Using mutual information to test from Finite State Machines: test suite generation[☆]

Alfredo Ibias[a]

[a]*Design and Testing of Reliable Systems Research Group, Complutense University of Madrid, Madrid, Spain*

Abstract

Mutual Information is an information theoretic measure designed to quantify the amount of similarity between two random variables ranging over two sets. In recent work we have used it as a base for a measure, called Biased Mutual Information, to guide the selection of a test suite among different possibilities. In this paper, we adapt this concept and show how it can be used to address the problem of generating a test suite with high fault finding capability, in a black-box scenario and following a *maximise diversity* approach. Additionally, we present a new Grammar-Guided Genetic Programming Algorithm that uses Biased Mutual Information to guide the generation of such test suites. Our experimental results clearly show the potential value of our measure when used to generate test suites. Moreover, they show that our measure is better in guiding test generation than current state-of-the-art measures, like Test Set Diameter (TSDm) measures. Additionally, we compared our proposal with classical completeness-oriented methods, like the H-Method and the Transition Tour method, and found that our proposal produces smaller test suites with high enough fault finding capability. Therefore, our methodology is preferable in an scenario where a compromise is necessary between fault detection and execution time.

Keywords: Formal approaches to testing, Information Theory, Mutual information, Finite State Machines.

1. Introduction

In order to increase the reliability of complex software systems, testing [1, 2] is the most widely used methodology. In a few words, testing is the process of applying inputs to the system that we are evaluating (usually called the System Under Test (SUT)), observe the outputs produced by the system and assess whether the obtained outputs coincide with the expected ones. One of the main problems to effectively apply testing to complex systems is that the amount of different inputs is usually huge (in most cases, this set is infinite). Thus, it is of the upmost importance to devise approaches that *generate* a set of inputs, that is, a *test suite*, that is small enough to be practically feasible and is likely to be effective in finding faults.

Our goal with the work that we present in this paper is to improve the testing process by generating *good* test suites, that is, test suites with high fault finding capability. We address this problem in the specific case where we need a finite test suite that have a bound on the number of applied inputs. This scenario appears in a number of contexts. For example, we might want to generate a test suite that has a limited execution cost and, therefore, we have to limit the number of inputs applied to the SUT.

We assume that the SUT is a black-box: we only know its input and output alphabets. In particular, we have no additional information about its internal structure nor can we access to the source code of the system. However, we assume that we do have a specification of the system that we want to build so that we can check whether the application of a certain input to the SUT has produced the expected output, according to the specification. In order to simplify the presentation, and following our previous work [3], we assume that the specification is given by a Finite State Machine (FSM) but it is possible to use other state-based formalisms, in particular, those containing data. Actually, FSMs are a powerful formalism that can be used to represent very different, software and hardware, systems. Finally, we also assume that the specification has the same number of states as the implementation.

It is important to note that the users of our approach do not need to provide a specification coded as an FSM. The users can provide their specification in any other formalism as long as it is state-based and can be mapped to an FSM that represents its semantics (possibly after some abstraction). One example of such state-based formalism are state-charts. This allows us to apply our FSM-based approaches to a wide range of state-based specifications. For example, classical FSM-based test generation techniques can be applied when testing from reactive I/O-state-transition systems (RIOSTS), a formalism that can be used with a range of embedded systems [4]. In par-

[☆]This work has been supported by the Spanish MINECO/FEDER (grant FAME, RTI2018-093608-B-C31); the Region of Madrid (grant FORTE-CM, S2018/TCS-4314) co-funded by EIE Funds of the European Union; and Santander – Complutense University of Madrid (grant number CT63/19-CT64/19).

Email address: aibias@ucm.es (Alfredo Ibias)

URL: https://alfredoibias.com/ (Alfredo Ibias)

ticular, RIOSTS has been applied to evaluate part of the European Train Control System and an airbag controller [5]. Finally, the use of FSMs is well supported by several tools that can be used to specify and analyse them (e.g. fsmlib-cpp[1], automatalib [6] and OpenFST [7]).

In this paper we consider a measure, called *Biased Mutual Information (BMI)*, that is inspired by the classical concept of Mutual Information [8]. The intuition underlying the formal definition of BMI is that if we have two tests that have common parts, then they will tend to traverse the same branches of the SUT. Therefore, test suites with smaller BMI values will include tests that are more different, with the corresponding impact on the increase of *diversity*. Let us note that there is a well-studied correspondence between test diversity and test quality [9, 10, 11, 12].

In our previous work [3] we showed that BMI could be successfully used to choose, between two test suites, the one with a higher expectation to detect faults. In this paper we use BMI to confront a more difficult problem: guide the generation of test suites with high fault finding capability. The difference lies in the fact that, when selecting between two test suites, it is only necessary to compare their performance. However, in order to generate a test suite, it is necessary to consider not only the performance of the current test suite, but also its potential as a good base for the generation of new test suites.

In this paper we consider the *intelligent* generation of test suites by using BMI as a fundamental component to choose between different tests. Specifically, within the family of Genetic Algorithms, we consider a special kind of algorithm, called *Genetic Programming Algorithms*, that is able to deal with complex structures. In short, we propose a Grammar-Guided Genetic Programming Algorithm to generate test suites with high fault finding capability, using BMI as guiding measure, more specifically, being used to define the fitness function.

Our experiments focus on comparing our BMI-based approach with current state-of-the-art measures, using the Genetic Algorithm as a common framework to test the effectiveness of each measure. With this setting, we obtained that BMI is consistently better to generate test suites with higher fault finding capability than some measures (at the cost of a small increase in computation time), and that BMI is preferable in a realistic scenario than the remaining measures (due to limitations in computation cost or execution cost).

The rest of the paper is structured as follows. In Section 2 we review previous work related to our research. In Section 3 we present the background theory used in the paper. We also define BMI in that section. In Section 4 we propose our methodology to generate *good* test suites. In Section 5 we report our experiments to evaluate the performance of our methodology. In Section 6 we discuss the threats to the validity of our results. Finally, in Section 7 we provide conclusions and propose lines for future work.

[1] https://github.com/agbs-uni-bremen/fsmlib-cpp

2. Related Work

Test suite generation is a fundamental problem in Software Testing and it has been addressed from multiple angles. Although the tester can manually build the tests included in the test suite, this is a time-consuming and prone to errors process. Therefore, it is essential to automate the generation of tests by following certain *quality* criteria [13]. In the case of testing from FSMs, test generation is also one of the most challenging problems and it was already considered in early work [14, 15, 16]. These first approaches were able to generate *complete* test suites, that is, they are able to determine the correctness of the SUT by simply assuming that the SUT can be represented by an (unknown) FSM belonging to a certain fault domain. A common assumption to limit the size of fault domains is to assume that the FSM that represents the behaviour of the SUT cannot have more than a certain number of states [17, 18, 19]. In addition to methods producing complete test suites, there are other approaches that focus on partially covering the specification [20]. In this paper we are interested in generating small (probably incomplete) test suites that cover as many paths as possible by avoiding, as much as possible, the repetition of the same path.

In order to increase automation and limit the test suite size, a very important line of research is the *intelligent* generation of tests. There are many proposals considering evolutionary computation [21, 22]. In particular, and related to the research reported in this paper, Genetic Algorithms have been extensively used [23]. Genetic Algorithms have been applied to state-based formalisms to guide test generation [24, 25, 26, 27, 28, 29]. Moreover, we have used Grammar-Guided Genetic Programming Algorithm, the variant used in this paper, to test from FSMs [30].

Information Theory has already been used in testing [31, 32, 33, 34, 35, 36, 37, 38, 39]. Concerning testing from FSMs, in our previous work we have used an information theoretic measure called Squeeziness to assess the likelihood of Failed Error Propagation [40, 41, 42]. In our work, the rationale to apply an information theoretic measure is to increase diversity. Interestingly enough, despite the vast amount of work on testing from FSMs, to the best of our knowledge there are no proposals to test from FSMs where diversity is considered other than our previous work[3] (and in that case we were selecting between test suites, not generating them).

3. Preliminaries

To develop our work, we needed some preliminary concepts. In this section we present the main topics we used during our work. Specifically, we introduce the concepts of Finite State Machine, to define our experimental subjects; of Test and Test Suite, to define the goal of our problem; of Genetic Programming Algorithm, to define the algorithm with which we will solve our problem; of Biased Mutual

Information, to define our proposed measure; and of Test Set Diameter, to define the state-of-the-art measure that we will use as baseline.

3.1. Finite State Machines

We model systems as *Finite State Machines* (FSMs), but to formally define an FSM we need to introduce the following notation: Given set A, A^* denotes the set of finite sequences of elements of A; A^+ denotes the set of non-empty finite sequences of elements of A; and $\epsilon \in A^*$ denotes the empty sequence. We let $|A|$ denote the size of set A. Given a sequence $\sigma \in A^*$, $|\sigma|$ 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.

Definition 1. *A* Finite State Machine *(FSM) is represented by a tuple $M = (Q, q_{in}, \mathcal{I}, \mathcal{O}, T)$ in which Q is a finite set of states, $q_{in} \in Q$ is the initial state, $\mathcal{I}$ is a finite set of input actions, $\mathcal{O}$ is a finite set of output actions, and $T \subseteq Q \times (\mathcal{I} \times \mathcal{O}) \times Q$ is the transition relation. The meaning of a* transition *$(q, (i, o), q') \in T$, also denoted by $(q, i/o, q')$, is that if M receives input action i when in state q then it can move to state q' and produce output action o. We write $(i, o) \in_m M$ to denote that the pair (i, o) appears in m transitions of M.*

We say that M is deterministic *if for all $q \in Q$ and $i \in \mathcal{I}$ there exists at most one pair $(q', o) \in Q \times \mathcal{O}$ such that $(q, i/o, q') \in T$.*

In this paper we work with the assumption that FSMs are deterministic. This allows us to compare our work with previous state-of-the-art measures like the information theoretic (white-box) TSDm measure [43]. We represent FSMs as diagrams in which nodes represent the states of the FSM and arcs represent the transitions between states, and the initial state is denoted by an incoming edge with no source.

Finally, in our work we also assume the *minimal test hypothesis* [44]: the SUT can be modelled as an (unknown) object described in the same formalism as the specification (here, an FSM). It is important to remark that we are in a black-box framework, therefore we can only assume the existence of such an FSM, but we do not need to have access to this description. Furthermore, we can weaken this assumption to only consider that for each input sequence given to the SUT it will return an output sequence.

3.2. Test Suites

When testing, we would like to determine if the behaviour of an SUT conforms to the specification of the system that we would like to build. To achieve this goal we need a tool to detect differences between the specification and the SUT by comparing their behaviours. The main tool to define such behaviours is the concept of *trace*.

Definition 2. *Let $M = (Q, q_{in}, \mathcal{I}, \mathcal{O}, T)$ be an FSM, $\sigma = (i_1, o_1) \dots (i_k, o_k) \in (\mathcal{I} \times \mathcal{O})^*$ 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 $q_0 = q$. 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 $\epsilon \in \texttt{traces}(M)$ for every FSM M.*

Following the notion of trace, we define the notion of *test*. A test consists of a sequence of (input action, output action) pairs. Finally, a test suite will be a set of tests.

Definition 3. *Let $M = (Q, q_{in}, \mathcal{I}, \mathcal{O}, T)$ be an FSM. We say that $t = (i_1, o_1) \dots (i_k, o_k) \in (\mathcal{I} \times \mathcal{O})^+$ is a* test *for M if $t \in \texttt{traces}(M)$. The* size *of t is the length of the sequence, that is, $|t| = k$. In addition, the sequence of input actions of t is $\lambda = i_1 \dots i_k$ and the sequence of output actions of t is $\mu = o_1 \dots o_k$. We will sometimes use the notation $t = (\lambda, \mu) \in (\mathcal{I}^+ \times \mathcal{O}^+)$. We write $(i, o) \in t$ to denote that the pair (i, o) appears in the test t; $(i, o) \in_n t$ denotes that the pair (i, o) appears n times in the test t.*

A test suite *for M is a set of tests for M. Given a test suite $\mathcal{T} = \{t_1, \dots, t_n\}$, the* size *of the test suite is the sum of the sizes of its tests, that is, $|\mathcal{T}| = \sum_{i=1,\dots,n} |t_i|$.*

Let $t = (\lambda, \mu)$ be a test for M. We say that the application of t to an FSM M' fails if there exists μ' such that $(\lambda, \mu') \in \texttt{traces}(M')$ and $\mu \neq \mu'$. 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.

The idea is that a test (λ, μ) for M denotes that applying the sequence of input actions λ to a correct system (with respect to M) should lead to the sequence of output actions μ. It is important to remark that if we have non-determinism, then the previous inequality should be replaced to express that the behaviour of the SUT must be one of those of the specification, and we will have a notion of conformance similar to ioco [45], non-separability [46] or quasi-reduction [47].

3.3. Genetic Programming Algorithms

The problem of generating test suites with high fault finding capability is a difficult problem, moreover when we want to measure how good they are through a measure. Specifically, this problem is NP-complete [48] due to the combinatorial explosion associated to determine which test suite is the best one. One kind of algorithms that have been successfully used to find good enough solutions to NP-complete problems [49] are *Genetic Programming* [50, 51] (in fact, any Genetic Algorithm). The main components of a Genetic Algorithm are detailed below (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 genes of existing ones.
- A *mutation method* that can modify some individuals in order to obtain new genes, which might have not been present before.

```
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.

We will be using Genetic Programming because it is an evolution of Genetic Algorithms that lets encode the population in genes using non-linear structures (i.e. tree-like structures [50]) instead of the classical lineal structures (i.e. vectors) used in classical Genetic Algorithms. This modification implies that all the elements of a Genetic Algorithm should be adapted to work with these structured types.

Regarding our problem, most of the previous approaches to generate test suites rely on a linear structure to represent the test suite, like a vector of the inputs of the test suite [52, 25, 24, 53]. Encoding the tests suites in this fashion presents a problem: if the specification is not input-enabled, then the algorithm could generate tests that will always fail when applied to the `SUT` because they are invalid. To overcome this problem from these classical approaches, and because we are using specifications that do not have to be input-enabled, we use Grammar-Guided Genetic Programming Algorithms. The grammar allows us to ensure the correctness of the generated test suites and to use not only the information of the input but also of the corresponding expected output when generating the test suite. However, not everything are benefits: the usual crossovers for Grammar-Guided Genetic Programming Algorithms do not keep the size of the generated individuals. Therefore, we will need to tackle this problem when developing our algorithm.

3.4. Biased Mutual Information

To guide the process of generating new test suites, we propose the concept of Biased Mutual Information (BMI). In order to proper introduce it, it is important to first recall the classical definition of mutual information [8], which was an inspiration for our measure.

Definition 4. *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*

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

where $\xi_{A,B}$ is the joint probability distribution, defined as usual, of ξ_A and ξ_B, and $\sigma_{\xi_X}(x)$ is the probability of the element x given the distribution ξ_X.

Ideally, our goal is to compute the mutual information between two tests. For this regard, each test is considered a set of input/output pairs (instead of a sequence), because we do not consider the position of each pair. To compute the mutual information $I(\xi_{t_1};\xi_{t_2})$ of two tests t_1 and t_2 we have to define the probability distribution $\sigma_{\xi_t}(x)$ (see Definition 4). However, as explained in [3], the notion of Biased Mutual Information do not define a probability distribution. This development comes from empirical results, where the non-probabilistic version got better results than several probabilistic ones. Despite that, in the definition of BMI we keep the notation to retain the structure of the original formulation. In our case, to properly mark that σ does not refer to a probability distribution, we use the notation $\sigma_t(x)$, even though t does not explicitly appear in the right hand side of the formulae, because t is an implicit parameter; $\sigma_t(x)$ is only defined for x if $x \in t$.

Taking into account these considerations, we defined the non-probability $\sigma_t(x)$ as the probability of the transition being one of the transitions of the `FSM` M with label x. This means that if the label x appears in m transitions of the `FSM`, $\sigma_t(x)$ will be $\frac{1}{m}$ (if we assume a uniform distribution). During this paper we will also refer to these probabilities as weights. Additionally, we redefined the notion of joint probability with a notion of *composition*: the *composition* of two transitions of the `FSM` with the same label (i,o) will be the *common* weight, that is, the weight of (i,o). Alternatively, the *composition* of two transitions of the `FSM` with different labels will be the product of the individual weights. In the rest of this section, as previously explained, we write $(i,o) \in_m M$ to denote that the pair (i,o) appears in m transitions of M and $(i,o) \in_n t$ denotes that the pair (i,o) appears n times in the test t.

Definition 5. *Let $M = (Q, q_{in}, \mathcal{I}, \mathcal{O}, T)$ be an `FSM`, t be a test for M and $x \in \mathcal{I}\times\mathcal{O}$ be an input/output pair such that $x \in t$. We let:*

$$\sigma_t(x) = \begin{cases} \dfrac{1}{m} & \text{if } x \in_m M,\ m \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

We define the composition of two tests t_1, t_2 of M, for input/output pairs $x_1 \in t_1$, $x_2 \in t_2$, as:

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

where $x_1 \in_{m_1} M$ and $x_2 \in_{m_2} M$. In the first case, note that $m_1 = m_2$ because we are looking for the same input/output pair in M. Also note that m_1 and m_2 are greater than zero because we request $x_1 \in t_1$ and $x_2 \in t_2$ to appear in M.

Finally, we redefine the mutual information of two tests as:

$$I(t_1;t_2) = \sum_{x_1 \in t_1} \sum_{x_2 \in t_2} \sigma_{t_1,t_2}(x_1,x_2) \cdot \log_2 \frac{\sigma_{t_1,t_2}(x_1,x_2)}{\sigma_{t_1}(x_1) \cdot \sigma_{t_2}(x_2)}$$

In our work we decided to assume a uniform distribution over the set of transitions of M with the same label. However, it is important to remark that this is not the only probability distribution we can assume for these values. For example, we could choose to assume a probability distribution where the probability associated with transitions further from the initial state is lower than those near the initial state. In our experiments these kind of probabilities did not show any improvement over the uniform probability, and they made the computation harder. Therefore, unless we know the *real* distribution (i.e. from probabilistic user models that indicate the probability of an user choosing each input [54]), using the uniform distribution is preferable. Also, note that uniform distributions have desirable properties, like that they maximise entropy [55]. Next we give a result allowing us to simplify the formulation (it has been proven at [3]).

Lemma 1. *Let M be an FSM and t_1, t_2 be tests for M. We have*

$$I(t_1;t_2) = \sum_{x \in t_2} n_x \cdot \frac{\log_2(m_x)}{m_x}$$

where m_x is such that $x \in_{m_x} M$ and n_x is such that $x \in_{n_x} t_1$.

Regarding this formula, it presents some problems to generate useful values for comparing test suites. The first one is that it is non-monotonic, and the second one is that it is equal to 0 if all transitions of the specification have different labels. To solve these problems we performed a simple transformation that will give its name to the measure. In Figure 1 the dashed curve shows the behaviour of the previous formula, and the solid one shows the behaviour of the same formula after performing a small translation in the X axis of the logarithm [3].

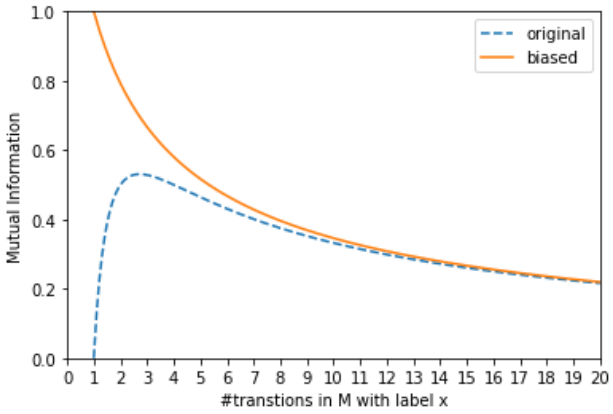

Figure 1: Measure Comparison Plot.

Definition 6. *Let M be an FSM and t_1, t_2 be tests for M. We say that the* biased mutual information *(bmi) of t_1 and t_2 is given by*

$$bmi(t_1;t_2) = \sum_{x \in t_2} n_x \cdot \frac{\log_2(m_x + 1)}{m_x}$$

where m_x is such that $x \in_{m_x} M$ and n_x is such that $x \in_{n_x} t_1$.

Using this formulation for the biased mutual information between two tests we will compute the cumulative amount of *bmi* between all the pairs of tests of a test suite. Given a test suite $\mathcal{T}$, this will be denoted by $\alpha(\mathcal{T})$ in the definition of the Biased Mutual Information of $\mathcal{T}$. Therefore, if we have a specification M and a test suite $\mathcal{T} = \{t_1, \ldots, t_k\}$, then we apply Definition 6 to all the pairs of tests included in $\mathcal{T}$:

$$\alpha(\mathcal{T}) = \sum_{i=1,\ldots,k} \sum_{j=i+1,\ldots,k} \sum_{x \in t_i} n_x \cdot \frac{\log_2(m_x + 1)}{m_x}$$

where m_x is such that $x \in_{m_x} M$ and n_x is such that $x \in_{n_x} t_j$.

Additionally, we would like to know how much repeated information are in each test of the test suite. The idea is that we need to penalise the test suites that have tests with repeated labels, even if these pairs do not appear in other tests of the test suite. Therefore, we add the *bmi* between a test and itself, that is, the value $\alpha(\mathcal{T})$ where $\mathcal{T} = \{t, t\}$ for the test t.

If we have a specification M and a test suite $\mathcal{T}$, then for each test $t \in \mathcal{T}$ we have that this *self-redundancy* factor, denoted by $\beta(t)$, is defined as:

$$\beta(t) = \sum_{x \in t} \frac{(n_x - 1) \cdot n_x}{2} \cdot \frac{\log_2(m_x + 1)}{m_x}$$

where m_x is such that $x \in_{m_x} M$ and n_x is such that $x \in_{n_x} t$.

In the previous formula, $\frac{(n_x-1)\cdot n_x}{2}$ is the sum of the first $n_x - 1$ integers. This represents the number of pairs

(x_1, x_2) of input/output pairs such that x_1 and x_2 are both in the test and $x_1 \neq x_2$. In addition, $\frac{\log_2(m_x+1)}{m_x}$ is the biased mutual information between those pairs.

If we appropriately combine these factors, then we obtain the final formula for the *Biased Mutual Information* (BMI) of a test suite.

Definition 7. *Let M be an FSM and $\mathcal{T} = \{t_1, \ldots, t_k\}$ be a test suite for M. We have*

$$BMI(\mathcal{T}) = \alpha(\mathcal{T}) + \sum_{i=1,\ldots,k} \beta(t_i)$$

An interesting property of this formula is that it only requires the information of how many times each input/output pair appears in the FSM. This allows us to use this formula even if we do not have an FSM but instead we have only the frequency of appearance of the input/output pairs. In following sections of this paper we will describe several experiments confirming that our measure works reasonably good to generate test suites that have more potential to detect faults in the SUT.

3.5. Test Set Diameter

In order to properly address the usefulness of our proposed measure, we need to compare it with state-of-the-art measures. Currently, for the generation of test suites with high fault finding capability, these measures would be the Test Set Diameter (TSDm) [43] measures. These measures are derived from the concept of Kolmogorov complexity [56]. The *Kolmogorov complexity* of a string is defined as the length of the shortest program that produces such string. Although properly computing the *Kolmogorov complexity* is really expensive in computational terms, recent work shows that it can be approximated using Normalised Compression Distance [57].

Definition 8. *Let x and y be two strings and $C(x)$ be the length of the string x after being compressed by a chosen compression program. We denote by $\texttt{ncd}(x, y)$ the Normalised Compression Distance of x and y and we define it as*

$$\frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}}$$

where xy denotes the concatenation of x and y.

The previous distance can be naturally extended to deal with multisets of strings.

Definition 9. *Let X be a multi-set of strings with at least two elements and $C(x)$ be the length of the string $x \in X$ after being compressed by a chosen compression program. We denote by $\texttt{NCD}(X)$ the Normalised Compression Distance of X and we define it as*

$$\begin{cases} \texttt{ncd}(x_1, x_2) & \text{if } X = \{\!\{x_1, x_2\}\!\} \\ \max\{\texttt{NCD}_1(X), \max_{Y \subset X}\{\texttt{NCD}(Y)\}\} & \text{otherwise} \end{cases}$$

where

$$\texttt{NCD}_1(X) = \frac{C(X) - \min_{x \in X}\{C(x)\}}{\max_{x \in X}\{C(X \backslash \{x\})\}}$$

and where $C(X)$ is the length of the compression of the concatenation of the strings belonging to X in any specific order as long as we use it for all the concatenations.

The definition of the Test Set Diameter of a test suite is based on the NCD of a (multi-)set of tests. To compute this metric there are multiple *multisets* that we can consider. Some examples are the multiset of test inputs (Input-TSDm), the one of test outputs (Output-TSDm) or even the one of execution traces (Trace-TSDm). However, in this paper we will stay true to the original work [43] and we will mainly use ITSDm to drive test generation.

Test Set Diameter is the current baseline for using Information Theory to drive test generation in a black-box setting because in an extensive study [58], it was found that TSDm outperformed many alternatives.

4. A Genetic Algorithm to Generate Test Suites

In order to generate test suites using our measure, we developed a Genetic Programming Algorithm. The idea is that this genetic algorithm will use BMI as a guide to generate test suites. Additionally, we will use this algorithm to compare different measures. For that goal we will generate two test suites using it but using as fitness functions different measures.

This algorithm is based on the one presented in [30], with several modifications to improve its performance. Specifically, among other minor tweaks, we improved the selection method to obtain a better population for the next generation, we introduced a new crossover method to increase the exchange of genetic material, and we selected new values for the crossover and mutation probability parameters in accordance with the previous modifications. In this section we will present all the components of the Genetic Algorithm.

4.1. Encoding

In order to properly represent the elements that we want to work with, it is critical to select a good encoding. We are working with test suites generated from an FSM. Therefore, we need to consider not only the input/output sequences that build a test, but also the structure of the FSM. We encode the test suites as grammar-guided trees and our Genetic Algorithm will follow a Grammar-Guided Genetic Programming approach [50, 51]. Then, the first step of our algorithm will be to generate the grammar that the FSM produces. In order to define this grammar, we follow the methodology used in [30], which allows us to automatically generate it. Formally, this grammar is defined with 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 $N \in \mathbb{N}$ is the state number.
- 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 $T \longrightarrow\ 0$ 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.

An example of a grammatically generated test suite is displayed in Figure 2.

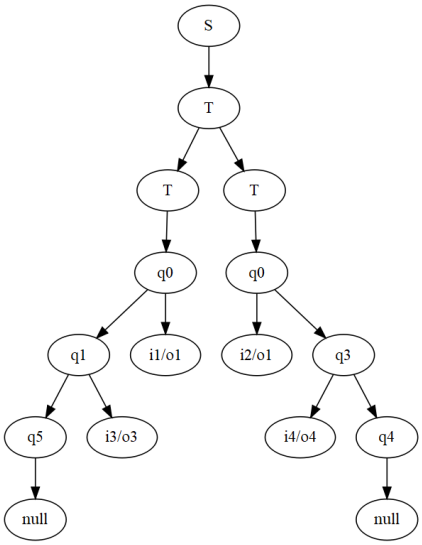

Figure 2: Example of Grammatically Generated Test Suite.

4.2. Initial Population

Our initial population is composed by 100 randomly generated test suites. Each one has as many inputs as decided by the user (in our case, for our experiments we decided to have between 100 and 500 inputs). Of course, the tests of the test suites have to conform to the `FSM`, and therefore the test suites are generated using the grammar depicted in the previous section.

4.3. Fitness Function

Our algorithm uses BMI as its (by default) fitness function, with the goal of minimising it. Additionally, it has the capability to choose between fitness functions in order to be able to compare the performance of different fitness functions, and it also has the capability to choose between maximising or minimising the fitness function. However, all the fitness function should receive a test suite and the `FSM` and should return a real value that represents how *good* is that test suite according to the measure.

4.4. Stopping Criterion

Our stopping criterion is twofold: in one hand we have a hard cap of a minimum of 20 epochs and a maximum of 100 epochs, and on the other hand we have a soft cap of having found the same best test suite along $0.2 \times NumberOfPassedEpochs$ epochs. We used this criterion with two goals in mind: first, to avoid a huge execution that takes a lot of time and resources, and second to force the algorithm to, at least, perform some iterations, but halt if there is no improvement after a while.

4.5. Selection Method

For our selection method we divided the test suites in two groups: the ones that obtained a fitness score under the mean (or over the mean if we want to maximise), and the other test suites. The test suites of the first group pass directly to the next epoch, while the ones from the second group do not pass. However, the test suites from the second group have a second opportunity to pass to the next epoch. To use this second opportunity their score should be lower (higher) than the mean plus (minus) a random number modulo the distance between the mean and the best score. This process is a variant of elitist reduction [59].

The goal of this complex selection method is not only to take the best test suites, but also to give to worse test suites an opportunity to pass to the next iteration. This way, we do not lose all the genetic information comprised in the worst test suites and we can potentially recover the best parts of such genetic information.

4.6. Crossover Method

For our crossover method we have to use a grammatical crossover. The current widely used grammatical crossovers from the literature are the Whigham crossover [51] and the standard grammatical crossover [60]. Based on these two, we produced a new crossover, presented in Algorithm 2. This crossover keeps the size of the generated individuals fixed in addition to performing a grammatical crossover. To do so, it takes the two generated offspring and crops the longer one to the desired size, and randomly extends the shorter one to the same size. To perform the crop, it just removes the excess nodes from the newly introduced branch.

We want to keep the sizes of the generated individuals fixed for two reasons: First, if it is not fixed, then the individuals with more nodes will tend to have better mutation score (as they can explore more transitions of the `FSM`) and thus, we will end with huge individuals (something we are trying to avoid to keep execution costs low). Second, we want to have a fixed test execution cost, to control the effort that will be put in the test execution phase, that we assume as the most time consuming.

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* **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 $p2$;
if $t2$ *size* $>$ $t1$ *size* **then**
  Crop $t2$ to $t1$ size;
  Extend $t1$ to $t2$ size;
end
else
  Crop $t1$ to $t2$ size;
  Extend $t2$ to $t1$ size;
end

Algorithm 2: Crossover Algorithm.

To set the probability of a crossover being produced, we consider the special characteristics of our crossover. Due to that, we set this probability to 75%. We set such probability after preliminary experiments, where we observed that this probability tends to give better results than using other probabilities.

4.7. Mutation Method

For our mutation we replace a test of the test suite with a newly generated test with the same size, therefore randomly introducing new genetic information. To perform such mutations we set a probability of 10% for each test of a test suite, which is a commonly used value in the literature [59].

5. Empirical Evaluation

To evaluate the proposed measure (BMI) we carry out several experiments. In this section we introduce the research questions and experimental design to evaluate the ability of BMI to generate test suites with high fault finding capability. We also present the results of the experiments and how they answer the research questions.

We made available, for the interested reader, all the code, benchmarks and results at `https://github.com/Colosu/BMI-Test-Generation`

5.1. Research Questions

The goal of our work is to generate test suites with high fault finding capability. Therefore, our first step is to assess how well BMI addresses this problem. Specifically, how well BMI guides the Genetic Algorithm generation process.

Research Question 1. *Is the performance of a test suite generated using BMI better than that of one randomly generated? Does this improvement pay back the extra time needed?*

We also wanted to see how BMI compares to completeness-oriented test suite generation methods, like the H-Method.

Research Question 2. *How different are the performance of a test suite generated using BMI and that of one generated by the H-Method? How important is the extra test suite size produced by the H-Method?*

Another kind of methods we wanted to compare BMI with are classical automata-based methods, like the Transition Tour method.

Research Question 3. *How different are the performance of a test suite generated using BMI and that of one generated by the Transition Tour method?*

We would like to see how BMI compares to the current state-of-the-art Information Theory approach, the Test Set Diameter (TSDm) measures.

Research Question 4. *Is the performance of a test suite generated using BMI better than that of one generated using TSDm? Does the difference in performance pay back the difference in computation time?*

Now, following a small concern from our previous work, we want to compare the performance of BMI compared to using transition cover as a guide for our Genetic Programming Algorithm.

Research Question 5. *Is the performance of a test suite generated using BMI better than that of one generated using Transition Cover? Does the difference in computation time pay back the difference in performance?*

Set size	Range # states	Range outgoing transitions	Range input alph. size	Range output alph. size
241	$[3, 156]$	$[0, 130]$	$[2, 130]$	$[2, 65]$

Table 1: Properties of the FSMs.

Finally, we would like to explore the evolution of all these alternatives with respect to the FSM size, to observe if any method has a correlation with the FSM size.

Research Question 6. *How perform the different methods when dealing with FSMs with different number of states? How this parameter affects mutation score and computation time?*

5.2. Experimental Subjects

We designed a series of experiments in order to address the research questions. We used a benchmark of 335 FSMs recently collected [61]. From this, we selected the 241 deterministic FSMs, whose parameters are displayed in Table 1. All these FSMs have a signalled initial state and, in case we need it, we can simulate the reset operation at computation time. We consider this set to be sufficiently diverse to be useful to compare the performance of our proposal in multiple scenarios. This benchmark comes with a Java library with utilities to read and interpret the FSMs, as well as a full set of utilities to generate and manipulate FSMs, called AutomataLib [6].

We will use mutants of the specification to approximate the fault detection capability of a test suite. It is well-known that Mutation Testing [62] is a suitable approach to assess the quality of a test (suite). In order to generate such mutants, we take the specification and we randomly modify the target state of one of its transitions. We only use this mutation operator because preliminary experiments showed that mutating the label of a transition (or deleting such transition) generates mutants that are easy to detect during testing [3], and introducing a new transition will not introduce any fault in the SUT given our working framework, as we will explain later. We say that a test t *kills* a mutant N of FSM M if either M and N produce different output sequences in response to t or $t = t'x/yt''$ for an input action x, output action y and prefix t' such that t' takes N to a state from which there is no transition with input action x.

For an FSM specification M, we generated test suites using the corresponding method. For the test suites generated using BMI (or TSDm or transition cover) as a guide, we used the Grammar-Guided Genetic Programming Algorithm described at Section 4. For the initial population of this Genetic Algorithm (as well as for the random generation method) we generated test suites by randomly traversing M.

We now describe the actual experiments used to address the research questions and the results of these.

5.3. Experiments

To answer our research questions, we performed different experiments. Specifically, we performed an experiment for each research question, in order to properly answer all of them. We wanted to compare our measure with previous proposals and state-of-the-art methods. In this regard, we compared our measure with the random generation, the exhaustive H-Method [63], the automata-based Transition Tour method, the alternative Test Set Diameter (TSDm) measure [43], and finally, with transition cover.

In all the experiments, we considered two assumptions: the number of states of the SUT is the same to the number of states of the specification FSM, and that we only want to know if the SUT conforms to the FSM. This last assumption implies that we will only care that the SUT can perform the same behaviour than the specification FSM, and any extra behaviour will be considered valid as long as it does not interfere with the behaviour defined by the specification. For example, if a SUT is equal to its specification plus additional transitions with no colliding inputs (that is, keeping the determinism), then we will consider such SUT as valid, because it conforms to the specification FSM.

5.3.1. Random Generation

The idea is to compare how a randomly generated test suite performs versus a test suite generated using BMI as a guide, and compare the increase of performance with the extra time needed.

In order to do so, we carried out an experiment that performs as follows. We get the benchmark FSMs and for each of them we generate a pair of test suites: one using BMI as a guide, and one randomly generated. We also computed the time needed to produce each test suite. We generated the randomly generated test suite by randomly traversing the FSM until the desired size is reached. Then, from the FSM we generate 1000 mutants, and we check which test suite killed more mutants. Finally, with the results, we report the percentage of times each test suite killed more mutants and the total percentage of killed mutants by each test suite.

In order to cover more possibilities when talking about test suite size, we alternate the size of the generated test suites as in [3], limiting those test suites to have size in $\{100, 200, 300, 400, 500\}$. Then, for each value of test suite size we performed the experiment 50 times in order to reduce the randomisation effect, performing a total of $50 \times 241 = 12050$ comparisons. With these 50 results we computed the final average results we display in Figure 3.

Using as reference the numbers for test suite size equal to 100 that are displayed in Table 2, we can observe that our measure got better results compared to the randomised method. Specifically, BMI killed more mutants at least 89% of the executions, killing an average of 37% of mutants. This is an important improvement from the 27% of mutants killed by the randomly generated test suite. Regarding times, as expected, the randomised method took

Comparison Measure(CM)	Ratio of success BMI	Ratio of success CM	Ratio of killed mutants by BMI	Ratio of killed mutants by CM	Average time of BMI (s)	Average time of CM (s)
Random	0.89	0.11	0.37	0.27	1	0
$H-method$	0.03	0.97	0.36	0.97	1	0
Transition Tour	0.96	0.04	0.38	0.11	1	0
$ITSDm$	0.86	0.14	0.37	0.28	1	1
$OTSDm$	0.69	0.31	0.37	0.33	1	1
$IOTSDm$	0.60	0.40	0.36	0.34	1	2
Coverage	0.34	0.66	0.36	0.39	1	2

Table 2: Results of the Comparison of BMI vs the Different Comparison Measures for test suite size equal to 100.

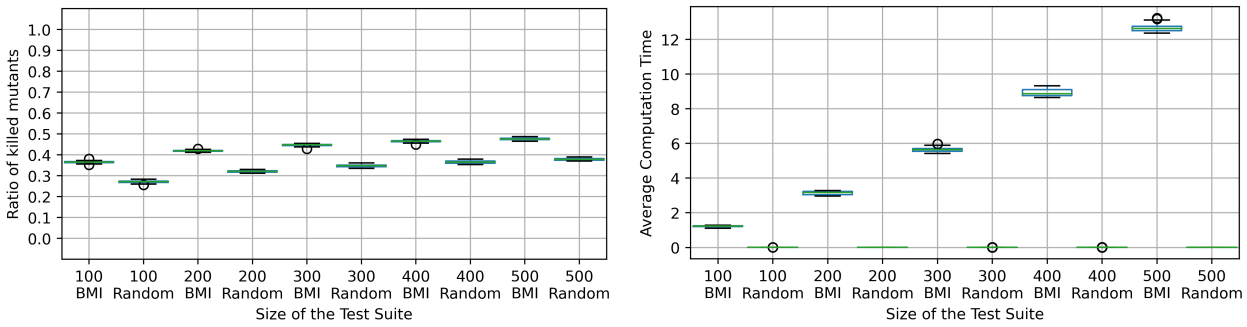

Figure 3: Random Method Results.

tenths of a millisecond to compute a test suite, while our methodology took 1 second in average to compute a good test suite. Therefore, based on the execution cost of a test suite, one could argue that a randomly generated test suite would be a better option due to its quicker computation, or that a test suite generated using BMI would be a better choice because the extra computation time is negligible compared to the cost of executing more test suites. In our scenario, we consider that using BMI is a better choice to avoid executing more test suites.

Finally, to check the validity of our results, we performed an ANOVA test to measure how probable is that the obtained percentages of killed mutants correspond to similar performances whose different results are due to the associated randomisation; or that they correspond to clearly different performances. The ANOVA test checked the null hypothesis, which corresponds to obtaining similar results using BMI and the random method, that is, both produced test suites with similar number of killed mutants. Specifically, we applied a one-way ANOVA test[2] and we computed the p-value for the experiment. We obtained p-values way lower than 0.01. Therefore, we can deny the null hypothesis for this experiment with a confidence higher than 0.99. In order to double-check our results, we performed a t-test and obtained the same p-value.

[2]Note that we could use the ANOVA test because we performed an homogeneity of variance check and it raised a positive result.

5.3.2. H-Method

The H-Method [63] is an exhaustive method which assumes that the specification is minimal, partially or completely specified, deterministic and that is has a reset operation. Also, it assumes that all states in the specification are reachable from the initial one. Although the `FSM`s from the benchmark [61] do not have a reset operation, this can be simulated during computation. Therefore, we can apply this method to our experimental subjects.

We performed an experiment to compare our methodology versus the H-Method, with the idea of comparing our proposal to a completeness oriented method. Our goal is to show how quickly the completeness oriented methods increase the test suite size with respect to the `FSM` size, while our methodology keeps the test suite size to a fixed size.

The experiment to do the comparison performs as follows. We get the benchmark `FSM`s and for each of them we generate a pair of test suites: one using BMI as a guide, and one generated by the H-Method. We also computed the time needed to produce each test suite. Then, from the `FSM` we generate 1000 mutants, and we check which test suite killed more mutants. Then, with the results, we report the percentage of times each test suite killed more mutants and the total percentage of killed mutants by each test suite.

Additionally, we alternate the size of the generated test suites as in [3], limiting those test suites to have size in $\{100, 200, 300, 400, 500\}$. The idea is to cover more possibilities when talking about test suite size. In this case,

the size of the H-Method could not be controlled, so we compared with respect to the full size test suite generated by the H-Method.

Then, for each value of test suite size we performed the experiment 50 times, reducing this way the randomisation effect, performing a total of $50 \times 241 = 12050$ comparisons. With these 50 results we computed the final average results we display in Figure 4.

Using as reference the numbers for test suite size equal to 100 that are displayed in Table 2, we can observe that our measure got, as expected, worse results compared to the H-Method. Specifically, BMI lost in both ratio of killed mutants (only 36% versus 97% for the H-Method) and in computation time (a second versus milliseconds for the H-Method). However, an important remark of this comparison is that we could not control the size of the test suite generated by the H-Method, and there is the key that explains why the H-Method is better at everything: the average test suite size of the H-Method is 1486.64, that is, more than 10 times more size than the test suites generated by our methodology, with a minimum size of 11 and a maximum size of $19,603$. This supposes a problem when executing such test suites, as they will take, in average, 10 times more time to be executed, what will not be ideal in most scenarios (especially those with big specifications, as they will lead to huge test suites). Therefore, the H-Method is not so useful in a real scenario due to the generated test suite size. Thus, based on these results, one could argue that a test suite generated by the H-Method would be a better option due to its completeness, or that a test suite generated using BMI would be a better choice because the limitation in the test suite size makes it feasible to be executed in limited time scenarios. In our scenario, we consider that using BMI is a better choice to avoid executing huge test suites.

One curious effect we have observed along our experiments is that the H-Method does not detect all the mutations in some cases. This supposedly impossible situation is produced by the kind of mutations we are using in our experiments. Specifically, changing the target state of a transition usually leads to a kind of mutation that is detected due to trying to execute a transition in a state where such transition does not exists (because the previous transition leaded to the wrong state). However, when you change the target state of a transition that was leading to a sink state (that is, a state without outgoing transitions), then no test will try to execute a transition in the new state, leading to the mutation to be effectively impossible to detect using a valid test. Thus, the H-Method could not detect this kind of mutated transitions neither.

Finally, to check the validity of our results, we performed an ANOVA test to measure how probable is that the obtained percentages of killed mutants correspond to similar performances whose different results are due to the associated randomisation; or that they correspond to clearly different performances. The ANOVA test checked the null hypothesis, which corresponds to obtaining similar results using BMI and the H-Method, that is, both produced test suites with similar number of killed mutants. Specifically, we applied a one-way ANOVA test[3] and we computed the p-value for the experiment. We obtained p-values way lower than 0.01. Therefore, we can deny the null hypothesis for this experiment with a confidence higher than 0.99. In order to double-check our results, we performed a t-test and obtained the same p-value.

5.3.3. Transition Tour

The Transition Tour is a test suite that traverses each transition of the specification `FSM` once. Ideally such test suite will be composed of only one test (the transition tour), but in case such test is impossible, we took the closer approach of using the minimum amount of tests in order to at least traverse each transition of the specification `FSM` once. Then, what we will call the *Transition Tour method* will consist on using such test suite as the test suite with which to compare our proposal.

We performed an experiment to compare our methodology versus the Transition Tour method, with the idea of comparing our proposal to a traditional automata-based method. Our goal is to show how our proposal is better than traditional methods. The experiment to do the comparison performs as follows. We get the benchmark `FSM`s and for each of them we generate a pair of test suites: one using BMI as a guide, and one generated by the Transition Tour method. We also computed the time needed to produce each test suite. Then, from the `FSM` we generate 1000 mutants, and we check which test suite killed more mutants. Then, with the results, we report the percentage of times each test suite killed more mutants and the total percentage of killed mutants by each test suite.

In order to cover more possibilities when talking about test suite size, we alternate the size of the generated test suites as in [3], limiting those test suites to have size in $\{100, 200, 300, 400, 500\}$. In this case, the size of the Transition Tour method could not be controlled, so we compared with respect to the full size test suite generated by such method.

Then, for each value of test suite size we performed the experiment 50 times in order to reduce the randomisation effect, performing a total of $50 \times 241 = 12050$ comparisons. With these 50 results we computed the final average results we display in Figure 5.

Using as reference the numbers for test suite size equal to 100 that are displayed in Table 2, we can observe that our measure got better results compared to the Transition Tour method. Specifically, BMI killed more mutants at least 96% of the executions, killing an average of 38% of mutants. This is an important improvement from the 11% of mutants killed by the Transition Tour test suite. However, an important point of this comparison is that we

[3]Note that we could use the ANOVA test because we performed an homogeneity of variance check and it raised a positive result.

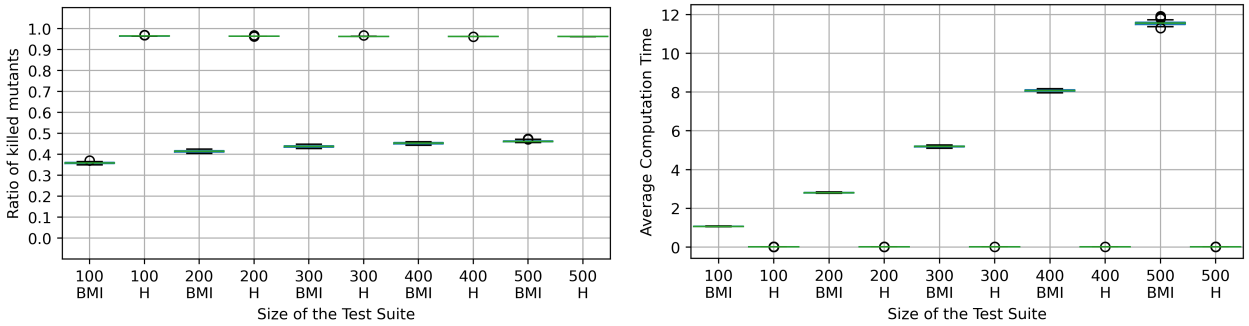

Figure 4: H-Method Results.

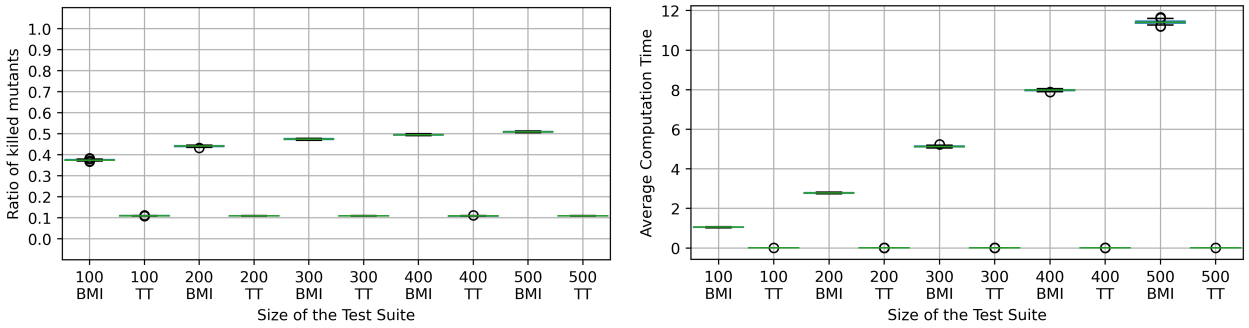

Figure 5: Transition Tour Method Results.

could not control the size of the test suite generated by the Transition Tour method, and thus we obtain an average test suite size of 366.08, that is, more than 3 times more size than the test suites generated by our methodology, with a minimum size of 6 and a maximum size of 2,816. This would be a problem when executing such test suites, as they will take, in average, 3 times more time to be executed, what will not be ideal in most scenarios (especially those with big specifications, as they will lead to huge test suites). Therefore, the Transition Tour method is not so useful in a real scenario due to the generated test suite size.

Regarding times, as expected, the Transition Tour method took only tenths of a millisecond to compute a test suite, while our methodology took 1 second in average to compute a good test suite. Therefore, based on the execution cost of a test suite, one could argue that a test suite generated by the Transition Tour method would be a better option due to its quicker computation, or that a test suite generated using BMI would be a better choice because the extra computation time is negligible compared to the cost of executing bigger test suites. In our scenario, we consider that using BMI is a better choice to avoid executing bigger test suites.

Finally, to check the validity of our results, we performed an ANOVA test to measure how probable is that the obtained percentages of killed mutants correspond to similar performances whose different results are due to the associated randomisation; or that they correspond to clearly different performances. The ANOVA test checked the null hypothesis, which corresponds to obtaining similar results using BMI and the Transition Tour method, that is, both produced test suites with similar number of killed mutants. Specifically, we applied a one-way ANOVA test[4] and we computed the p-value for the experiment. We obtained p-values way lower than 0.01. Therefore, we can deny the null hypothesis for this experiment with a confidence higher than 0.99. In order to double-check our results, we performed a t-test and obtained the same p-value.

5.3.4. TSDm Measure

Another comparison we wanted to do was between BMI and the current state-of-the-art measures: the TSDm measures [43]. To compare them, we compared their performance guiding our Genetic Programming Algorithm to generate test suites with high fault finding capability. Specifically, we compared our measure with the Input-TSDm (ITSDm), the Output-TSDm (OTSDm) and the InputOutput-TSDm (IOTSDm), so we cover all the possible uses of the TSDm measures in a black-box scenario.

The experiments to do the comparison perform as follows. We get the benchmark `FSM`s and for each of them we generate a pair of test suites: one using BMI as a guide,

[4]Note that we could use the ANOVA test because we performed an homogeneity of variance check and it raised a positive result.

and one using the corresponding TSDm measure as a guide for the Genetic Programming Algorithm. We also computed the time needed to produce each test suite. Finally, from the FSM we generate 1000 mutants, and we check which test suite killed more mutants. Then, with the results, we report the percentage of times each test suite killed more mutants and the total percentage of killed mutants by each test suite.

Following the reasoning of previous experiments, we alternate the size of the generated test suites limiting it to be in $\{100, 200, 300, 400, 500\}$. Then, for each value of test suite size we performed the experiment 50 times in order to reduce the randomisation effect, performing a total of $50 \times 241 = 12050$ comparisons. With these 50 results we computed the final average results we display in Figures 6, 7, and 8.

Using as reference the numbers for test suite size equal to 100 that are displayed in Table 2, we can observe that our measure got better results in all the three comparisons. For ITSDm, BMI killed more mutants at least 86% of the times, killing an average of 37% of the mutants, while ITSDm killed an average of 28% of the mutants. Similarly, for OTSDm, BMI killed more mutants at least 69% of the times, killing an average of 37% of the mutants, while OTSDm killed an average of 33% of the mutants. Finally, for IOTSDm, BMI killed more mutants at least 60% of the times, killing an average of 36% of the mutants, while IOTSDm killed an average of 34% of the mutants.

More interesting is the comparison in computation time, because BMI gets not only better results, but also lower computation times when compared with IOTSDm, as we can observe in the right plot of Figure 8. This contradicts the results obtained in our previous work ([3]), because there we obtained higher computation times for BMI than for IOTSDm. This is easily explained by the fact that we used, to compare both measures, a methodology derived from a method to approximate the value of TSDm of a test suite. This method, instead of computing all the steps for getting the TSDm measure of the test suite, considered the test suite as a initial pool of test and therefore the subset of the test suite obtained by the method with the desired size will be the one with higher TSDm score. Therefore, the computation was done along the method used to compare both measures, while that case did not apply to the computation of BMI. However, in the experiments presented in this paper, we have to compute the TSDm score properly for each test suite, and the method used to generate the test suite (the Genetic Programming Algorithm) does not simplifies this computation. Therefore, this induces an increase in the computation time for the TSDm scores.

However, when comparing with ITSDm and OTSDm BMI obtains higher computation times. This can question how preferable is our measure if it takes more time to compute a good test suite, but we consider that this decision depends a lot in the cost of executing a test. In a similar way to what we explained in Section 5.3.1: a test suite generated using ITSDm or OTSDm would be a better option due to its quicker computation, or a test suite generated using BMI would be a better choice because the extra computation time is negligible compared to the cost of executing more test suites. As previously explained, in our scenario using BMI would be a better option if it can avoid the execution of extra tests.

The clear conclusion from the results is that the best contender against BMI is IOTSDm. However, this measure failed to kill more mutants than our measure in any run. Then, the clear conclusion is that our measure is better for generating test suites than the TSDm measures.

Finally, to check the validity of our results, we performed another ANOVA test. The ANOVA test checked the null hypothesis, which corresponds to obtaining similar results using BMI and the TSDm measures, that is, both produced test suites with similar number of killed mutants. Specifically, we applied a one-way ANOVA test[5] and we computed the p-value for the experiments. We obtained p-values way lower than 0.01 in all three cases. Therefore, we can deny the null hypothesis for our experiments with a confidence higher than 0.99. In order to double-check our results, we performed a t-test and obtained the same p-values.

5.3.5. Transition Cover

Finally, the last comparison we wanted to do was between our proposal and the transition cover. Following the concern found in our previous work, we wanted to explore how well our methodology performs when compared with the transition cover criteria. To compare them we compared their performance guiding our Genetic Programming Algorithm to generate test suites with high fault finding capability. That is, we used the transition cover score of each individual as their fitness value. Our notion of transition cover score consist in counting how many different transitions are covered by the test suite, and dividing this number by the total number of transitions of the FSM. Thus, to compute it we need to have access to the FSM to properly identify the different transitions, and to know if two equal input/output pairs from the test suite correspond to the same transition or to different transitions.

The experiment to do the comparison performs as follows. We get the benchmark FSMs and for each of them we generate a pair of test suites: one using BMI as a guide, and one using transition cover as a guide. We also computed the time needed to produce each test suite. Then, from the FSM we generate 1000 mutants, and we check which test suite killed more mutants. Then, with the results, we report the percentage of times each test suite killed more mutants and the total percentage of killed mutants by each test suite.

Same as previous experiments, we alternate the size of the generated test suites as in [3], limiting those test suites

[5]Note that we could use the ANOVA test because we performed an homogeneity of variance check and it raised a positive result.

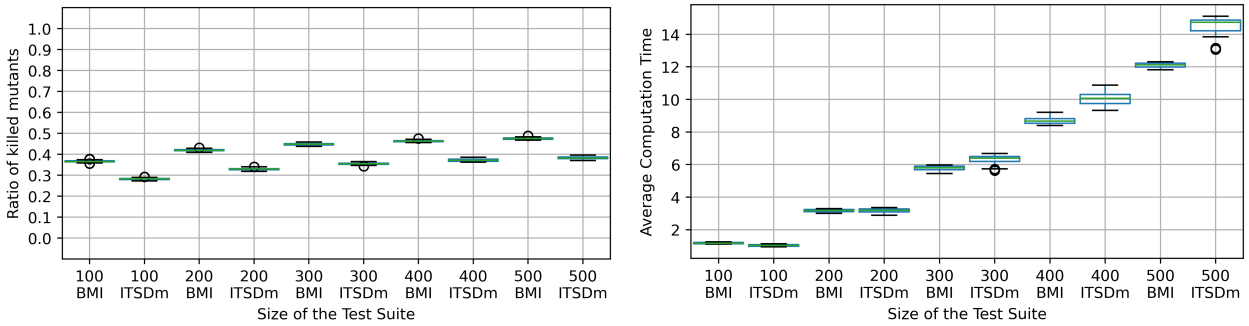

Figure 6: ITSDm Results.

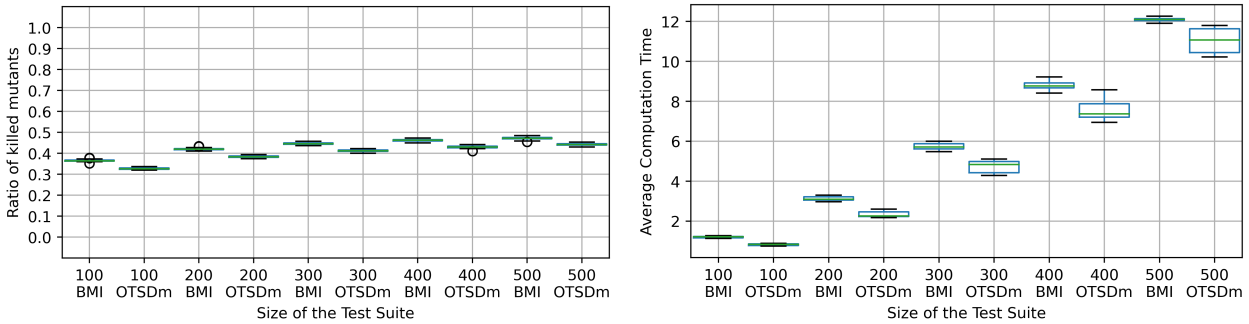

Figure 7: OTSDm Results.

to have size in $\{100, 200, 300, 400, 500\}$. The idea is to cover more possibilities when talking about test suite size. Then, for each value of test suite size we performed the experiment 50 times, reducing this way the randomisation effect, performing a total of $50 \times 241 = 12050$ comparisons. With these 50 results we computed the final average results we display in Figure 9.

Using as reference the numbers for test suite size equal to 100 that are displayed in Table 2, we can observe that our measure got slightly worse results compared to the transition cover. Specifically, BMI killed more mutants at least 34% of the executions, killing an average of 36% of mutants. This is only slightly worse than the 39% of mutants killed by the transition cover test suite. Regarding times, our methodology took 1 second in average to compute a good test suite, while the transition cover took 2 seconds in average. Additionally, it is important to remark that the transition cover needs to have access and explore the actual specification at runtime, while our proposal does not need as many information to be computed; it can be computed only with the information of how many times each transition label appears in the `FSM`. Therefore, based on the execution cost of a test suite, one could argue that a test suite generated using transition cover would be a better option due to its better results, or that a test suite generated using BMI would be a better choice because the saving in computation time plus the reduced required information is better suited for their scenario. In our scenario, we consider that using BMI is a better choice to avoid the extra computation time and the extra information required.

Finally, to check the validity of our results, we performed an ANOVA test to measure how probable is that the obtained percentages of killed mutants correspond to similar performances whose different results are due to the associated randomisation; or that they correspond to clearly different performances. The ANOVA test checked the null hypothesis, which corresponds to obtaining similar results using BMI and transition cover, that is, both produced test suites with similar number of killed mutants. Specifically, we applied a one-way ANOVA test[6] and we computed the p-value for the experiment. We obtained p-values way lower than 0.01. Therefore, we can deny the null hypothesis for this experiment with a confidence higher than 0.99. In order to double-check our results, we performed a t-test and obtained the same p-value.

5.3.6. `FSM` Size Evaluation

To evaluate how well all the measures and methods used in the previous experiments evolve with respect to the `FSM` size, we computed the mutation score and computation time of each method, averaged along 50 iterations, for each `FSM` from the benchmark. Then, for the `FSM`s that

[6]Note that we could use the ANOVA test because we performed an homogeneity of variance check and it raised a positive result.

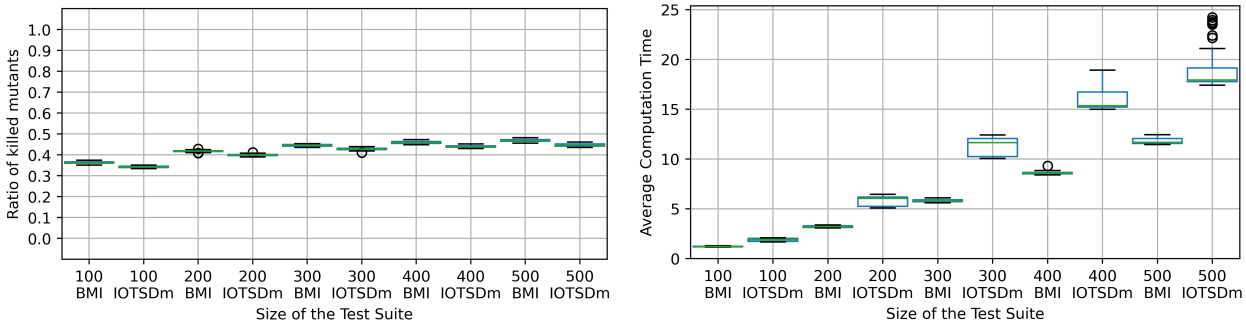

Figure 8: IOTSDm Results.

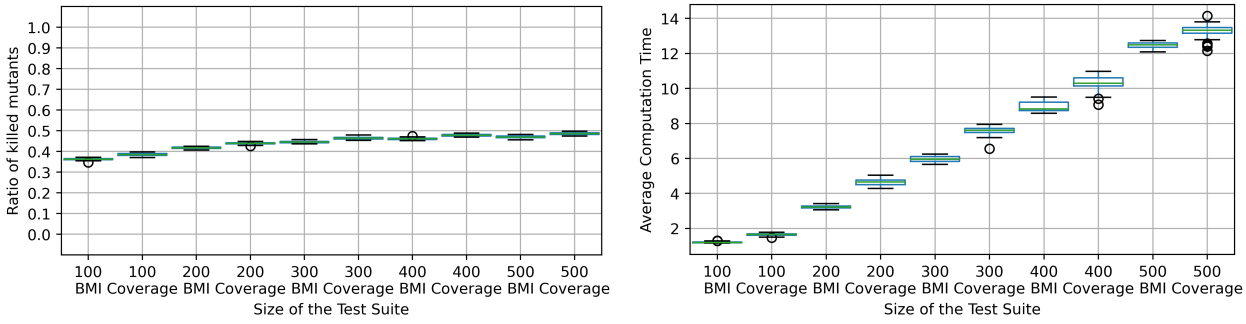

Figure 9: Transition Cover Results.

had the same number of states, we averaged their values to obtain a value for that FSM size. The results of such computation are displayed in Figure 10.

In these results there are two clear trends: the H-Method almost always kills 100% of the mutants while the Transition Tour method only kills a few mutants (even for the smaller FSMs), and their computation times are always less than a second (along with the random method). Then, we have the other methods, that are bounded by the test suite size (that was set to 100 for this experiment). These methods get better mutation scores when dealing with smaller test suites (as expected), mainly due to the limitations of the test suite size. However, when we talk about computation times, we can see that these methods are the ones that take more time, with IOTSDm always having the highest score, followed by BMI, OTSDm and ITSDm. What is more interesting is that these methods do not show an increase in computation time correlated with the increase on FSM size, but instead we can see that the computation time is more or less between the same boundaries. We consider that this is an effect of the Genetic Programming Algorithm not actually dealing with the FSM size, but instead dealing with values and representations of the FSM where the FSM size does not directly determine the size of such representations. Finally, one method that gets out of all these trends is the transition cover, whose computation time gets really high for some small FSMs, and in general is less stable than the computation time of the other methods. We consider that this behaviour could be due to the necessity of using the reset operation more frequently, but further research is needed to properly address this behaviour.

5.3.7. Test Suite Evaluation

Finally, as a final side experiment we wanted to explore how BMI performs given different test suite sizes. For such task we computed the mutation score and computation time of our proposal for different test suite sizes. Specifically, we computed them for sizes in $\{10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700, 800, 900\}$, and we obtained the results displayed in Figure 11.

As we can observe in these results, the ratio of killed mutants increases with the test suite size, although at a greater rate for sizes lower than 100 than for sizes greater than 100. In fact, we can consider the size 100 to be a good trade-of between computation time and mutation score, as the average computation time increases quicker than the mutation score once reached a size of 100. This could suppose a problem for our measure when dealing with the generation of really big test suites, but we are confident that novel advances in the Information Theory field, especially those related with computing approximations efficiently, will reduce these computation times in the future.

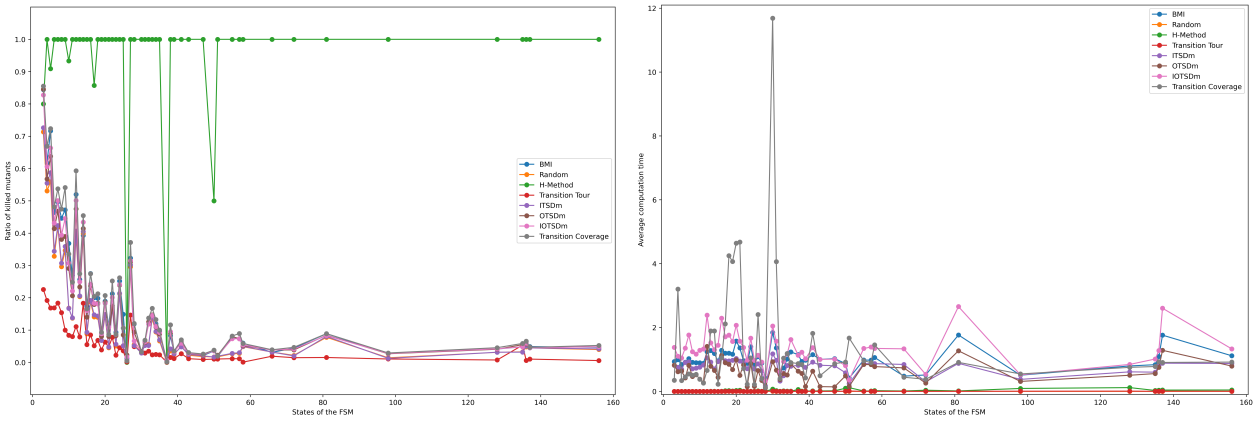

Figure 10: FSM Size Results.

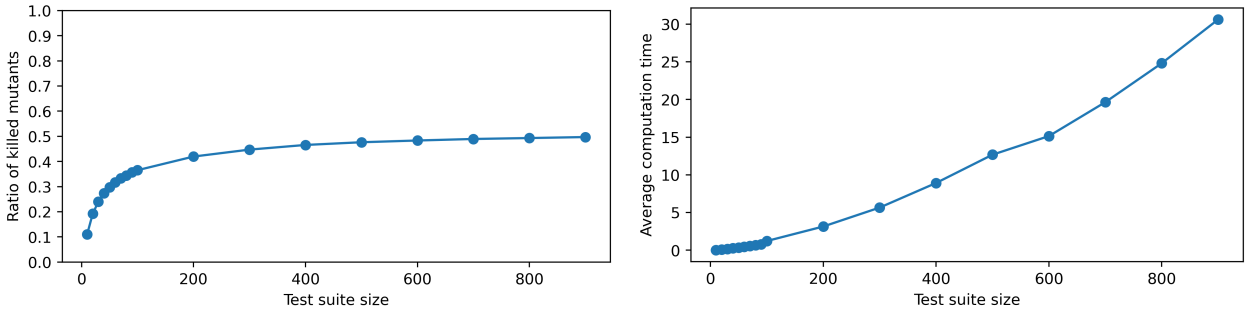

Figure 11: Test Suite Size Results for BMI.

5.4. Research Questions Answers

We now summarise what the results tell us about the research questions.

Research Question 1. *Is the performance of a test suite generated using BMI better than that of one randomly generated? Does this improvement pay back the extra time needed?*

As it was expected, the answer to this question is affirmative. Our methodology generates better test suites than a randomised method, by an important margin. It is important to remark that our methodology takes more time to compute a test suite, but we know that in many cases our methodology would be preferably than a randomised one due to the cost of executing a test suite.

Research Question 2. *How different are the performance of a test suite generated using BMI and that of one generated by the H-Method? How important is the extra test suite size produced by the H-Method?*

As expected, the performance of the H-Method is considerably better, concerning fault detection, than that of our methodology because it generates complete test suites. However, given the huge size of the generated test suites, it is clear that in a real scenario our methodology is preferable than using the H-Method. Of course, this implies that the extra test suite size is critical to the effectiveness of the H-Method, something that in our scenario is a problem due to the assumed higher cost of executing a test.

Research Question 3. *How different are the performance of a test suite generated using BMI and that of one generated by the Transition Tour method?*

In this case, the performance of the Transition Tour method was worse than expected. It not only got worse results than BMI, but even than the random method, and all that given that it produced longer test suites than the other two methods. The only place where the Transition Tour is better is in the computation time, although the worse results make this insufficient. Therefore, BMI is a better option in all cases.

Research Question 4. *Is the performance of a test suite generated using BMI better than that of one generated using TSDm? Does the difference in performance pay back the difference in computation time?*

Here, we got that BMI performs better than using any TSDm measure as a guide, obtaining a positive answer to this question. However, it is important to remark that

our methodology takes more time to compute a test suite, but, same as with the randomised method, we know that in many cases our methodology would be preferably than one using TSDm measures due to the cost of executing extra test suites.

Research Question 5. *Is the performance of a test suite generated using BMI better than that of one generated using Transition Cover? Does the difference in computation time pay back the difference in performance?*

In this case, we found that transition cover obtained better results than BMI. It kills more mutants, although at the expense of increasing its computation time and needing more information from the FSM. This leads to us preferring our methodology instead of the transition cover, due to its dependence to the FSM information and its extra computation time.

Research Question 6. *How perform the different methods when dealing with FSMs with different number of states? How this parameter affects mutation score and computation time?*

In general, more states mean lower mutation scores for all methods (except the H-Method, given its completeness-oriented approach). However, we can observe that for most methods, the computation time keeps in the same boundaries independently of the FSM size, except for the transition cover, whose computation time is less stable.

6. Threats to Validity

To ensure the validity of the results of our work, we need to address the potential threats that can invalidate them. We start with the threats to internal validity, which refers to uncontrolled factors that can affect the output of the experiments, either in favour or against our hypothesis. The main threat in this category is the possibility of having faults in the code of the experiments. To diminish this threat we carefully tested the code, even using small examples for which we know what the expected results were. Additionally, to reduce the impact of the randomness associated with our methodology, we repeated the experiments several times. Finally, the last internal threat was the poor performance of Normalised Compress Distance (NCD) with short strings. For this threat we used relatively long strings: we will have 400 characters for a test suite of size 100.

The next category of threats is the threats to external validity, which refers to the generality of our findings to other situations. The main threat in this category is the choice of experimental subjects. As the population of FSMs is unknown, then this threat is not fully addressable. However, we used a carefully constructed benchmark that aims to represent real systems to try to diminish this threat.

Finally, the last category of threats is the threats to construct validity, which refers to the relevance of the properties we are measuring for the extrapolation of the results to real-world examples. The main threat in this category is whether the faults used during our experiments are real or not. However, this threat is impossible to be addressed without a proper benchmark of FSMs with faulty versions, and we are not aware of the existence of one of these. We know that state-based specifications are widely used in certain areas of industry (e.g. automotive and avionics) but associated companies appear not to have provided faulty versions. The open source community provides a source of faulty code but not faulty models.

7. Conclusions

In this work we have confronted a fundamental task for software testing when resources and time are scarce: the automatic generation of test suites with high fault finding capability problem. We addressed this problem developing a Grammar-Guided Genetic Programming Algorithm that guides the test suite generation using BMI.

Having developed BMI, and analysed a number of its properties, we reported on experiments that evaluated it. First, we compared test suites generated using BMI to randomly generated test suites and found that BMI generates better test suites. Similarly, we compared test suites generated using BMI to test suites generated using TSDm measures (the current state-of-the-art measures) and the Transition Tour method, and found again that BMI generates better test suites. Finally, we compared our methodology to the H-Method and transition cover measure and found that ours is preferable in a realistic scenario where time and resources are scarce.

These positive results confirm that our measure is preferable when generating test suites with respect to their test suite size. Moreover, when comparing BMI with TSDm the positive results point out that we can improve the diversity of the tests if we introduce knowledge about the specification. Further debate about this effect can be found at [3].

Regarding future work, one really interesting research line is the use of BMI for the sequential generation of test suites. There is also the challenge of devising measures for non-deterministic specifications. We would like to work further in the comparison between BMI and TSDm measures, too. Finally, another interesting research line would be the application of BMI in more complex scenarios or with other specification systems (i.e. having extended finite state machines).

Acknowledgements

I would like to thank Robert M. Hierons and Manuel Núñez for very useful discussions on the topic of this paper.

References

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

[2] G. J. Myers, C. Sandler, T. Badgett, The Art of Software Testing, 3rd Edition, John Wiley & Sons, 2011.

[3] A. Ibias, M. Núñez, R. M. Hierons, Using mutual information to test from Finite State Machines: Test suite selection, Information & Software Technology 132 (2021) 106498.

[4] W. Huang, J. Peleska, Complete model-based equivalence class testing for nondeterministic systems, Formal Aspects of Computing 29 (2) (2017) 335–364.

[5] F. Hübner, W. Huang, J. Peleska, Experimental evaluation of a novel equivalence class partition testing strategy, Software and Systems Modeling 18 (1) (2019) 423–443.

[6] M. Isberner, F. Howar, B. Steffen, The open-source LearnLib, in: 27th Int. Conf. on Computer Aided Verification, CAV'15, LNCS 9206, Springer, 2015, pp. 487–495.

[7] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, 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, Vol. 4783, Springer, 2007, pp. 11–23.

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

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

[10] E. G. Cartaxo, P. D. L. Machado, 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) (2011) 75–100.

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

[12] H. Hemmati, Z. Fang, 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, IEEE Computer Society, 2015, pp. 1–10.

[13] S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, An orchestrated survey of methodologies for automated software test case generation, Journal of Systems and Software 86 (8) (2013) 1978–2001.

[14] F. C. Hennie, Fault-detecting experiments for sequential circuits, in: 5th Annual Symposium on Switching Circuit Theory and Logical Design, IEEE Computer Society, 1964, pp. 95–110.

[15] M. P. Vasilevskii, Failure diagnosis of automata, Cybernetics 4 (1973) 653–665.

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

[17] R. M. Hierons, H. Ural, Optimizing the length of checking sequences, IEEE Transactions on Computers 55 (5) (2006) 618–629.

[18] F. Ipate, Bounded sequence testing from deterministic finite state machines, Theoretical Computer Science 411 (16-18) (2010) 1770–1784.

[19] A. Simão, A. Petrenko, N. Yevtushenko, On reducing test length for FSMs with extra states, Software Testing, Verification and Reliability 22 (6) (2012) 435–454.

[20] A. V. Aho, A. T. Dahbura, D. Lee, M. Ü. Uyar, An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours, IEEE Transactions on Communications 39 (11) (1991) 1604–1615.

[21] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, A. Arcuri, An empirical evaluation of evolutionary algorithms for unit test suite generation, Information and Software Technology 104 (2018) 207–235.

[22] D. Griñán, A. Ibias, Generating tree inputs for testing using evolutionary computation techniques, in: 22nd IEEE Congress on Evolutionary Computation, CEC'20, IEEE Computer Society, 2020, pp. E–24267: 1–8.

[23] D. S. Rodrigues, M. E. Delamaro, C. G. Corrêa, F. L. S. Nunes, Using genetic algorithms in test data generation: A critical systematic mapping 51 (2).

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

[25] K. Derderian, M. G. Merayo, R. M. Hierons, 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, Springer, 2011, pp. 396–403.

[26] A. Núñez, M. G. Merayo, R. M. Hierons, M. Núñez, Using genetic algorithms to generate test sequences for complex timed systems, Soft Computing 17 (2) (2013) 301–315.

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

[28] M. Benito-Parejo, M. G. Merayo, An evolutionary algorithm for selection of test cases, in: 22nd IEEE Congress on Evolutionary Computation, CEC'20, IEEE Computer Society, 2020, pp. E–24535: 1–8.

[29] R. Zhao, W. Wang, Y. Song, Z. Li, Diversity-oriented test suite generation for EFSM model, IEEE Transactions on Reliability 69 (2) (2020) 611–631.

[30] A. Ibias, D. Griñán, 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, Springer, 2019, pp. 716–728.

[31] K. Androutsopoulos, D. Clark, H. Dan, R. Hierons, M. Harman, An analysis of the relationship between conditional entropy and failed error propagation in software testing, in: 36th Int. Conf. on Software Engineering, ICSE'14, ACM Press, 2014, pp. 573–583.

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

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

[34] D. Clark, R. M. Hierons, Squeeziness: An information theoretic measure for avoiding fault masking, Information Processing Letters 112 (8-9) (2012) 335–340.

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

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

[37] K. R. Pattipati, S. Deb, M. Dontamsetty, A. Maitra, START: System testability analysis and research tool, IEEE Aerospace and Electronic Systems Magazine 6 (1) (1991) 13–20.

[38] R. Sagarna, A. Arcuri, X. Yao, Estimation of distribution algorithms for testing object oriented software, in: 9th IEEE Congress on Evolutionary Computation, CEC'07, IEEE Computer Society, 2007, pp. 438–444.

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

[40] A. Ibias, R. M. Hierons, M. Núñez, Using Squeeziness to test component-based systems defined as Finite State Machines, Information & Software Technology 112 (2019) 132–147.

[41] A. Ibias, M. Núñez, Estimating fault masking using Squeeziness based on Rényi's entropy, in: 35th ACM Symposium on Applied

Computing, SAC'20, ACM Press, 2020, pp. 1936–1943.
[42] A. Ibias, M. Núñez, SqSelect: Automatic assessment of failed error propagation in state-based systems, Expert Systems with Applications 174 (2021) 114748.
[43] R. Feldt, S. M. Poulding, D. Clark, 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, IEEE Computer Society, 2016, pp. 223–233.
[44] 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).
[45] J. Tretmans, Model based testing with labelled transition systems, in: Formal Methods and Testing, LNCS 4949, Springer, 2008, pp. 1–38.
[46] N. Shabaldina, K. El-Fakih, N. Yevtushenko, Testing nondeterministic finite state machines with respect to the separability relation, in: Testing of Software and Communicating Systems, TestCom'07, Vol. 4581 of Lecture Notes in Computer Science, Springer, 2007, pp. 305–318.
[47] A. Petrenko, N. Yevtushenko, Conformance tests as checking experiments for partial nondeterministic FSM, in: 5th Int. Workshop on Formal Approaches to Software Testing, FATES'05, LNCS 3997, Springer, 2006, pp. 118–133.
[48] C. T. Cheng, The test suite generation problem: Optimal instances and their implications, Discret. Appl. Math. 155 (15) (2007) 1943–1957.
[49] K. A. D. Jong, W. M. Spears, Using genetic algorithms to solve NP-complete problems, in: 3rd Int. Conf. on Genetic Algorithms, ICGA'89, Morgan Kaufmann Publishers Inc., 1989, pp. 124–132.
[50] J. R. Koza, Genetic programming, MIT Press, 1993.
[51] R. I. McKay, N. X. Hoai, P. A. Whigham, Y. S., M. O'Neill, Grammar-based genetic programming: a survey, Genetic Programming and Evolvable Machines 11 (3-4) (2010) 365–396.
[52] K. Derderian, M. G. Merayo, R. M. Hierons, 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, Springer, 2009, pp. 327–334.
[53] A. Samarah, A. Habibi, S. Tahar, 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, IEEE Computer Society, 2006, pp. 19–26.
[54] C. Andrés, M. G. Merayo, 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, IEEE Computer Society, 2009, pp. 145–154.
[55] T. M. Cover, J. A. Thomas, Elements of Information Theory, Wiley Interscience, 1991.
[56] M. Li, P. M. B. Vitányi, An Introduction to Kolmogorov Complexity and Its Applications, 4th Edition, Springer, 2019.
[57] R. Cilibrasi, P. M. B. Vitányi, Clustering by compression, IEEE Transactions on Information Theory 51 (4) (2005) 1523–1545.
[58] C. Henard, M. Papadakis, M. Harman, Y. Jia, Y. L. Traon, Comparing white-box and black-box test prioritization, in: 38th Int. Conf. on Software Engineering, ICSE'16, ACM Press, 2016, pp. 523–534.
[59] M. Mitchell, An introduction to genetic algorithms, MIT Press, 1998.
[60] J. Couchet, D. Manrique, J. Rios, A. Rodríguez-Patón, Crossover and mutation operators for grammar-guided genetic programming, Soft Computing 11 (10) (2007) 943–955.
[61] D. Neider, R. Smetsers, F. W. Vaandrager, H. Kuppens, Benchmarks for automata learning and conformance testing, in: T. Margaria, S. Graf, K. G. Larsen (Eds.), Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday, Springer, 2019, pp. 390–416.
[62] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, M. Harman, Mutation testing advances: An analysis and survey, Vol. 112 of Advances in Computers, Elsevier, 2019, pp. 275 – 378.
[63] R. Dorofeeva, K. El-Fakih, N. Yevtushenko, An improved conformance testing method, in: Formal Techniques for Networked and Distributed Systems - FORTE'05, Vol. 3731 of Lecture Notes in Computer Science, Springer, 2005, pp. 204–218.

Appendix A. Appendix: Additional Material

This appendix does not constitute an integral part of the paper and it is included to facilitate the work of the reviewers. You can find the tables with the results at this link: `https://github.com/Colosu/BMI-Test-Generation/tree/main/Results`