of integer numbers, which it used both for inputs and outputs.

I changed the implementation to allow for two separate input and output alphabets made of strings with the goal of giving the user more freedom in their usage of the tool. For this, I implemented both alphabets as HashMaps. This choice would allow us to map the strings that the user wanted to use for their alphabets to unique integer numbers. Thus, there was no need to change how the system dealt with inputs and outputs beyond changing how it accessed alphabets because it could still treat them as unique integers and only use their string representation when receiving information or when outputting information from and to the user.

This proved to have some problems as in different moments we wanted to access the key of the HashMap or its value with operations unique to only one of them. In order to tackle this problem, we implemented the *Translator* class which kept and managed two HashMaps for each alphabet. This let us access both the integers and the strings with all the operations available to HashMaps. However, this was not the most practical solution and Jose María changed the alphabet implementation to a list of strings and changed how the system worked with inputs and outputs to accommodate for this change.

The other main drawback of the first implementation of our system was its handling of the comparison of tests. The utilities in the *TestComparer* class where initially divided into two separate classes: *TestComparer* and *TestTester*. *TestTester* called for the creation of the mutants of the FSM and applied a test to the mutants and the FSM storing the obtained results. *TestComparer* would then compare the results obtained by each *TestTester*, one for each test case.

The main cause of this problem was that each instance of *TestTester* only managed one of the test cases. This meant that the mutants created for the application of one test were not guaranteed to be the same mutants to which we applied any other test. As a fix, I merged both classes into *TestComparer*, changing its list of instances of *TestTester* to a list of test cases and a list of mutants of the FSM. This allowed for consistency in the mutants to which all the test cases were applied and for a faithful representation of the information obtained through mutation testing in the form of a comparison matrix where each row corresponds to one of the test cases and each column corresponds to one of the mutants of the FSM.

Jose María implemented the rest of new and important functionalities such as non-determinism and the necessary changes to the system in order to handle it and the multiple modifications and additions to the interface. I helped on some small problems and increased the readability of the code.

Lastly I was the main *editor* of this document, with the collaboration of Jose María and with great help and guidance from our tutor.

Chapter 6

Conclusions and future work

We have developed a tool that allows users to apply all the steps involved in the process of mutation testing ranging from the creation of the finite state machines and its mutants to the creation and application of the test suites. We first looked at the Automata Wiki[1] for a text format to represent the FSMs the system would be working with. Our second step was the decision to separate the classes of the system into three main packages that would permit us to work with the machines, the test cases we would be applying to the machines and the interface for users to use the tool in a more organized and self contained manner.

Following the division of the code into its main packages we divided the structure of the finite state machines further down into its core components. This allowed the system to more easily modify an aspect of a finite state machine without compromising the integrity of the rest of the FSM. This meant dividing a machine into its states and transitions with each state having a list of the transitions of which it is the origin state. To further take advantage of this division, we implemented the process of an FSM receiving inputs and carrying out the corresponding transitions at the lowest level possible through the *State* and *Transition* objects. This ability to only modify an aspect of the FSM while maintaining its functionality let us implement the necessary functions to create mutants of the FSM.

During development, the scope of the system was increased to also offer the ability to manage non-determinism. While the tool was successful so far, its implementation proved working with non-determinism to be cumbersome. This led us to implement two more complex classes in *TransitionFunction* and *TreeAux* to control the transitions of each state.

Extensive work has been done in regards to test cases in the scope of FSMs [14]. Test cases were initially implemented as a list of inputs, a list of outputs and the finite state machine that would generate said outputs when receiving the list of inputs. In order to compare the outputs generated when applying the same inputs to two different FSMs, and present the information obtained in an easily understandable manner, we implemented

[1] https://automata.cs.ru.nl/

the *TestComparer* class. In the same way as it happened with our implementation of the FSMs, the addition of the non-deterministic functionality to the system meant that new classes needed to be implemented to adjust for non-determinism. We added a new type of test *NonDeterministicTest*, which instead of having a list of outputs had a list of lists of outputs. This change, together with the addition of the *Tree* class, allowed us to manage tests being applied to non-deterministic FSMs.

Lastly, for our tool to be of practical use, we implemented an understandable and easy to use interface. This gave the user the ability to input into the system varying amounts of information regarding the FSMs, the mutants and the test cases and have the system create the rest of the information following a set of guidelines. Throughout development of the system the interface was tweaked to allow for a better user experience. We repeatedly tested the tool to examine its usability and effectiveness.

The final system developed as the result of this project, and presented in this document, already includes features that were not initially planned. However, during its development we have identified several extensions that could further improve the tool. First, it would be interesting to consider other textual formats to represent FSMs. The integration of a graphical interface could facilitate the definition of FSMs. However, our experience with this type of interfaces showed that this is only useful for very small FSMs. Although completely out of the scope of this project, our system could be extended with other type of formalisms based on FSMs. For example, it would be interesting to consider FSMs with time and/or probabilistic information [14, 15].

Bibliography

[1] A. Dar Aziz, J. Cackler, and R. Yung. Basics of automata theory. `https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html`, 2004.

[2] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[3] A. Church and E.F. Moore. Gedanken-experiments on sequential machines. *J. Symbolic Logic*, 23(1):60, 1958.

[4] G.H. Mealy. A method for synthesizing sequential circuits. *The Bell System technical journal*, 34:1045–1079, 1958.

[5] F. Arts, P. van den Bos, A. Fedotov, P. Fiterau-Brostean, F. Howkar, H. Kuppens, J. Moerman, D. Neider, E. Poll, J. de Ruiter, and F. Vaandrager. Automata wiki. `https://automata.cs.ru.nl/`.

[6] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practical programmer. *IEEE Computer*, 11:31–41, 1978.

[7] J. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley and Sons: New York, 1998.

[8] R. DeMillo, D. Guindi, K. King, M.M McCracken, and A.J. Offutt. An extended overview of the Mothra software testing enviroment. In *2nd Workshop on Software Testing, Verification, and Analysis*, pages 142–151. IEEE Computer Society Press, 1988.

[9] K.N. King and A.J Offutt. A fortran language system for mutation-based software testing. *Software Practice and Experience*, 21:686–718, 1991.

[10] Y.-S. Ma, M.-J. Harrold, and Y.R. Kwon. Evaluation of mutation testing for object-oriented programs. In *28th International Conference on Software Engineering*, pages 869–872. ACM Press, 2006.

[11] A.J. Offutt, Y.-S Ma, and Y.-R Kwon. The class-level mutants of mujava. In *Workshop of Automation of Software Test*, pages 78–84. ACM Press, 2006.

[12] R.M. Hierons, M.G. Merayo, and M. Núñez. Mutation testing. In Phillip A. Laplante, editor, *Encyclopedia of Software Engineering*, pages 594–602. Taylor & Francis, 2010.

[13] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[14] M. G. Merayo, M. Núñez, and I. Rodríguez. Formal testing from timed finite state machines. *Computer Networks*, 52(2):432–460, 2008.

[15] N. López, M. Núñez, and I. Rodríguez. Specification, testing and implementation relations for symbolic-probabilistic systems. *Theoretical Computer Science*, 353(1–3):228–248, 2006.

Appendix A

Implementation of core functions

A.1 Mutation functions

```
public void mutateOutput(){
     Random r=new Random();
     int randomInt1=r.nextInt(this.stateList.size());
     while(this.stateList.get(randomInt1).getSize()==0) {
        randomInt1=r.nextInt(this.stateList.size());
     }
     int randomInt2=r.nextInt(this.stateList.get(randomInt1).getSize());
     int randomInt3=r.nextInt(this.outAlphabetSize);
     this.stateList.get(randomInt1).getTransition(randomInt2).
        setOutput(this.outAlph.get(randomInt3));
  }
  public void mutateDestiny(){
     int randomInt1=r.nextInt(this.stateList.size());
     while(this.stateList.get(randomInt1).getSize()==0) {
        randomInt1=r.nextInt(this.stateList.size());
     }
     int randomInt2=r.nextInt(this.stateList.get(randomInt1).getSize());
     int randomInt3=r.nextInt(this.stateList.size());
     this.stateList.get(randomInt1).getTransition(randomInt2).setDest(randomInt3);
  }
  public void mutateSpecial1(double prob) {
     int i=this.getNumStates();
     Integer estadoComunicador=r.nextInt(this.getNumStates());
     this.stateList.add(new State(Integer.toString(i)));

     this.stateList.get(estadoComunicador).addTransition(
          new Transition(estadoComunicador,i,
```

```
                this.inAlph.get(r.nextInt(this.getInAlphSize())),
                this.outAlph.get(r.nextInt(this.getOutAlphSize())));
      double caso;
      for(int j=0;j<this.inAlphabetSize;j++) {
         caso=r.nextDouble();
         if(caso<=prob) {
            this.stateList.get(i).addTransition(new Transition(
                  i,r.nextInt(i+1), this.inAlph.get(r.nextInt(this.inAlphabetSize)),
                  this.outAlph.get(r.nextInt(this.outAlphabetSize))));
         }
      }
   }
```

A.2 TreeAux attributes and children generation

```
public class TreeAux {
   private String inp;
   private Integer state;
   private ArrayList<TreeAux> hijos;

   public TreeAux(String a, Integer b) {
      this.inp=a;
      this.state=b;
      hijos=new ArrayList<TreeAux>();
   }

   public void addChilds(MealyMachine m) {
      //HashSet of pairs of <destination states, input>
      //with the accepted inputs from this state
      HashSet<Pair<Integer, String>> aux=m.acceptedInputs(state);
      Iterator<Pair<Integer, String>> it=aux.iterator();
      Pair<Integer, String> act;
      while(it.hasNext()) {
         act=it.next();
         hijos.add(new TreeAux(
               act.getValue(),act.getKey()
               ));
      }
   }
```

A.3 setStatesI implementation

```
public HashSet<Pair<Integer,String>> setStatesI(String input) {
     HashSet<Pair<Integer,String>> ret=new HashSet<Pair<Integer,String>>();
     if(this.contains(input)) {
        Transition caso;
        for(int j=0;j<this.mapa.get(input).size();j++) {
           caso=this.mapa.get(input).get(j);
           ret.add(new Pair<Integer, String>(caso.getDest(),caso.getOutput()));
        }
     }
     return ret;
  }
```

A.4 First implementation of mutation testing applied to deterministic FSMs

```
  public void runAll(){
     for(int i = 0; i < testList.size(); i++) {
        ArrayList<Integer> tempComps = new ArrayList<Integer>();
        testList.get(i).runTest();
        for(int j = 0; j < mutList.size(); j++) {
           tempComps.add(compare(testList.get(i), this.m, mutList.get(j)));
        }
        compMatrix.add(tempComps);
     }
  }

  public Integer compare(Test t, MealyMachine m1, MealyMachine mutation){
     ArrayList<String> m1Out = t.runTestMachined(m1);
     ArrayList<String> mutOut = t.runTestMachined(mutation);
     for(int i = 0; i< m1Out.size() && i < mutOut.size(); i++) {
        if(!m1Out.get(i).equals(mutOut.get(i))) {
           return i + 1;
        }
     }
     return 0;
  }
```

A.5 Core functions of Tree.java

```
public void generateChilds(MealyMachine m, String i) {
   this.input=i;
   HashSet<Pair<Integer,String>> aux=m.setStatesI(state, input);
    Iterator<Pair<Integer,String>> it = aux.iterator();
    Tree caso;
    hijos=new ArrayList<Pair<Tree,String>>();
   while(it.hasNext()) {
      Pair<Integer,String> act=it.next();
      caso=new Tree(act.getKey(),i);
      hijos.add(new Pair<Tree, String>((Tree) caso.clone(),act.getValue()));
   }
}
public ArrayList<ArrayList<String>> outputs() {
   ArrayList<String> outs=new ArrayList<String>();
   ArrayList<ArrayList<String>> ret=new ArrayList<ArrayList<String>>();
   ArrayList<ArrayList<String>> aux=new ArrayList<ArrayList<String>>();
   if(hijos==null || hijos.size()<=0)return ret;
   for(int i=0;i<hijos.size();i++) {
      outs.add(hijos.get(i).getValue());
   }
   for(int i=0;i<hijos.size();i++) {
      aux=hijos.get(i).getKey().outputs();
      for(int j=0;j<aux.size();j++) {
         aux.get(j).add(0, hijos.get(i).getValue());
      }
      if(aux.size()<=0) {
         ArrayList<String> soloUno=new ArrayList<String>();
         soloUno.add(hijos.get(i).getValue());
         aux.add(soloUno);
      }
      ret.addAll(aux);
   }
   if(ret.size()==0) {
      for(int i=0;i<outs.size();i++) {
         ret.add(new ArrayList<String>());
         ret.get(i).add(outs.get(i));
      }
   }
   return ret;
}
```

A.6 NonDeterministicTest.java

```
public ArrayList<ArrayList<String>> runTest(MealyMachine m) {
     Tree trace=new Tree(0);
     LinkedList<Tree> cola=new LinkedList<Tree>();
     LinkedList<Tree> cola2=new LinkedList<Tree>();
     cola.push(trace);
     Tree caso;
     int i=0;
     while(!cola.isEmpty() && i<inputs.size()) {
        while(!cola.isEmpty()) {
           caso=cola.pop();
           caso.generateChilds(m, inputs.get(i));
           for(int j=0;j<caso.getNumChilds();j++) {
              cola2.push(caso.getChildTrees().get(j));
           }
        }
        cola=cola2;
        cola2=new LinkedList<Tree>();
        i++;
     }
     outputs=trace.outputs();
     return outputs;
  }
```

A.7 NonDeterministicTestComparer.java

```
public void runAll(){
     for(int i = 0; i < testList.size(); i++) {
        ArrayList<Integer> tempComps = new ArrayList<Integer>();
        for(int j = 0; j < mutList.size(); j++) {
           tempComps.add(compare(testList.get(i), mutList.get(j)));
        }
        compMatrix.add(tempComps);
     }

  }

  public Integer compare(NonDeterministicTest t, MealyMachine mutation){
     ArrayList<ArrayList<String>> outs = t.getOutputs();
     ArrayList<ArrayList<String>> outs2 = mutation.generateOutputs(t.getInputs());
```

```
    for(int i=0;i<outs2.size();i++) {
       if(!outs.contains(outs2.get(i)))return 1;
    }
    return 0;
 }
```