USING MUTUAL INFORMATION TO SELECT TEST SUITES IN A BLACK-BOX FRAMEWORK

ALFREDO IBIAS MARTÍNEZ

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

Trabajo Fin de Máster en Métodos Formales en Ingeniería Informática

Junio 2019

Director:

Manuel Núñez

Convocatoria:

Junio-Julio 2019

Calificación:

Sobresaliente - 10

Autorización de difusión

Alfredo Ibias Martínez

Junio 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: "Using mutual information to select test suites in a black-box framework", realizado durante el curso académico 2018-2019 bajo la dirección de Manuel Núñez 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

La *información mutua* es una medida usada en Teoría de la Información para cuantificar la cantidad de similaridad entre dos variables aleatorias que actúan sobre dos conjuntos. En este trabajo, adaptamos este concepto y mostramos cómo puede usarse para seleccionar un *buen* conjunto de tests, en un escenario de caja negra y usando el enfoque de *maximizar la diversidad*. Aportamos evidencias experimentales para mostrar la utilidad de la medida propuesta. También mostramos que el tiempo necesario para calcular la medida es despreciable cuando lo comparamos con el tiempo necesario para aplicar tests adicionales. Finalmente, comparamos nuestra medida con las mejores medidas disponibles para priorización de tests y mostramos que nuestra propuesta las supera.

Además, en este trabajo presentamos un enfoque basado en Programación Genética, respaldado por una herramienta, para generar conjuntos de tests utilizando medidas basadas en la Teoría de la Información.

Palabras clave

Enfoques formales de testing; Teoría de la información; Información mutua

Abstract

Mutual Information is an information theoretic measure designed to quantify the amount of similarity between two random variables ranging over two sets. In this paper, we adapt this concept and show how it can be used to select a *good* test suite, in a black-box scenario and following a *maximize diversity* approach. We provide experimental evidence to show the usefulness of the measure. We also show that the time needed to compute the measure is negligible when compared to the time needed to apply extra testing. Finally, we compare our measure with current test prioritization measures and show that our proposal outperforms them.

As a side result, in this thesis we present a Genetic Programming approach, fully supported by a tool, to generate test suites using Information Theory based measures.

Keywords

Formal approaches to testing; Information Theory; Mutual information

Contents

Acknowledgements

I would like to thank my thesis supervisor for his invaluable help and advice and for all the work he did to help me during this thesis. I would also like to thank Professor Robert M. Hierons and David Griñan for their support and their revisions of the work. Finally, I would like to thank my family for his moral and economical support.

Dedication

To my family, my friends and my supervisor, for their invaluable help.

Chapter 1

Introduction

Software testing [3, 36] is the main technique to validate complex systems with the goal of increasing their reliability. Traditionally, software testing has been a mainly manual process, strongly depending on the knowledge of the specific group of testers. However, for more than 20 years, it has been shown that testing can have a formal basis [19]. Actually, formal testing is currently an active research area [6, 9, 24] and the existence of several tools that support formal testing has led to the recognition that the combination of formal methods and testing strongly facilitates test automation [41].

One of the problems that testing activities confront is that, being a time/resources consumption part of the software development process, an appropriate testing process usually needs more time than the available one (mainly due to budget constraints because testing can cost more than 50% of the development budget [36]). Therefore, it is very important to define methodologies that reduce the time without notably decreasing the confidence on the reliability of the system. A good starting point to diminish the time devoted to testing is to reduce the number of tests that we apply to the System Under Test (`SUT`). In fact, the possible number of tests needed to exhaustively test even the simplest systems is exorbitant. For example, exhaustive testing of a black-box implementation of a method adding two numbers on a 32-bit machine needs around $8 \cdot 10^{28}$ tests. Therefore, it is imperative to *select* a reduced number of tests having good capabilities to detect faults in the `SUT`. We can rephrase this problem, and this is the main goal of this work, as the problem of

choosing among different test suites, with the same size, the one that might have a greater expectation of finding faults.

We work within a black-box framework, that is, the SUT is a black-box from which we only assume that we know its alphabet. Therefore, we cannot make any assumptions about the internal structure of the SUT. A side-effect of this setting is that we will not be sure that we are totally covering the SUT because we cannot know whether a certain internal state has been reached, despite the length of the applied tests. On the contrary, we have complete access to the specification of the system that we want to build. In order to simplify the presentation, we will assume that the specification of a system is given by a Finite State Machine (FSM) but our work can be easily adapted to deal with other state-based formalisms. In this context, it makes perfect sense to choose between different finite test suites, but with the same number of inputs, which one we apply to the SUT. We will define an information theoretic measure to perform this choice. Specifically, we will define a notion inspired by *mutual information* [42] and use it to compute, among different test suites, the one with the lowest mutual information. Our intended goal is to *indirectly* maximize diversity because it has been widely recognised that diversity has a strong impact on test quality [8, 17, 21, 22]. In order to assess the validity of our approach we performed experiments using the following approach. We started with a specification that we used with two goals:

- Derivation of test suites. We randomly traverse the specification to generate test suites. As usual when testing from FSMs, a test will be a sequence of (input, output) pairs.

- Generation of mutants. We will generate mutants from the specification. The rationale is that a test suite is better than another one if the first one kills more mutants than the second one. In this setting, we do not need to worry about the problem of equivalent mutants [29, 32] because we will compare numbers of killed mutants: none of the test suites will kill equivalent mutants.

Next we describe our scenario, which we consider to be highly realistic in practical terms. We consider that our testing process has the following properties:

- It is easy to generate different test suites.
- It is relatively easy to select which one of them is the *best* one (where being the best is equivalent to get the best score in a given measure).
- It is expensive to apply a lot of tests (so that we cannot perform exhaustive testing).

For our experiments, we fix the size of the test suites, the number of generated mutants and the size of the `FSM`s (number of states, size of alphabet, outgoing transitions, etc...). Then, we generate test suites and compare their capabilities to detect mutants versus our measure score and show that there is a correlation.

Summarizing the results of our experiments, we repeated the process 50 times and we obtained that our method chooses the test suite that kills more mutants around 75% of the times. Moreover, we obtained a correlation of our measure with the mutation score of -0.650256. These results allow us to claim that there is a relevant correlation between our measure and the ability to kill mutants. In addition, we provide a cheap method (in terms of computational power) to choose between test suites. Note that if the application of test suites were cheap (for example, in terms of time), then it would be better to apply all the available tests. However, the application of tests can be time intensive (in addition to the proper monetary costs). In this case, we really need to choose a reduced number of test. Our work provides a method that almost 75% of the times chooses the *a priori* best test suite. Moreover, our experiments show that the time needed to compute our measure is negligible when compared to the time needed to apply a single test suite. Finally, our experiments show that our measure gets better results than the current best information theory based measures.

Information Theory has been already used for testing [7, 11, 34, 37–39, 44]. In particular, the problem of choosing among different test suites has been addressed before [16, 17, 20]

and different test prioritization techniques have been compared [23].

We have already used Information Theory for black-box testing in order to detect failed error propagation in the execution of a test [26]. However, to the best of our knowledge, this is the first work where a measure inspired in Information Theory is specifically made to choose between test suites in a black-box testing framework.[1] Also, the measure proposed in this work improves the results obtained by previously proposed test prioritization measures.

The rest of the thesis is structured as follows. In Section 2 we review some basic concepts and notation that we will use along the thesis. In Section 3 we formally define our measure and show some simple examples to motivate the usefulness of its current definition. In Section 4 we report on our experiments. In Section 5 we analyse some threats to the validity of our results. In Section 6 we discuss some decisions that we took during the research presented in this thesis. Finally, in Section 7 we give our conclusions and some lines for future work.

[1] Some work has somehow adapted measures specifically designed in white-box testing to a black-box testing setting [23].

Chapter 2

Preliminaries

In this thesis, systems will be modelled as *Finite State Machines* (FSMs). In order to define an FSM, we first need to set some previous notations. Given a set A, we let A^* denote the set of finite sequences of elements of A; $\epsilon \in A^*$ denotes the empty sequence. We let A^+ denote the set of non-empty sequences of elements of A. We let $|A|$ denote the cardinal of set A. Given a sequence $\sigma \in A^*$, we have that $|\sigma|$ denotes its length. Given a sequence $\sigma \in A^*$ and $a \in A$, we have that σa denotes the sequence σ followed by a and $a\sigma$ denotes the sequence σ preceded by a.

Throughout this thesis we let I be the set of input actions and O be the set of output actions. It is important to differentiate between input actions and *inputs* of the system. In our context an input of a system will be a non-empty sequence of input actions, that is, an element of I^+ (similarly for outputs and output actions).

An FSM is a (finite) labelled transition system in which transitions are labelled by an input/output pair. We use this formalism to define specifications.

Definition 1. *We say that $M = (Q, q_{in}, I, O, T)$ is a* Finite State Machine *(FSM), where Q is a finite set of states, $q_{in} \in Q$ is the initial state, I is a finite set of inputs, O is a finite set of outputs, and $T \subseteq Q \times (I \times O) \times Q$ is the transition relation. A transition $(q, (i, o), q') \in T$, also denoted by $q \xrightarrow{i/o} q'$ or by $(q, i/o, q')$, means that from state q after receiving input i it is possible to move to state q' and produce output o.*

We say that M is deterministic *if for all $q \in Q$ and $i \in I$ there exists at most one pair*

$(q', o) \in Q \times O$ *such that* $(q, i/o, q') \in T$. *We say that* M *is* input-enabled *if for all* $q \in Q$ *and* $i \in I$ *there exists* $(q', o) \in Q \times O$ *such that* $(q, i/o, q') \in T$.

We let $\mathtt{FSM}(I, O)$ *denote the set of finite state machines with input set* I *and output set* O.

As usual, we assume that SUTs are input-enabled: the SUT should be able to react, somehow, to any external stimulus. In particular, if the tester applies an input action at a certain stage, then the SUT should be able to provide a response (that is, an output action). This assumption is, actually, realistic because if an input could not be applied in some state of the SUT, then we could always assume that there is a response to the input that reports that this input is blocked. We assume that the FSMs are deterministic, but we do not force specifications to be input-enabled. We have this restriction mainly for compatibility with work on white-box testing that we would like to use as comparison.

A model can be identified with its initial state and we can define a process corresponding to a state q of M by making q the initial state. Thus, we use states and processes and their notation interchangeably. An FSM can be represented by a diagram in which nodes represent states of the FSM and transitions are represented by arcs between the nodes. We use a double circle to denote the initial state.

We will assume the *test hypothesis* [27]: the SUT can be modelled as an object described in the same formalism as the specification (in our case, an FSM). Note that we do not need to have access to this description; we are indeed in a black-box testing framework because we only assume the existence of such FSM. Actually, it would be enough to assume that each time that the SUT receives a sequence of input actions, it returns a sequence of output actions. This will be clear when we use mutants, to simulate possible SUTs, and we test them applying sequences of input actions and observing sequences of output actions.

Our main goal while testing is to decide whether the behaviour of an SUT conforms to the specification of the system that we would like to build. In order to detect differences between specifications and SUTs, we need to compare their behaviours and the main notion to define such behaviours is given by the concept of *trace*.

Definition 2. *Let $M = (Q, q_{in}, I, O, T)$ be an* FSM*, $\sigma = (i_1, o_1) \dots (i_k, o_k) \in (I \times O)^*$ be a sequence of pairs and $q \in Q$ be a state. We say that M can perform σ from q if there exist states $q_1 \dots q_k \in Q$ such that for all $1 \leq j \leq k$ we have $(q_{j-1}, i_j/o_j, q_j) \in T$, where $q_0 = q$. We denote this by either $q \stackrel{\sigma}{\Longrightarrow} q_k$ or $q \stackrel{\sigma}{\Longrightarrow}$. If $q = q_{in}$ then we say that σ is a* trace *of M. We denote by $\texttt{traces}(M)$ the set of traces of M. Note that for every state q we have that $q \stackrel{\epsilon}{\Longrightarrow} q$ holds. Therefore, $\epsilon \in \texttt{traces}(M)$ for every* FSM *M.*

Next we define the notion of test. As we have already explained, a test is a sequence of (input action, output action) pairs. A test suite will be a set of tests.

Definition 3. *Let $M = (Q, q_{in}, I, O, T)$ be an* FSM*. We say that $t = (i_1, o_1) \dots (i_k, o_k) \in (I \times O)^+$ is a* test *for M if $t \in \texttt{traces}(M)$. We define the* length *of t as the length of the sequence, that is, $|t| = k$. We define the sequence of inputs of t as $\alpha = i_1 \dots i_k$ and the sequence of outputs of t as $\beta = o_1 \dots o_k$ (we will sometimes use the notation $t = (\alpha, \beta) \in (I^+ \times O^+)$). A* test suite *for M is a set of tests for M. Given a test suite $\mathcal{T} = \{t_1, \dots, t_n\}$, we define the* length *of the test suite as the sum of the lengths of its tests, that is, $|\mathcal{T}| = \sum_{i=1,\dots,n} |t_i|$.*

Let $t = (\alpha, \beta)$ be a test for M. We say that the application of t to an FSM *M' fails if there exists β' such that $(\alpha, \beta') \in \texttt{traces}(M')$ and $\beta \neq \beta'$. Similarly, let $\mathcal{T}$ be a test suite for M. We say that the application of $\mathcal{T}$ to an* FSM *M' fails if there exists $t \in \mathcal{T}$ such that the application of t to M' fails.*

Intuitively, a test (α, β) for M denotes that the application of the sequence of input actions α to a correct system (with respect to M) should show the sequence of output actions β. Note that if we would allow non-determinism, then the previous inequality must be appropriately replaced to express that the behaviours of the SUT must be a subset of those of the specification, and we will have a notion of conformance similar to ioco [43].

Next we introduce some notation for random variables. First we recall the notion of *entropy* associated with a random variable, which we will use to inspire our measure. The concept of entropy is a "measure of the average uncertainty in the random variable. It is the

number of bits on average required to describe the random variable" [42]. In other words, entropy is a measure of the amount of information of a given set with a random variable ranging over it.

Definition 4. *Let A be a set and ξ_A be a random variable over A. We denote by σ_{ξ_A} the probability distribution induced by ξ_A. The* entropy *of the random variable ξ_A, denoted by $\mathcal{H}(\xi_A)$, is defined as:*

$$\mathcal{H}(\xi_A) = -\sum_{a \in A} \sigma_{\xi_A}(a) \cdot \log_2(\sigma_{\xi_A}(a))$$

In order to select *good* test suites, which can detect a high amount of faults in the system, it is useful to have a *measure* on the goodness of a test suite. Let us emphasize that measures will be, in general, heuristics to find good solutions and that each measure should be validated with experiments. Usually, higher values of a measure will be associated with better solutions, but this relation can be the other way around. We introduce a general notion of measure.

Definition 5. *A* measure *is a function*

$$f : \mathtt{FSM}(I, O) \times \mathcal{P}(I^+ \times O^+) \to \mathbb{R}^+ \cup \{0\}$$

Intuitively, a measure is a function that receives an `FSM` and a test suite and returns a real number representing how good the measure considers that this test suite is to detect faults in an `SUT`. This notion of measure allows us to use information both from the specification and from the test suite that we are evaluating, although it not necessarily has to use information from both, that is, a measure could work only with the information from the test suite and not use the specification at all, or the other way around.

Finding the best test suite according to a measure (that is, the test suite that gets the best score) is usually an NP-complete problem (due to the combinatorial explosion). *Genetic Programming* [30, 33] (in fact, any genetic algorithm) has been a successfully used to find good enough solutions to NP-complete problems [28]. Therefore, we could rely on Genetic

Programming in order to obtain *relatively good* test suites. Next we briefly describe the main components of a genetic algorithm (the basic structure is given in Algorithm 1):

- An encoding of the population in *genes*.
- An *initial population* composed by randomly generated individuals expressed in the selected codification.
- A *fitness function* to evaluate the individuals of the population.
- A *stopping criterion*.
- A *next population selection method*, which usually keeps the best individuals and discards the worst ones (with respect to the fitness function values).
- A *crossover method* that generates new individuals from the mixture of the genes of the existing ones.
- A *mutation method* that can modify some individuals in order to obtain new genes, which might have not been present before.

Genetic Programming consists in using a genetic algorithm where the codification of the population in genes does not use a linear structure (as a vector) but a tree-like structure [30]. Therefore, all the elements of a genetic algorithm have to be adapted to work with this structured types.

Most of the work using genetic algorithms to generate test suites rely on a linear structure to represent the test suite. Specially, they usually rely on a vector of the inputs of the test suite [14, 15, 31, 40]. This encoding of a test suite presents a problem: if the specification is not input-enabled, then the algorithm could generate invalid tests that will always fail when applied to the `SUT`, even if it is totally equivalent to the specification. As we are working with deterministic but not necessarily input-enabled specifications, we have to face this problem, and grammar-guided Genetic Programming gives a solution to it. Grammar-guided Genetic

```
Initialize population;
Evaluate population;
while termination criterion not reached do
    Select next population;
    Perform crossover;
    Perform mutation;
    Evaluate population;
end
```

Algorithm 1: Genetic algorithm: general scheme.

Programming allows us to ensure the correctness of the generated test suites. It also allows us to use the information from the output that each input generates in each state of the `FSM` (as the inputs do not have to generate the same output in all the states). Finally, a problem that comes with this kind of algorithms is that current crosses for grammar-guided Genetic Programming does not keep the size of the solution. Therefore, as we will see later, we will have to adapt the crosses that we perform to keep this size within the limits that we need.

Chapter 3

Our measure

In order to choose between test suites, we might consider test suites with low mutual information between the tests conforming the test suite. As we already explained in the introduction of the thesis, our goal is to increase test diversity with the hope that this will be reflected in the capability of the test suite to detect faults (in our experiments, to kill mutants). Therefore, lower values of mutual information should be associated with higher diversity.

First, let us remind the classical definition of mutual information [42].

Definition 6. *Let A and B be two sets and ξ_A and ξ_B be two discrete random variables ranging, respectively, over A and B. We denote by $I(\xi_A;\xi_B)$ the mutual information of ξ_A and ξ_B and we define it as*

$$\sum_{b\in B}\sum_{a\in A}\sigma_{\xi_{A,B}}(a,b)\cdot\log_2\frac{\sigma_{\xi_{A,B}}(a,b)}{\sigma_{\xi_A}(a)\cdot\sigma_{\xi_B}(b)}$$

where $\xi_{A,B}$ is the joint probability distribution, defined as usual, of ξ_A and ξ_B.

In order to compute our measure, we might consider both the input part of the test and the expected output with the goal of also increasing output diversity [2]. The intuition behind maximizing diversity as our goal is very simple. Assume that we have an `FSM` with a set of (input/output) pairs labelling its transitions. Obviously, if we select two different (input/output) pairs, then we can ensure that this selection corresponds to traverse two

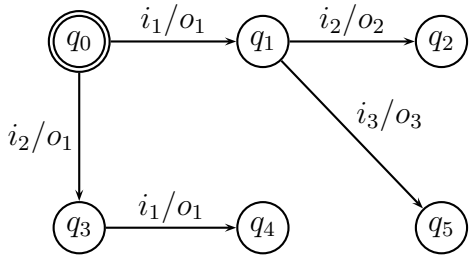

Figure 3.1: Example of FSM.

different transitions of the FSM once. However, selecting one (input/output) pair twice leads to a scenario where we might traverse the same transition of the FSM twice [1]. Actually, this happens with probability $\frac{1}{n}$, where n is the number of times that the (input/output) pair labels a transition of the FSM. This implies that, even if we have a big n, we will have a non-zero probability of being traversing the same transition more than once. This scenario also shows that we have to be careful when looking for diversity, as the probability decreases when n increases. For example, we cannot consider to be the same to have $n = 2$ than having $n = 200$. Therefore, our algorithms will not automatically discard test suites where the same pair appears more than once. Instead, our algorithms should take that into account and weight somehow the difference between repeating a transition that appears fewer times in the FSM versus repeating a transition that appears a lot of times in the FSM. We illustrate this idea with a simple example.

Example 1. *Consider the FSM given in Figure 3.1 and the following two test suites for M:*

$$\mathcal{T}_1 = \{(i_2 i_4, o_1 o_4), (i_1 i_2, o_1 o_2)\} \qquad \mathcal{T}_2 = \{(i_1 i_3, o_1 o_3), (i_1 i_2, o_1 o_2)\}$$

On the one hand, $\mathcal{T}_1$ has a mutual information of 0. Even though the input i_2 appears twice in the test suite, we know that the pairs (i_2, o_1) and (i_2, o_2) represent completely different behaviours. On the other hand, $\mathcal{T}_2$ has a mutual information equal to 0.53 and this value

[1] Note that systems may have the same answer to the same input at different stages of their behaviour. However, and the experiments reported in this thesis confirmed this, it is more likely that different occurrences of the same (input action, output action) pair in a test will be associated with traversing the same transition after a loop has been performed.

is, obviously, greater than 0. *Therefore, our measure should choose the first test suite if we consider a measure based on mutual information.*

First, we would like to compute the mutual information of two tests. Each test is a sequence of (input action, output action) pairs. If we abstract the position of the pair in the sequence, we obtain a set of pairs. We use the notation $(i, o) \in_n t$ to denote that the pair (i, o) appears n times in the test t. In a similar way, we will overload this notation ($(i, o) \in_m M$) to also denote that the pair (i, o) appears in m transitions of the `FSM` M. Given two tests t_1 and t_2, in order to compute $I(\xi_{t_1}; \xi_{t_2})$ we have to give a proper definition of $\sigma_t(x)$. Our first attempt was to give an *intuitive* definition for σ. Therefore, we used the uniform distribution as the probability distribution used in the mutual information formula. That is, if a label appears in m transitions of the machine, then the probability of this label will be $\frac{1}{m}$. Intuitively, $\frac{1}{m}$ is the probability that the pair x corresponds to a specific pair labelling a transition of the specification M. With this definition we are ensuring that the weight of each occurrence is proportional to the probability of corresponding to the same transition.

Unfortunately, this choice does not induce a probability distribution over the pairs appearing in the tests. Therefore, it is not a mathematically valid formulation for the mutual information between two tests. Then, we have to search for alternatives that keep, as long as possible, the initial intuition. After several proposals and tries (that we will discuss in Chapter 6), the best alternative that we obtain was the following one.

Definition 7. *Let $M = (Q, q_{in}, I, O, T)$ be an `FSM`, t be a test for M and $x \in I \times O$ be an input/output pair appearing in t. We say that $\xi^M_{t,x}$ (or simply $\xi_{t,x}$ to not overload the notation) is the random variable corresponding to x according to M if its probability distribution is given by:*

$$\sigma_{\xi_{t,x}}(id) = \begin{cases} \frac{1}{m \cdot s} & \text{if } label(id) = x,\ x \in_m M,\ m \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

where $id \in [1, |T|]$ *is a unique identificator for each transition of* M*,* $label(id)$ *is the input/output pair labelling the transition identified by* id*, and* $s = \sum_{\substack{x \in t \\ x \in_m M}} \frac{1}{m}$.

We define the joint probability of two tests t_1, t_2 *of* M *as:*

$$\sigma_{\xi_{t_1,x_1},\xi_{t_2,x_2}}(id) = \begin{cases} \frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2} \cdot \frac{P}{S2} & \text{if } x_1 = label(id) = x_2,\ m_1 = m_2 \\ \frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2} & \text{otherwise} \end{cases}$$

where $x_1 \in_{n_1} t_1$, $x_2 \in_{n_2} t_2$, $x_1 \in_{m_1} M$, $x_2 \in_{m_2} M$, $s_1 = \sum_{\substack{x_1 \in t_1 \\ x_1 \in_{m_1} M}} \frac{1}{m_1}$, $s_2 = \sum_{\substack{x_2 \in t_2 \\ x_2 \in_{m_2} M}} \frac{1}{m_2}$,

$P = 1 - S_1$, $S_1 = \sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 \neq x_2}} \frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2}$ *and* $S_2 = \sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 = x_2}} \frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2}$

This formulation has a downside (in addition to its complexity). Actually, the experiments showed that it gets worse results than the initial intuition that we had. Therefore, we decided to reconsider our initial intuition and considered a *pseudo-random variable*. This type of variable has a *pseudo-probability distribution* associated with it. Intuitively, a pseudo-probability distribution is a function such that given an element returns a value between 0 and 1, although it does not have to sum up to 1.

Definition 8. *Let* $M = (Q, q_{in}, I, O, T)$ *be an* `FSM`*,* t *be a test for* M *and* $x \in I \times O$ *be an input/output pair present in* t*. We define* $\xi^M_{t,x}$ *(or simply* $\xi_{t,x}$ *to not overload the notation) as the pseudo-random variable corresponding to* x *according to* M*, with a pseudo-probability distribution given by:*

$$\sigma_{\xi_{t,x}}(id) = \begin{cases} \frac{1}{m} & \text{if } label(id) = x,\ x \in_m M,\ m \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

where $id \in [1, |T|]$ *is a unique identificator for each transition of* M *and* $label(id)$ *is the input/output pair labelling the transition identified by* id*.*

We define the composition of two tests t_1, t_2 *of* M *as the pseudo-random variable having*

as pseudo-probability distribution function:

$$\sigma_{\xi_{t_1,x_1},\xi_{t_2,x_2}}(id) = \begin{cases} \frac{1}{m_1} \cdot \frac{1}{\cancel{m_2}} \cdot \cancel{m_2} & \text{if } x_1 = label(id) = x_2,\ m_1 = m_2 \\ \frac{1}{m_1} \cdot \frac{1}{m_2} & \text{otherwise} \end{cases}$$

where $x_1 \in t_1$, $x_2 \in t_2$, $x_1 \in_{m_1} M$ *and* $x_2 \in_{m_2} M$.

Note that our intuition assumes a uniform distribution over the set of transitions of M with the same label. We could choose another distribution for those probabilities[2], for example, increasing the probability associated with transitions that are reached from the initial state after a smaller number of transitions. However, this would complicate the computation of our measure and our preliminary experiments did not show a significant improvement of the measure. Therefore, we will stick to uniform distributions for simplicity and for its good properties when we do not have additional information about the *real* distributions (in particular, this distribution maximizes entropy [13]). Once said this, it is important to remark that we should not consider uniform distributions if we had evidence that they are not appropriate, because using the true distribution leads to better results. For example, this is the case if we had probabilistic user models indicating the experimental probabilities with which users will chose each input [4].

The following result allows us to simplify the formula given in Definition 6.

Lemma 1. *Let* M *be an* `FSM` *and* t_1, t_2 *be tests for* M. *Let* ξ_{t_1}, ξ_{t_2} *be the pseudo-random variables corresponding, respectively, to* t_1 *and* t_2 *and according to* M. *We have*

$$I(\xi_{t_1}; \xi_{t_2}) = \sum_{\substack{y \in t_2 \\ y \in_m M}} \sum_{\substack{x \in t_1 \\ x = y}} \frac{\log_2(m)}{m}$$

Proof. In order to compute the terms of the sum defining mutual information, that is,

$$\sigma_{\xi_{t_1,t_2}}(x,y) \cdot \log_2 \frac{\sigma_{\xi_{t_1,t_2}}(x,y)}{\sigma_{\xi_{t_1}}(x) \cdot \sigma_{\xi_{t_2}}(y)}$$

[2]It is important to remark that we are assuming a scenario where we have no information about the true probability distribution governing those transitions. We use the uniform distribution because it leads to better results in these situations.

we will distinguish two cases: $x = y$ and $x \neq y$. First, let us consider $x = y$. Note that $\sigma_{\xi_{t_1}}(x) = \sigma_{\xi_{t_2}}(x)$, because these values depend only on M and x. In addition, the composition of an element of a test and itself is the probability of the element (as we stated in Definition 8). Therefore, $\sigma_{\xi_{t_1,t_2}}(x,x) = \sigma_{\xi_{t_1}}(x)$. Now, taking into account Definition 8 we have that if $x = y$ then the previous term is equal to

$$\frac{1}{m} \cdot \frac{1}{m} \cdot m \cdot \log_2 \left(\frac{\frac{1}{m} \cdot \frac{1}{m} \cdot m}{\frac{1}{m} \cdot \frac{1}{m}} \right) = \frac{1}{m} \cdot \log_2 \left(\frac{1}{\frac{1}{m}} \right)$$

and, simplifying, we conclude that this term is equal to

$$\frac{\log_2(m)}{m}$$

Now, let us consider $x \neq y$. In this case, $\sigma_{\xi_{t_1,t_2}}(x,y) = \sigma_{\xi_{t_1}}(x) \cdot \sigma_{\xi_{t_2}}(y)$ and, therefore, we have $\log_2(1)$ and this is equal to 0. Therefore,

$$\sigma_{\xi_{t_1,t_2}}(x,y) \cdot \log_2 \frac{\sigma_{\xi_{t_1,t_2}}(x,y)}{\sigma_{\xi_{t_1}}(x) \cdot \sigma_{\xi_{t_2}}(y)} = 0$$

The result immediately follows from these two cases. □

An important remark about this formulation to compute our measure inspired in mutual information is that it is not monotonous and it is equal to 0 if all the transitions of the specification are pairwise different. Since we are interested in values that are useful to compare test suites (therefore, we need monotony and we should avoid to "divide by zero"), we can solve this problem with a simple transformation. The dashed curve in Figure 3.2 shows the behaviour of the previous formula. We will do a small translation in the X axis of the logarithm of the formula, so that its behaviour is the one given by the solid curve.

Definition 9. *Let M be an `FSM` and t_1, t_2 be tests for M. Let ξ_{t_1}, ξ_{t_2} be the pseudo-random variables corresponding, respectively, to t_1 and t_2 and according to M. We have that the modified mutual information formula will be*

$$BMI(\xi_{t_1}; \xi_{t_2}) = \sum_{\substack{y \in t_2 \\ y \in_m M}} \sum_{\substack{x \in t_1 \\ x = y}} \frac{\log_2(m+1)}{m}$$

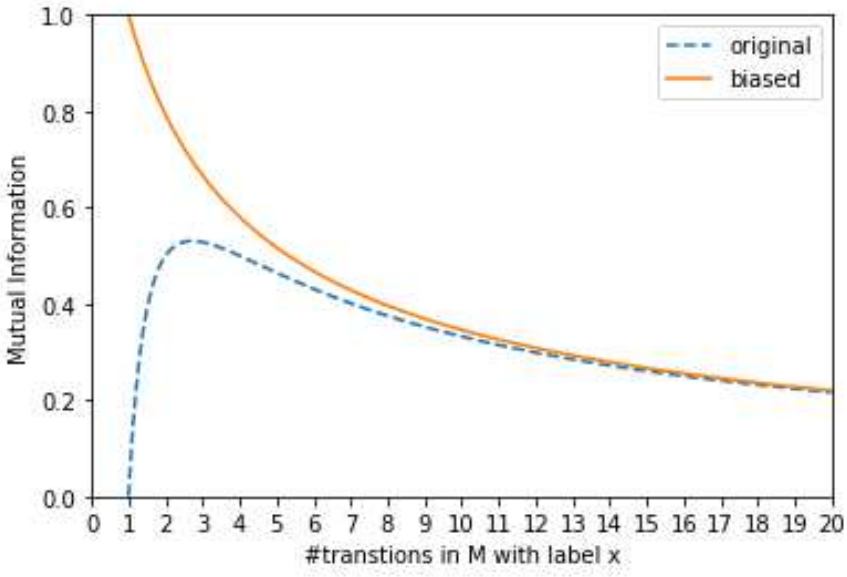

Figure 3.2: Measure comparison plot.

We will call this formula biased mutual information (BMI).

In the following example we illustrate the importance of this translation.

Example 2. *Consider the* `FSM` M *depicted in Figure 3.3 and the test suites* $\mathcal{T}_1 = \{t_1 = (i_2i_1i_1, o_2o_1o_1),\ t_2 = (i_2i_4i_4, o_2o_4o_4)\}$ *and* $\mathcal{T}_2 = \{t_3 = (i_3i_1i_1, o_3o_1o_1),\ t_4 = (i_3i_2i_2, o_3o_2o_2)\}$. *Note that the only pair appearing in both* t_1 *and* t_2 *is* (i_2, o_2) *(this transitions appears* 9 *times in* M*); similarly, the only common pair for* t_3 *and* t_4 *is* (i_3, o_3) *(this transitions appears* 1 *time in* M*). The (biased) mutual information of each test suite can be computed as follows:*

$$BMI(\xi_{t_1}; \xi_{t_2}) = \frac{\log_2(9+1)}{9} = \frac{\log_2(10)}{9} \approx 0.3691$$

$$BMI(\xi_{t_3}; \xi_{t_4}) = \frac{\log_2(1+1)}{1} = \frac{\log_2(2)}{1} = 1$$

$$I(\xi_{t_1}; \xi_{t_2}) = \frac{\log_2(9)}{9} \approx 0.3522$$

$$I(\xi_{t_3}; \xi_{t_4}) = \frac{\log_2(1)}{1} = 0$$

Therefore, the first test suite would be better if we consider biased mutual information, but would be worse if we consider mutual information. In principle, we should prefer $\mathcal{T}_1$ *because*

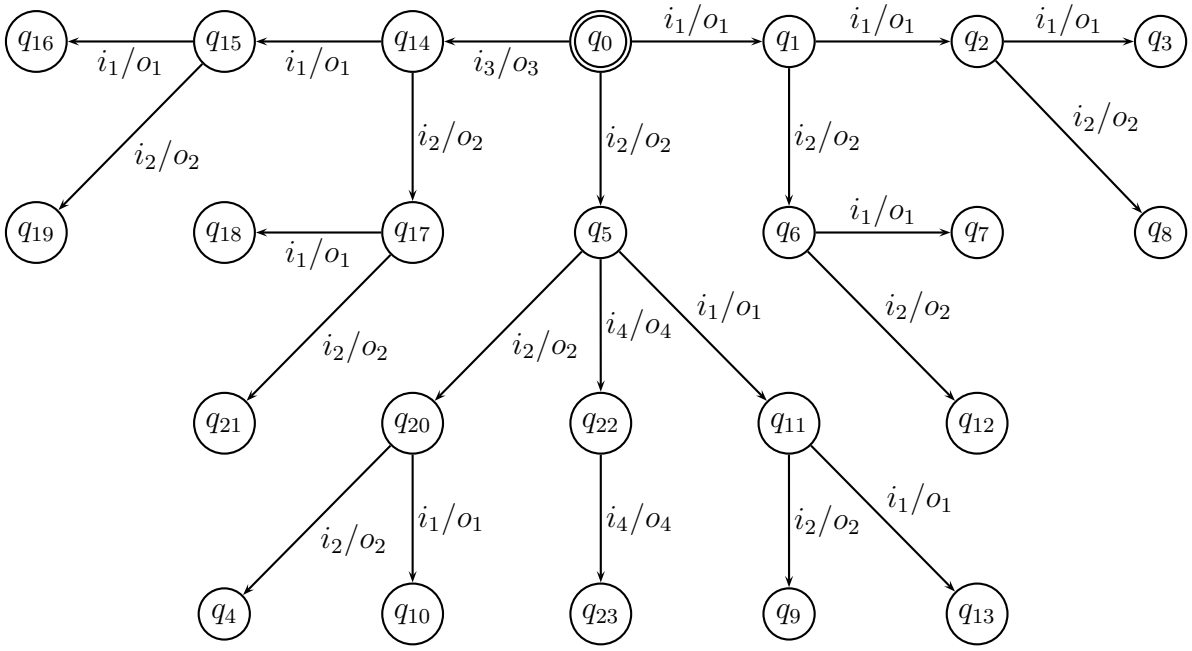

Figure 3.3: Example of `FSM`.

it is more likely that it will check more transitions than $\mathcal{T}_2$. In fact, in this example we know that the second test suite will traverse the same transition twice.

We are aware that the previous example shows that BMI works in one (ad hoc) example. In the next section, we will present experimental evidence to show that this measure works for randomly generated systems.

So far, we have defined biased mutual information for a test suite consisting of two tests. Fortunately, it is very easy to extend our notion to deal with more than 2 tests. This is so because we do not need to know the biased mutual information between all the sequences of (input/output) pairs of the test suite; we need to compute the cumulative amount of biased mutual information between all the possible pairs of these sequences (so it is easily done with a summation). Therefore, if we have a specification M (as usually, we omit M in the following notation) and a test suite $\mathcal{T} = \{t_1, \ldots, t_k\}$ then

$$I(\mathcal{T}) = I(\xi_{t_1}, \ldots, \xi_{t_k}) = \sum_{i=1,\ldots,k} \sum_{j=i+1,\ldots,k} I(\xi_{t_i}, \xi_{t_j}) =$$

$$= \sum_{i=1,\ldots,k} \sum_{j=i+1,\ldots,k} \sum_{y \in t_i} \sum_{x \in t_j} \sigma_{\xi_{t_j,t_i}}(x,y) \cdot \log_2 \frac{\sigma_{\xi_{t_j,t_i}}(x,y)}{\sigma_{\xi_{t_j}}(x) \cdot \sigma_{\xi_{t_i}}(y)}$$

Now, using Lemma 1 and Definition 9, we have that the biased mutual information of $\mathcal{T}$ is

$$BMI(\xi_{t_1}, \ldots, \xi_{t_k}) = \sum_{i=1,\ldots,k} \sum_{j=i+1,\ldots,k} \sum_{\substack{y \in t_i \\ y \in_m M}} \sum_{\substack{x \in t_j \\ x = y}} \frac{\log_2(m+1)}{m}$$

We have to add to the previous expression the biased mutual information of each test of the test suite. This is computed as:

$$BMI(\xi_t; \xi_t) = \sum_{\substack{y \in n\, t \\ y \in_m M}} \frac{(n-1) \cdot n}{2} \cdot \frac{\log_2(m+1)}{m}$$

This addition allows us to give a more precise formulation because it will give lower priority to tests corresponding to the repeated traversal of a loop in the specification. In fact, we are penalising the repetition of the same (input/output) pair (and therefore, potentially the same transition) in the test.

Therefore, the final formula to compute the biased mutual information of a test suite $\mathcal{T}$ is given in the following definition.

Definition 10. *Let M be an `FSM` and $\mathcal{T} = \{t_1, \ldots, t_k\}$ be a test suite for M. Let $\xi_{t_1}, \ldots, \xi_{t_k}$ be the pseudo-random variables corresponding, respectively, to $t_1, \ldots, t_k$ and according to M. We have*

$$BMI(\xi_{t_1}, \ldots, \xi_{t_k}) = \sum_{i=1,\ldots,k} \sum_{\substack{y \in n\, t_i \\ y \in_m M}} \left(\sum_{j=i+1,\ldots,k} \sum_{\substack{x \in t_j \\ x = y}} \frac{\log_2(m+1)}{m} + \frac{(n-1) \cdot n}{2} \cdot \frac{\log_2(m+1)}{m} \right)$$

In the next chapter of this thesis we will describe several experiments confirming that our measure works reasonably good to choose among two different test suites the one that has more potential to detect faults in the `SUT`.

Chapter 4

Empirical evaluation

In this chapter we show our results concerning the usefulness of our measure to choose *good* test suites. First, we clearly state our research questions.

4.1 Research questions

In order to *test* our measure, we will first check how *well* it works.

Research Question 1. *Are lower levels of biased mutual information associated with higher fault coverage?*

If we have a positive answer (with statistical relevance) to this question, then we would like to see how our strategy compares to a purely random choice and with currently proposed Information Theory measures.

Research Question 2. *Have test suites selected by biased mutual information higher fault coverage than the ones randomly selected?*

Research Question 3. *Have test suites selected by biased mutual information higher fault coverage than the ones selected by test set diameter (TSDm)?*

Finally, we would like to check whether our measure is time consuming in a way such that it might be better to use the time needed to compute the measure to apply additional tests.

Research Question 4. *How does the time to execute the selection method scale as the size of the test suite increases? How does the time needed to compute the selection method relate to the time needed to apply a test suite?*

Before we show the results of the experiments, we briefly describe how our `FSM` *subjects* are computed.

4.2 FSM Generator

In order to perform our experiments we need to generate `FSM`s. We developed an `FSM` generator that randomly generates `FSM`s given some parameters. We used the OpenFST library [1][1] to represent `FSM`s. Once we had a proper representation of our `FSM`s, we developed a tool to generate these `FSM`s. In order to have a general tool that can be used in a range of experiments, we included some basic parameters:

- $NREP$: the number of `FSM`s that we want to generate.
- MAX_STATES: the maximum number of states an `FSM` can have.
- MIN_STATES: the minimum number of states an `FSM` must have.
- $MAX\_TRANSITIONS$: the maximum number of transitions each state of an `FSM` can have.
- $MIN\_TRANSITIONS$: the minimum number of transitions each state of an `FSM` must have.
- NUM_INPUTS: the number of inputs.
- $NUM\_OUTPUTS$: the number of outputs.

[1]This library is intended to work with Finite State Transducers (as its name indicates). These are a kind of `FSM`s with an (input/output) pair in each transition and a weight. Therefore, we simply ignored the weight.

```
Result: NREP FSMs.
machine = 0;
while machine < NREP do
    Set a random number S of states between MIN_STATES and
     MAX_STATES for the FSM;
    Choose the state 0 as initial state;
    for each state 0 ≤ i < S − 1 of the machine do
        Set a random number T of transitions between MIN_TRANSITIONS
         and MAX_TRANSITIONS for the state;
        for each transition 0 ≤ j < T of the state do
            if j == 0 then
                Set the state i + 1 as the end of the transition;
            else
                Set a random state as the end of the transition;
            end
            Set a random input label for the transition;
            Set a random output label for the transition;
            Save this transition to the FSM file;
        end
    end
    Create the binary file that the OpenFST library uses;
    machine + +;
end
```

Algorithm 2: FSM generation algorithm.

After setting these basic parameters, the program can be executed. The execution flow for generating an FSM using our tool is given in Algorithm 2. Note that, by construction, the tool returns connected FSMs (all states are reachable from the initial state).

In order to create input-enabled FSMs with our tool we simply set:

$$MIN\_TRANSITIONS = MAX\_TRANSITIONS = NUM\_INPUTS$$

4.3 Experiments

In order to answer our research questions, we performed different experiments. The first ones tries to address RQ1, RQ2 and RQ4, while the last ones tries to answer RQ3. The code developed to answer these questions has an MIT license and can be downloaded from https://github.com/Colosu/Master-Thesis.

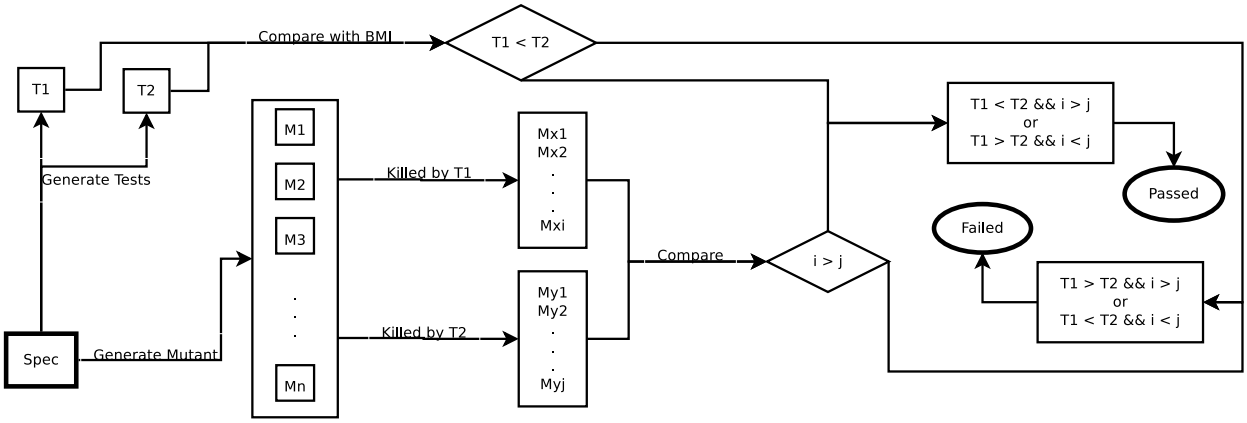

Figure 4.1: Schema of our experiment scenario.

4.3.1 Measure convenience

First, we need to know if our measure is valid for our purposes. In order to do so we will follow the schema presented in Figure 4.1. We started with 100 randomly generated FSMs. These FSMs had 50 states. Each state has a random number, between 2 and 5, of outgoing transitions. Each transition will have a pair (input/output) where the input and output alphabets have 5 elements [2].

For each FSM, we first go through it storing all the possible (input/output) labels of the transitions and the number of times a label appears. Then, we generate 2 test suites, of size 100, from the FSM representing the specification[3] and compute our measure to chose the *best* one. Finally, we produce 1000 mutants of the original FSM. We only use the mutation operator consisting in modifying the incoming state of a transition because our experiments showed that mutations on the labels of the transitions are easier to detect during testing.

We apply the test suites to the mutants and compute which test suite kills more mutants. We repeat this procedure for each FSM and then compute how many times our selected test suite was the one that killed more mutants. We produce a *percentage of success*, that is, how many times the test suite chosen by our measure was the one killing more mutants (in case

[2]Later, we will see experiments with alphabets of different size, FSMs with different number of states and/or different number of outgoing transitions. We do not report results on these scenarios because they are similar to the ones presented in this thesis.

[3]We produce random test suites by randomly traversing the specification.

of ties, either concerning the value of our measure or the number of killed mutants, we avoid this result). We repeated the whole process 50 times, getting 50 different percentages of success (each percentage is obtained from 100 different FSMs). Therefore, the final percentage is computed after repeating the experiment 5000 times.

The results of our experiments are given in Table 4.1. First, we randomly generated test suites without any checking. Therefore, our initial test suites had repeated and/or redundant tests [4]. In this setting, the results show that our measure is very good: the percentage of success was 62.2402%. Then, we ensured that our test suites did not have either repeated tests or prefixes of another test. Our results where almost the same: we got, on average, a percentage of success equal to 62.4924%. Therefore, our measure is equally good given any kind of test suite and, very important, none of the experiments showed a result with a percentage of success lower than 50%. An important remark we have to do is that this scenario is a very *complex* one, due to the small alphabet. The complexity comes from the fact that with FSMs with smaller alphabets, it is easier that in a test a (input/output) pair is repeated. This leads to bigger similarity in the test suites and therefore it is harder to detect differences. In the next section we will discuss what happens when we have bigger alphabets.

In Table 4.2 we display a summary of these results, where we can see that they are almost the same. We have only a small difference between the percentages of success. We think that it was be produced due to the randomness associated with the process. Also, we can see that in both cases, all the values are in the range $[0.5 - 0, 8)$, with the majority of them belonging to the range $[0.6 - 0.7)$. Finally, we can see that there are almost no differences between the maximum and minimum values. These observations reinforce the idea that our measure is equally good if we allow the test suite to have repeated tests or not. Therefore, and thinking as we would do when doing proper testing, during the rest of the experiments we will always consider the case where we do not allow the test suite to

[4]Note that in this specific situation, a test suite will be a multi-set of tests instead of a set. This is so because we need to take into account how many times a certain test appears in the test suite.

Run Number	With possible repetition of tests	without repetition of tests
1	0.65	0.555556
2	0.59596	0.62
3	0.666667	0.535354
4	0.602041	0.62
5	0.71	0.642857
6	0.58	0.59596
7	0.66	0.612245
8	0.56	0.65
9	0.65	0.585859
10	0.656566	0.71
11	0.69697	0.520408
12	0.59	0.646465
13	0.63	0.656566
14	0.63	0.636364
15	0.602041	0.737374
16	0.58	0.71
17	0.663265	0.636364
18	0.606061	0.646465
19	0.616162	0.581633
20	0.65	0.608247
21	0.585859	0.686869
22	0.59	0.57
23	0.540816	0.666667
24	0.653061	0.61
25	0.565657	0.626263
26	0.602041	0.61
27	0.61	0.587629
28	0.670103	0.63
29	0.639175	0.618557
30	0.632653	0.636364
31	0.540816	0.59596
32	0.626263	0.57
33	0.618557	0.632653
34	0.68	0.585859
35	0.6	0.646465
36	0.606061	0.64
37	0.58	0.57
38	0.66	0.61
39	0.59	0.626263
40	0.69697	0.663265
41	0.656566	0.69697
42	0.545455	0.656566
43	0.59	0.63
44	0.714286	0.656566
45	0.6	0.540816
46	0.571429	0.642857
47	0.63	0.61
48	0.606061	0.61
49	0.61	0.628866
50	0.714286	0.68

Table 4.1: Percentages of success of our selected test suite.

Type of test suite	# runs $[0.5, 0.6)$	# runs $[0.6, 0.7)$	# runs $[0.7, 0.8)$	min value	max value	% success (mean)
with possible repetitions	15	32	3	0.540816	0.714286	62.2402%
without repetitions	13	34	3	0.520408	0.737374	62.4924%

Table 4.2: Summary of the results of the first experiment.

have neither repeated nor redundant tests.

4.3.2 Measure analysis

Once we can claim, with experimental evidence, that our measure works, we would like to do a *sanity check*. We repeated the experiments using the same `FSM`s but different test suites and mutants, and evaluated what is the correlation between our measure score and the mutation score (i.e. the number of killed mutants). We obtained an average Pearson correlation of -0.369134 and an average Spearman correlation of -0.356978. The fact that the correlations are negative implies that lower scores of our measure will obtain higher mutation scores, what gives us an affirmative answer to RQ1. More important, these correlations are consistent with the results from the previous experiment. Specifically, if we select the test suite with lower biased mutual information, then we will be selecting the test suite that kills more mutants approximately 62% of the times. This implies that the correlation should be negative. The full results are displayed in Table 4.3.

In order to decide how well our measure performs in terms of time, we realized another experiment. For this experiment, we produced 100 new `FSM`s, with the same parameters as before, but with a bigger alphabet (specifically, of size 25). We wanted to check how the results can change in a more relaxed scenario. We generated test suites without repetitions because the previous experiments show that the results are nearly the same. This time we also recorded the time that it takes to compute our measure (we will use these values to assess the performance of our measure). The results are given in Table 4.4.

On average, the percentage of success was 75.0605%. These results are much better than the results from the first experiments. So we can conclude that our method works better in

Run Number	Pearson correlation	Spearman correlation
1	−0.378529	−0.447121
2	−0.407239	−0.308503
3	−0.305347	−0.299361
4	−0.338248	−0.257336
5	−0.342747	−0.374436
6	−0.634282	−0.541008
7	−0.731647	−0.731102
8	−0.273694	−0.312782
9	−0.247209	−0.220384
10	−0.395459	−0.409929
11	−0.259926	−0.298081
12	−0.170457	−0.0308619
13	−0.414295	−0.456907
14	−0.507343	−0.693233
15	−0.58775	−0.624765
16	−0.44744	−0.545865
17	0.123024	0.232103
18	−0.383787	−0.300752
19	−0.534142	−0.471783
20	0.00977185	−0.0647103
21	−0.505013	−0.486649
22	−0.354695	−0.484575
23	−0.492013	−0.408578
24	−0.357769	−0.355907
25	−0.460478	−0.391877
26	−0.584937	−0.660399
27	−0.425394	−0.415194
28	−0.224411	−0.227905
29	−0.495339	−0.596992
30	−0.674246	−0.603391
31	−0.0656146	0.00300865
32	−0.468773	−0.438511
33	−0.433851	−0.395637
34	−0.564793	−0.456735
35	−0.553955	−0.548872
36	−0.383386	−0.40271
37	−0.285205	−0.221302
38	0.110759	0.0721805
39	−0.689946	−0.61203
40	−0.664481	−0.54778
41	−0.333295	−0.275188
42	−0.328676	−0.34501
43	−0.158634	−0.202484
44	−0.0459975	0.0428894
45	−0.245797	−0.300113
46	−0.380434	−0.359398
47	−0.325745	−0.26968
48	−0.462855	−0.37594
49	−0.132061	−0.17833
50	−0.242898	−0.248966

Table 4.3: Correlation between BMI and mutation score with alphabet size of 5.

Run Number	Percentage of success	Elapsed Time
1	0.76	0.000842141
2	0.721649	0.000793936
3	0.85	0.000822785
4	0.767677	0.000821019
5	0.826531	0.000813211
6	0.680412	0.000838903
7	0.77	0.000841091
8	0.74	0.000789195
9	0.806122	0.000826917
10	0.707071	0.000835529
11	0.717172	0.000853626
12	0.74	0.00082674
13	0.8	0.000840832
14	0.787879	0.000840821
15	0.76	0.00086412
16	0.75	0.000841077
17	0.714286	0.000850253
18	0.69	0.00082661
19	0.747475	0.0008403
20	0.714286	0.000846815
21	0.707071	0.000871455
22	0.65	0.000806566
23	0.757576	0.000827078
24	0.7	0.000842363
25	0.77	0.000819572
26	0.76	0.000836346
27	0.693878	0.00083186
28	0.717172	0.00085203
29	0.74	0.000836536
30	0.76	0.00085497
31	0.767677	0.000844704
32	0.795918	0.000847846
33	0.767677	0.000852514
34	0.78	0.000832202
35	0.71	0.000828946
36	0.795918	0.000863082
37	0.767677	0.000832604
38	0.83	0.000833852
39	0.83	0.000831776
40	0.838384	0.000874901
41	0.636364	0.000850119
42	0.737374	0.000826633
43	0.83	0.000844154
44	0.767677	0.000823715
45	0.686869	0.000835953
46	0.747475	0.000843442
47	0.69	0.000844199
48	0.73	0.000821044
49	0.7	0.000852965
50	0.814433	0.000873662

Table 4.4: Results of BMI computation time.

FSMs with big alphabets.

Concerning time, on average we needed 0.00083786 seconds for the computation of our measure for test suites of size 100.[5] Therefore, we needed a time of the order of thousandths of a second to compute our measure for all the test suites used in our experiment. In principle, we can conclude that the time needed to compute our measure is acceptable for our purposes but we would like to compare this time with the time needed to apply a test suite, so we performed another experiment.

As a performance check, in order to answer RQ4 we repeated the previous experiment as follows. First, we recorded the time needed to compute our measure for test suites of size 100, 200, 300, 400, 500, 600, 700, 800, 900 and 1000. We also recorded the time needed to apply these test suites over 1000 mutants assuming that the time needed to execute a transition is 0, 0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1 and 1 seconds. Note that in our setting, the execution of a transition is instantaneous but if we test a real system, then there will be a delay, depending on the computations needed to compute the result, between the application of an input and the reception of an output. Also note that we do not need to assume these *delays* when computing our measure because we compute it using only the specification and the involved test suites: neither test application nor transition executions are needed. With all these values, we compared the average time needed to compute our measure over one test suite versus the average time needed to apply a test suite over a mutant and we got the results that we can see in Figure 4.2. It is important to note that the average time to compute our measure includes the time needed to generate the map of I/O pairs from the specification. This time depends on the size of the specification and it is computed only once for all the measure computations. Therefore, although we added it to the time needed to compute our measure for one test suite, computing it for two test suites will not double the time. Instead, it will simply add the time that is reflected on the plot

[5]The experiments were run on a GNU/Linux machine with an AMD® Ryzen threadripper 1920X at 3.50GHz × 12 cores and with 32GB of RAM (although only one core was running at a time and we did not use more than 4Gb of RAM).

for transition time equal to 0 because, in this case, the time needed to generate the I/O pairs map is assumed to be 0.

As we can deduce from Figure 4.2, as long as the time needed to perform a transition is higher than 0.001 seconds, computing our measure is much faster than applying test suites. If this time is higher than 0.1 seconds then we need minutes (or even hours if it is higher than 1 second) to apply a test suite. As an additional result of our experiments, we validated that the time needed to compute our measure scales (approximately) quadratically with respect to the test suite size.

This last fact is important because the time to execute a test suite follows a nearly logarithmic scale with respect to the test suite size. This is so because we are cutting the execution of a test whenever we found a failure, as this implies that the implementation does not pass the test. Note that if we would continue the execution after a failure is found, then the scale in time should be (nearly) lineal. Anyway, there will always be a test suite size where the time needed to select a test suite will be equally to the time needed to execute a test suite. Therefore, we should take into account that for some test suites it will be better to execute two test suites than to select one using our measure.

Finally, in order to observe the consistency in the Pearson and Spearman correlations between our measure and mutation scores, we repeated the experiment that started this section. However, this time we used the `FSM`s having an alphabet of 25 elements that we used in the last two experiments. The results show that the consistency holds because we got even better correlations. Specifically, we obtained an average Pearson correlation of -0.650256 and an average Spearman correlation of -0.634711. The full results are displayed on Table 4.5.

4.3.3 Measure comparison

Finally, we wanted to compare our measure with previous proposals. In this regard, we compared our measure with the Test Set Diameter (TSDm) measure and method [16].

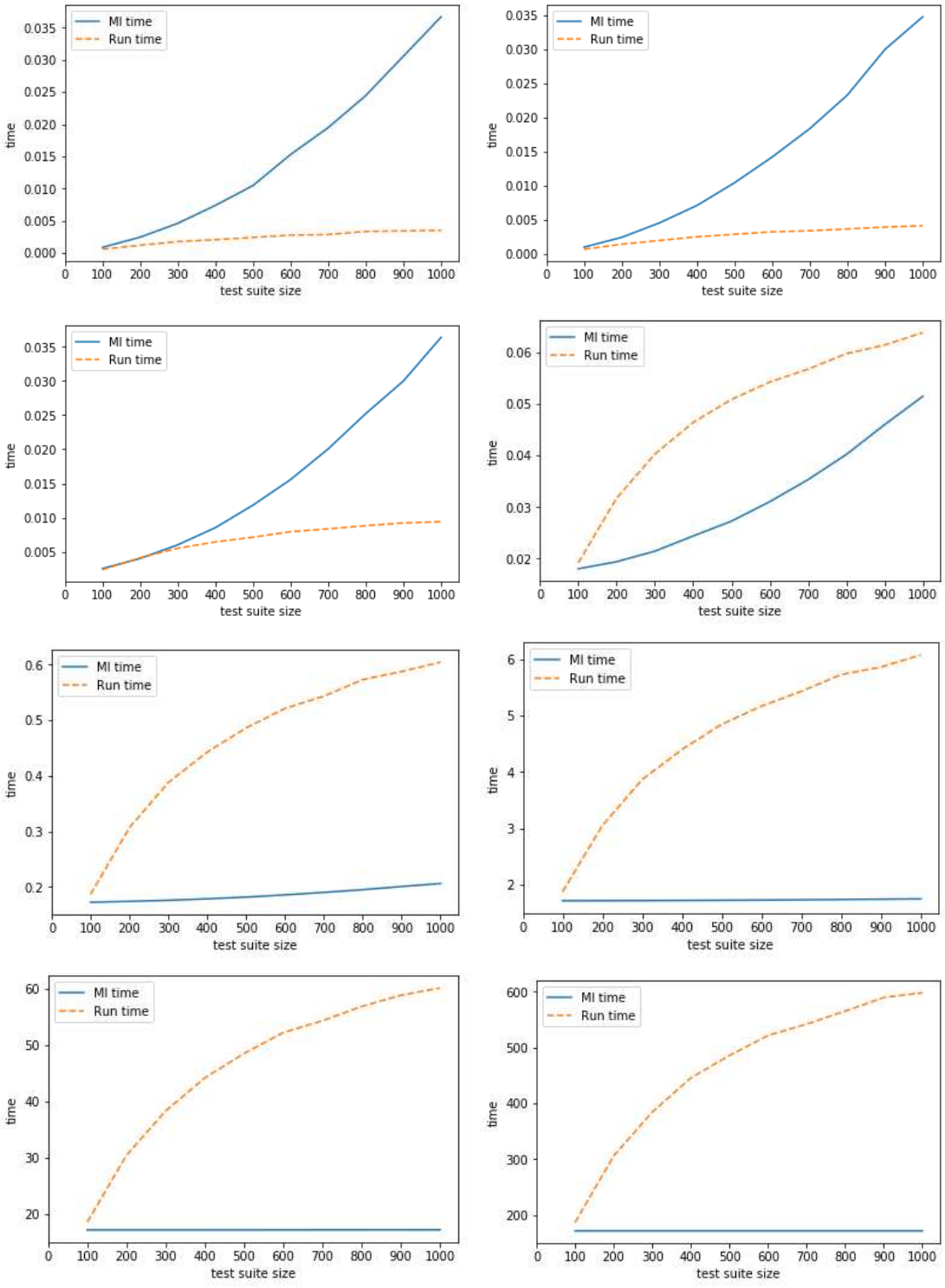

Figure 4.2: Time comparison plots (from left to right, from top to bottom, with transition time of 0, 0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1 and 1 seconds).

Run Number	Pearson correlation	Spearman correlation
1	−0.571537	−0.426476
2	−0.517691	−0.501696
3	−0.692478	−0.634825
4	−0.539586	−0.556391
5	−0.705762	−0.602183
6	−0.671802	−0.555305
7	−0.704822	−0.77924
8	−0.589333	−0.631064
9	−0.56587	−0.519744
10	−0.642352	−0.643851
11	−0.828483	−0.817908
12	−0.614509	−0.62147
13	−0.749209	−0.749906
14	−0.649231	−0.659135
15	−0.583118	−0.360286
16	−0.620321	−0.729323
17	−0.783434	−0.809184
18	−0.761556	−0.733358
19	−0.810923	−0.838661
20	−0.531118	−0.562406
21	−0.347714	−0.341353
22	−0.561493	−0.454306
23	−0.558292	−0.62754
24	−0.662369	−0.72009
25	−0.804059	−0.774436
26	−0.723854	−0.748683
27	−0.798174	−0.7567
28	−0.624052	−0.647347
29	−0.605207	−0.543675
30	−0.525772	−0.561324
31	−0.73993	−0.827379
32	−0.520028	−0.697744
33	−0.541205	−0.496989
34	−0.643269	−0.58443
35	−0.789753	−0.864459
36	−0.843549	−0.767784
37	−0.787704	−0.774436
38	−0.706133	−0.678706
39	−0.666575	−0.715789
40	−0.596623	−0.585844
41	−0.721583	−0.762406
42	−0.725915	−0.708804
43	−0.528187	−0.541008
44	−0.596918	−0.491347
45	−0.694988	−0.527109
46	−0.583214	−0.487585
47	−0.730042	−0.712782
48	−0.60586	−0.64812
49	−0.744956	−0.613996
50	−0.402239	−0.340986

Table 4.5: Correlation between BMI and mutation score with alphabet size of 25.

TSDm method

The TSDm method [16] produces a test suite from a pool of tests. This method uses *test set diameter* (TSDm) measures as a guide to generate test suites. In order to assess the suitability of our measure, we will use it in combination with the TSDm method to evaluate its goodness with respect to different test set diameter measures.

Specifically, we select in each step, from a pool of tests, the test that will be removed from the test suite. In other words, if we take out this test from the pool, then the test suite conformed by the remaining tests of the pool has the best score[6] of the measure when compared to the test suites that will be produced from taking out another test from the pool. This process is iteratively performed until we obtain a test suite of the desired size.

In order to perform the comparison, we started using the same 100 `FSM`s from the first experiment. For each `FSM`, we first go through it storing all the possible (input/output) labels of the transitions and the number of times a label appears. Then we produced two test suites of size 100 executing the TSDm method over the same pool of 100 tests, using the Input-TSDm (ITSDm) measure for one test suite and our BMI for the other test suite. After that, we produce 1000 mutants of the original `FSM`, with mutations consisting, as before, in changing the target state of a transition to another state. The states of the mutants are completed with *bogus* self-loop transitions to ensure input-enabledness.

Finally, we computed the mutation score applying the test suites to the mutants. We repeated this procedure for each `FSM` and then computed how many times the test suite generated using BMI was the one with higher mutation score. With this result we produced a *percentage of success*, that is, how many times the test suite generated with our measure was the one with higher mutation score (in case of ties on the number of killed mutants, as usual, we avoid this result). We repeated the whole process 50 times, getting 50 different percentages of success (each percentage is obtained from 100 different `FSM`s). Therefore, the final percentage is computed after repeating the experiment 5000 times.

[6]The maximum for ITSDm and the minimum for BMI.

Run Number	Percentage of success BMI	Percentage of success ITSDm
1	0.857143	0.142857
2	0.880952	0.119048
3	0.875	0.125
4	0.864865	0.135135
5	0.891892	0.108108
6	0.871795	0.128205
7	0.906977	0.0930233
8	0.809524	0.190476
9	0.904762	0.0952381
10	0.828571	0.171429
11	0.948718	0.0512821
12	0.875	0.125
13	0.809524	0.190476
14	0.853659	0.146341
15	0.909091	0.0909091
16	0.863636	0.136364
17	0.847826	0.152174
18	0.886364	0.113636
19	0.933333	0.0666667
20	0.875	0.125
21	0.918919	0.0810811
22	0.888889	0.111111
23	0.833333	0.166667
24	0.833333	0.166667
25	0.844444	0.155556
26	0.897436	0.102564
27	0.9	0.1
28	0.854167	0.145833
29	0.926829	0.0731707
30	0.930233	0.0697674
31	0.893617	0.106383
32	0.893617	0.106383
33	0.911111	0.0888889
34	0.945946	0.0540541
35	0.906977	0.0930233
36	0.883721	0.116279
37	0.883721	0.116279
38	0.897436	0.102564
39	0.866667	0.133333
40	0.909091	0.0909091
41	0.92	0.08
42	0.837209	0.162791
43	0.894737	0.105263
44	0.904762	0.0952381
45	0.956522	0.0434783
46	0.857143	0.142857
47	0.930233	0.0697674
48	0.854167	0.145833
49	0.869565	0.130435
50	0.948718	0.0512821

Table 4.6: Results of the TSDm method with half of the pool.

The results of the experiment took a lot of time to compute. So, we only managed to obtain the first result and the percentage of success was 100%. However, we realized that our measure took around 78% more time to produce the better test suite than the ITSDm. This could be considered as a kind of *cheating*: our measure is much better but it takes much longer to compute it. Therefore, we decided to perform two quicker experiments.

The first one consists in cutting the times to the half. In order to do so, as we cannot cut off the computation of the measure, we decided to reduce the pool for the BMI measure to a 50% of the pool that the ITSDm will have to produce the test suite. Then, the pool for BMI corresponds to the first 50 tests from the pool of the ITSDm, although our experiments show that it would be essentially the same to select any 50 tests from the ITSDm pool. The results of this experiment are given in Table 4.6. They show that our measure is very good: the percentage of success was 88.5836%, what implies that with half of the resources, our measure clearly outperforms ITSDm.

The last experiment comes from the decision to give both measures the same time to produce the test suite. In order to do so, we shrink the pool for BMI to 22 tests from the pool of the ITSDm. The results of the experiment are given in Table 4.7. As we can see, a curious effect appears: our measure is better *only* 46.89% of the time. So, we have slightly worse results but we have a *handicap* of 78%. More important, there are runs where our measure gets a success score of more than 50%.

TSDm measure

We wanted also to test our measure versus the TSDm measure in a more active way. Instead of choosing from a pool of randomly generated test suites, we wanted to compare how good is each measure in order to create a *good* test suite. Then, we wanted to compare their performance as a measure to guide an evolutive programming method to generate test suites.

In order to do so, we will use the tool that we developed during this academic year. The tool is able to generate test suites using Genetic Programming [25]. The framework and

Run Number	Percentage of success BMI	Percentage of success ITSDm
1	0.46	0.54
2	0.442623	0.557377
3	0.461538	0.538462
4	0.473684	0.526316
5	0.508772	0.491228
6	0.365385	0.634615
7	0.438596	0.561404
8	0.509091	0.490909
9	0.45614	0.54386
10	0.47541	0.52459
11	0.457627	0.542373
12	0.442308	0.557692
13	0.446429	0.553571
14	0.5	0.5
15	0.446429	0.553571
16	0.423077	0.576923
17	0.444444	0.555556
18	0.586207	0.413793
19	0.403509	0.596491
20	0.545455	0.454545
21	0.508475	0.491525
22	0.457627	0.542373
23	0.465517	0.534483
24	0.527273	0.472727
25	0.4375	0.5625
26	0.418182	0.581818
27	0.491525	0.508475
28	0.357143	0.642857
29	0.533333	0.466667
30	0.45614	0.54386
31	0.520833	0.479167
32	0.518519	0.481481
33	0.40678	0.59322
34	0.534483	0.465517
35	0.473684	0.526316
36	0.388889	0.611111
37	0.483333	0.516667
38	0.518519	0.481481
39	0.5	0.5
40	0.471698	0.528302
41	0.454545	0.545455
42	0.481481	0.518519
43	0.490566	0.509434
44	0.377358	0.622642
45	0.5	0.5
46	0.55	0.45
47	0.415094	0.584906
48	0.388889	0.611111
49	0.517241	0.482759
50	0.490566	0.509434

Table 4.7: Results of the TSDm method with “equal time”.

implementation of the tool are explained later on Section 4.5; here we will only discuss the results of this comparison.

We decided to make an extensive comparison of our measure versus the possible versions of the TSDm measure. Specifically, we decided to compare BMI with the Input-TSDm (ITSDm), the Output-TSDm (OTSDm) and the InputOutput-TSDm (IOTSDm). This way, we cover all the possibilities that we have to use the TSDm measure when working in a black-box scenario. Also, in order to address some of the threats to the validity of our results that we will discuss later, we changed the `FSM`s that the tool provides to make the comparison by the 100 `FSM`s that we have been using in our experiments.

The results can be seen in Tables 4.8 (ITSDm), 4.9 (OTSDm) and 4.10 (IOTSDm). As we can see, our measure got better results in all the three comparisons. For ITSDm, BMI killed more mutants 70.5488% of the times, killing an average of 48.5651% of the mutants, while ITSDm killed more mutants 29.4512% of the times, killing an average of 45.5551% of the mutants. Similarly, for OTSDm, BMI killed more mutants 71.0744% of the times, killing an average of 49.2625% of the mutants, while OTSDm killed more mutants 28.9256% of the times, killing an average of 45.1924% of the mutants. Finally, for IOTSDm, BMI killed more mutants 66.9494% of the times, killing an average of 48.8414% of the mutants, while IOTSDm killed more mutants 33.0506% of the times, killing an average of 46.5447% of the mutants.

With these results, we can conclude that the measure that gets a better performance against BMI is IOTSDm. In any case, none of them managed to get better results than our measure in the majority of the runs. Therefore, we can claim that our measure is better, in order to generate tests suites, than the test set diameter measures.

4.4 Research questions answers

As a recap of all the results that we obtained along all the experiments, we can answer the Research Questions we performed at the beginning of this chapter.

Run Number	Percentage of success BMI	Percentage of success ITSDm	Percentage of killed mutants by BMI	Percentage of killed mutants by ITSDm
1	0.72619	0.27381	0.501798	0.446298
2	0.666667	0.333333	0.466654	0.479782
3	0.730769	0.269231	0.493308	0.418359
4	0.725	0.275	0.512737	0.462038
5	0.666667	0.333333	0.425722	0.459042
6	0.682353	0.317647	0.468518	0.498447
7	0.692308	0.307692	0.466859	0.454359
8	0.702703	0.297297	0.452919	0.431595
9	0.733333	0.266667	0.498853	0.42044
10	0.830986	0.169014	0.509408	0.368718
11	0.653846	0.346154	0.464744	0.481641
12	0.75	0.25	0.453184	0.436276
13	0.613333	0.386667	0.48644	0.520147
14	0.703704	0.296296	0.500951	0.459642
15	0.696203	0.303797	0.518215	0.463215
16	0.707317	0.292683	0.515537	0.459268
17	0.75641	0.24359	0.475551	0.440897
18	0.670886	0.329114	0.509975	0.478494
19	0.728571	0.271429	0.477114	0.426729
20	0.7	0.3	0.513737	0.475263
21	0.701299	0.298701	0.525195	0.437857
22	0.736842	0.263158	0.458303	0.414711
23	0.666667	0.333333	0.485393	0.477429
24	0.7	0.3	0.525543	0.476471
25	0.691358	0.308642	0.412543	0.472506
26	0.68	0.32	0.47492	0.440453
27	0.736111	0.263889	0.437694	0.44525
28	0.697368	0.302632	0.513342	0.466895
29	0.746667	0.253333	0.507933	0.440147
30	0.666667	0.333333	0.490747	0.432093
31	0.6375	0.3625	0.425737	0.48605
32	0.675676	0.324324	0.488392	0.473122
33	0.679487	0.320513	0.517949	0.469654
34	0.642857	0.357143	0.496655	0.521512
35	0.773333	0.226667	0.513773	0.422067
36	0.75	0.25	0.560369	0.419286
37	0.662338	0.337662	0.496286	0.493156
38	0.842105	0.157895	0.52675	0.373513
39	0.763158	0.236842	0.455658	0.415053
40	0.696203	0.303797	0.519494	0.454
41	0.682927	0.317073	0.490073	0.503963
42	0.618421	0.381579	0.425632	0.477
43	0.769231	0.230769	0.457026	0.405397
44	0.707317	0.292683	0.518951	0.436354
45	0.684932	0.315068	0.397822	0.486041
46	0.782051	0.217949	0.510462	0.407321
47	0.675	0.325	0.443737	0.498713
48	0.728395	0.271605	0.484185	0.45637
49	0.692308	0.307692	0.502397	0.483474
50	0.671053	0.328947	0.490763	0.485961

Table 4.8: Results of the comparison BMI vs ITSDm.

Run Number	Percentage of success BMI	Percentage of success OTSDm	Percentage of killed mutants by BMI	Percentage of killed mutants by OTSDm
1	0.671053	0.328947	0.469737	0.469513
2	0.716049	0.283951	0.443963	0.444654
3	0.706667	0.293333	0.509373	0.433827
4	0.6625	0.3375	0.51235	0.470938
5	0.772152	0.227848	0.492114	0.413241
6	0.7375	0.2625	0.4797	0.432113
7	0.82716	0.17284	0.514617	0.376037
8	0.662162	0.337838	0.497973	0.474189
9	0.65	0.35	0.455862	0.497288
10	0.72	0.28	0.424533	0.427547
11	0.768293	0.231707	0.500756	0.433927
12	0.674699	0.325301	0.516904	0.496217
13	0.688312	0.311688	0.478065	0.439857
14	0.716216	0.283784	0.507824	0.455189
15	0.743902	0.256098	0.509695	0.420878
16	0.74026	0.25974	0.467909	0.428377
17	0.73494	0.26506	0.500229	0.452181
18	0.697368	0.302632	0.479408	0.489118
19	0.693333	0.306667	0.49524	0.4992
20	0.717949	0.282051	0.429705	0.445808
21	0.631579	0.368421	0.414434	0.493724
22	0.675	0.325	0.511787	0.420313
23	0.675325	0.324675	0.504247	0.472584
24	0.783784	0.216216	0.589486	0.415959
25	0.6	0.4	0.4587	0.516563
26	0.649351	0.350649	0.459545	0.469156
27	0.739726	0.260274	0.515932	0.43737
28	0.703704	0.296296	0.496481	0.437605
29	0.691358	0.308642	0.477136	0.486815
30	0.717949	0.282051	0.466821	0.448103
31	0.684932	0.315068	0.47074	0.491438
32	0.72	0.28	0.547787	0.481773
33	0.712329	0.287671	0.493822	0.439699
34	0.74359	0.25641	0.554	0.423615
35	0.674419	0.325581	0.49086	0.461756
36	0.695122	0.304878	0.534622	0.466829
37	0.704225	0.295775	0.482155	0.462211
38	0.782051	0.217949	0.457526	0.398359
39	0.74359	0.25641	0.451513	0.433064
40	0.771429	0.228571	0.517071	0.401386
41	0.692308	0.307692	0.472885	0.468487
42	0.689189	0.310811	0.521703	0.475824
43	0.671053	0.328947	0.475263	0.488539
44	0.675	0.325	0.511925	0.458288
45	0.779221	0.220779	0.512506	0.425143
46	0.72973	0.27027	0.493946	0.460041
47	0.68	0.32	0.52008	0.442267
48	0.716216	0.283784	0.519041	0.453149
49	0.797297	0.202703	0.504824	0.411757
50	0.716049	0.283951	0.522815	0.452642

Table 4.9: Results of the comparison BMI vs OTSDm.

Run Number	Percentage of success BMI	Percentage of success IOTSDm	Percentage of killed mutants by BMI	Percentage of killed mutants by IOTSDm
1	0.567901	0.432099	0.418852	0.513173
2	0.625	0.375	0.4545	0.475488
3	0.641026	0.358974	0.492064	0.464551
4	0.684932	0.315068	0.467438	0.476863
5	0.716216	0.283784	0.530662	0.389378
6	0.716049	0.283951	0.52179	0.436543
7	0.675676	0.324324	0.447986	0.473419
8	0.679012	0.320988	0.529741	0.475951
9	0.597403	0.402597	0.438831	0.529481
10	0.7125	0.2875	0.527137	0.447488
11	0.658228	0.341772	0.492418	0.459937
12	0.753247	0.246753	0.470026	0.410766
13	0.697368	0.302632	0.463329	0.444382
14	0.65	0.35	0.437888	0.494113
15	0.719512	0.280488	0.543902	0.4525
16	0.577465	0.422535	0.443915	0.557099
17	0.60274	0.39726	0.431137	0.497493
18	0.609756	0.390244	0.455159	0.463098
19	0.690476	0.309524	0.502607	0.451488
20	0.641975	0.358025	0.409284	0.486914
21	0.632911	0.367089	0.500949	0.534797
22	0.658824	0.341176	0.491482	0.454553
23	0.717949	0.282051	0.512449	0.433615
24	0.679487	0.320513	0.475872	0.445218
25	0.581081	0.418919	0.443757	0.523919
26	0.716216	0.283784	0.523041	0.425973
27	0.689189	0.310811	0.469608	0.437378
28	0.690141	0.309859	0.47493	0.414662
29	0.620253	0.379747	0.508367	0.489722
30	0.728395	0.271605	0.52042	0.422926
31	0.64557	0.35443	0.44357	0.478582
32	0.75	0.25	0.476512	0.42345
33	0.62963	0.37037	0.450123	0.449432
34	0.64	0.36	0.493253	0.500293
35	0.658228	0.341772	0.472646	0.49681
36	0.714286	0.285714	0.546117	0.456221
37	0.662338	0.337662	0.487714	0.489195
38	0.654321	0.345679	0.476864	0.465383
39	0.686747	0.313253	0.497819	0.446771
40	0.734177	0.265823	0.52738	0.424582
41	0.666667	0.333333	0.487097	0.464514
42	0.657143	0.342857	0.456143	0.465343
43	0.765432	0.234568	0.554852	0.431062
44	0.716216	0.283784	0.553243	0.434959
45	0.726027	0.273973	0.564192	0.445164
46	0.620253	0.379747	0.49138	0.498835
47	0.616279	0.383721	0.492628	0.533244
48	0.689189	0.310811	0.521905	0.4385
49	0.688312	0.311688	0.550091	0.456753
50	0.623377	0.376623	0.471182	0.487455

Table 4.10: Results of the comparison BMI vs IOTSDm.

Research Question 1. *Are lower levels of biased mutual information associated with higher fault coverage?*

The answer to this question is affirmative: lower levels of BMI are associated (even correlated) with higher fault coverage. We can conclude this from Tables 4.3 and 4.5, where we can observe that our measure is negatively correlated to the mutation score of the tests, with a correlation of -0.369134 for `FSM`s with an alphabet of size 5 and a correlation of -0.650256 for `FSM`s with an alphabet of size 25.

Research Question 2. *Have test suites selected by biased mutual information higher fault coverage than the ones randomly selected?*

The answer to this question is affirmative because BMI selects test suites with higher fault coverage than random selection. We can conclude this from Tables 4.1 and 4.2, where we can observe how our measure selects the best test suite 62.4924% of the times in a scenario including `FSM`s with an alphabet of size 5 (even if the test suite has repeated tests) and from Table 4.4, where we can observe how our measure selects the best test suite 75.0605% of the times in an scenario having `FSM`s with an alphabet of size 25.

Research Question 3. *Have test suites selected by biased mutual information higher fault coverage than the ones selected by test set diameter (TSDm)?*

The answer to this question is affirmative. In fact, BMI selects test suites with higher fault coverage than TSDm in any of its black-box versions. We can conclude this from Tables 4.6 and 4.7, where we can observe that our measure outperforms the ITSDm measure using the TSDm selection method, even when we restrict the selecting pool for our measure. Tables 4.8, 4.9 and 4.10 show that our measure gets better results than ITSDm, OTSDm and IOTSDm, respectively.

Research Question 4. *How does the time to execute the selection method scale as the size of the test suite increases? How does the time needed to compute the selection method relate to the time needed to apply a test suite?*

Finally, the answer to this question is that the time needed to compute our measure scales (approximately) quadratically with respect to the size of the test suite. Also, the time needed to compute the selection method is smaller than the time needed to apply a test suite as long as we need at least 0.001 seconds to apply each transition. We can conclude this from Table 4.4, where we can observe that our measure needs an average of 0.00083786 seconds to be computed. Figure 4.2 also shows how the curve of the time to compute our measure scales approximately quadratically with respect to the size of the test suite, while the time to execute a test suite is much higher than the time needed to compute our selection method as long as we need at least 0.001 seconds to apply each transition.

4.5 A tool to compare different measures

In order to compare our measure with other measures, we developed a comparison tool based on Genetic Programming. The idea is to generate two tests suites using a Genetic Programming algorithm but using as fitness functions different measures. The results appearing in this part of the document have been recently published [25].

4.5.1 The Genetic Programming algorithm

In this section we will present all the components of our genetic algorithm.

Encoding

The first and most important choice of an approach based on genetics is to select a good encoding. As we are working with test suites generated from an `FSM`, we need to preserve the structure of the `FSM` in order to generate correct tests for it. Therefore, we decided to use a tree structure as an encoding of our tests and we use a Genetic Programming algorithm. Specifically, we decided to use a grammar-guided Genetic Programming approach [30, 33], which solves the correctness issues from just using Genetic Programming. This implies that the first step of our algorithm will be to generate the grammar that the `FSM` produces. We have the following components:

- A start non-terminal symbol S that starts the grammar.
- A non-terminal symbol T that introduces each test of the test suite.
- A non-terminal symbol N for each state, where N is the state name.
- A terminal symbol $'a/b'$ for each (input/output) pair present on the `FSM`, where a is the input and b is the output.
- A terminal symbol $'null'$ to represent the end of a test.
- A production rule $S \longrightarrow T$ to generate the initial test.
- A production rule $T \longrightarrow T + T$ to introduce a new test.
- A production rule $T \longrightarrow q_0$ to start each test in the `FSM` initial state.
- A production rule $N \longrightarrow 'a/b' + M$ for each transition from the state N to a state M with (input/output) pair (a, b).
- A production rule $N \longrightarrow 'null'$ for each state N to a terminal to represent the end of the test.

With this components we produce the following meta-grammar:

$$\begin{array}{lcl} S & \longrightarrow & T \\ T & \longrightarrow & T + T \mid q_0 \\ N & \longrightarrow & 'a/b' + M \mid 'null' \quad with\ N, M \in Q,\ a \in I,\ b \in O \end{array}$$

where Q will be the set of states of the `FSM`, I will be the set of inputs and O will be the set of outputs of the `FSM`.

This meta-grammar will define all the possible grammars that we can generate, one for each possible specification of a system. Given an `FSM`, the generation of the associated grammar is automatic (and it has been implemented as part of our tool). For example, if we consider the `FSM` depicted in Figure 4.3, our tool would produce the grammar depicted in 4.4.

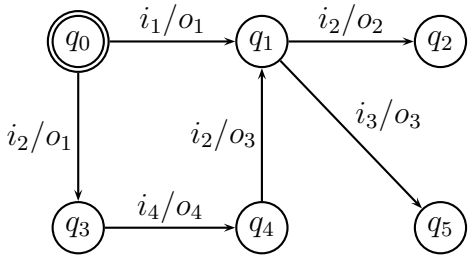

Figure 4.3: Example of `FSM` for grammar generation.

- $S \mapsto T$
- $T \mapsto T + T \mid q_0$
- $q_0 \mapsto$ 'i_1/o_1' $+ q_1 \mid$ 'i_2/o_1' $+ q_3 \mid$ '$null$'
- $q_1 \mapsto$ 'i_2/o_2' $+ q_2 \mid$ 'i_3/o_3' $+ q_5 \mid$ '$null$'
- $q_2 \mapsto$ '$null$'
- $q_3 \mapsto$ 'i_4/o_4' $+ q_4 \mid$ '$null$'
- $q_4 \mapsto$ 'i_2/o_3' $+ q_2 \mid$ '$null$'
- $q_5 \mapsto$ '$null$'

Figure 4.4: Grammar generated by our meta-grammar.

Note that the non-terminals S and T are artificially introduced to get a grammar that defines a test suite, although a test is defined only with the elements derived from the `FSM`. We can see clearly this behaviour in Figure 4.5. It is an example of a test suite generated by the grammar defined in Figure 4.4.

Initial population

As initial population we randomly generate 100 test suites, with a total number of inputs given by the user, using the grammar previously derived from the `FSM`. Each rule in the grammar has the same probability of being triggered. This allows a uniform random initialization.

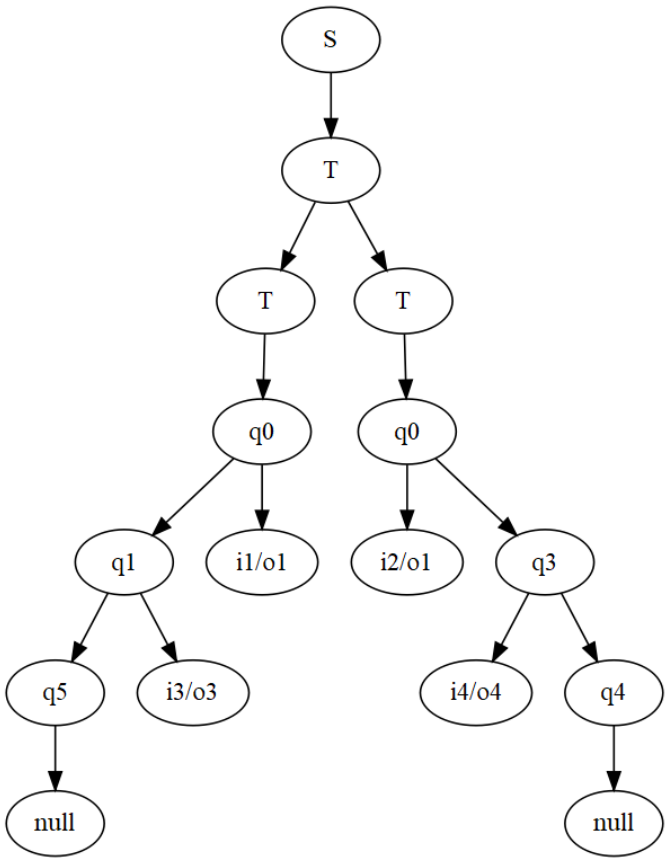

Figure 4.5: Example of test suite generated by the grammar.

Fitness function

The fitness function of our algorithm will be one of the available measures. As previously explained, the fitness function will receive the test suite and the `FSM` and will return a real value that represents how *good* is this test suite according to the measure. An important remark about fitness functions is that they should be easy to compute, as they will be invoked many times during the execution of the algorithm. Therefore, fitness functions with high computational cost will lead to a higher computational cost of the algorithm.

We decided to give the users the capability to select the fitness function that better suites their problem, along with the decision on whether the score should be maximized or minimized. As we explained before, fitness functions should have similar performances

and this is the case for the measures based on Information Theory that we include in our tool. Among them, we can mention, due to the big improvement with respect to previous measures, the *Test Set Diameter* (TSDm) based measures [16]. We implemented the Input-TSDm, the Output-TSDm and the InputOutput-TSDm. Also, we implemented our *Biased Mutual Information*. Let us emphasize that users of our tool can add their own measures. So, our tool can be used to evaluate the usefulness of new proposals because they can be compared with the existing ones.

Stopping criterion

The algorithm performs at most 100 epochs and at least 20 epochs. Once we have passed the 20 epochs, the stop criterion will be fulfilled if the best test suite is the same along $0.2 \times NumberOfPassedEpochs$ epochs.

Selection method

We use a variant of elitist reduction [35]. First, the test suites that obtained a fitness score over the mean (or under the mean if we want to minimize) go directly to the next epoch. In addition, the ones that are under the mean can pass to the next epoch if their score is higher than the mean minus a random number modulo the distance between the mean and the best score.

Crossover method

The choice of the crossover method depends on our encoding and the characteristics that we want the produced test suites to have. As we use a grammatical encoding, we need to use a grammatical crossover. We have considered a mixture between the Whigham crossover [33] and the standard grammatical crossover [12]. Also, as we want all our test suites to have the same length (as previously said), we need to slightly modify crossovers in order to achieve a crossover that keeps the length fixed. Algoritm 3 shows how crossover is performed.

Finally, we need to set the probability of producing a crossover. In our case, giving how

```
Data: TS1, TS2 test suites
Result: Crossover of TS1 and TS2
match = false;
while !match do
    Select a random node t1 from TS1;
    for each node t2 of TS2 do
        if t2 non-terminal == t1 non-terminal and t2 length == t1 length then
            Set t2 as valid node.
        end
    end
    if valid nodes > 0 then
        match = true;
    end
end
Select a random valid node t2;
Get parent p1 of t1;
Get parent p2 of t2;
Set t2 as child of p1;
Set t1 as child of p2;
```

Algorithm 3: Crossover algorithm.

hard is that two test suites match the conditions to perform a crossover, we decided to set this probability to 90%, so that we favour the mixture between test suites.

Mutation method

A mutation consists in generating a new test with the same length. The probability of performing a mutation will be, as usual [35], equal to 5% for each test of each test suite of the population.

4.5.2 The tool

We have implemented a tool[7] supporting this framework. The tool has two main uses: generate a test suite with a giving length according to a selected measure and compare different measures. In order to develop the tool, we also used the OpenFST library [1] (as we did in the experiments). Therefore, input files must be in OpenFST format, with the

[7]The tool can be downloaded from https://github.com/Colosu/gptsg.

.fst extension. The tool will have two types of calls: *generate* and *compare*. The syntax of the two calls is:

```
gptsg generate inputFile length {max|min} fitness
gptsg compare length {max|min} fitness {max|min} fitness
```

and two examples of calls are:

```
gptsg generate ./test/binary.fst 50 max ITSDm
gptsg compare 50 max ITSDm min OTSDm
```

Currently, as we have already said, our tool supports the following fitness functions:

- BMI: Biased Mutual Information.
- ITSDm: Input Test Set Diameter.
- OTSDm: Output Test Set Diameter.
- IOTSDm: Input-Output Test Set Diameter.
- Own: For your own developed measure.
- Random: generates a totally random test suite.

Let us emphasize that an important feature of our tool is the possibility of defining new measures, so that they can be compared with the already existing ones. The user only needs to open the `src/Measures.cpp` file and modify the `OwnFunction` method. Once the code is compiled, the inserted measure can be called as the *Own* fitness function.

Test suite generation

In order to generate a *good* enough test suite, we implemented the Genetic Programming algorithm explained in Section 4.5.1, giving some configuration options to the user. Specifically, the tool needs to know the file containing the `FSM`, the length of the expected test suite in terms of input actions, the measure to use as a fitness function and whether it needs to maximize or minimize the measure. Then, the user will receive a .txt file with the generated

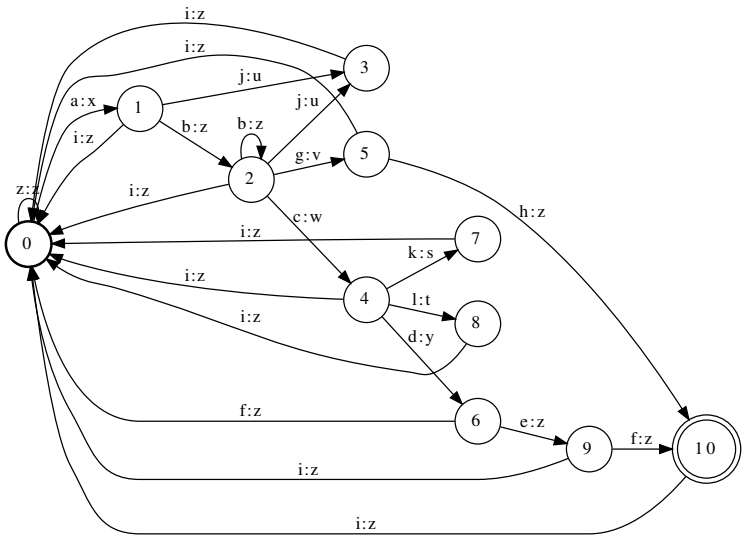

Figure 4.6: Example of `FSM`.

test suite, with each test conformed by a succession of (input/output) pairs. An example of this file with a test suite of length 50 generated for the `FSM` given in Figure 4.6 can be found in Figure 4.7. Note that Figure 4.6 is automatically generated by the OpenFST tool.

Test suite comparison

The tool allows users to compare two measures. For this option of our tool, the user needs to give the length of the desired test suite, the two measures to be compared and whether the tool should maximize or minimize each measure. Essentially, the tool takes the set of 100 `FSM`s that are shipped with the tool, representing different and diverse scenarios and characteristics, and it generates, for each of them, two test suites according to the corresponding measures. Then, the tool produces 1000 mutants of the corresponding `FSM` and checks which test suite kills more mutants. With the results for each `FSM`, the tool produces an output indicating the percentage of cases where each test suite has killed more mutants, along with a percentage of how many mutants where killed by each test suite. This process is repeated 50 times, getting 50 results, and the program returns a mean of all the results obtained for the 50 repetitions. In Algorithm 4 we present an algorithmic view

```
Test 0:
97 —> 120
98 —> 122
98 —> 122
99 —> 119
100 —> 121
101 —> 122
102 —> 122
105 —> 122
97 —> 120
105 —> 122
97 —> 120
98 —> 122
99 —> 119
100 —> 121
101 —> 122
102 —> 122
105 —> 122
122 —> 122
97 —> 120
106 —> 117
105 —> 122
97 —> 120
106 —> 117
105 —> 122
122 —> 122
122 —> 122
122 —> 122
```

```
Test 1:
97 —> 120
106 —> 117
105 —> 122
97 —> 120
98 —> 122
99 —> 119
100 —> 121
101 —> 122
102 —> 122
105 —> 122
122 —> 122
97 —> 120
98 —> 122
98 —> 122
103 —> 118
104 —> 122
105 —> 122
122 —> 122
97 —> 120
98 —> 122
103 —> 118
104 —> 122
105 —> 122
```

Figure 4.7: Example of test suite generated by the tool.

of this process.

Data: $length, measure1, measure2$
Result: .txt file with the values
$REP = 50$;
`FSM` $= 100$;
for *each* REP **do**
    Set control values to 0;
    for *each* `FSM` F **do**
        Generate TS_1 genetic test suite using measure $measure1$;
        Generate TS_2 genetic test suite using measure $measure2$;
        Generate 1000 mutants of F;
        Check which test suite kills more mutants;
    end
    Output the percentage of runs TS_1 killed more mutants;
    Output the percentage of runs TS_2 killed more mutants;
    Output the percentage of mutants killed by TS_1;
    Output the percentage of mutants killed by TS_2;
end
Output the average percentage of runs TS_1 killed more mutants;
Output the average percentage of runs TS_2 killed more mutants;
Output the average percentage of mutants killed by TS_1;
Output the average percentage of mutants killed by TS_2;

Algorithm 4: Test suite comparison algorithm.

Chapter 5

Threats to validity

In this chapter we discuss the possible threats to the validity of the results of our experiments.

Concerning threats to internal validity, which consider uncontrolled factors that might be responsible for the obtained results, the main threat is associated with the possible faults in the developed tools because they could lead to misleading results. In order to reduce the impact of this threat we tested our code with carefully constructed examples for which we could manually check the results. In addition, we repeated the experiment many times in order to get a mean so that the randomization impact is reduced. Another important threat was the processor reschedule policy for our process because it can affect the registered times. In order to reduce the impact of this threat, we abstracted the time computation and only computed small enough time values so that the reschedule policy does not affect them. In addition, we computed the mean of these values to reduce the impact of outliers. Another threat was that the NCD that we used to compare our measure works bad with short strings. In order to address this threat we worked with long enough strings so that the performance of NCD was not deteriorated by it (for a test suite of size 100 we are talking about strings of $(100\ inputs + 100\ outputs) \times 2\ characters\ per\ output = 400\ characters$). Finally, the last identified threat is the difficulty of the scenario used for the experiments. In this case, the size of the alphabet of the `FSM`s is specially relevant. In order to address this threat we performed the experiments with different alphabet sizes, as we showed in Chapter 4.

The main threat to external validity, which concern conditions that allow us to generalise our findings to other situations, is the different possible representations of a black-box component as an FSM. Such a threat cannot be entirely addressed since the population of FSMs is unknown and it is not possible to sample from this (unknown) population. In order to reduce the impact of this threat we used randomly generated FSMs. Also, a minor external threat is the fact that our measure relays in the combination of inputs and outputs into pairs to differentiate the transitions. It is likely that using bigger alphabets for our randomly generated FSMs, it will be harder to find an (input/output) pair repeated. Therefore, to address this threat we developed our FSMs with small alphabets, as stated in Chapter 4

Finally, we considered threats to construct validity, which are related to the *reality* of our experiments, that is, whether our experiments reflect real-world situations. In our work, the main construct threat is whether the FSMs used in the experiments correspond to possible system components. In order to reduce the impact of this threat, we restricted our range of FSM samples to connected deterministic machines. In future work we plan to extend our framework and experiments to deal with features such as non-determinism and consider a set of FSMs representing real systems.

Chapter 6

Final discussion: alternative definitions

We have shown that our measure is interesting, useful and that the time needed to compute it is negligible when compared to the time needed to apply extra testing. However, it is possible to consider that some of the design decisions were not optimal. For example, one might consider that the translation of the mutual information formula and the computation of the $\sigma_{\xi_A}(x)$ pseudo-probability distribution values are not justified. In order to evaluate these concerns, we repeated the initial experiment with the new considerations; we used `FSM`s with an alphabet of 25 elements (as in the experiment where we computed time) in order to be able to see bigger differences.

We also studied alternative options for one formula. Specifically, we may use the original mutual information formula together with a proper translation in the X axis. Formally, the new formulation would be given by:

$$BMI_2(\xi_{t_1};\xi_{t_2}) = \sum_{\substack{y\in t_2 \\ y\in_m M}} \sum_{\substack{x\in t_1 \\ x=y}} \frac{\log_2(m+2)}{m+2}$$

Another variation consist in computing the probabilities $\sigma_{\xi_A}(z)$ as the probability of the x pair to be one of the (input/output) pairs with the same value on the test suite, assuming a uniform distribution over them. Formally, we might consider that these probabilities are given by:

$$\sigma_{\xi_A}(x) = \frac{1}{\#\text{test suite } IO \text{ pairs with label } x}$$

Test	**# 0.4%**	**# 0.5%**	**# 0.6%**	**# 0.7%**	**# 0.8%**	**min value**	**max value**	**% success (mean)**
MI based on spec	4	33	13	0	0	0.459184	0.680851	56.9662%
MI based on test suite	0	0	3	41	6	0.666667	0.818182	75.0757%
BMI based on spec	**0**	**0**	**5**	**36**	**9**	**0.673469**	**0.838384**	**75.3883%**
BMI based on test suite	0	0	5	32	13	0.666667	0.848485	76.4054%
BMI_2 based on spec	0	0	14	31	5	0.66	0.816327	74.2696%
BMI_2 based on test suite	0	0	1	43	6	0.670103	0.848485	75.1613%
BMI_3	3	22	24	1	0	0.42268	0.7	58.9412%

Table 6.1: Comparing different alternative approaches.

A final variation uses as pseudo-probability distribution values the formula $\frac{n}{m}$ instead of $\frac{1}{m}$ (where $x \in_n t$ and $x \in_m M$). This way, we take into account also how many times the test suite is repeated in the test. This formula is defined by:

$$BMI_3(\xi_{t_1};\xi_{t_2}) = \sum_{\substack{y \in_{n_2} t_2 \\ y \in_m M}} \sum_{\substack{x \in_{n_1} t_1 \\ x=y}} n_1 \cdot n_2 \cdot \frac{\log_2(m+1)}{m}$$

The results are given in Table 6.1 and show that from the seven possible combinations, five are more or less equally *good*, and the other two are clearly worse. Therefore, we decided to keep our approach as we think that it keeps a good balance between intuition and being faithful to the Information Theory original formulas.

Finally, another important choice was the decision to not use true random variables and its probability distributions at all in the mutual information formula. This was done for two reasons: the difficulty to find two random variables over two distinct tests with some kind of correlation (as we already explained in Chapter 3) and the difficulty to use alternative methods to define the random variables that get better results than the alternative proposed in this work.

The alternative methods that we explored were combinations of some mechanisms for generating those correlated random variables.

The first and more important mechanism was *normalization*. With the application of normalization we have a probability distribution summing up to 1. It consists on summing up all the values of the (input/output) pairs of the test and then divide each (input/output) pair by this factor. This way, the sum of the probabilities of all the (input/output) pairs of

the test is equal to 1.

Another mechanism was the use of the *number of times* each (input/output) pair appears in the test. Using this number we have that the values are reweighed by the test. Therefore, we simplify the task of defining random variables over the (input/output) pairs of the test as now it depends more on the test properties. The downside of this mechanism was that we lose the intuition that we followed during the thesis.

Finally, the last mechanism was to force *correlation*. In order to do so, we give a random variable and its probability distribution to each test. Then, we compute the joint probability of both tests with the uncorrelated (input/output) pairs of each test and add all these values. This is an easy process because the joint probability of uncorrelated (input/output) pairs is given by the product of the probabilities of each (input/output) pair. As the sum of all the values of the joint probability should sum up to 1, we know the amount of probability corresponding to the correlated (input/output) pairs (we will call it P). Then, we defined s as the product of the probabilities of each (input/output) pair modified by a factor and we define the joint probability of the correlated (input/output) pairs as s divided by the sum of all the s's and multiplied by P. This give us a joint probability for each pair of correlated (input/output) pairs that we can call "of correlated (input/output) pairs". It is important to remark that this joint probability is different from the product of the probabilities of each (input/output) pair; otherwise, in the mutual information formula we would get $\log_2(1)$ and giving the fact that this is equal to 0 we would have that mutual information would be 0.

With these mechanisms, we tried different options by combining them. Those alternatives are displayed in Table 6.2. The column "dist" corresponds to the probability distribution formula, while "joint" corresponds to the joint distribution formula for the correlated (input/output) pairs (for the uncorrelated (input/output) pairs, the joint distribution is the product of the individual distributions). In all the table we assume $x_1 \in_{n_1} t_1$, $x_2 \in_{n_2} t_2$, $x_1 \in_m M$, $x_2 \in_m M$ and $P = 1 - S_1$.

Despite all the alternative formulations that we considered, some of them notably more

#	dist	joint	$\mathbf{s_1}$	$\mathbf{s_2}$	$\mathbf{S_1}$	$\mathbf{S_2}$	success
1	$\frac{1}{m \cdot s}$	$\frac{n_1}{m \cdot s_1} \cdot \frac{n_2}{m \cdot s_2} \cdot \frac{P}{S2}$	$\sum_{\substack{x_1 \in t_1 \\ x_1 \in M}} \frac{1}{m_1}$	$\sum_{\substack{x_2 \in t_2 \\ x_2 \in M}} \frac{1}{m_2}$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 \neq x_2}} \frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2}$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 = x_2}} \frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2}$	64%
2	$\frac{n}{s}$	$\frac{min(n_1, n_2)}{S_2}$	$\sum_{\substack{x_1 \in t_1 \\ x_1 \in M}} n_1$	$\sum_{\substack{x_2 \in t_2 \\ x_2 \in M}} n_2$	0	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 = x_2}} min(n_1, n_2)$	55%
3	$\frac{n}{s}$	$\frac{n_1}{s_1} \cdot \frac{n_2}{s_2} \cdot \frac{1}{m_1} \cdot \frac{P}{S2}$	$\sum_{\substack{x_1 \in t_1 \\ x_1 \in_{m_1} M}} n_1$	$\sum_{\substack{x_2 \in t_2 \\ x_2 \in_{m_2} M}} n_2$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 \neq x_2}} \frac{n_1}{s_1} \cdot \frac{n_2}{s_2}$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 = x_2}} \frac{n_1}{s_1} \cdot \frac{n_2}{s_2} \cdot \frac{1}{m_1}$	54%
4	$\frac{n}{s}$	$\frac{n_1}{s_1} \cdot \frac{n_2}{s_2} \cdot m_1 \cdot \frac{P}{S2}$	$\sum_{\substack{x_1 \in t_1 \\ x_1 \in_{m_1} M}} n_1$	$\sum_{\substack{x_2 \in t_2 \\ x_2 \in_{m_2} M}} n_2$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 \neq x_2}} \frac{n_1}{s_1} \cdot \frac{n_2}{s_2}$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 = x_2}} \frac{n_1}{s_1} \cdot \frac{n_2}{s_2} \cdot m_1$	58%
5	$\frac{1}{m \cdot s}$	$\frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2} \cdot \frac{1}{m_1} \cdot \frac{P}{S2}$	$\sum_{\substack{x_1 \in t_1 \\ x_1 \in_{m_1} M}} \frac{1}{m_1}$	$\sum_{\substack{x_2 \in t_2 \\ x_2 \in_{m_2} M}} \frac{1}{m_2}$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 \neq x_2}} \frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2}$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 = x_2}} \frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2} \cdot \frac{1}{m_1}$	57%
6	$\frac{1}{m \cdot s}$	$\frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2} \cdot m_1 \cdot \frac{P}{S2}$	$\sum_{\substack{x_1 \in t_1 \\ x_1 \in_{m_1} M}} \frac{1}{m_1}$	$\sum_{\substack{x_2 \in t_2 \\ x_2 \in_{m_2} M}} \frac{1}{m_2}$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 \neq x_2}} \frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2}$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 = x_2}} \frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2} \cdot m_1$	55%
7	$\frac{n}{m \cdot s}$	$\frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2} \cdot \frac{1}{m_1} \cdot \frac{P}{S2}$	$\sum_{\substack{x_1 \in t_1 \\ x_1 \in_{m_1} M}} \frac{n_1}{m_1}$	$\sum_{\substack{x_2 \in t_2 \\ x_2 \in_{m_2} M}} \frac{n_2}{m_2}$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 \neq x_2}} \frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2}$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 = x_2}} \frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2} \cdot \frac{1}{m_1}$	56%
8	$\frac{n}{m \cdot s}$	$\frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2} \cdot m_1 \cdot \frac{P}{S2}$	$\sum_{\substack{x_1 \in t_1 \\ x_1 \in_{m_1} M}} \frac{n_1}{m_1}$	$\sum_{\substack{x_2 \in t_2 \\ x_2 \in_{m_2} M}} \frac{n_2}{m_2}$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 \neq x_2}} \frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2}$	$\sum_{\substack{x_1 \in t_1 \\ x_2 \in t_2 \\ x_1 = x_2}} \frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2} \cdot m_1$	55%

Table 6.2: Comparing different probability distributions.

involved than our approach, all the results that we obtained where worse than the results we got with the notion presented in this thesis. All of the alternatives achieve a mean score between 50% and 60%, with only one distribution getting more than 60% (the one that we presented in Chapter 3) while the result for the same experiment of our chosen approach was 75.0605%.

This lack of good results using different alternatives based on mutual information actually confirms the claim stating that Normalized Compression Distance (based on Normalized Information Distance) is *universal* in the sense that it "discovers all effective feature similarities or cognitive similarities between two objects" [5, 16]. Mutual information should not be able to get better results than the Test Set Diameter measure (that is based on NCD) and, therefore, it should get worse results. However, our proposal of measure, that is not a true mutual information, gets better results than TSDm, as shown in the experiments.

This can be explained due to the fact that it does not rely on the test as the Information Theory based measures do. Therefore, it works out of the framework that sets TSDm as a universal measure.

Chapter 7

Conclusions

The selection of a test suite can be a critical task because the time and resources devoted to testing are limited. In this thesis we focused on developing a measure to select between test suites. We have proposed a measure based on mutual information. As we have shown with our experiments, our measure is clearly better both than random selection and than Test Set Diameter. In addition, the overhead needed to compute the measure can be assumed.

An intuition that explains why our measure is better than a true Information Theory based measure can be the following one: our measure gives a proportional value to each (input/output) pair independently of the rest of the test. That is, we are giving the same weight to the same (input/output) pair independently of the test size. However, when using Information Theory based measures, we rely on a probability distribution over the (input/output) pairs of the test. Therefore, the weights of the same (input/output) pair appearing in two different tests will be different. This produces undesirable effects like the decrease of the weight of a (input/output) pair due to being in a longer test than if it where in a shorter test. This means that longer test should have more repetitions of the same (input/output) pair in order to be as penalized as a shorter one. This can lead to situations where, for example, a test suite with a longer test with many repeated (input/output) pairs could be preferred instead of a test suite with many shorter tests with only one repeated (input/output) pair between all of them.

We have also addressed the problem of the automatic generation of good test suites.

This is a fundamental task when limitations in testing complex systems come into play. We have developed a Genetic Programming algorithm in order to generate *good enough* test suites measured using our measure (or any other test prioritization measure). We have implemented the algorithm in a tool that also allows us to compare the performance of different test prioritization measures.

Finally, we have discussed some elections made along the process of definition of our measure. We have shown that our proposal is better than several rational alternative options that we could have taken. Therefore, we can claim that our decisions are grounded on empirical evidence.

For future work we would like to compare our measure with (the adaption to our black-box framework of) other measures. We would like to consider exhaustive techniques such as the classical Wp method [10, 18]. We would also like to apply our measure in more complex scenarios. Also, we consider to explore how to take into account the number of times an (input/output) pair appears in the test, in a way that gives better results than the options explored in Chapter 6. Finally, another line of future work is the use of our measure, and even our tool, with real `FSM`s, that is, `FSM`s representing systems.

Considering our tool, we also have lines of future work for it. One of them is the definition and implementation of new measures so that we can extend the catalogue included by default in our tool.

Bibliography

[1] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In *9th Int. Conf. on Implementation and Application of Automata, CIAA'07, LNCS 4783*, volume 4783, pages 11–23. Springer, 2007.

[2] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *24th ACM SIGSOFT Int. Symposium on Software Testing and Analysis, ISSTA'14*, pages 181–192. ACM Press, 2014.

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

[4] C. Andrés, M. G. Merayo, and M. Núñez. Supporting the extraction of timed properties for passive testing by using probabilistic user models. In *9th Int. Conf. on Quality Software, QSIC'09*, pages 145–154. IEEE Computer Society, 2009.

[5] C. H. Bennett, P. Gács, M. Li, P. M. B. Vitányi, and W. H. Zurek. Information distance. *IEEE Transactions on Information Theory*, 44(4):1407–1423, 1998.

[6] R. V. Binder, B. Legeard, and A. Kramer. Model-based testing: where does it stand? *Communications of the ACM*, 58(2):52–56, 2015.

[7] J. K. Blundell, M. L. Hines, and J. Stach. The measurement of software design quality. *Annals of Software Engineering*, 4(1–4):235–255, 1997.

[8] E. G. Cartaxo, P. D. L. Machado, and F. G. de Oliveira Neto. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability*, 21(2):75–100, 2011.

[9] A. R. Cavalli, T. Higashino, and M. Núñez. A survey on formal active and passive testing with applications to the cloud. *Annales of Telecommunications*, 70(3-4):85–93, 2015.

[10] T. S. Chow. Testing software design modeled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.

[11] D. Clark, R. Feldt, S. M. Poulding, and S. Yoo. Information transformation: An underpinning theory for software engineering. In *37th IEEE/ACM International Conference on Software Engineering, ICSE'15*, pages 599–602, 2015.

[12] J. Couchet, D. Manrique, J. Rios, and A. Rodríguez-Patón. Crossover and mutation operators for grammar-guided genetic programming. *Soft Computing*, 11(10):943–955, 2007.

[13] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley Interscience, 1991.

[14] K. Derderian, M. G. Merayo, R. M. Hierons, and M. Núñez. Aiding test case generation in temporally constrained state based systems using genetic algorithms. In *10th Int. Conf. on Artificial Neural Networks, IWANN'09, LNCS 5517*, pages 327–334. Springer, 2009.

[15] 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.

[16] R. Feldt, S. M. Poulding, D. Clark, and S. Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *9th IEEE Int. Conf. on Software Testing, Verification and Validation, ICST'16*, pages 223–233. IEEE Computer Society, 2016.

[17] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *1st IEEE Int. Conf. on Software Testing Verification and Validation Workshops*, pages 178–186. IEEE Computer Society, 2008.

[18] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite-state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[19] M.-C. Gaudel. Testing can be formal, too! In *6th Int. Joint Conf. CAAP/FASE, Theory and Practice of Software Development, TAPSOFT'95, LNCS 915*, pages 82–96. Springer, 1995.

[20] A. González-Sanchez, É. Piel, H.-G. Groß, and A. J. C. van Gemund. Prioritizing tests for software fault localization. In *10th Int. Conf. on Quality Software, QSIC'10*, pages 42–51. IEEE Computer Society, 2010.

[21] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology*, 22(1):6:1–6:42, 2013.

[22] H. Hemmati, Z. Fang, and M. V. Mantyla. Prioritizing manual test cases in traditional and rapid release environments. In *8th IEEE Int. Conf. on Software Testing, Verification and Validation, ICST'15*, pages 1–10. IEEE Computer Society, 2015.

[23] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. Comparing white-box and black-box test prioritization. In *38th Int. Conf. on Software Engineering, ICSE'16*, pages 523–534. ACM Press, 2016.

[24] R. M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A.J.H Simons, S. Vilkomir, M.R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2):9:1–9:76, 2009.

[25] A. Ibias, D. Griñán, and M. Núñez. GPTSG: a Genetic Programming Test Suite Generator using Information Theory measures. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 716–728. Springer, 2019.

[26] A. Ibias, R. M. Hierons, and M. Núñez. Using squeeziness to test component-based systems defined as finite state machines. *Information & Software Technology*, 112:132–147, 2019.

[27] ISO/IEC JTCI/SC21/WG7, ITU-T SG 10/Q.8. Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing. Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500. ISO – ITU-T, 1996.

[28] K. A. De Jong and W. M. Spears. Using genetic algorithms to solve NP-complete problems. In *3rd Int. Conf. on Genetic Algorithms, ICGA'89*, pages 124–132. Morgan Kaufmann Publishers Inc., 1989.

[29] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Transactions on Software Engineering*, 44(4):308–333, 2018.

[30] J. R. Koza. *Genetic programming*. MIT Press, 1993.

[31] R. Lefticaru and F. Ipate. Automatic state-based test generation using genetic algorithms. In *9th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC'07*, pages 188–195. IEEE Computer Society, 2007.

[32] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, 2014.

[33] R. I. McKay, N. X. Hoai, P. A. Whigham, Y. S., and M. O'Neill. Grammar-based

genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010.

[34] A. V. Miranskyy, M. Davison, R. M. Reesor, and S. S. Murtaza. Using entropy measures for comparison of software traces. *Information Sciences*, 203:59–72, 2012.

[35] M. Mitchell. *An introduction to genetic algorithms*. MIT Press, 1998.

[36] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, 3rd edition, 2011.

[37] K. R. Pattipati and M. G. Alexandridis. Application of heuristic search and information theory to sequential fault diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(4):872–887, 1990.

[38] K. R. Pattipati, S. Deb, M. Dontamsetty, and A. Maitra. Start: System testability analysis and research tool. In *IEEE Conference on Systems Readiness Technology, 'Advancing Mission Accomplishment'*, pages 395–402, 1990.

[39] R. Sagarna, A. Arcuri, and X. Yao. Estimation of distribution algorithms for testing object oriented software. In *9th IEEE Congress on Evolutionary Computation, CEC'07*, pages 438–444. IEEE Computer Society, 2007.

[40] A. Samarah, A. Habibi, S. Tahar, and N. N. Kharma. Automated coverage directed test generation using a cell-based genetic algorithm. In *11th Annual IEEE Int. High-Level Design Validation and Test Workshop*, pages 19–26. IEEE Computer Society, 2006.

[41] M. Shafique and Y. Labiche. A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer*, 17(1):59–76, 2015.

[42] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[43] J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing, LNCS 4949*, pages 1–38. Springer, 2008.

[44] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transanctions on Software Enginnering and Methodology*, 22(3):19:1–19:29, 2013.