

UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA



TESIS DOCTORAL

**Applications of information theory and artificial intelligence
to software testing**

**Aplicaciones de la teoría de la información y la inteligencia
artificial al testing de software**

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Alfredo Ibias Martínez

Director

Manuel Núñez García

Madrid

UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA



TESIS DOCTORAL

Applications of Information Theory and Artificial Intelligence to Software Testing
Aplicaciones de la Teoría de la Información y la Inteligencia Artificial al Testing de Software

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Alfredo Ibias Martínez

DIRECTOR

Manuel Núñez García

Applications of Information Theory
and Artificial Intelligence
to Software Testing

Aplicaciones de la Teoría de la Información
y la Inteligencia Artificial
al Testing de Software



Ph.D. Thesis

Alfredo Ibias Martínez

Facultad de Informática

Universidad Complutense de Madrid

December 2021

Document layout with T_EX_S v.1.0+.

This document is ready to be printed double-sided.

Applications of Information Theory
and Artificial Intelligence
to Software Testing

Aplicaciones de la Teoría de la Información
y la Inteligencia Artificial
al Testing de Software

Thesis presented to qualify for the title of Computer Science Doctor

Alfredo Ibias Martínez

Directed by

Manuel Núñez García

**Facultad de Informática
Universidad Complutense de Madrid**

December 2021

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

To the memory of my father.

Acknowledgements

*Every genius stands on the shoulders
of a Social Network,
not the shoulders of Giants.*

Sal Restivo

I would like to thank my thesis supervisor for his invaluable help and advice and for all the work he did to help me during this thesis. I would also like to thank Professor Robert M. Hierons for his support and his revisions of the work, he could be considered a co-director of this work due to his inestimable help. Additionally, I would like to thank Professor David Clark for his help during my stay at UCL. I would also like to thank all my co-authors for their help during our research. Finally, I would like to thank my family for his moral and economical support.

This thesis has been developed within the Design and Testing of Reliable Systems research group of the Complutense University of Madrid (group number 910606 of the catalogue of groups recognised by the UCM) and has been supported by the following research projects:

- UK EPSRC (grant number InfoTestSS EP/P006116/2).
- Spanish MINECO-FEDER (grant numbers DArDOS TIN2015-65845-C3-1-R and FAME RTI2018-093608-B-C31).
- The Region of Madrid (grant numbers SICOMORo-CM ICE/3006 and FORTE-CM S2018/TCS-4314).
- The Santander – Complutense University of Madrid grant (CT63/19-CT64/19).

Abstract

*The art and science of asking questions
is the source of all knowledge.*

Thomas Berger

Software Testing is a critical field for the software industry, as it has the main tools used to ensure the reliability of the produced software. Currently, more than 50% of the time and resources for creating a software product are diverted to testing tasks, from unit testing to system testing. Moreover, there is a huge interest into automatising this field, as software gets bigger and the amount of required testing increases. However, Software Testing is not only an industry oriented field; it is also a really interesting field with a noble goal (improving the reliability of software systems) that at the same time is full of problems to solve. Therefore, it leaves space for imagination to dream and try to address such problems through the application of tools from other fields. In this thesis, such fields are *Information Theory* and *Artificial Intelligence*. Information Theory is a field with a strong mathematical basis. Its main goal is to measure the information of a string based on the *commonality* of its components. Artificial Intelligence is an algorithmic field that tries to approximate solutions for exponentially complex problems. Both fields are full of tools and methodologies that could help addressing some of the problems that Software Testing arise. Moreover, although both fields can seem disparate, with tools that would be better fitted to solve different kinds of problems, in fact that is not always the case. Along the research carried out during this thesis we found multiple situations where the use of tools from Information Theory improves an Artificial Intelligence-based solution and vice versa. Actually, these synergies make this thesis a compact work more than a compilation of methods.

The main goal of this thesis is, therefore, to address different problems from the Software Testing field and devise ways of solving (or approximate a solution for) such problems using tools and results coming from the Information Theory and Artificial Intelligence fields. Specifically, this thesis addresses the Failed Error Propagation (FEP) problem, the test case generation problem, the Integration Testing of Software Product Lines (SPLs)

problem, and the selection of hard-to-kill mutants for Mutation Testing problem. These four problems are addressed from different perspectives, looking for the best method to try to solve each of them.

This way, for the test case generation problem we propose both an evolutionary method based on a Grammar-Guided Genetic Programming Algorithm and an Information Theory-based measure (initially developed to choose between test cases) to guide such algorithm, with the goal of generating test cases with high fault finding capability. This is one of those cases where both fields join forces to obtain really good solutions. Additionally, we develop a Grammar-Guided Genetic Programming Algorithm to generate test cases guided by coverage metrics, with the goal of increasing the coverability of the produced test cases.

For the Failed Error Propagation problem our work focuses on the use of Information Theory-based measures to address it. Specifically, we focus on a previously proposed information theoretic measure called Squeeziness that measures the likelihood of FEP in a System Under Test (SUT), and we adapt it to work in a black-box scenario, in a non-deterministic one, and even to work with notions of entropy different from the original Shannon's entropy. Additionally, we develop a tool to automatically compute this last version. It is inside this tool where another case of these two fields helping each other can be found: we implement an Artificial Neural Network to automatically estimate the best notion of entropy to use for the given SUT.

In another line of work, our research to address the selection of hard-to-kill-mutants problem delves in the idea of using swarm intelligence to solve a complex problem. Specifically, with the goal of reducing the amount of *useful* mutants, we develop a swarm intelligence algorithm, inspired in the Particle Swarm Optimisation one, to decide which mutants are the harder-to-kill ones. Finally, in order to solve the Integration Testing of SPLs problem we use an Ant Colony Optimisation algorithm to select features either with a low testing cost or with a high probability of being requested. The goal is to simplify the testing processes through the reduction of the number of feature combinations needed to test an SPL.

The outcomes of all these proposals are relevant, improve the state-of-the-art and set new precedents for future work. Moreover, they open new lines of work for further development of the proposals and for improving the obtained solutions. Thus, this thesis makes its humble contribution to the aforementioned fields, for the enjoyment of whoever find it interesting.

Key Words: Software Testing, Information Theory, Artificial Intelligence, Evolutionary Algorithms, Machine Learning, Failed Error Propagation (FEP), Test Case Generation, Software Product Lines, Mutation Testing.

Resumen

*La ciencia y el arte de hacer preguntas
es la fuente de todo conocimiento.*

Thomas Berger

El Testing de Software es un campo crítico para la industria del software, ya que éste contiene las principales herramientas que se usan para asegurar la fiabilidad del software producido. Hoy en día, más del 50% del tiempo y recursos necesarios para crear un producto software son dirigidos a tareas de testing, desde el testing unitario al testing a nivel de sistema. Más aún, hay un gran interés en automatizar este campo, ya que el software cada vez es más grande y la cantidad de testing requerido crece. Sin embargo, el Testing de Software no es solo un campo orientado a la industria; también es un campo muy interesante con un objetivo noble (mejorar la fiabilidad de los sistemas software) que al mismo tiempo está lleno de problemas por resolver. Por tanto, éste deja espacio a la imaginación para soñar e intentar afrontar dichos problemas a través de la aplicación de herramientas de otros campos. En esta tesis, dichos campos son la *Teoría de la Información* y la *Inteligencia Artificial*. La Teoría de la Información es un campo con una fuerte base matemática. Su principal objetivo es medir la información de una cadena de caracteres basándose en lo *común* de sus componentes. La Inteligencia Artificial es un campo algorítmico que intenta aproximar soluciones para problemas exponencialmente complejos. Ambos campos están llenos de herramientas y metodologías que pueden ayudar a afrontar algunos de los problemas que el Testing de Software presenta. Más aún, aunque ambos campos puedan parecer dispares, con herramientas que están preparadas para solucionar problemas de distintos tipos, este no es siempre el caso. A lo largo de la investigación realizada durante esta tesis hemos encontrado múltiples situaciones donde el uso de herramientas de la Teoría de la Información mejoran una solución basada en Inteligencia Artificial y viceversa. De hecho, estas sinergias hacen de esta tesis un trabajo compacto más que una compilación de métodos.

El principal objetivo de esta tesis es, por tanto, afrontar diferentes problemas provenientes del campo del Testing de Software y divisar formas de

solucionar (o de aproximar una solución a) dichos problemas usando herramientas y resultados provenientes de los campos de la Teoría de la Información y la Inteligencia Artificial. Específicamente, esta tesis afronta el problema del Fallo en la Propagación de Errores (FEP por sus siglas en inglés), el problema de la generación de casos de test, el problema del Testing de Integración de Líneas de Producción de Software (SPLs por sus siglas en inglés), y el problema de la selección de mutantes difíciles-de-matar para el Testing de Mutación. Estos cuatro problemas son afrontados desde distintas perspectivas, buscando el mejor método para intentar solucionar cada uno de ellos.

De esta forma, para el problema de la generación de casos de test proponemos tanto un método evolutivo basado en un Algoritmo de Programación Genética Guiado por Gramática como una medida basada en la Teoría de la Información (inicialmente desarrollada para elegir entre casos de test) para guiar dicho algoritmo, con el objetivo de generar casos de test con una gran capacidad para encontrar fallos. Este es uno de esos casos en los que ambos campos unen fuerzas para obtener soluciones muy buenas. Adicionalmente, también desarrollamos un Algoritmo de Programación Genética Guiado por Gramática para generar casos de test guiado por medidas de cobertura, con el objetivo de aumentar la cobertura de los casos de test producidos.

Para el problema del Fallo en la Propagación de Errores nuestro trabajo se centra en el uso de medidas basadas en la Teoría de la Información para afrontarlo. Específicamente, nos centramos en una medida de la Teoría de la Información propuesta previamente llamada Squeeziness que mide la potencialidad de tener FEP en un Sistema Bajo Test (SUT por sus siglas en inglés), y la adaptamos para funcionar en un escenario de caja negra, en uno no determinista, e incluso para trabajar con nociones de entropía diferentes de la entropía original de Shannon. Adicionalmente, desarrollamos una herramienta para calcular automáticamente esta última versión. Es dentro de esta herramienta donde podemos encontrar otro caso de ambos campos ayudándose el uno al otro: implementamos una Red Neuronal Artificial para estimar automáticamente la mejor noción de entropía a usar para la SUT dada.

En otra línea de trabajo, nuestra investigación para afrontar el problema de la selección de mutantes difíciles-de-matar ahonda en la idea de usar inteligencia de enjambre para resolver un problema complejo. Específicamente, con el objetivo de reducir la cantidad de mutantes *útiles*, desarrollamos un algoritmo de inteligencia de enjambre, inspirado por la Optimización de Enjambre de Partículas, para decidir qué mutantes son los más difíciles de matar. Finalmente, para resolver el Testing de Integración de SPLs usamos un Algoritmo de Colonia de Hormigas para seleccionar los artículos que o bien tienen un bajo coste de testing o bien tienen una alta probabilidad de

ser requeridos. El objetivo es simplificar el proceso de testing a través de la reducción del número de combinaciones de artículos necesarias para testear una SPL.

Los resultados de todas estas propuestas son relevantes, mejoran el estado del arte y sientan nuevos precedentes para trabajo futuro. Es más, abren nuevas líneas de trabajo para un mayor desarrollo de las propuestas y para mejorar las soluciones obtenidas. Por todo esto, esta tesis hace su humilde contribución a los campos antes mencionados, para el disfrute de quien la encuentre interesante.

Palabras Clave: Testing de Software, Teoría de la Información, Inteligencia Artificial, Algoritmos Evolutivos, Aprendizaje Automático, Fallo en la Propagación de Errores (FEP), Generación de Casos de Test, Líneas de Producción de Software, Testing de Mutación.

Contents

Acknowledgements	xiii
Abstract	xv
Resumen	xvii
I Introduction	1
1 Introduction	3
II State of the Art	11
2 Software Testing Background	13
2.1 General Overview of the Field	13
2.2 State-of-the-Art	15
2.2.1 Test Case Generation	16
2.2.2 The Detection of Failed Error Propagation	20
3 Information Theory Background	21
3.1 General Overview of the Field	21
3.2 State-of-the-Art	25
3.2.1 Generic Theories	26
3.2.2 Using Markov Chains	26
3.2.3 Test Case Generation and Selection	27
3.2.4 Software Quality	28
3.2.5 Failed Error Propagation	29
4 Artificial Intelligence Background	31
4.1 General Overview of the Field	31
4.2 State-of-the-Art	34
4.2.1 Machine Learning for Software Testing	34

4.2.2	Evolutionary Algorithms	36
III	Integrative Discussion	37
5	The Detection of Failed Error Propagation	39
5.1	Theoretical Background	39
5.2	The Deterministic Case	42
5.2.1	Maximum Entropy Principle	43
5.2.2	Maximum Loss of Information	43
5.3	The Generic Deterministic Case	44
5.3.1	Maximum Entropy Principle	46
5.3.2	Maximum Loss of Information	47
5.4	The Non-Deterministic Case	48
5.4.1	Maximum Entropy Principle	52
5.4.2	Maximum Information Balance (Loss and Gain)	52
5.5	Associated Papers	53
6	Test Case Generation	55
6.1	Theoretical Background	56
6.2	Using Test Set Diameter	59
6.2.1	Encoding	60
6.2.2	Initial population	61
6.2.3	Fitness function	61
6.2.4	Stopping criterion	61
6.2.5	Selection method	61
6.2.6	Crossover method	61
6.2.7	Mutation method	61
6.2.8	Replacement method	62
6.3	Using Biased Mutual Information	63
6.3.1	Fitness Function	67
6.3.2	Crossover Method	67
6.3.3	Mutation Method	67
6.4	Using Coverage-Based Metrics	68
6.5	Associated Papers	70
7	Integration Testing of Software Product Lines	71
7.1	Theoretical Background	72
7.2	Software Product Lines with Probabilities	73
7.3	Software Product Lines with Costs	75
7.4	Associated Papers	78

8	Detecting Hard-to-Kill Mutants	79
8.1	Associated Papers	82
IV	Conclusions	83
9	Conclusions	85
V	Publications	89
10	Publications	91
10.1	Using Squeeziness to test component-based systems defined as Finite State Machines	93
10.2	Estimating fault masking using Squeeziness based on Rényi's entropy	111
10.3	SqSelect: Automatic assessment of Failed Error Propagation in state-based systems	121
10.4	GPTSG: A Genetic Programming test suite generator using Information Theory measures	139
10.5	Using mutual information to test from Finite State Machines: Test suite selection	153
10.6	Coverage-Based Grammar-Guided Genetic Programming Generation of Test Suites	175
10.7	Feature Selection using Evolutionary Computation Techniques for Software Product Line Testing	185
10.8	Using Ant Colony Optimisation to Select Features having Associated Costs	195
10.9	Using a swarm to detect hard-to-kill mutants	213
VI	Bibliography	221

List of Figures

3.1	Entropy values for a random variable with two elements. . . .	22
3.2	Relationship between different entropy formulas.	23
3.3	Rényi's entropy values for a random variable with two elements.	25
5.1	Definition of \mathcal{S}_M (top) and \mathcal{S}'_M (bottom).	49
5.2	Definition of $\text{NDS}_{\mathbf{q}_k}(M)$ under maximum entropy (top) and under maximum information balance (loss and gain) (bottom).	51
6.1	Comparison plot between mutual information and biased mu- tual information.	65
7.1	Examples of translation from FODA Diagrams into SPLA. . .	72

List of Algorithms

1	Genetic algorithm: general scheme.	59
2	Crossover algorithm.	62
3	Improved Crossover Algorithm.	68
4	Ant Colony Optimisation algorithm: general scheme.	73
5	Hard-to-kill mutants heuristic: general scheme.	80

Part I

Introduction

This part presents a brief introduction to the thesis. Such introduction includes a brief overview of the importance of the fields addressed in this thesis, the motivation of the research presented in this thesis, and the goals that this thesis tries to achieve. Additionally, this introduction will include the general lines, scenarios and assumptions that the research presented in this thesis uses.

Chapter 1

Introduction

*Every story has
a beginning, a middle, and an end.
Not necessarily in that order.*

Tim Burton

Software Testing [9, 194] addresses a problem inherent to humans: the generation of errors (or bugs, as they are commonly known) through their imperfection. As we are working with Turing machines, the space of possible doable programs is infinite. In such infinite space is really hard to code the desired one. Most of the time, we code an approximation of such desired program and with the use of testing and debugging tools we improve it, hoping to get as near as possible to our goal. It is in this approximation process where the Software Testing field rises its problems.

Software Testing focuses on how to assess the reliability and correctness of a system, without being able to ensure that such system is correct. As a validation field, it focuses on finding as many faults as possible, but it can never ensure that there are no more faults (maybe hidden, maybe in the non-tested code, etc.). This nature gives a huge complexity to the problems that we have to address when testing software. Moreover, Software Testing is not only an abstract field for academic minds hungry for a challenge; it is also a very experimental field with a lot of work involving the industry. Currently, testing is a time consuming phase of software development, with costs that exceed the 50% of the development budget [194]. Thus, the funding of new research able to devise methodologies that reduce the cost and time taken without notably decreasing effectiveness has raised a lot of interest in the industry.

This duality in the Software Testing field is easily observable looking at how different researchers approach one of its main problems: test case generation. On the one hand, the more theoretical research brought approaches that focused on completeness, that is, approaches that generate test cases

that will detect all possible faults. The paradigmatic case in this scenario is the W-Method [52] (and its improvement, the Wp-Method [86]), able to completely test a given System Under Test (SUT). However, a big problem of these approaches are that, although they can ensure the conformance of the SUT with its specification in a systematic way, they are infeasible in reality due to the amount of executions needed to prove all the generated test cases. On the other hand, the more practical research brought approaches focused on practicality, that is, approaches that manage to generate a limited number of test cases, but trying that such test cases will be the ones that reveal more faults. Actually, there exist many approaches dealing with this problem using simulations or representative benchmarks and datasets, but we also have reports on the real application of testing to industrial cases. For example, we have applications of testing to industrial protocols such as the Path Computation Element Communication Protocol (PCEP) [126] (an industrial protocol for constraint-based path computation), to games like Hearthstone [87], to entire industrial sectors like the railway sector [23] (with multiple, different individual applications), and to end user systems like videoconferencing systems [7].

The innate nature of the Software Testing field as a validation field brought some caveats to the early research: such research was more focused on practicality and therefore was not formalised. This led to what we could call *informal testing*, that is, the development of ad-hoc methodologies for specific systems and rules of thumb that helped practitioners to find faults in their Systems Under Test (SUTs), but that were not formalised in any sense, neither reproducible in multiple different systems. From that initial research, a huge effort has been made to formalise testing techniques to show that testing can be formal [88]. With this effort, the field of formal approaches to testing arose [45, 115], including many tools supporting the theoretical frameworks [174, 235].

Due to the complex nature of the problems presented in the Software Testing field, researchers have resorted to not only use the tools of the field, but also to use tools coming from other fields that have been used in multiple scenarios. This is how we can observe the intersection of Software Testing with fields as diverse as state base systems [69] and evolutionary algorithms [101]. Following this aim of finding tools that can help to solve, or at least address, the problems that arise from the Software Testing field is why this thesis focuses on the application of two different but complementary fields: Information Theory and Artificial Intelligence.

Information Theory is a mathematical field that aims to measure the concept of information. To that end, it defines a measure, called *entropy*, that provides an information value for each element of a random variable. Specifically, the entropy measures the amount of information of the elements of a random variable based on their probability, with the elements with

higher probability having lower information than the more uncommon ones. However, there is not always a random variable at hand and, therefore, one has to be constructed. The most common way to do so is through the use of a bag (or multiset) of elements as the random variable and defining a probability distribution based on how frequent each element is in the bag to complete it. This way, the information comprised in each element of a bag is based on its *commonality*. Finally, the entropy formula gives the information in bits as its basic measure unit.

Information theory has been originally used to measure the amount of information a channel can transfer, given its size in number of bits. However, since its initial conceptualisation for solving such problem, it has been used in multiple scenarios, including software. Moreover, it has been used in multiple Artificial Intelligence algorithms to discriminate the amount of information a given data provides to the model. Therefore, its application to a field like Software Testing is very appropriate because it can really benefit from managing information in a more meaningful way. Moreover, there is previous applications of this field to Software Testing for a couple of problems, like Failed Error Propagation (FEP) [56] and test case generation [80].

Artificial Intelligence is a broad field widely recognised in the Computer Science world for its usefulness. However, it starts with an apparently simple although really complex goal: the generation of a real artificial intelligence, either creating a consciousness or by simulating an intelligent behaviour. In the search for an artificial consciousness, a lot of researchers have given their best years and a lot of alternatives have been provided to solve the difficult problems that this task arises. However, we still have not managed to get it. It is in the second approach where the best results have been obtained: simulating the intelligent behaviour we have managed to produce really intelligent programs that can perform (in a relatively intelligent way) a specific task. It is from this approach where things as diverse as facial recognition [76], autonomous robots [138], film reconstruction [153], and music generation [193] come, and for which Artificial Intelligence is currently a trending field. Following this more practical approach, the Artificial Intelligence field has developed a lot of useful tools for different goals, like dealing with complex data, learning patterns from data, generating new data, and representing data in a meaningful way. Among such tools arise the Machine Learning and Evolutionary Algorithms fields, both of them with a lot of experience in the application of their tools to new, sometimes unexpected problems.

Machine Learning is a statistical field focused on the learning of patterns from (a huge amount of) data. Usually, these patterns are used to classify the data in different classes, with the goal of being able to differentiate between data that apparently does not have a clear division in classes. Alternatively, they can be used to obtain a value for each data point, which can be inter-

preted later. Additionally, lately some new methods have arose that not only classify data, but that can modify (in a meaningful way) or even generate them. Therefore, Machine Learning is a useful toolbox to have when facing new problems. Actually, its use for addressing Software Testing problems is not new.

Evolutionary Algorithms is an evolutionary field that focuses on the iterative optimisation of a solution. As such, all of its tools are based on nature-inspired iterative processes, either for evolution or for exploration. The trick of these tools is that they can easily handle huge search spaces with low computational cost. Thus, their suitability for solving exponentially complex problems has no discussion. Due to their iterative nature, they are approximation methods and as such they do not always (in fact, rarely) find the optimal solution for a problem. However, they usually obtain solutions near such optima that are enough for addressing the problem in a real context. All of these properties make Evolutionary Algorithms a really useful toolbox for solving search problems in huge search spaces. Thus, they have been applied to multiple different problems, including Software Testing ones, as the only requisite is to be able to present them as a search in a definite (but not necessarily finite) search space.

Given the suitability of the tools provided by the Artificial Intelligence field to solve complex problems, it is of no surprise that one of the goals of this thesis would be to apply them to try to solve, or at least to address, problems from the Software Testing field. Moreover, given their nature, such tools rely a lot in the management of information. Thus, there is an intrinsic relationship between Artificial Intelligence and Information Theory. It is from this relationship from where new synergies between both fields can arise to better solve certain problems. Specifically, the synergies between both fields could reach solutions for certain problems that were not at reach when using only one field. Finally, Information Theory is a field that can solve some Software Testing problems due to its own nature. Summing this all up, there is a strong motivation to try to use the tools from these fields to solve, or at least address, some of the complex problems that arise from the Software Testing field and that is the main goal of this thesis. More specifically, the goal of this thesis is to study how to use the tools available in these different fields to solve problems from the Software Testing field. Additionally, if possible, the search for synergies between the Information Theory and Artificial Intelligence fields would be a critical goal, with the aim of improving the results of the proposed solutions. However, knowing that this synergy is not always possible to obtain, the application of individual tools will not be relegated to a lower priority.

In the first stages of this thesis, a basic framework is established: SUTs will be black boxes. That means that we do not know their internal structure, neither their current running state. Therefore, we have to work with

specifications and the input/output behaviour of the system. Another important assumption is that the specification of the SUT is given by a Finite State Machine (FSM). This is not always the case, but we can usually transform a state-based formalism into an FSM. All the research performed in this thesis follows these assumptions as they allow us to abstract our results from any implementation formalism and therefore contribute to the generality of our solutions.

After carefully studying the open problems from Software Testing, some proposals to apply tools from Information Theory and Artificial Intelligence arose. From them, along the process of researching for this thesis only a couple could be developed. For that reason, in this thesis there are only four problems from Software Testing that are addressed (some with multiple solutions): Failed Error Propagation (FEP), test case generation, Integration Testing of Software Product Lines (SPLs), and detection of hard-to-kill mutants.

Failed Error Propagation (FEP) is a fundamental problem in Software Testing: if a fault is masked, then it is hard to find it and fix it. Therefore, a lot of research has focused in the detection and assessment of the presence of FEP in an SUT. To achieve such goal, a line of work defined measures such that, given an SUT, they estimate the likelihood of having cases of FEP. In that line, previous work presented an information-theoretic measure called Squeeziness, which measured the likelihood of having cases of FEP in a white-box SUT with high success. In this thesis this work is adapted to deal with black-box SUTs. In addition, the work is expanded in two directions: the improvement of the measure and the reduction of the requirements of the measure. In the first direction, Squeeziness is extended to deal with new notions of entropy: Squeeziness was defined using Shannon's entropy but in this thesis it is extended to use Rényi's entropy, a parametric generic notion of entropy such that when the parameter tends to 1, Rényi's entropy converges into Shannon's one. Additionally, a tool that automatically computes the best value of the parameter to assess the likelihood of having cases of FEP is developed. In the second direction, Squeeziness relaxes one of its prerequisites (that the SUT should be deterministic) and it is adapted to deal with non-deterministic systems, giving raise to Non-Deterministic Squeeziness.

Test case generation is a common issue in Software Testing, as the execution of test cases is one of the main ways to find faults. The easiest method to generate test cases is to randomly traverse the specification, annotating the input/output pairs that appear. In this case, the efficacy of the generated test cases is also random. A way to improve the efficacy of the generated test cases to find faults is through the generation of test cases in an *intelligent* way, that is, generating them with the goal of achieving some properties. In this line, this thesis presents three different approaches to this problem. First a Grammar-Guided Genetic Programming Algorithm to gen-

erate test suites that maximise (or minimise) a certain measure is devised. Such measure will guide the evolution of the algorithm and, therefore, will be the proxy to measure the fault finding capability of the generated test suites. The first approach consists in the use of the Test Set Diameter (TSDm) [80] measures as measures based on the diversity of the generated test suites. The second approach is to define an improved measure for diversity, based on the concept of Mutual Information, that outperforms the TSDm ones. This measure is called Biased Mutual Information (BMI). Finally, the third approach changes a bit the orientation and focuses on the coverage of certain elements of the SUT's specification, defining some measures based on the t-way coverage ones.

The Integration Testing of Software Product Lines (SPLs) is a combinatorial explosion related problem. The idea of testing SPLs is that the tester should ensure that each feature of the SPL can correctly function when composed with any other feature of the SPL. This simple definition hides a really complex problem: once the number of features starts to increase, the number of feature combinations exponentially rises. Therefore it is of the utmost importance to find methods able to reduce the number of feature combinations to test. In this thesis, two of such methods are proposed: the first one focuses on finding the feature combination that has the highest probability (under the assumption that such probability represents the probability of such feature of being selected by a user), while the second method focuses on finding the combination of features that require less testing when adding a new feature to the SPL. Both methods use an Ant Colony Optimisation (ACO) algorithm to achieve their goals.

Finally, the detection of hard-to-kill mutants is an efficiency focused problem. In Mutation Testing, not all the mutants are killed in the same proportion by the test cases. Under the assumption that the number of killed mutants is a proxy for the capability of a test suite to finding faults, it is easily understandable that a test case that kills mutants harder to kill will be more efficient detecting faults. Therefore, finding which mutants are the hardest to kill can help in the use of Mutation Testing to assess the generation of test cases with high fault finding capability. In this thesis a swarm approach is developed to find such mutants in an efficient way.

As reported in the following chapters, all of these new ways to apply Information Theory and Artificial Intelligence to Software Testing led to satisfactory results. All of them achieve their goals, setting new state-of-the-art in their respective fields. The code for the experiments of all the research presented in this thesis can be found inside different repositories at <https://github.com/Colosu>.

To present this work, the rest of this thesis is organised as follows: in Part II the State-of-the-Art of the different fields used in this thesis is presented, with Chapter 2 focusing on the situation in the Software Testing field,

Chapter 3 focusing on the Information Theory field and Chapter 4 on the Artificial Intelligence field. In Part III, an integrative discussion is presented to organise the research performed in this thesis, with four chapters focusing on the four problems addressed in this thesis: Chapter 5 considers the **FEP** problem, Chapter 6 focuses on the test case generation problem, Chapter 7 focuses on the Integration Testing of **SPLs** problem and Chapter 8 focuses on the detection of hard-to-kill mutants problem. Finally, in Part IV, Chapter 9 presents the conclusions that derive from the work performed in this thesis as well as some lines of future work.

Part II

State of the Art

This part presents the thesis revision of the current state-of-the-art in the research fields relevant for it. Specifically, those fields are Software Testing, Information Theory and Artificial Intelligence. To better organise the presentation of these state-of-the-arts, each chapter provides a general overview of the field with the goal of getting acquainted to the terminology of the field. In addition, a brief summary of the research performed in the corresponding field will follow, focused on the research that will be relevant to put into perspective the contributions of this thesis.

Chapter 2

Software Testing Background

*It's more about good enough
than it is about right or wrong.*

James Bach

Software Testing is the main application field of this thesis. This chapter includes an overview of the field, in Section 2.1, together with a brief explanation of its main concepts. Section 2.2 presents the state-of-the-art in the field for the problems addressed in this thesis.

2.1 General Overview of the Field

Software Testing [9, 194] is a broad field whose main goal is to assess the correctness of a System Under Test (SUT) through the detection of faults. In Software Testing, a *fault* is a static defect in the software and its execution produces an error. An *error* is the incorrect internal state of an SUT that is the manifestation of the executed fault. It usually produces a *failure*, that is, an external, incorrect behaviour with respect to a description of the expected behaviour.

This definition of the Software Testing goal is what leads the testing process. This process starts by defining the requirements and scenario in which it will work. The process follows by producing *test cases* that will be applied to the SUT. It continues with the application of these test cases to the SUT, obtaining outputs, and concludes with the comparison of the obtained outputs and the expected ones to determine if a failure is identified. It is important to note that in all this process we only have evaluated the SUT by observing its execution: we have not searched for the faults that produced the observed failures. This goal corresponds to the *debugging* process where we search for the detected faults and try to fix them. However, this process is not part of the testing process and it is not considered in this thesis.

The first step of the testing process is to define the requirements and scenario in which we will work. There are two main categories in which each scenario can fall: white-box testing and black-box testing. In a white-box testing scenario we have access to the SUT code and/or internal state values (during execution). On the contrary, in a black-box scenario we do not have access to any of these. In this case, we can only observe the introduced inputs and the obtained outputs. Additionally, we can have a specification of the expected behaviour of the SUT. This specification can be used as an *oracle*, and will define the requirements of the SUT. As an oracle, it should determine if an observed input/output pair is an expected behaviour. As some SUTs can be really huge, sometimes there is no oracle available or it is extremely expensive to produce one [257, 48]. This situation is called the *Oracle Problem* and can appear in multiple situations, like when the software is designed to solve a complex problem and there is no alternative solution.

An approach that was developed to address the Oracle Problem is Metamorphic testing [49, 232]. It focuses on checking properties that should hold between different test case executions, comparing the outputs corresponding to different inputs instead of checking whether, given an input, the observed output is the expected one. A clear example arises with the sine function: it is hard to know the sine of an arbitrary number, but we know some relations between the expected outputs. For example, we know that for the same input with an offset of 180 degrees we expect that the outputs will be the opposite (negation) of one another. These kind of properties (that should hold over multiple executions) are called *metamorphic relations*.

The second step of the testing process is to generate test cases. In testing, a test case is an input to be given to the SUT, together with an oracle to check whether the obtained output represents a correct behaviour. If the observed output does not represent a correct behaviour, then we say that we have found a failure. This failure has been observed due to a fault in the code being executed (and if we are in a white-box scenario we should also observe errors in the internal state of the SUT). A formal explanation of this phenomenon is outlined in the RIPR model [9], where *Reachability* denotes the situation when a test case reaches the location or locations where the fault resides, that is, executing a fault; *Infection* denotes the internal state of the program being incorrect after the location is executed, that is, having an error; *Propagation* denotes the infected state propagating through the rest of the execution and causing some output or final state of the program to be incorrect, that is, having a failure; and, finally, *Revealability* denotes the tester observing part of the incorrect portion of the final program state, that is, observing such failure.

Generating test cases randomly is usually easier than generating them in an intelligent way, as we only need to pair inputs with their expected outputs and feed them to the SUT. However, executing test cases has an associated

cost, usually higher than the cost of generating test cases. Therefore, in order to reduce such cost, it is important to generate test cases with high fault finding capability, that is, *good* test cases. There are multiple proposals on how to generate such test cases, ranging from random generators to exhaustive methods [86, 52], passing through evolutive or expert (informed) generators. However, it is not enough to generate a lot of test cases: we also need a way to know which ones are *good*. To that end, a lot of measures have been proposed, like the TSDm ones [80], and also some methods, like *Mutation Testing* [63, 116, 147, 205]. This last one provides a widely employed method to estimate how good a test case is. It uses modifications (mutants) of the SUT to check how many of these seeded *faults* are found (killed) by the test case, with the idea that the more mutants are killed, the better a test case is.

A set of test cases, or test suite, is the main element used in the following steps of the testing process. These steps usually are very straightforward and, therefore, there is much less research dealing with them. The third step consists in executing the SUT with the test inputs and observing the resultant outputs, while the fourth step is to decide whether the obtained outputs are product of a correct program or a faulty one. However, after all this process, we cannot ensure the correctness of the SUT because it can still be faulty and we have not detected such faults either due to not executing them, or due to their errors being masked in the execution. This last situation corresponds to another big problem of Software Testing: *Failed Error Propagation* [160, 263] (FEP). FEP occurs if a fault has been executed, this leads to an error, but the error does not propagate to incorrect output. In terms of the RIPR model previously mentioned, this is the case where the fault has been reached, infection has occurred, but there is a failure to propagate this infection to the output.

The testing process can be modified depending on the specific problem that we are addressing. For example, if we have multiple versions of the same system released over time, we can take advantage of such versions. That is the case of *regression testing* [267, 222, 170], an approach that consists in testing the new versions of an SUT using the previous ones as a kind of oracle, checking that the new changes have not introduced new faults. Sometimes, this approach involves the use of a regression test suite to check that the new version of the SUT behaves in the same way as the previous one for the inputs of the test suite.

2.2 State-of-the-Art

In the previous section five problems were identified in Software Testing: the oracle problem, test case generation, determining which test cases are *good*, the detection of Failed Error Propagation and the testing process as a whole.

Although this is not a comprehensive list, this state-of-the-art will only focus on the problems that are addressed in this thesis: test case generation and detection of Failed Error Propagation. These are two of the most relevant in the field and this section will present the state-of-the-art to solve them.

2.2.1 Test Case Generation

The literature around test case generation is really broad but it can be divided into five categories [10]: test case generation by symbolic execution, test case generation in model-based testing, test case generation in combinatorial testing, test case generation by adaptive random testing, and test case generation in search-based Software Testing. This thesis contributes to two of these categories: model-based testing and search-based Software Testing.

With respect to the test case generation in model-based testing category, three main branches can be identified [10]: axiomatic approaches, Finite State Machine (FSM) approaches and Labelled Transition Systems (LTS) approaches. This thesis only contributes to the FSM approaches branch, so the state-of-the-art with respect to model-based testing will be limited to it.

Regarding test case generation in search-based Software Testing, it is possible to identify ten main branches [106]: structural testing, model based testing, Mutation Testing, temporal testing, exception testing, regression testing, configuration and interaction testing, stress testing, Integration Testing and other testing-based applications. This thesis mainly contributes to Mutation Testing and Integration Testing. So, the state-of-the-art with respect to search-based Software Testing will be limited to them.

2.2.1.1 Test Case Generation in Model-Based Testing: FSM Approaches

FSM approaches use an FSM, formalised as a *Mealy machine* [181], to model the SUT. In this formalisation, the input and output of each transition are paired and the transitions are directed from one state to another. The idea is that FSM approaches derive sequences from that machine by using some kind of coverage criteria. Most of them only deal with deterministic FSMs, what can be considered a restriction if those FSM are supposed to represent reactive or under-specified systems.

This research area started with a seminal paper about experiments in sequential machines [192] and continued with the introduction of the first FSM-based test case generation algorithm [112]. From them, many FSM-based test case generation algorithms have been proposed [52, 125, 155, 213]. These algorithms were initially used to address problems arising in functional testing of hardware circuits. Later on, the theory was adapted to be used in the context of communication protocols, where FSMs were used to reason about

behaviour. The interested reader can find a survey on this area in [162]. More recently, FSMs have been used to test a wide variety of systems, including embedded systems [33, 209] and parts of operating systems [95, 96]. Due to the limitations of the FSMs to model data, an FSM is usually generated from a model by either applying an abstraction or expanding out the data (possibly after applying an abstraction).

Test case selection from FSMs is an area of study that has been extensively researched. It focuses on discovering assumptions that would make testing exhaustive. In this line, it has been shown that completeness can be achieved if the number of states of the SUT has a known maximum [52, 252, 162]. Also, a common assumption to limit the size of fault domains is that the FSM that represents the behaviour of the SUT cannot have more than a certain number of states [121, 139, 240]. This helps to overcome the fact that the SUT is a black-box with unknown characteristics, from which the only observable behaviour is its I/O one. Additionally to the completeness methods, a lot of work was done regarding the optimisation of test cases using its length, overlap, and other goals. This work resulted in methods like Transition-Tour [195] and Unique-Input-Output [5].

In contrast to the completeness methods, there is a huge amount of work in practical FSM approaches that do not aim at completeness. Instead, they use structural coverage criteria (like transition coverage, state coverage, path coverage, etc.) as a test case selection strategy [6, 198, 85]. A good overview of these coverage criteria (and more) for FSMs can be found in the book *Practical model-based testing* [251]. Additionally, there are research lines regarding the intelligent generation of test cases to increase automation and limit the test suite size. Some of these lines consider evolutionary computation [41, 97] and Genetic Algorithms [220]. In particular, genetic algorithms have used state-based formalisms to guide test case generation [164, 68, 197, 24, 25, 271]. Other lines consider diversity as a proxy for test case quality [81, 44, 108, 109]. Most of them consider only input diversity, but there is also some work regarding output diversity [8]. It is also important to note that some work found that both white-box and black-box notions of diversity are effective when ordering a test suite [110], that is, when addressing the test case prioritisation problem.

Finally, there have been some proposals to refine the FSM approach and problems have been studied based on them. For example, some work investigated the implications of the SUT using queues for buffering inputs and outputs [124] as a way to deal with systems that are not input and output enabled. Other work investigated distributed testing [117, 123, 122]. In particular, it is undecidable to know whether there is a strategy for each local tester that guarantees that it will force the SUT into a particular model state when testing from an FSM [114].

2.2.1.2 Test Case Generation in Search-Based Software Testing: Mutation Testing

Mutation Testing is a technique where *mutants* of the original program are created with the goal of improving the testing of such program. Mutants are created through the insertion of a single fault in the original SUT. A test case kills a mutant if it distinguishes the behaviour of both the mutant and the original program (i.e. they produce different outputs for the same input). The idea is that a test case that kills multiple mutants is supposed to be good at detecting faults (either real or artificially inserted) and experimental evidence corroborates that test suites produced using Mutation Testing approaches were significantly better than the (high quality) manually written ones [84].

Initially, Mutation Testing was used to assess the quality of a test suite, but soon it was used as a mechanism to generate test cases through the generation of test cases for killing mutants [154]. Some of these approaches include using Genetic Programming to generate and evaluate test cases for Mutation Testing [78], using data state mutation (i.e. mutating the state of the computation instead of the SUT) for test cases generation [236], using Ant Colony Optimisation (ACO) to generate test input data in automatic Mutation Testing [17], using Genetic Algorithms for co-evolution (i.e. evolving both the mutants and the test cases that can kill the mutants) of mutants and test cases [2], using Genetic Algorithms for test case generation in Mutation Testing [189], and using Model-Checkers to generate test cases for Mutation Testing [83].

Additional applications of techniques closely related to evolutionary approaches have been tried. Some of these applications included using Bacteriological Algorithms (i.e. Genetic Algorithms that retain a population and remove the cross-over operation) for test case generation for mutation-based testing [22, 19, 20, 21], using Artificial Immune Systems (i.e. a technique inspired by the behaviour of animal immune system responses) to Mutation Testing [176], and using an Artificial Immune System-based test case evolution approach to compare it with Genetic Algorithms in Mutation Testing [177].

Finally, some work has been done in higher order approaches to Mutation Testing [146]. These approaches consist in inserting multiple faults to the same mutant, considering therefore exponentially more mutants than if we consider only first order mutants. Some of these approaches focus on finding high quality mutants (i.e. those that subsume their first order constituents) under the assumption that this way the explosion in size can become manageable [145].

Regarding mutant quality, research has been done in the problem of mutant selection with the goal of minimising execution cost and time while still generating meaningful mutants that can be used to generate test cases

with high fault finding capabilities. Classically, the solution to this problem come from the reduction of the amount of mutants used in the process, using mutant reduction strategies such as selective mutation [199, 262] and random mutant selection [1]. However, the classification of mutants into two categories (hard-to-kill and easy-to-kill) brought a new research field focused on finding the hardest to kill mutants. The hard-to-kill mutants are usually defined as those killed by a small fraction of the considered test cases, but some work also defines them based on the internal structure of the code of the given SUT [253]. Also, multiple mutant classification criteria (hard to kill, subsuming, hard to propagate and fault revealing) have been compared and shown to classify different mutants as the preferable ones for Mutation Testing [204].

2.2.1.3 Test Case Generation in Search-Based Software Testing: Integration Testing

Integration testing focus on finding faults produced by the interaction of multiple components that have been integrated into an SUT [141]. Regarding Search-Based Software Testing, Integration Testing has not been explored very much. There is some work in comparing Integration Testing strategies, either using a Genetic Algorithm to minimise the stubs and optimise testing resources allocation [105], or using a Genetic Algorithm to optimise the orders of test cases with the goal of minimising the complexity of stubbing in Integration Testing [35]. There is also some work that looks at testing systems that are composed of components for which we have FSM models [14, 77, 214].

Regarding the work presented in this thesis, Integration Testing is used within a Software Product Line (SPL) framework. A Software Product Line (SPL) is, as defined by The Carnegie Mellon Software Engineering Institute, “a set of software-intensive systems that share a common managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” [178]. In the literature there are different approaches to represent SPLs, such as FODA [149], RSEB [99], PLUS [79] and SPLA [11]. This last one has been extended to deal with probabilities [39] (SPLA^P) and costs [38] (SPLA-CRIS).

There is some work dealing with testing for SPLs [58], including some evolutionary approaches for test case selection and prioritisation in SPLs [111, 169]. Regarding Integration Testing, there is some work for SPLs [43], including the use of Model-Based Testing [218] and using compositional symbolic execution [238]. In Integration Testing, independently testing all possible (sometimes redundant) products is impossible [159], and the complexity (and costs) of the testing process depends on the order of the selected features [245]. That implies that this kind of testing has its own idiosyncrasy,

with its own challenges.

2.2.2 The Detection of Failed Error Propagation

The literature around the detection of Failed Error Propagation (FEP) is relatively limited. Empirical studies have been conducted to show that many systems suffer from FEP [160, 263, 255, 229, 13, 175]. Specifically, one study showed that in 13% of the examined programs, 60% or more of the test cases suffered from FEP [175].

There is some work on FEP and fault masking for both white-box testing [15, 175, 255, 263] and black-box testing [102, 211, 212, 256], including some studies that propose different alternatives to measure FEP [175, 13, 56, 263] and to generate test suites that avoid it [102, 211, 255]. Finally, in the measuring of FEP there is some interesting work [13, 56, 263]. Most of this work focuses on Squeeziness [13, 56], a measure that uses Information Theory to assess FEP. Squeeziness has been found to have a rank correlation of close to 0.95 with the likelihood of FEP [13]. Additionally, it has been checked that Squeeziness correlates to the likelihood of having FEP more strongly than the Domain to Range Ratio [56]. Therefore, Squeeziness can be considered the state-of-the-art regarding FEP assessment.

Finally, there is an important study regarding the incidence of FEP in real programs [142]. This study measured FEP on Defects4J, the reference benchmark for Java programs with real faults. It found that the prevalence of FEP is negligible when testing is performed at the unit level. However, when system-level inputs are provided, the prevalence of FEP substantially increases. This indicates that it is enough for method post-conditions to consider only the externally observable state/data and that intermediate steps should be checked when testing at system level.

Chapter 3

Information Theory Background

*For Wiener, entropy was
a measure of disorder;
for Shannon,
of uncertainty.
Fundamentally,
as they were realizing,
these were the same.*

James Gleick

Information Theory is the mathematical toolbox that will be used in this thesis. In Section 3.1 we include a brief overview of the field, together with the presentation of the main concepts, and in Section 3.2 we review the state-of-the-art for the problems addressed in this thesis.

3.1 General Overview of the Field

Information Theory is a field that tries to measure information through the probability distribution of a random variable. The first relevant concept is called *entropy*, which is a measure of the average uncertainty of a random variable.

Definition 1. *The entropy of a discrete random variable X with a probability mass function $p(x)$ is defined with the following formula:*

$$\mathcal{H}(X) = - \sum_{x \in X} p(x) \cdot \log_2 p(x)$$

Note that logarithms in base 2 are used because entropy is measured in bits.

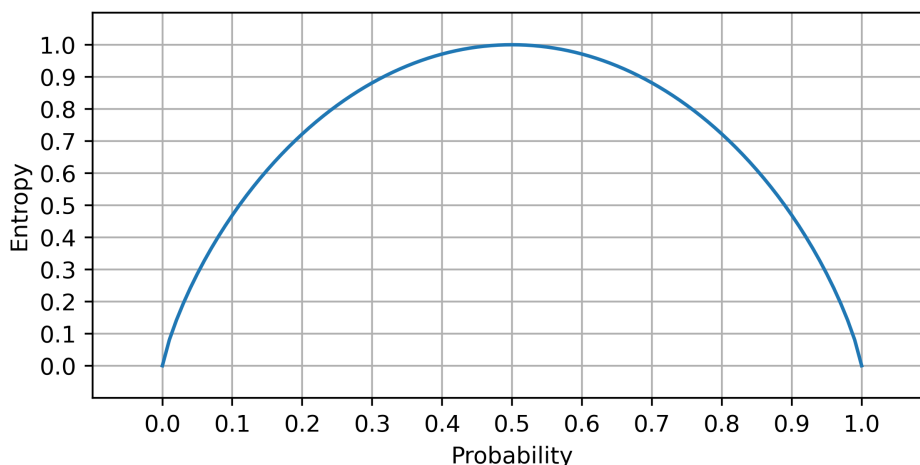


Figure 3.1: Entropy values for a random variable with two elements.

Given a random variable with two elements, we can observe the entropy of such random variable depending on the probability of one of them (versus the other) in Figure 3.1. As we can observe, the maximum entropy (and therefore, the maximum information) happens when both elements have the same probability because in such situation there is maximum uncertainty. Meanwhile, the minimum entropy (and therefore minimum information) happens when one of the elements has probability 1 and the other has probability 0. In this case the entropy is 0 because there is no uncertainty.

Using this simple formula, the information of different combinations of random variables can be computed. For example, using their joint entropy we can measure the information of two random variables together, while using conditional entropy we can measure the information that one random variable provides conditioned to the knowledge of another one.

Definition 2. Let X and Y be two discrete random variables with a joint distribution $p(x, y)$. The joint entropy of X and Y is defined by the following formula:

$$\mathcal{H}(X, Y) = - \sum_{x \in X, y \in Y} p(x, y) \cdot \log_2 p(x, y)$$

The conditional entropy of X conditioned to the knowledge of Y is defined by the following formula:

$$\mathcal{H}(X|Y) = - \sum_{x \in X, y \in Y} p(y, x) \cdot \log_2 p(x|y)$$

In order to compute the information shared by two variables, the concept of *mutual information* is introduced. It is a measure of the dependence between variables. Therefore, it is symmetric in X and Y and it is equal

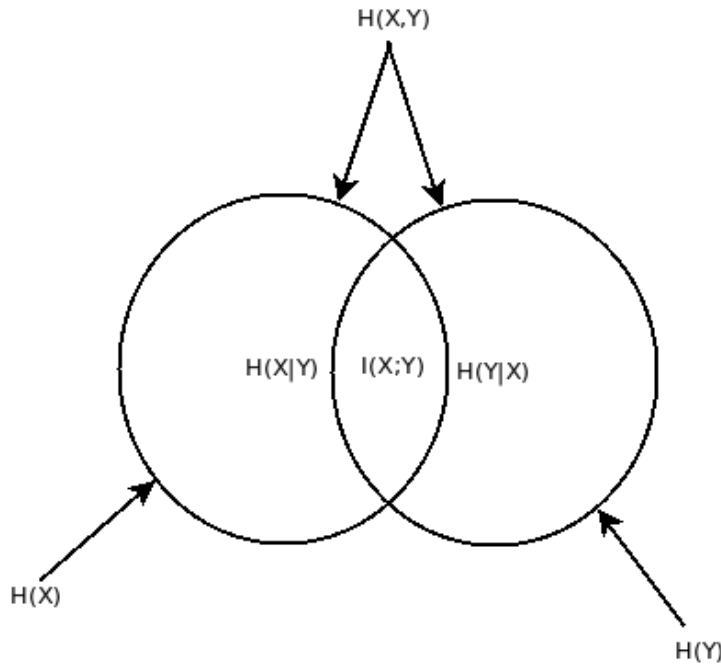


Figure 3.2: Relationship between different entropy formulas.

to 0 if and only if both random variables are independent (otherwise, it is always positive).

Definition 3. Let X and Y be two discrete random variables with a joint distribution $p(x, y)$. The mutual information between X and Y is defined by the following formula:

$$\mathcal{I}(X; Y) = \mathcal{H}(X) - \mathcal{H}(X|Y) = \sum_{x \in X, y \in Y} p(x, y) \cdot \log_2 \frac{p(x, y)}{p(x) \cdot p(y)}$$

An easy way of understanding these formulas can be found in Figure 3.2, where a Venn diagram representing the relationship between the different entropy formulas explains them at a glance. There, we can observe how the mutual information is the entropy of the intersection between random variables. Therefore, it only considers the values that are shared. Joint entropy is the entropy of the union of random variables. Therefore, it considers all the values of all random variables. Finally, conditional entropy is the entropy of the difference between random variables. Therefore, it considers only the values that are not affected by the dependence with respect to the other random variable, that is, the values that are not shared.

There are other interesting concepts coming from the Information Theory field. One of them is *relative entropy*, also known as the Kullback-Leibler distance between two probability mass functions. It is always non-negative

and it is equal to 0 if and only if both probability mass distributions are the same, but it is not a true distance because it is not symmetric (and therefore it does not satisfy the triangle inequality). This “distance” is sometimes called *Cross-Entropy*.

Definition 4. Let $p(x)$ and $q(x)$ be two probability mass functions. The relative entropy, or *Kullback-Leibler distance*, between p and q is defined by the following formula:

$$\mathcal{D}(p||q) = \sum_{x \in X} p(x) \cdot \log_2 \frac{p(x)}{q(x)}$$

Another interesting concept is *Kolmogorov Complexity*, which measures the complexity of a string (of data) based on the size of the shortest binary computer program that computes it. In order to model a computer, this concept uses a Universal Turing Machine. Briefly, a Universal Turing Machine has a program tape containing a binary program, which is fed left to right to an FSM; an FSM modelling its behaviour; and an empty work tape. The machine then reads from the program tape, writes to the work tape, changes its state according to the FSM model, and calls for more program.

Definition 5. Let x be a finite-length binary string and let \mathcal{U} be a universal computer. Let $l(x)$ denote the length of the string x . Let $\mathcal{U}(p)$ denote the output of the computer \mathcal{U} when presented with a program p . The Kolmogorov complexity $K_{\mathcal{U}}(x)$ of x with respect to \mathcal{U} is defined as

$$K_{\mathcal{U}}(x) = \min_{p:\mathcal{U}(p)=x} l(p)$$

that is, the minimum length over all programs that print x and halt. Thus, $K_{\mathcal{U}}(x)$ is the shortest description length of x over all descriptions interpreted by computer \mathcal{U} .

A final interesting general concept that we can find when working within Information Theory is *Markov Chain*, which is a sort of sequence of conditional random variables. Random variables X, Y, Z are said to form a Markov chain in that order if the conditional distribution of Z depends only on Y and is conditionally independent of X .

Definition 6. Random variables X, Y, Z are said to form a Markov chain in that order, denoted by $X \rightarrow Y \rightarrow Z$, if the joint probability mass function can be written as

$$p(x, y, z) = p(x) \cdot p(y|x) \cdot p(z|y)$$

As a final note, it is important to mention that the classical notion of entropy introduced by Shannon (i.e. the one previously defined) is not the only

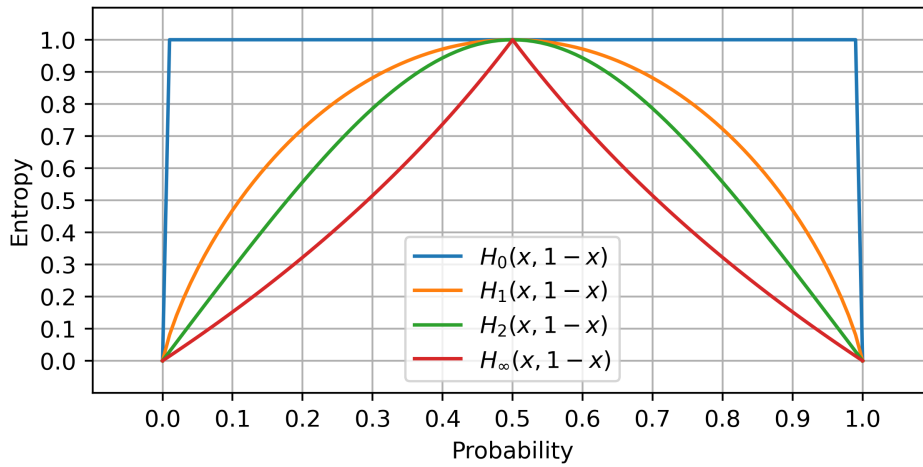


Figure 3.3: Rényi's entropy values for a random variable with two elements.

existing one. In fact, there are multiple notions of entropy proposed along the years, for multiple different goals. Over all of them, there is a proposal of a general notion that contains all of the others thanks to a parameterisation: *Rényi's entropy*.

Definition 7. *The Rényi's entropy of a discrete random variable X with a probability mass function $p(x)$ is defined by the following formula:*

$$\mathcal{H}_\alpha(X) = \frac{1}{1-\alpha} \cdot \log_2 \left(\sum_{x \in X} p(x)^\alpha \right)$$

In the previous definition, $\alpha \in \mathbb{R}_+ \setminus \{1\}$ is the parameter that will define which specific notion of entropy is used. Setting this parameter to different values, we can obtain different entropy notions, being the most relevant the ones displayed at Figure 3.3. There, we can observe how different notions approach the concept of information, and how all of them give the maximum entropy value to the uniform distribution and 0 entropy to both ends, when there is no uncertainty. Finally, note that if α tends to 1, then we have Shannon's entropy.

3.2 State-of-the-Art

The research regarding the use of Information Theory in Software Testing can be divided into multiple fields. However, this state-of-the-art presents only the five that are more important for the research performed in this thesis, namely: generic theories, use of Markov chains, test case generation and selection, software quality and Failed Error Propagation assessment.

3.2.1 Generic Theories

Some of the research dealing with Information Theory and Software Testing focuses on the development of frameworks and theories about how to join both fields. These theories include: modelling the Software Testing process as a Markov decision process to optimise it, using a Cross-Entropy based learning method with the goal of getting an optimal testing profile for the SUT [269]; comparing different software complexity measures for the different stages of the testing process (including some based on Information Theory) [28]; mining software defect data to support Software Testing management, using entropy-based discretisation to discretise the obtained data (so it can be used by machine learning classifiers) [113]; using word entropies for the classification of software traces, in order to be able to compare them [188]; studying the relationship between entropy, information gain and the uncertainty regarding the random generation of test cases [62]; and using entropy region graphs to triage crash-types, that is, a group of similar unexpected terminations of an application [151]. There is also work on the development of information-theoretic frameworks for Software Testing. For example, it is possible to define a syntax-independent coverage criterion for Software Testing (modelling the SUT as a random variable) [266, 264] and use different alternative ways to introduce Information Theory into Software Testing [55].

Finally, there is some work in testing campaigns, where entropy and statistical methods are used to determine how probably is to find another program branch or (crashing) path in the future using new inputs, so that the campaign should continue [30]. A campaign consist in the execution of random inputs to find new program branches or (crashing) paths until a statistical measure determines that the probability of finding such new elements is lower than a threshold.

3.2.2 Using Markov Chains

Research focusing on the use of Markov chains mainly propose methods to model either the software system [259] or its usage through Markov chains. For example, there is research in producing test cases using Markov chains that model the software's usage [258]; in introducing Software Testing techniques improved using Markov chain usage models of the program [231]; in using entropy-based methods to define the probability distributions of a Markov chain modelling the usage model of the SUT [216]; and in the generation of Markov usage models of software systems for their use in the software reliability process [272].

3.2.3 Test Case Generation and Selection

The main body of work regarding the application of Information Theory to Software Testing focuses on the development of information-theoretic measures to prioritise between test cases, to generate test cases, to determine how many test cases are needed for testing and to guide the test case application process.

Regarding research in the prioritisation of test cases, there is work dealing with Kolmogorov complexity to define a measure (Normalized Compression Distance) to prioritise between two test cases based on their diversity [81] and in the use of such measure to devise another one (Test Set Diameter) to determine the diversity of sets of test cases [80]. There is also work in using conditional entropy for profile reduction for test case selection [167]; in fuzzy entropy-based multi-faceted measurement frameworks for test cases classification and fitness evaluation [157]; in comparing information-theoretic measures as a guide to select test cases from a pool of test cases [110]; in defining distances based on entropy to measure the diversity of a test set [239]; and in using Rough Set Theory-Similarity Relation to reduce the size of the test cases and Conditional Entropy-Based Similarity Measure to obtain a minimum subset of requirements for minimising test cases and requirement attributes of a testing process [196]. There is also work in the use of Information Theory to select test cases based on the amount of information gained (or the reduction of the uncertainty) after the application of a test case [93, 92, 265, 275]. Finally, an interesting research line introduced the Fault Localisation Prioritisation problem, which combines fault prioritisation and fault localisation, and proposed a Fault Localisation Prioritisation technique that uses the entropy of the structural elements executed by a test suite to order the test cases that are going to be executed [268].

Regarding the generation of test cases, there are multiple proposals in the literature: use of conditional entropy to generate a heuristic for sequencing a test case [207]; the generation of an information-theoretic tool for automatic test case sequencing and testability analysis of complex, hierarchically-described modular systems [208]; and the use of entropy as the fitness function for a search based test suite generation focused on minimising the uncertainty of the diagnostic ranking of candidate fault locations [40]. Additionally, there is a body of work focused on the use of TREE (a bivariate estimation of a distribution algorithm based on an adaptation of the Combining Optimizers with Mutual Information Trees algorithm [210]), which is based on mutual information, for either evolutionary testing [225], the automatic generation of test suites (combined with scatter search) [226], or the generation of test cases for object oriented programs (specifically, for containers) [224]. Finally, some of the applications where the generation of test cases using Information Theory has been applied include the validation of machine learning-based systems [250], the improvement of the quality of web service

composition test cases through the search of paths that lead to the greatest probability of service combination failure (using Cross-Entropy) [242], and the automatically generation of software test cases based on user interaction data (using the Kullback-Leibler “distance”) [203].

Regarding the search for a limit in the number of required test cases, the literature provides little research. We can mention the use of Shannon’s source coding theorem and binary entropy to set the lower bound on the number of test cases in an optimal test plan for a system [261] and the use of Shannon’s source coding theorem to detect the lower bound of the order of the number of test cases required to perform group testing over a system [31].

Finally, regarding the guidance of the test case application process, there is some work around the ideas of measuring the difficulty (for a computer) to generate test cases based on a Markov model of the SUT [82] and of measuring the robustness of different test set categories through the comparison of the possible compression of the generated error messages (using Kolmogorov complexity) [217].

3.2.4 Software Quality

Assessing the quality of software using Information Theory is another of the main bodies of work in the intersection between Information Theory and Software Testing. We can find from direct measures of the quality of a software in an abstract sense to more concrete measures of the maintainability or reliability of a system.

In this line of work, the research focused on devising abstract software quality measures include: the use of entropy to measure the quality of a system’s design [191]; the use of statistical testing, based on usage models, to test and evaluate software intensive systems, using source entropy to measure how many test cases are needed to obtain a sample path representative of usage as defined by the model, and using trajectory entropy to measure the complexity of the specification [215]; the use of entropy to quantify the uncertainty of the operational profile, uncertainty of the overall system reliability, and component uncertainties [94]; devise the *Concept Coherence Metric* (based on Mutual Information) to measure the quality of software modularisation [230]; the use of Kolmogorov Complexity to measure the information shared between two artefacts, to quantify the software’s evolution in terms of quantified change, that is, lack of similarity [16]; and the use of entropy as a formal approach to software measurement [163]. There is also work in the characterisation of software structures and systems as either uncertainty products, where the design process consists in reducing such uncertainty [66], or as probability distributions, with the goal of finding the probability distribution that yields the *best* probability distribution for a particular object [34].

Research focused on devising measures for maintainability and/or reliability of a system include: the use of relative entropy to combine the initial assessments of the software development manager with past development data for predicting software reliability [36]; the use of the Maximum-Entropy Principle to quantify the uncertainties (of the parameters) in the software reliability modelling of a single software component with correlated parameters and in a large system with numerous components [61]; devise some advanced parametric models for assessment and prediction of software reliability (based on statistics of bugs at the initial stage of testing) and the use of Cross-Entropy Global Optimisation Methods to support the optimisation of such complex models [29]; rank Software Reliability Growth Models (for assessing the reliability of the SUT) using a weighted entropy-distance based approach [103]; and the use of the concept of complexity based on entropies to estimate the maintainability of a system [274].

Finally, there is a body of work using Akaike Information Criterion (based on Information Theory). Specifically, it is used to select the best model for a system in order to predict the number of remaining errors [152]; to evaluate the validity of the software reliability model obtained through the mix of different ones using the Expectation-Maximisation principle [202]; to prove the efficiency of a proposal of multi-factor software reliability model based on logistic regression [200]; and to evaluate the logistic regression-based software reliability growth models generated to quantify the effectiveness of testing efforts on software fault detection [201].

3.2.5 Failed Error Propagation

The last body of work to consider is the use of Information Theory to estimate or detect Failed Error Propagation. This body of work was small at the start of the thesis, but since then it has attracted some attention from researchers. Originally, there were two proposals to measure the likelihood of having FEP in a SUT: the use of Squeeziness [56] and the use of conditional entropy [13]. Squeeziness has received more attention, both in this thesis [129, 132, 134] and outside of it, with the normalisation of Squeeziness to allow for the comparison of the likelihood of FEP from two SUTs with different input domains [57].

Chapter 4

Artificial Intelligence Background

*People worry that computers
will get too smart
and take over the world,
but the real problem is that
they're too stupid
and they've already
taken over the world.*

Pedro Domingos

Artificial Intelligence is the algorithmic toolbox that will be used in this thesis. This chapter includes a brief overview of the field in Section 4.1, with a brief explanation of the main concepts of the field, and in Section 4.2 presents the state-of-the-art in the field for the problems addressed in this thesis.

4.1 General Overview of the Field

Artificial Intelligence is a broad field and highly researched since the origin of computers. Its main goal is to solve complex problems in an intelligent and automatic way. Its applications range from the use of statistical tools or expert systems to solve specific decision problems, to the use of databases and ontologies to store and depict knowledge. Given the broad nature of this field, this overview is going to be limited to the subfields that provided the tools used along this thesis. In particular, the overview is going to focus on Machine Learning and Evolutionary Algorithms.

Machine Learning is a field strongly supported by statistical methods for the discovery of patterns that solve complex problems, through the iterative

learning over a bunch of data. Machine Learning algorithms can be divided, depending on the learning framework they use, into supervised learning, semi-supervised learning, reinforcement learning, and unsupervised learning. The different frameworks operate as follows: they start with a learning phase where the algorithm takes data from the training set and tries to return the correct label for such data. In the supervised learning case the data has the expected label and the algorithm only has to learn based on how far from the expected label it felt; in the semi-supervised learning case only part of the data has the expected label (so the previous framework applies) and then the rest of the data should be carefully used to improve the performance of the algorithm; in the reinforcement learning case we do not have any labelled data but we have a formula or reward function that would guide the learning; and finally in the unsupervised learning we do not have any kind of guide about what is the correct label of the data and we rely in mathematical tools to cluster the data. After the training phase, the frameworks that have labelled data have a testing phase where new, previously unseen, but labelled data is feed to the algorithm to compute how well the algorithm has learned the desired model instead of the individual training data.

Depending on the task, this labelled data would represent values or classes. In a classification task, the labels are the classes in which the data is classified, and the algorithm learns to differentiate data from different classes in order to correctly classify new data. In a regression task, the labels are real values that represent the regression value of the data, and the algorithm learns to obtain the regression value of new data. As these are two totally different kinds of tasks, there are few algorithms that can be tweak to solve both of them, although any regression can be transformed into a classification using a sigmoid function. In fact, most algorithms usually solve only one kind of tasks and, therefore, we have a subdivision of all the learning frameworks into classification and regression methods.

Inside supervised learning, there are a myriad of algorithms and methods for classification and for regression. First, there are simple mathematical models like linear regression (where we use straight lines to either model the data or to divide the data into classes) and its derivatives (polynomial regression, Lasso regression, Ridge regression, etc...), and logistic regression (where we model the probability of a certain event happening given the inputs). Next, there are tree inspired methods like decision trees (where a tree is generated with classes as leaves and probabilities in its edges and at each level a different value of the input determines which branch it would follow) and random forest (where multiple classification methods are used to “democratically” determine the class of a new data). Finally, there are matrix based methods like Artificial Neural Networks (ANNs) (where perceptrons are used to try to simulate the brain behaviour) and Support Vector Machines (SVMs) (where kernels are used to non-linearly classify between

different classes, trying to maximise the spatial gap between them to improve generality).

ANNs [143, 223] are one of the most successful Machine Learning techniques, having recently develop its own field called *Deep Learning*. The basic concept behind ANNs is that using *perceptrons* (a compressed linear classifier unit that makes the weighted sum of the received values) and *activation functions* (a non-linear function) and stacking them into layers would allow for a better fit of the training data, specially data with many dimensions. Depending on the number of layers we can talk about shallow ANNs (with few layers) and deep ANNs (with multiple layers). Finally, the learning algorithm used by the ANNs is *back-propagation*, a gradient-descent step algorithm that updates the perceptrons weights in an appropriate and efficient manner.

Regarding unsupervised learning, there are several methods that can be further divided between parametric and non-parametric. Parametric methods assume that the data has been generated from a mixture of probability distributions of a given shape (usually normal distributions) and try to estimate the parameters of such distributions using different measures. Two of these methods are the maximum-likelihood, where the mixture of distributions that maximise the likelihood function is computed, and expectation maximisation, where the mixture of distributions that maximise the expectation of the likelihood function is computed. Non-parametric methods do not assume anything about the data and try to gather the points based on a *similarity measure*. Most of these methods are clustering methods, where the data points are gathered in classes. One of such methods is k-means, where the most common class between the k nearest neighbours is the one selected for each point.

Evolutionary Computation is a field composed by iterative optimisation methods where a pool of randomly generated solutions (individuals) is considered a population and, in each iteration, that population evolves through different “natural selection” methods that improve the most promising solutions and discard the less promising ones. They are also called bioinspired algorithms because they are usually inspired in the evolution or behaviour of biological populations. The main three algorithms of this field are Genetic Algorithms (GAs), Particle Swarm Optimisation (PSO) and Ant Colony Optimisation (ACO).

A Genetic Algorithm (GA) [89, 244] consists in the evolution of a population through the crossover of its individuals and their random mutation. The idea is that the crossover of individuals will exploit the best solutions already found, while random mutations will explore the search space looking for new elements that could improve the existing solutions. Additionally to these two fundamental operations, GAs also need to choose a proper representation for its solutions. In the basic case, a solution will be an array

of numbers, but that is not always enough to solve a problem. Some alternatives include representing the solutions as trees (and hence the Genetic Programming Algorithms [156]) and limiting the structure of such trees using a grammar (and hence the Grammar-Guided Genetic Programming Algorithms [59, 171, 180]).

A Particle Swarm Optimisation (PSO) [150, 254] algorithm consists in the evolution of a swarm of points by moving them towards the most promising solution, with a certain velocity and momentum. This velocity and momentum will help to avoid local optima allowing for a more extensive exploration of the search space, while the direction of the movement towards the most promising solution allows for the exploitation of the obtained knowledge. Additionally, this movement is performed in two levels: individual and global. In the individual level, the individual moves towards the best solution that it has found, while in the global level the individual moves towards the best solution found by the swarm. The sum of these two movements will be the total movement of the individual in the current iteration. This also helps to avoid local optima.

An Ant Colony Optimisation (ACO) [71, 72, 73] algorithm consists in a swarm of ants that explore a graph following random paths and leaving behind pheromones. Such pheromones will be stronger if the path was a successful solution and closer to the optimal one. This way, in successive iterations the ants will give more probability of being chosen to the paths with stronger pheromones, allowing for a balance between exploring the search space and exploiting the acquired knowledge.

4.2 State-of-the-Art

As explained before, this state-of-the-art will be limited to applications of Machine Learning and Evolutionary Computation to Software Testing.

4.2.1 Machine Learning for Software Testing

According to a recent survey [74], we can divide the applications of Machine Learning to Software Testing into the following topics: test case design, the oracle problem, test case evaluation, test case prioritisation, test case refinement, test cost estimation, and Mutation Testing automation. The ones that are more relevant for this thesis are test case design, test cost estimation, and Mutation Testing automation.

4.2.1.1 Test Case Design

There has been a lot of interest in applying Machine Learning to automate test case generation [234]. The research performed in this line in-

cluded a test case generation approach based on the inductive learning of programs from finite sets of input/output pairs [26]; a test case generation approach for Android applications for which there is no existing model of the GUI [51, 173, 221], where Machine Learning was used to learn a model of the application; a test case generation approach for web-applications [228], where Machine Learning is used to turn the user session data into a model of the web application; and a test case generation approach through test suite reduction using k-means clustering [50].

4.2.1.2 Test Cost Estimation

As Software Testing accounts for a significant proportion of the total cost of software development, testers have managed to effectively test software systems within the allotted time and budget. Some Machine Learning approaches have been proposed to help testers to better estimate what can affect the cost of Software Testing efforts. For example, some research has been done to estimate the effort of executing test suites [75, 273]. Other research has focused into determining which attributes that influenced testing time were the most important ones to predict testing time [46]. Finally, some research focused into predicting test case code size for object-oriented software in terms of test case lines of code [18], which is a key indicator of testing effort.

4.2.1.3 Mutation Testing Automation

Mutation Testing is a costly and time-consuming technique for which automation did not ease enough its resource needs. Therefore, some research effort has focused into overcoming these hurdles by using Machine Learning algorithms to expedite some steps of the process. The most common step researchers tried to make it faster was mutant execution, with approaches that either run a subset of randomly selected mutants and compute the result for the non-selected ones based on their similarity to the executed ones [246], approaches that predict the effectiveness of a given test suite based on a combination of source code and test suite metrics [144], and, finally, approaches that predict whether a test suite will kill a mutant based on a previously generated model based on features related to mutants and test cases [270]. There is also another line of research focusing on selecting the most meaningful mutants for the mutant execution phase, including the development of a Machine Learning approach named FaRM [47] and the development of algorithms to predict where defects may appear [148, 182].

4.2.2 Evolutionary Algorithms

The application of Evolutionary Algorithms to Software Testing can be divided [172] into Coverage Testing, test case generation, testing program dynamics, black box testing and software quality categories. For this thesis, the relevant categories are Coverage Testing and test case generation.

4.2.2.1 Coverage Testing

Coverage Testing focuses on executing different elements of an SUT, like traces, states, transitions, statements, etc. For that end, it is necessary to generate test cases and test suites that execute such elements and Evolutionary Algorithms attracted attention given their evolutive nature. For example, some researchers focused on studying test case coverage through the use of a hybrid version of a Genetic Algorithm and hill-climbing local search [241], others on condition/decision coverage proposing tools like the genetic algorithm data generation tool [186, 185], others on statement and branch coverage, using a control-dependence graph to guide optimisation [206], and, finally, others on detecting the potentially infeasible program paths [37].

4.2.2.2 Test Case Generation

The use of Evolutionary Algorithms for test case generation is a widely researched application of Evolutionary Algorithms [27, 100, 233, 260]. Some researchers use a Genetic Algorithm to estimate the parameters of a so-called “hyper-geometric distribution software reliability growth model”, where the increase of the number of errors is observed as a function of time [187]. Others use Evolutionary Algorithms to automatically generate test cases for testing a chosen subpath of the SUT [161, 168], for testing FSMs [24], for testing Extended FSMs [165, 247, 248, 249], for testing temporal systems [67, 68, 197], for testing IoT with recorded and generated events [104], for testing Event-B models [70], for passive testing [12], for Mutation Testing [64], and for search-based testing [107, 164]. Finally, there is also research in using Evolutionary Algorithms for unit test case generation [41], for state-based test case generation [164], and for coverage-based test case generation [227].

Part III

Integrative Discussion

This part presents the thesis integrative discussion. This discussion aims to briefly explain the achievements of the research presented in this thesis, explaining the aim and goal of it, the theoretical development performed on it, and the experimental results that corroborate such theory. Additionally, this discussion will include a brief resume of the debates raised by the results of such research.

This discussion is divided into four chapters, one for each problem that is addressed in this thesis.

Chapter 5

The Detection of Failed Error Propagation

*An error does not become truth
by reason of multiplied propagation,
nor does truth become error
because nobody sees it.*

Mahatma Gandhi

Addressing the detection of Failed Error Propagation (**FEP**) is a research line focused on the quality of the generated programs. This problem tackles the quality of an **SUT** in the sense of how easy it is for a fault to be masked in such **SUT** in a way that a tester could easily miss it. This problem is fundamental in Software Testing, as the quality of any solution that finds faults is hampered by the presence of this phenomenon.

In Section 5.1 we present the theoretical background common to this research line. Then, in Section 5.2 we adapt Squeeziness to work in a black-box scenario, in Section 5.3 we extend Squeeziness to use a new notion of entropy (Rényi's entropy), and in Section 5.4 we extend Squeeziness to be used in a non-deterministic scenario. Finally, in Section 5.5 we list the papers related to the work presented in this chapter.

5.1 Theoretical Background

An information-theoretic measure was proposed to address the **FEP** problem: Squeeziness [56]. This measure considered the entropy of the inputs of the program and the entropy of the outputs of the program and measured the difference in entropy or entropy loss. Then, they used this entropy loss as the proxy to assess how likely was that the **SUT** suffered from cases of **FEP**, with the idea that **FEP** appears due to a loss of information in the system.

In an empirical study [13] where 30 programs and more than $7 \cdot 10^6$ test cases were used, the authors concluded that the Spearman rank correlation of Squeeziness with FEP is close to 0.95.

In order to introduce the concept of Squeeziness, first it is important to remember the concept of entropy of a set.

Definition 8. Let S be a set and ξ_S be a random variable over S . We denote by σ_{ξ_S} the probability distribution induced by ξ_S . The entropy of the random variable ξ_S , denoted by $\mathcal{H}(\xi_S)$, is defined as:

$$\mathcal{H}(\xi_S) = - \sum_{s \in S} \sigma_{\xi_S}(s) \cdot \log_2(\sigma_{\xi_S}(s))$$

Using this entropy, Squeeziness is easily defined as the difference between the entropy of two sets. Specifically, the input and output sets of a total function.

Definition 9. Let $f : S \rightarrow \Theta$ be a total function and consider two random variables ξ_S and ξ_Θ ranging, respectively, over S and Θ . The Squeeziness of f , denoted by $\mathbf{Sq}(f)$, is defined as the loss of information after applying f to S , that is, $\mathcal{H}(\xi_S) - \mathcal{H}(\xi_\Theta)$.

This definition presents Squeeziness as the amount of information lost between the sets S and Θ through the function f . However, in our case we are working with SUTs, not functions. Moreover, Squeeziness was defined for a white-box scenario in which we could access the code during runtime and this is easily transform into a function. Luckily, in a black-box scenario the SUTs can be represented as Finite State Machines (FSMs) and FSMs can be seen as functions that transform sequences of input actions into sequences of output actions.

Definition 10. A Finite State Machine (FSM) is represented by a tuple $M = (Q, q_{in}, I, O, T)$ in which Q is a finite set of states, $q_{in} \in Q$ is the initial state, I is a finite set of input actions, O is a finite set of output actions, and $T \subseteq Q \times (I \times O) \times Q$ 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 say that M is deterministic if for all $q \in Q$ and $i \in I$ there exists at most one pair $(q', o) \in Q \times O$ such that $(q, i/o, q') \in T$; otherwise, we say that M is non-deterministic.

Although the previous definition of Squeeziness could be applied to the function induced by an FSM, this approach is not practical in most cases. The size of the SUTs usually limits the applicability of this formula and, for that end, a limited version was defined. First, we present some auxiliary concepts.

Definition 11. Let $M = (Q, q_{in}, I, O, T)$ be an FSM. We use the following notation:

1. Let $\sigma = (i_1, o_1) \dots (i_k, o_k) \in (I \times O)^*$ be a sequence of input/output actions and q be a state. We say that M can perform σ from q if there exist states $q_1 \dots q_k \in Q$ such that for all $1 \leq j \leq k$ we have $(q_{j-1}, i_j/o_j, q_j) \in T$, where $q_0 = q$. We denote this by either $q \xrightarrow{\sigma} q_k$ or $q \xrightarrow{\sigma}$. If $q = q_{in}$ then we say that σ is a trace of M .
2. Let $s = i_1 \dots i_k \in I^*$ be a sequence of input actions and q be a state. We define $\text{out}_M(q)s$ as the set

$$\{o_1 \dots o_k \in O^* | q \xrightarrow{(i_1, o_1) \dots (i_k, o_k)}\}$$

Note that if M is deterministic then this set is either empty or a singleton.

3. Let $q \in Q$ be a state. We define $\text{dom}_M(q)$ as the set

$$\{s \in I^* | \text{out}_M(q)s \neq \emptyset\}$$

If $q = q_{in}$ then we simply write dom_M . Similarly, we define $\text{image}_M(q)$ as the set

$$\{o_1 \dots o_k \in O^* | \exists i_1 \dots i_k \in I^* : q \xrightarrow{(i_1, o_1) \dots (i_k, o_k)}\}$$

If $q = q_{in}$ then we simply write image_M . We denote by $\text{dom}_{M,k}$ the set $\text{dom}_M \cap I^k$. Similarly, We denote by $\text{image}_{M,k}$ the set $\text{image}_M \cap O^k$.

4. We define $f_M : \text{dom}_M \rightarrow \mathcal{P}(\text{image}_M)$ as the function such that for all $s \in \text{dom}_M$ we have $f_M(s) = \{t \in O^* | t \in \text{out}_M(q_{in})s\}$. Note that if M is deterministic then this set is a singleton and we could define $f_M : \text{dom}_M \rightarrow \text{image}_M$.
5. Let $k > 0$. We define $f_{M,k}$ to be the function $f_M \cap (I^k \times O^k)$, where f_M denotes the associated set of pairs. Let $t \in \text{image}_M$. We define $f_M^{-1}(t)$ as $\{s \in I^* | t \in f_M(s)\}$.

Now we can define the limited version of Squeeziness that will be used along this work [129].

Definition 12. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider two random variables $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$ ranging, respectively, over the domain and image of $f_{M,k}$. The Squeeziness of M at length k is defined as

$$\text{Sq}_k(M) = \mathcal{H}(\xi_{\text{dom}_{M,k}}) - \mathcal{H}(\xi_{\text{image}_{M,k}})$$

Squeeziness for FSMs has some unexpected properties. For example, it is not monotonic with respect to k . That is, there exist FSMs where longer sequences can loss less information than shorter ones. Another interesting property is that bijective functions have a null Squeeziness.

Lemma 1. *Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. If $f_{M,k}$ is bijective then $\mathbf{Sq}_k(M) = 0$.*

Additionally, this limited version of Squeeziness gives us a potential to inform the choice of test cases. For example, we can use the Squeeziness values to determine, for a given test case length, if the likelihood of having FEP (once all the possible inputs of such length have been tested) is greater than 0.

5.2 The Deterministic Case

Focusing now in the deterministic case, some interesting properties of Squeeziness arise. Among them, a fundamental one states that given an FSM M and $k > 0$, the probability distribution of the random variable $\xi_{\mathbf{image}_{M,k}}$ is completely determined by the probability distribution of the random variable $\xi_{\mathbf{dom}_{M,k}}$. This is based on the fact that for each element $t \in \mathbf{image}_{M,k}$ it is true that

$$\sigma_{\xi_{\mathbf{image}_{M,k}}}(t) = \sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\mathbf{dom}_{M,k}}}(s) \quad (5.1)$$

Using this property and the entropy partition property [60], we can rewrite Squeeziness in terms of the inverse images partition of the input space, that is, using only the probability distribution on inputs given by $\xi_{\mathbf{dom}_{M,k}}$.

Corollary 1. *Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider a random variable $\xi_{\mathbf{dom}_{M,k}}$ ranging over the domain of $f_{M,k}$. We have that*

$$\mathbf{sq}_k(M) = - \sum_{t \in \mathbf{image}_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\mathbf{dom}_{M,k}}}(s) \right) \cdot \mathcal{R}_M(t) \quad (5.2)$$

where the term $\mathcal{R}_M(t)$ is equal to

$$\left(\sum_{s \in f_M^{-1}(t)} \frac{\sigma_{\xi_{\mathbf{dom}_{M,k}}}(s)}{\sigma_{\xi_{\mathbf{dom}_{M,k}}}(f_M^{-1}(t))} \cdot \log_2 \left(\frac{\sigma_{\xi_{\mathbf{dom}_{M,k}}}(s)}{\sigma_{\xi_{\mathbf{dom}_{M,k}}}(f_M^{-1}(t))} \right) \right) \quad (5.3)$$

This new formulation of Squeeziness is totally parameterised by the distribution over the inputs of the function. The optimal situation would be to know which distribution is that one (for example, using user behaviour data).

However, if we do not know the distribution then we have to set it. Between all the possible distributions, there are two that arise interesting values: the uniform distribution over the inputs (that maximises entropy [60]) and the distribution uniformly distributed in the largest inverse image of an element of the outputs and zero elsewhere (that maximises entropy loss [56]).

5.2.1 Maximum Entropy Principle

Using a uniform distribution over the inputs maximises the entropy value of the set of inputs. In this case, the weight of a single element of $\sigma_{\xi_{\text{dom}_{M,k}}}$ is $\frac{1}{|\text{dom}_{M,k}|}$. Thus, the weight of the inverse image of an output $t \in \text{image}_{M,k}$ is equal to $\frac{|f_M^{-1}(t)|}{|\text{dom}_{M,k}|}$. Finally, Squeeziness under this assumption is equal to

$$\begin{aligned} \text{Sq}_k(M) &= - \sum_{t \in \text{image}_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \frac{1}{|\text{dom}_{M,k}|} \right) \\ &\quad \cdot \left(\sum_{s \in f_M^{-1}(t)} \frac{\frac{1}{|\text{dom}_{M,k}|}}{\frac{|f_M^{-1}(t)|}{|\text{dom}_{M,k}|}} \cdot \log_2 \left(\frac{\frac{1}{|\text{dom}_{M,k}|}}{\frac{|f_M^{-1}(t)|}{|\text{dom}_{M,k}|}} \right) \right) \\ &= \frac{1}{|\text{dom}_{M,k}|} \cdot \sum_{t \in \text{image}_{M,k}} |f_M^{-1}(t)| \cdot \log_2(|f_M^{-1}(t)|) \end{aligned}$$

5.2.2 Maximum Loss of Information

Using the distribution that is uniformly distributed in the largest inverse image of an element of the outputs and zero elsewhere maximises the loss of information, that is, it gets the maximum possible value for Squeeziness. This case is considered the worst case scenario, as we are assuming that we are losing as much information as possible. In this case, we have to consider $t' \in \text{image}_{M,k}$ such that for all $t \in \text{image}_{M,k}$ we have that $|f_M^{-1}(t')| \geq |f_M^{-1}(t)|$. Then,

$$\sigma_{\xi_{\text{dom}_{M,k}}}(s) = \begin{cases} \frac{1}{|f_M^{-1}(t')|} & \text{if } s \in f_M^{-1}(t') \\ 0 & \text{otherwise} \end{cases}$$

Using this probability distribution, Squeeziness is defined as follows:

$$\begin{aligned} \mathbf{Sq}_k(M) &= - \left(\sum_{s \in f_M^{-1}(t')} \frac{1}{|f_M^{-1}(t')|} \right) \\ &\quad \cdot \left(\sum_{s \in f_M^{-1}(t')} \frac{1}{|f_M^{-1}(t')|} \cdot \log_2 \left(\frac{1}{|f_M^{-1}(t')|} \right) \right) \\ &= \log_2(|f_M^{-1}(t')|) \end{aligned}$$

It is important to remark that this distribution maximises Squeeziness because for any other possible distribution $\xi_{\text{dom}_{M,k}}$ we have $\mathbf{Sq}_k(M) \leq \log_2(|f_M^{-1}(t')|)$. This result is an immediate consequence of the following result [56].

Lemma 2. *Let us consider $2 \cdot n$ non-negative real numbers $a_1, \dots, a_n, p_1, \dots, p_n \in \mathbb{R}^+$. If for all $1 \leq i \leq n$ we have that $a_1 \geq a_i$ and $\sum_i p_i \leq 1$, then $\sum_i (p_i \cdot a_i) \leq a_1$.*

To finalise with the deterministic case, it is important to mention that we used two types of experiments to assess the suitability of this notion of Squeeziness as a proxy for the likelihood of having cases of FEP, in a black-box scenario. Specifically, we performed simulated experiments in the same fashion as those presented in [56], and also we performed real experiments with automatically generated FSMs. In both cases, we observed a strong correlation between the likelihood of FEP and Squeeziness. Moreover, in the experiments with FSMs we observed a slight improvement when increased the number of states.

5.3 The Generic Deterministic Case

One limitation of Squeeziness is that it was defined using Shannon's entropy [237]. However, there are many alternative notions to define what is intended as entropy. Rényi's entropy [219] is not only one of such alternative notions, but it also provides an infinite family of *entropies* due to the parameterisation of its definition by a positive real value α . Moreover, Rényi's entropy includes Shannon's entropy (as well as other notions appearing in the literature) as one of its notions, specifically, the notion corresponding to $\alpha = 1$.

Definition 13. *Let S be a set and ξ_S be a random variable over S . Let $\alpha \in \mathbb{R}_+ \setminus \{1\}$. The Rényi's entropy of the random variable ξ_S with respect to α , denoted by $\mathcal{H}_\alpha(\xi_S)$, is defined as:*

$$\mathcal{H}_\alpha(\xi_S) = \frac{1}{1 - \alpha} \cdot \log_2 \left(\sum_{s \in S} \sigma_{\xi_S}(s)^\alpha \right)$$

It is well-known that when α tends to 1, Rényi's entropy becomes Shannon's entropy, that is,

$$\lim_{\alpha \rightarrow 1} \mathcal{H}_\alpha(\xi_S) = \mathcal{H}(\xi_S) = - \sum_{s \in S} \sigma_{\xi_S}(s) \cdot \log_2(\sigma_{\xi_S}(s))$$

It is easy to expand the notion of Squeeziness to take into account this new notion of entropy [132].

Definition 14. Let S and Θ be sets and $f : S \rightarrow \Theta$ be a total function. Let us consider two random variables ξ_S and ξ_Θ ranging, respectively, over S and Θ , and $\alpha \in \mathbb{R}_+ \setminus \{1\}$. Rényi's Squeeziness of f with respect to α , denoted by $\text{Sq}_\alpha(f)$, is defined as the loss of information after applying f to S taking into account α , that is, $\mathcal{H}_\alpha(\xi_S) - \mathcal{H}_\alpha(\xi_\Theta)$.

Now, we can also define the limited version of Rényi's Squeeziness of an FSM in the same fashion than in Definition 12

Definition 15. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider two random variables $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$ ranging, respectively, over the domain and image of $f_{M,k}$. Let $\alpha \in \mathbb{R}_+ \setminus \{1\}$. Rényi's Squeeziness of M at length k with respect to α is defined as

$$\text{Sq}_{\alpha,k}(M) = \mathcal{H}_\alpha(\xi_{\text{dom}_{M,k}}) - \mathcal{H}_\alpha(\xi_{\text{image}_{M,k}})$$

Using again the total definition of the probability distribution over the outputs by the one over the inputs presented in Equation 5.1, we can define Rényi's Squeeziness in the following way

Lemma 3. Let $M = (Q, q_{in}, I, O, T)$ be an FSM, $k > 0$ and $\alpha \in \mathbb{R}_+ \setminus \{1\}$. Let us consider a random variable $\xi_{\text{dom}_{M,k}}$ ranging over the domain of $f_{M,k}$. We have that

$$\text{Sq}_{\alpha,k}(M) = \frac{1}{1 - \alpha} \cdot \log_2 \left(\frac{\sum_{s \in \text{dom}_{M,k}} \left(\sigma_{\xi_{\text{dom}_{M,k}}}(s) \right)^\alpha}{\sum_{t \in \text{image}_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s) \right)^\alpha} \right)$$

If α tends to 1 then we obtain Shannon's entropy [219] and we have

$$\text{Sq}_{1,k}(M) = - \sum_{t \in \text{image}_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s) \right) \cdot \mathcal{R}_M(t)$$

where the term $\mathcal{R}_M(t)$ is equal to

$$\sum_{s \in f_M^{-1}(t)} \frac{\sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\sigma_{\xi_{\text{dom}_{M,k}}}(f_M^{-1}(t))} \cdot \log_2 \left(\frac{\sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\sigma_{\xi_{\text{dom}_{M,k}}}(f_M^{-1}(t))} \right)$$

If α tends to ∞ then we obtain min-entropy [219] (that is, $\mathcal{H}_\infty(X) = -\log_2(\max_i p_i)$) and we have

$$\text{Sq}_{\infty,k}(M) = \log_2 \left(\frac{\max_{t \in \text{image}_{M,k}} \sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\max_{s \in \text{dom}_{M,k}} \sigma_{\xi_{\text{dom}_{M,k}}}(s)} \right)$$

This definition of Rényi's Squeeziness keeps some of the interesting properties of the original Squeeziness definition. For example, the nullification of the value when the function is bijective remains.

Lemma 4. *Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. If $f_{M,k}$ is bijective then $\text{Sq}_{\alpha,k}(M) = 0$.*

Similar to the original formulation, this extension of Squeeziness is totally parameterised by the distribution over the inputs of the function. Therefore, we present the cases for maximum entropy and maximum information loss.

5.3.1 Maximum Entropy Principle

The distribution that maximises entropy is the uniform distribution over the inputs. Then, under this distribution, the weight of a single element of $\text{dom}_{M,k}$ is $\frac{1}{|\text{dom}_{M,k}|}$ and the weight of the inverse image of an output $t \in \text{image}_{M,k}$ is equal to $\frac{|f_M^{-1}(t)|}{|\text{dom}_{M,k}|}$. Finally, the formula for Rényi's Squeeziness becomes:

$$\text{Sq}_{\alpha,k}(M) = \frac{1}{1-\alpha} \cdot \log_2 \left(\frac{|\text{dom}_{M,k}|}{\sum_{t \in \text{image}_{M,k}} (|f_M^{-1}(t)|)^\alpha} \right)$$

As usual, we have two special cases: α tending to 1 and to ∞ . If α tends to 1, then we are using Shannon's entropy and we are in the case of the previous section. If α tends to ∞ , then we are using min-entropy and we obtain the following formulation:

$$\text{Sq}_{\infty,k}(M) = \log_2 \left(\max_{t \in \text{image}_{M,k}} |f_M^{-1}(t)| \right)$$

5.3.2 Maximum Loss of Information

The distribution that maximises information loss (and therefore, that considers the worst case scenario) is the one uniformly distributed over the largest inverse image of an element of the outputs and zero elsewhere. In this case we have to consider $t' \in \mathbf{image}_{M,k}$ such that for all $t \in \mathbf{image}_{M,k}$ we have that $|f_M^{-1}(t')| \geq |f_M^{-1}(t)|$. Then,

$$\sigma_{\xi_{\text{dom}_{M,k}}}(s) = \begin{cases} \frac{1}{|f_M^{-1}(t')|} & \text{if } s \in f_M^{-1}(t') \\ 0 & \text{otherwise} \end{cases}$$

Using this probability distribution, Rényi's Squeeziness is defined as follows:

$$\text{Sq}_{\alpha,k}(M) = \log_2(|f_M^{-1}(t')|)$$

In this case, unlike the previous ones, Squeeziness does not depend on the value of α . In particular, the two special cases (α tending to 1 and α tending to ∞) have the same formulation.

To finalise with the generic deterministic case, it is important to note that we explored the effectiveness of this new notion of Squeeziness with multiple experiments. We computed the correlations between Rényi's Squeeziness and the likelihood of having cases of FEP for a set of randomly generated FSMs with different number of states. Moreover, we computed correlation for different values of α , specifically, we computed the extreme cases ($\alpha \in \{0, 1, \infty\}$) and uniformly distributed values in the ranges $[0, 1]$, $[1, 10]$ and $[10, 100]$. All the correlations fall in the range 0.5 – 0.9, with the best correlations achieved for values of $\alpha \in (2, 3)$. Moreover, we also observed a slight improvement when increasing the number of states of the generated FSMs while keeping the value of α constant.

Additionally, after observing that the *best* α value for different FSMs was different, we considered the development of a tool to determine which α value is the one that will correlate more with the likelihood of FEP for a specific FSM. This tool should receive a set of parameters from the FSM (for example, the number of states, the input alphabet size, etc...) or an FSM in a valid format and return the *best* α to assess the likelihood of FEP, or even Rényi's Squeeziness of the FSM for such α . To that end, we created a tool that used an Artificial Neural Network (ANN) to perform the selection of α for the given FSM [134].

This is one of the cases where the union of Information Theory and Artificial Intelligence helps to solve a problem. In this case, through the use of an ANN we are improving the use of Rényi's Squeeziness to assess the likelihood of FEP for a given FSM. Moreover, we are facilitating its use to a wide range of users that otherwise would not use Rényi's Squeeziness due to the complexities associated to select a proper value for α .

5.4 The Non-Deterministic Case

Another limitation of Squeeziness was that it was defined to deal only with deterministic systems. This limits its applicability as using it for systems with non-determinism returns nefarious results. Therefore, an extension of Squeeziness to deal with non-deterministic systems was much needed.

In order to define this extension, it is important to analyse the new scenario: up until now we considered that FEP was produced due to a loss of information in the system. However, in the non-deterministic case an increase of information (produced by the inputs leading to multiple outputs) also generates FEP. Specifically, the fact that multiple outputs can be produced by the same input produces that a fault can be masked as one of the valid outputs. Also, it is important to notice that the Squeeziness formulation previously presented (in Equation 5.2) is simplified assuming a deterministic situation. Therefore, the definition of an appropriate non-deterministic version of Squeeziness needs to derive again the formula from Definition 9. Using the following result [129]

$$\begin{aligned} & \mathcal{H}(\xi_{\text{image}_{M,k}}) + \mathcal{H}(\xi_{\text{dom}_{M,k}} | \xi_{\text{image}_{M,k}}) \\ & \quad \parallel \\ & \mathcal{H}(\xi_{\text{dom}_{M,k}}) + \mathcal{H}(\xi_{\text{image}_{M,k}} | \xi_{\text{dom}_{M,k}}) \end{aligned}$$

we can rewrite Squeeziness as

$$\text{Sq}_k(M) = \mathcal{H}(\xi_{\text{dom}_{M,k}} | \xi_{\text{image}_{M,k}}) - \mathcal{H}(\xi_{\text{image}_{M,k}} | \xi_{\text{dom}_{M,k}})$$

and using some auxiliary results [129] and the total definition of the probability distribution over the outputs by the one over the inputs presented in Equation 5.1 we can finally rewrite Squeeziness as

$$\text{Sq}_k(M) = - \sum_{t \in \text{image}_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s) \right) \cdot \mathcal{R}_M(t) + \mathcal{S}_M \quad (5.4)$$

where the term $\mathcal{R}_M(t)$ is equal to

$$\sum_{s \in f_M^{-1}(t)} \frac{\sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\sigma_{\xi_{\text{dom}_{M,k}}}(f_M^{-1}(t))} \cdot \log_2 \left(\frac{\sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\sigma_{\xi_{\text{dom}_{M,k}}}(f_M^{-1}(t))} \right) \quad (5.5)$$

and the term \mathcal{S}_M is given in Figure 5.1.

The equivalent formulation of Squeeziness presented in Equation 5.2 was possible due to the fact that \mathcal{S}_M is equal to 0 if the FSM is deterministic. However, as we are in a non-deterministic scenario, $\mathcal{S}_M \neq 0$ and we have to start from Equation 5.4. In that equation, the term \mathcal{S}_M accounts for

$$\mathcal{S}_M = \sum_{s \in \text{dom}_{M,k}} \sigma_{\xi_{\text{dom}_{M,k}}}(s) \cdot \sum_{t \in f_M(s)} \frac{\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\sum_{t \in f_M(s)} \sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s)} \cdot \log_2 \left(\frac{\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\sum_{t \in f_M(s)} \sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s)} \right)$$

$$\mathcal{S}'_M = \sum_{t \in \text{image}_{M,k}} \sigma_{\xi_{\text{image}_{M,k}}}(t) \cdot \sum_{s \in f_M^{-1}(t)} \frac{\sum_{t \in f_M(s)} \sigma_{\xi_{\text{image}_{M,k}}}(t)}{\sum_{s \in f_M^{-1}(t)} \sum_{t \in f_M(s)} \sigma_{\xi_{\text{image}_{M,k}}}(t)} \cdot \log_2 \left(\frac{\sum_{t \in f_M(s)} \sigma_{\xi_{\text{image}_{M,k}}}(t)}{\sum_{s \in f_M^{-1}(t)} \sum_{t \in f_M(s)} \sigma_{\xi_{\text{image}_{M,k}}}(t)} \right)$$

Figure 5.1: Definition of \mathcal{S}_M (top) and \mathcal{S}'_M (bottom).

the increase of information introduced by non-determinism and, since it is subtracted, it is reducing the total loss of information of the FSM.

This new formulation of Squeeziness measures the loss of information produced by the FSM. However, in a non-deterministic scenario we also obtain FEP from the increase of information generated by non-determinism. Therefore, we need to measure such increment and the most natural way to do this is to use an *alternative* Squeeziness.

Definition 16. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider two random variables $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$ ranging, respectively, over the domain and image of $f_{M,k}$. The *Alternative Squeeziness* of M at length k is defined as

$$\text{ALSq}_k(M) = \mathcal{H}(\xi_{\text{image}_{M,k}}) - \mathcal{H}(\xi_{\text{dom}_{M,k}})$$

For this notion, we can also define an alternative definition of the probabilities for inputs and outputs. Specifically, in the same fashion as Equation 5.1, we can set that the probabilities of the inputs of the FSM are totally defined by the probabilities of their corresponding outputs, that is,

$$\sigma_{\xi_{\text{dom}_{M,k}}}(s) = \sum_{t \in f_M(s)} \sigma_{\xi_{\text{image}_{M,k}}}(t) \quad (5.6)$$

Using this fact, we can provide the following formulation for Alternative Squeeziness.

Corollary 2. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider

a random variable $\xi_{\text{image}_{M,k}}$ ranging over the image of $f_{M,k}$. We have that

$$\text{AlSq}_k(M) = - \sum_{s \in \text{dom}_{M,k}} \left(\sum_{t \in f_M(s)} \sigma_{\xi_{\text{image}_{M,k}}}(t) \right) \cdot \mathcal{R}'_M(s) + \mathcal{S}'_M \quad (5.7)$$

where the term $\mathcal{R}'_M(s)$ is equal to

$$\sum_{t \in f_M(s)} \frac{\sigma_{\xi_{\text{image}_{M,k}}}(t)}{\sigma_{\xi_{\text{image}_{M,k}}}(f_M(s))} \cdot \log_2 \left(\frac{\sigma_{\xi_{\text{image}_{M,k}}}(t)}{\sigma_{\xi_{\text{image}_{M,k}}}(f_M(s))} \right) \quad (5.8)$$

and the term \mathcal{S}'_M is given in Figure 5.1.

Similarly to what happened with Equation 5.4, in this equation the term \mathcal{S}'_M accounts for the decrease of information introduced by determinism and, as it is subtracted, it is reducing the total gain of information of the FSM. Additionally, this factor is equal to 0 when each output is produced by only one input, that is, when there is no possible loss of information.

Obviously, Squeeziness and Alternative Squeeziness are the opposite of one another. Therefore, their addition is always equal to 0. However, there is a *trick* that we can use to obtain a useful value. First, it is important to note that the correcting factors (S_M and S'_M) are diminishing the effect of the source of FEP that we are evaluating with each formula. Therefore, a quick fix would be to eliminate such terms and work with the rest of the formulas. This way, we are obtaining, on the one hand, the maximum possible information loss due to determinism and, on the other hand, the maximum possible information gain due to non-determinism.

However, this quick fix is not the only option. Other options are to use the full Squeeziness formula (see Equation 5.4) or add, instead of subtract, the correction factor (S_M) in such formula. However, after some experiments, we found that neither of these options obtained better solutions than to erase the correction factors and add the remaining formulas. We think that this difference is due to how the probability distributions are handled in each option, but further research is needed. An extended discussion about this concern can be found in [135]. Then, using these considerations, we defined the notion of Non-Deterministic Squeeziness as follows.

Definition 17. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider two random variables $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$ ranging, respectively, over the domain and image of $f_{M,k}$. We have that

$$\begin{aligned} \text{NDSq}_k(M) = & - \sum_{t \in \text{image}_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s) \right) \cdot \mathcal{R}_M(t) \\ & - \sum_{s \in \text{dom}_{M,k}} \left(\sum_{t \in f_M(s)} \sigma_{\xi_{\text{image}_{M,k}}}(t) \right) \cdot \mathcal{R}'_M(s) \end{aligned}$$

$$\begin{aligned}
\text{NDSq}_k(M) &= - \sum_{t \in \text{image}_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \frac{1}{|\text{dom}_{M,k}|} \right) \cdot \left(\sum_{s \in f_M^{-1}(t)} \frac{1}{\frac{|f_M^{-1}(t)|}{|\text{dom}_{M,k}|}} \cdot \log_2 \left(\frac{1}{\frac{|\text{dom}_{M,k}|}{|f_M^{-1}(t)|}} \right) \right) \\
&\quad - \sum_{s \in \text{dom}_{M,k}} \left(\sum_{t \in f_M(s)} \frac{1}{|\text{image}_{M,k}|} \right) \cdot \left(\sum_{t \in f_M(s)} \frac{1}{\frac{|f_M(s)|}{|\text{image}_{M,k}|}} \cdot \log_2 \left(\frac{1}{\frac{|\text{image}_{M,k}|}{|f_M(s)|}} \right) \right) \\
&= \frac{1}{|\text{dom}_{M,k}|} \cdot \sum_{t \in \text{image}_{M,k}} |f_M^{-1}(t)| \cdot \log_2(|f_M^{-1}(t)|) + \\
&\quad + \frac{1}{|\text{image}_{M,k}|} \cdot \sum_{s \in \text{dom}_{M,k}} |f_M(s)| \cdot \log_2(|f_M(s)|) \\
\text{NDSq}_k(M) &= - \left(\sum_{s \in f_M^{-1}(t')} \frac{1}{|f_M^{-1}(t')|} \right) \cdot \left(\sum_{s \in f_M^{-1}(t')} \frac{1}{|f_M^{-1}(t')|} \cdot \log_2 \left(\frac{1}{|f_M^{-1}(t')|} \right) \right) \\
&\quad - \left(\sum_{t \in f_M(s')} \frac{1}{|f_M(s')|} \right) \cdot \left(\sum_{t \in f_M(s')} \frac{1}{|f_M(s')|} \cdot \log_2 \left(\frac{1}{|f_M(s')|} \right) \right) \\
&= \log_2(|f_M^{-1}(t')|) + \log_2(|f_M(s')|)
\end{aligned}$$

Figure 5.2: Definition of $\text{NDSq}_k(M)$ under maximum entropy (top) and under maximum information balance (loss and gain) (bottom).

where the terms $\mathcal{R}_M(t)$ and $\mathcal{R}'_M(s)$ are given in Equations 5.5 and 5.8, respectively.

Now, same as with previous cases, this formulation of Non-Deterministic Squeeziness is parameterised by the distribution over the inputs. However, unlike previous cases, it is also parameterised by the distribution over the outputs. Therefore, in case we do not have the real distributions, it is necessary to set ones manually. Moreover, it is also necessary to decide if the distribution of the outputs will be fixed by the one of the inputs (or viceversa) or if we will set a different distribution for each set. Due to preliminary experiments, we decided to set a different distribution for each inputs and outputs, as this configuration gives the best results. Here, we present the distributions that maximise entropy (for both inputs and outputs) and that generate the maximum balance (maximum information loss and gain).

5.4.1 Maximum Entropy Principle

The maximisation of entropy is achieved by setting both distributions to follow the uniform one. This way, the weight of a single element of $\sigma_{\xi_{\text{dom}_{M,k}}}$ would be $\frac{1}{|\text{dom}_{M,k}|}$ and of $\sigma_{\xi_{\text{image}_{M,k}}}$ would be $\frac{1}{|\text{image}_{M,k}|}$. Thus, the weight of the inverse image of an output $t \in \text{image}_{M,k}$ would be equal to $\frac{|f_M^{-1}(t)|}{|\text{dom}_{M,k}|}$ and the weight of the image of an input $s \in \text{dom}_{M,k}$ would be equal to $\frac{|f_M(s)|}{|\text{image}_{M,k}|}$.

Taking into account these values, the Non-Deterministic Squeeziness formula can be found in Figure 5.2 (top).

5.4.2 Maximum Information Balance (Loss and Gain)

In order to maximise information balance, it is necessary to set the distribution over the inputs to the one that is uniformly distributed over the largest inverse image of an element of the outputs and zero elsewhere, and the one over the outputs to the one that is uniformly distributed over the largest image of an input and zero elsewhere.

Formally, consider $t' \in \text{image}_{M,k}$ such that for all $t \in \text{image}_{M,k}$ we have that $|f_M^{-1}(t')| \geq |f_M^{-1}(t)|$. Then,

$$\sigma_{\xi_{\text{dom}_{M,k}}}(s) = \begin{cases} \frac{1}{|f_M^{-1}(t')|} & \text{if } s \in f_M^{-1}(t') \\ 0 & \text{otherwise} \end{cases}$$

Similarly, consider $s' \in \text{dom}_{M,k}$ such that for all $s \in \text{dom}_{M,k}$ we have that $|f_M(s')| \geq |f_M(s)|$. Then,

$$\sigma_{\xi_{\text{image}_{M,k}}}(t) = \begin{cases} \frac{1}{|f_M(s')|} & \text{if } t \in f_M(s') \\ 0 & \text{otherwise} \end{cases}$$

After using these probability distributions in the definition of Non-Deterministic Squeeziness, the formulation can be found in Figure 5.2 (bottom).

Finally, it is important to mention that two types of experiments were used to assess the suitability of this Non-Deterministic Squeeziness notion as a proxy for the likelihood of having cases of FEP in a non-deterministic system. Specifically, there was a set of simulated experiments, in the same fashion as those presented in [56], and a set of experiments using FSMs obtained from a benchmark. In both cases, there was a strong correlation between the likelihood of FEP and Squeeziness, with only few cases with not so strong correlation. After a careful analysis, it was possible to conclude that the source of those cases were FSMs with a high number of input sequences and low potential non-determinism. This happens due to the logarithmic nature of the formula, but we do not consider it a big flaw, although

further research is needed. An extended discussion about this concern can be found in [135].

We also explored how Non-Deterministic Squeeziness compares to the classical Squeeziness in non-deterministic systems and found that Non-Deterministic Squeeziness clearly outperforms Squeeziness, with only a negligible increment in computation time. Moreover, as Non-Deterministic Squeeziness is conservative with respect to Squeeziness when facing deterministic systems, it can also be applied to deterministic systems without effectiveness loss.

5.5 Associated Papers

- **Alfredo Ibias, Robert M. Hierons and Manuel Núñez.** *Using Squeeziness to test component-based systems defined as Finite State Machines.* Information and Software Technology 112, pages: 132-147. ([129])
- **Alfredo Ibias and Manuel Núñez.** *Estimating fault masking using Squeeziness based on Rényi's entropy.* 35th ACM/SIGAPP Symposium on Applied Computing, SAC '20, pages: 1936-1943, ACM. ([132])
- **Alfredo Ibias and Manuel Núñez.** *SqSelect: Automatic assessment of Failed Error Propagation in state-based systems.* Expert Systems With Applications 174, pages: 114748. ([134])
- **Alfredo Ibias and Manuel Núñez.** *Squeeziness for Non-Deterministic Systems.* Unpublished. ([135])

Chapter 6

Test Case Generation

*Program testing can be
a very effective way
to show the presence of bugs,
but is hopelessly inadequate
for showing their absence.*

Edsger Dijkstra

Test case generation is a research line focused on the performance of the tools used to test a given SUT. Generating test cases is a fundamental problem of Software Testing as executing test cases is the main way of finding faults. Therefore, it is important to generate test cases with high fault finding capability. Usually, in order to measure the fault finding capability of a test case, a proxy measure is used.

There exist many algorithms and measures to address the test case generation problem. Among them, the measures based on the Test Set Diameter (TSDm) [80] measures are the state-of-the-art. This is a family of Information Theory based measures focused on diversity. The idea of diversity-based measures is that obtaining test cases as diverse as possible in the test suite will lead to a higher fault finding capability. Following this idea, in the research conducting to this thesis, a new Information Theory based measure, also focused on diversity, was introduced: Biased Mutual Information (BMI).

In Section 6.1 we present the theoretical background underlying this research line. Then, in Section 6.2 we propose an algorithm to generate test suites guided by the Test Set Diameter measures, in Section 6.3 we propose a new measure called Biased Mutual Information and an algorithm to generate test suites guided by such measure, and in Section 6.4 we propose an algorithm to generate test suites guided by coverage-based measures. Finally, in Section 6.5 we list the papers related to the work presented in this chapter.

6.1 Theoretical Background

The context in which test case generation will be addressed is a black-box approach: the tester does not have access to the source code of the SUTs. Fortunately, the tester has access to a specification of the SUT in the form of an FSM, a finite labelled transition system in which input/output pairs label transitions.

Definition 18. We say that $M = (Q, q_{in}, I, O, T)$ is a Finite State Machine (FSM), where Q is a finite set of states, $q_{in} \in Q$ is the initial state, I is a finite set of inputs, O is a finite set of outputs, and $T \subseteq Q \times (I \times O) \times Q$ is the transition relation. A transition $(q, (i, o), q') \in T$, also denoted by $q \xrightarrow{i/o} q'$ or by $(q, i/o, q')$, means that from state q after receiving input i it is possible to move to state q' and produce output o .

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

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

In our work, we assume that the FSMs that specify our SUTs are deterministic. Moreover, we assume the *test hypothesis* [140]: the SUT can be modelled as an object described in the same formalism as its specification (in our case, an FSM). As we are in a black-box framework, we do not need to have access to such model, we only need to assume that it exists. Actually, it would be enough to assume that the SUT reacts with an outputs sequence to an inputs sequence.

In order to compare the behaviour of the SUT with respect to its specification we need to define such behaviours. We can do so with the concept of *trace*.

Definition 19. Let $M = (Q, q_{in}, I, O, T)$ be an FSM, $q \in Q$ be a state and $\sigma = (i_1, o_1) \dots (i_k, o_k) \in (I \times O)^*$ be a sequence of pairs. We say that M can perform σ from q if there exist states $q_1 \dots q_k \in Q$ such that for all $1 \leq j \leq k$ we have $(q_{j-1}, i_j/o_j, q_j) \in T$, where $q_0 = q$. We denote this by either $q \xrightarrow{\sigma} q_k$ or $q \xRightarrow{\sigma}$. If $q = q_{in}$ then we say that σ is a trace of M . We denote by $\mathbf{traces}(M)$ the set of traces of M . Note that for every state q we have that $q \xRightarrow{\epsilon} q$ holds. Therefore, $\epsilon \in \mathbf{traces}(M)$ for every FSM M .

The notion of trace can be used to define the notion of test case: a (input action, output action) pairs sequence. A test suite will be a set of test cases.

Definition 20. Let $M = (Q, q_{in}, I, O, T)$ be an FSM. We say that a sequence $\tau = (i_1, o_1) \dots (i_k, o_k) \in (I \times O)^+$ is a test case for M if $\tau \in \mathbf{traces}(M)$. We define the length of τ as the length of the sequence, that is, $|\tau| = k$. We define the sequence of inputs of τ as $\alpha = i_1 \dots i_k$ and the sequence of outputs of τ as

$\beta = o_1 \dots o_k$ (we will sometimes use the notation $\tau = (\alpha, \beta) \in (I^+ \times O^+)$). We write $(i, o) \in \tau$ to denote that the pair (i, o) appears in the test case τ ; $(i, o) \in_n \tau$ denotes that the pair (i, o) appears n times in the test case τ .

A test suite for M is a set of test cases for M . Given a test suite $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$, we define the length of the test suite as the sum of the lengths of its test cases, that is, $|\mathcal{T}| = \sum_{i=1, \dots, n} |\tau_i|$.

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

The idea is that given a test case (α, β) for M , the application of the inputs sequence α to a correct system with respect to M should return the outputs sequence β .

When generating test cases, it is not enough to generate any test case. We are usually looking for a test case that fulfils some properties (ideally, properties that are a proxy of having a high fault finding capability). To assess these properties, a common tool is to define a measure notion that would tell how much a test case fulfils a property. We can define the concept of measure as a function that receives an FSM and a test suite and returns a real number representing how good the test suite is considered by the measure.

Definition 21. A measure is a function

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

Although this definition of measure needs the information of both the test suite and the specification, we can also define measures that only use the test suite.

Generating a test case or a test suite that obtains a high score for a measure is usually an NP-hard problem (due to a combinatorial explosion). A useful tool to find approximations to NP-hard problems (that comes from the Artificial Intelligence field) is the Genetic Algorithm. Therefore, in our work we decided to rely on Genetic Algorithms to address this problem. A genetic algorithm is composed by:

- An encoding of the population in *genes*.
- An *initial population*, that is, randomly generated individuals expressed in the selected codification.
- A *fitness function* to evaluate the population.
- A *stopping criterion*.

- A *next population selection method*, which pairs the individuals for the next steps.
- A *crossover method* that generates new individuals from the mixture of the genes of the existing ones.
- A *mutation method* that can modify some individuals in order to obtain new genes that might have not been present before.
- A *replacement method*, which usually keeps the best individuals and discards the worst ones (with respect to the fitness function values).

The structure of a genetic algorithm is given in Algorithm 1. This algorithm is divided into the following steps:

- *Initialisation* step: generates the initial *population*, acting as a seed for the whole process. This initialisation is usually random, in order not to bias the behaviour of the algorithm.
- *Selection* step: focus on obtaining the most suited individuals to perform the following steps and achieving a better solution in next generations.
- *Crossover* step: pairs the individuals obtained in the selection step and exchanges parts of the structure within each couple.
- *Mutation* step: considers each individual after the crossover step and, with a small probability, performs slight variations. This process, although might seem counter-intuitive, tends to avoid obtaining local optima solutions by possibly substituting the negative-impact elements of the individual for new ones.
- *Replacement* step: takes the current population and its offspring and decides which individuals amongst them conform the following generation.

The Genetic Algorithm iterates these steps to evolve the population to produce a better solution. Therefore, it requires a termination criteria, which usually considers a bound on the number of iterations.

Due to the graph nature of the FSM formalism, we need an expressive Genetic Algorithm. Specifically, we need a Genetic Programming Algorithm in which the codification of the population in genes does not use a linear structure (as a vector) but a tree-like structure [156]. Additionally, in order to ensure the correctness of the generated test suites, the population will be constructed from a grammar that can build all the valid test cases and only valid test cases. This grammar will be generated from the specification of the SUT. This way, along this research line we will use Grammar-Guided Genetic Programming Algorithms.

```

Initialise population;
Evaluate population;
while termination criterion not reached do
  | Select next population;
  | Perform crossover;
  | Perform mutation;
  | Evaluate population;
  | Replace population for next iteration;
end

```

Algorithm 1: Genetic algorithm: general scheme.

6.2 Using Test Set Diameter

The first approach that we explored to measure the fault finding capability of a test suite was based on Test Set Diameter (TSDm) [80], a measure based on Kolmogorov complexity [166] and focused on the diversity of the test cases conforming the test suite. The *Kolmogorov complexity* of a string is the length of the shortest program that produces such string. It can be approximated using Normalised Compression Distance [54].

Definition 22. *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 $\text{ncd}(x, y)$ 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 .

This measure is easily extendable to deal with bags (multi-sets) of strings.

Definition 23. *Let X be a bag 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 $\text{NCD}(X)$ the Normalised Compression Distance of X and we define it as*

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

where

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

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.

Finally, we define the Test Set Diameter of a test suite as the NCD of a bag of elements of a test case. For example, such bag could be the one of test inputs (Input-TSDm), the one of test outputs (Output-TSDm), or even the bag of execution traces (Trace-TSDm). In our work, as we are working in a black-box scenario, we compare between the bag of test inputs (Input-TSDm), the one of test outputs (Output-TSDm), and the bag of test input/test output pairs (Input/Output-TSDm).

Test Set Diameter is the current baseline for the generation of test suites in black-box testing. In an extensive study [110] it was shown that it outperformed many black-box alternatives and even some white-box ones. It is for this reason that we developed a framework to generate the best test suite with respect to their TSDm value. For this framework, we use a Grammar-Guided Genetic Programming Algorithm with the subsequent configuration.

6.2.1 Encoding

We use a tree structure to encode test suites. This tree will be generated from a grammar that conforms to the FSM, so that we ensure the correctness of the produced test cases. This grammar has the following components:

- A start non-terminal symbol S that starts the grammar.
- A non-terminal symbol T that introduces each test case of the test suite.
- A non-terminal symbol N for each state, where $N \in \mathbf{N}$ 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 case.
- A production rule $S \rightarrow T$ to generate the initial test case.
- A production rule $T \rightarrow T + T$ to introduce a new test case.
- A production rule $T \rightarrow 0$ to start each test case in the FSM initial state.
- A production rule $N \rightarrow 'a/b' + M$ for each transition from the state N to a state M with input/output pair (a, b) .
- A production rule $N \rightarrow 'null'$ for each state N to a terminal to represent the end of the test case.

Note that, given an FSM, this grammar can be automatically generated.

6.2.2 Initial population

For the initial population, we randomly generate 100 test suites of the desired length, using the grammar previously derived from the FSM. Each rule from the grammar can be triggered with the same probability, with the goal of having a uniform random initialisation.

6.2.3 Fitness function

The fitness function will be a Test Set Diameter (TSDm) measure, defined as a function that receives the FSM and a test suite and computes the TSDm value of the test suite. It will return a real value representing how *good* is the test suite according to the measure. In this case, the *best* test suites will be those that have a higher TSDm value. We implemented the Input-TSDm, the Output-TSDm and the InputOutput-TSDm measures.

6.2.4 Stopping criterion

We stop our algorithm after 100 epochs at most and at least after 20 epochs. We stop before the epoch 100 if the *best* test suite is the same along $0.2 \cdot \text{NumberOfPassedEpochs}$ epochs.

6.2.5 Selection method

As selection method we simply take the individuals of this iteration and pair them for the following steps. We do this simple selection method because we aim at an exploratory goal. Therefore, we consider that our selection method should take in account all the population to avoid losing genetic diversity.

6.2.6 Crossover method

We need a grammatical crossover to keep the population inside the space of the test suites generated by the grammar. We use a mixture between the Whigham crossover [180] and the standard grammatical crossover [59]. Additionally, we need all the test suites to have the same length. With all these requisites, we generated the crossover presented in Algorithm 2.

The probability of crossover is set to 90% due to the difficulty to find a pair of valid points where to cut the parents and create an offspring.

6.2.7 Mutation method

The mutation method consist in erasing one test case of the test suite and generating a new one with the same length. The probability in this case is set to 5% for each test case of each test suite, what is a common value in the literature [190].

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

Algorithm 2: Crossover algorithm.

6.2.8 Replacement method

Finally, as replacement method we use a variant of elitist reduction [190]. First, we directly pass to the next epoch those test suites with a fitness score over the mean. Then, for those that fall under the mean we give them a second chance of passing them to the next epoch if their score is higher than the mean minus a random number (modulo the distance between the mean and the best score). This replacement method aims at keeping both the information from the best test suites and some of the information of the worst ones, with the idea that not losing all the genetic information comprised in the worst test suites will improve the results in next iterations.

We performed multiple experiments to compare the results of the TSDm measures with respect to using a random algorithm and also between them. In general, we found that a genetically generated test suite kills more mutants than a randomly generated test suite (this was the case in a 75.3% of the cases), both with the same length. The genetically generated test suite killed an average of 47.1% of the mutants, while the randomly generated test suite killed an average of 43.9% of the mutants.

6.3 Using Biased Mutual Information

In order to improve the results obtained when using Test Set Diameter, we need to devise new measures that better approximate the fault finding capability of a test case. Following the trend of measures based on a diversity approach, we propose a new measure inspired by the notion of mutual information [237].

Definition 24. *Let A and B be two sets and ξ_A and ξ_B be two discrete random variables ranging, respectively, over A and B . We denote by $I(\xi_A; \xi_B)$ the mutual information of ξ_A and ξ_B and we define it as*

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

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

We expect that a measure based on mutual information increases diversity by penalising the similarity between test cases. This way, we should obtain test suites with as low mutual information as possible. In this work we consider both the input and the output of a test case for the mutual information computation, with the goal of increasing input and output diversity [8]. Note that we have access to both the inputs and outputs of the SUT because we have a specification (in the form of an FSM).

Maximising diversity is a good proxy for fault detection. If we are testing an FSM with a set of input/output pairs labelling its transitions, whenever we take two input/output pairs we can have two scenarios. In the first scenario, both pairs are different and we know that they correspond to different transitions. This is the desired scenario. In the second scenario, both pairs are equal and we are in a situation where we do not know if they correspond to different transitions or not. In this last scenario, the probability that both pairs correspond to the same transition is equal to $\frac{1}{m}$, where m is the number of transitions of the FSM labelled with that same input/output pair. Therefore, even although for high values of m the probability tends to 0, it is never 0, but the difference between a low value or a high value of m is important. This also implies that we cannot automatically discard test suites where a pair is repeated, but instead we have to consider the probability of such pair of corresponding to the same transition.

In order to compute the mutual information of two test cases, we have to transform each of them into a set of pairs abstracting their position in the sequence. Additionally, given two test cases τ_1 and τ_2 , we need a definition of the probability distribution $\sigma_{\xi_\tau}(x)$ to compute the mutual information $I(\xi_{\tau_1}; \xi_{\tau_2})$. After multiple considerations (that are explained in [136]), the best solution for computing these probabilities is to use a formula that does not arise a probability distribution over the elements of the test case, but

that instead uses the probability presented before (that ranges over the set of transitions of the FSM with the same input/output pair). In order to retain the structure of the original formulation, we will keep the notation σ although we should be aware that it does not refer to a probability distribution. Moreover, we will use the term $\sigma_\tau(x)$, with τ as an implicit parameter, to represent that $\sigma_\tau(x)$ is only defined for x if $x \in \tau$. Finally, it is important to mention that we write $(i, o) \in_m M$ to denote that the pair (i, o) appears in m transitions of M and $(i, o) \in_n \tau$ denotes that the pair (i, o) appears n times in the test case τ .

Definition 25. Let $M = (Q, q_{in}, \mathcal{I}, \mathcal{O}, T)$ be an FSM, τ be a test case for M and $x \in \mathcal{I} \times \mathcal{O}$ be an input/output pair such that $x \in \tau$. We let:

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

We define the composition of two test cases τ_1, τ_2 of M , for input/output pairs $x_1 \in \tau_1, x_2 \in \tau_2$, as:

$$\sigma_{\tau_1, \tau_2}(x_1, x_2) = \begin{cases} \frac{1}{m_1} \cdot \frac{1}{m_2} & \text{if } x_1 = x_2 \\ \frac{1}{m_1} \cdot \frac{1}{m_2} & \text{otherwise} \end{cases}$$

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 $x_1 \in \tau_1$ and $x_2 \in \tau_2$ to appear in M .

Finally, we redefine the mutual information of two test cases as:

$$I(\tau_1; \tau_2) = \sum_{x_1 \in \tau_1} \sum_{x_2 \in \tau_2} \sigma_{\tau_1, \tau_2}(x_1, x_2) \cdot \log_2 \frac{\sigma_{\tau_1, \tau_2}(x_1, x_2)}{\sigma_{\tau_1}(x_1) \cdot \sigma_{\tau_2}(x_2)}$$

It is important to remark that we are assuming a uniform distribution over the set of transitions of the FSM with the same input/output pair. We could choose another distribution for those probabilities, but preliminary experiments did not show any improvement and it would only complicate the computation. Therefore, we keep the uniform distribution unless a *real* distribution is known. Thanks to this uniform distribution, we can simplify the formulation in the following way.

Lemma 5. Let M be an FSM and τ_1, τ_2 be test cases for M . We have

$$I(\tau_1; \tau_2) = \sum_{x \in \tau_2} n_x \cdot \frac{\log_2(m_x)}{m_x}$$

where m_x is such that $x \in_{m_x} M$ and n_x is such that $x \in_{n_x} \tau_1$.

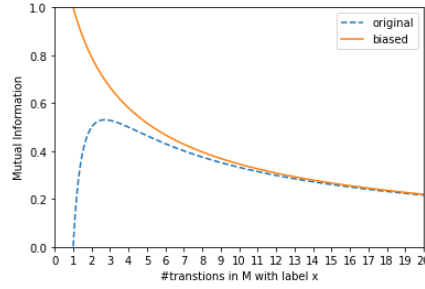


Figure 6.1: Comparison plot between mutual information and biased mutual information.

However, this formulation has some undesirable properties. Specifically, it is not monotonic and it is equal to 0 if all the transitions of the specification have different input/output pairs. For this formula to be useful we would like it to be monotonic and that it avoids “division by zero”. Therefore, we need to slightly modify it. Specifically, we decided to perform a small translation in the X axis of the logarithm of the formula. To observe the difference between both options, Figure 6.1 shows the behaviour of the un-translated formula as a dashed curve, and of the translated one as a solid curve. Using this translated version we define biased mutual information.

Definition 26. Let M be an FSM and τ_1, τ_2 be test cases for M . We say that the biased mutual information (bmi) of τ_1 and τ_2 is given by

$$bmi(\tau_1; \tau_2) = \sum_{x \in \tau_2} n_x \cdot \frac{\log_2(m_x + 1)}{m_x}$$

where m_x is such that $x \in_{m_x} M$ and n_x is such that $x \in_{n_x} \tau_1$.

We will use the biased mutual information of two test cases as the basis of a measure for a test suite. Specifically, we will compute the bmi of each pair of test cases of the test suite. That is, given a specification M and a test suite $\mathcal{T} = \{\tau_1, \dots, \tau_k\}$, we apply

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

where m_x is such that $x \in_{m_x} M$ and n_x is such that $x \in_{n_x} \tau_j$. We will call this formula $\alpha(\mathcal{T})$ in the definition of the Biased Mutual Information of \mathcal{T} .

In addition to the biased mutual information between two test cases, it is also desirable to account for the repetitions inside each test case of the test suite. The goal is to penalise test suites having test cases with many repeated input/output pairs, even if these pairs do not appear in other test

cases of the test suite. This way, given a specification M and a test suite $\mathcal{T} = \{\tau_1, \dots, \tau_k\}$, for each test case $\tau \in \mathcal{T}$ we have:

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

where m_x is such that $x \in_{m_x} M$ and n_x is such that $x \in_{n_x} \tau$. $\beta(\tau)$ will be the *self-redundancy* factor in the definition of the Biased Mutual Information of a test suite \mathcal{T} .

In the definition of this *self-redundancy* factor, $\frac{(n_x-1) \cdot n_x}{2}$ is the sum of the first $n_x - 1$ integers, what represents the number of input/output pairs (x_1, x_2) such that x_1 and x_2 are both in the test case and $x_1 \neq x_2$. Additionally, $\frac{\log_2(m_x+1)}{m_x}$ is the biased mutual information between those pairs.

Finally, combining these two factors, we obtain the formula for the *Biased Mutual Information* (BMI) of a test suite.

Definition 27. Let M be an FSM and $\mathcal{T} = \{\tau_1, \dots, \tau_k\}$ be a test suite for M . We have

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

Before generating test suites using BMI, we evaluated its performance selecting between two test cases in multiple experiments. Specifically, we compared it with a random selection approach and with the Test Set Diameter (TSDm). In both cases BMI outperformed the selection performed by the other measures. Moreover, we found a (negative) correlation between the fault detecting ability of a test suite and the BMI of such test suite. These results suggest that the diversity of a test suite can be improved using knowledge about the specification. An additional result was that transition coverage was (slightly) more effective than BMI, although its computation time is higher than the one of BMI. Moreover, BMI can be used only with the information about the input/output pairs frequency, what means that there is potential to use BMI without having a specification, something that transition coverage cannot do. Therefore, BMI is more widely applicable than transition coverage.

During the development of BMI there were two major decisions that raised some concerns. The first one was the translation performed over the original mutual information formula, where other transformations were possible. In this case, we performed an exhaustive exploration of possible alternatives and concluded that the one that we choose contained the best balance between intuition, simplicity and being faithful to the original Information Theory formulas. The second decision was the definition of a probability distribution for the elements of the test cases, where real distributions were possible options. In this case we considered multiple approaches to define such probability distributions, but none of them was better than

our non-probability option (by an appreciable margin). This behaviour can be explained because the weight (probability) of each element of a test case does not change with the length of the test case, while any probability distribution will be influenced by the length of the test case over which it is defined. An extended discussion about these concerns can be found in [136].

In order to generate test suites using BMI we devise an improved version of the Grammar-Guided Genetic Programming Algorithm used in the previous section. Specifically, this version uses a new fitness function (BMI), an improved crossover method, and new probabilities for both the crossover and the mutation methods.

6.3.1 Fitness Function

The fitness function will be the BMI measure, defined as a function that receives the FSM and a test suite, and computes the BMI value of the test suite. It will return a real value representing how *good* is the test suite according to the measure. In this case, the *best* test suites will be those that have a lower BMI value.

6.3.2 Crossover Method

We need a grammatical crossover to keep the population inside the space of the test suites generated by the grammar. We improved the crossover method presented in Section 6.2.6 by increasing its facility to be performed. Specifically, the new crossover generates two offspring and then crops the longer one to the desired size and randomly extends the shorter one to the same size. This crossover is presented in Algorithm 3.

The probability of this crossover is set to 75% due to the increased facility to happen in comparison with the previous crossover.

6.3.3 Mutation Method

The mutation method still consists in erasing one test case of the test suite and generating a new one with the same length. However, in this case, the probability is set to 10% because, in preliminary experiments, this was the probability that better worked with the new crossover.

We performed multiple experiments to compare the performance of the new approach with previous state-of-the-art measures for generating test suites. Specifically, we compared our Grammar-Guided Genetic Programming Algorithm guided by BMI with a random generation algorithm and with the algorithm from the previous section guided by the TSDm measures. In both cases BMI obtained test suites with higher fault finding capability, outperforming both alternatives. Additionally, we compared our methodology with the classical W [52] and Wp [86] methods and found that ours is

```

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 3: Improved Crossover Algorithm.

preferable in a realistic scenario where time and resources are scarce.

6.4 Using Coverage-Based Metrics

Following the point raised in the previous section about the coverage metrics outperforming BMI, we decided to explore how to use them to generate test suites in a reasonable time. To that end, we have to fix the notion of coverage. In other words, we have to define how much of the FSM a test suite traverses. We used as a basis the t -way coverage criterion and defined four different coverage criteria:

- t -way transition coverage: the percentage of sets of t consecutive transition labels that the test suite traverses. We define transition label as an input/output pair of the FSM (that is, an input/output pair that

corresponds to the execution of a transition).

- *t*-way extended transition coverage: the percentage of sets of *t* consecutive transitions that the test suite traverses. We define transition as the origin and target states and the input/output pair corresponding to the transition of the FSM between those states.
- *t*-way state coverage: the percentage of sets of *t* consecutive states that the test suite visits. We define state as a state of the FSM.
- *t*-way action coverage: the percentage of sets of *t* consecutive actions that the test suite executes. We define action as an input of the FSM.

We decided to use these four types of coverage because they are widely used in the literature [179, 243]. Each one of them focuses on a different component of the FSM and, therefore, a comparison between them is necessary. It is important to remark that state coverage can only be used in a white-box scenario, due to it needing the information about internal states. Action and transition coverage, in contrast, can be used in a black-box scenario as they only use observable information. Finally, the extended transition coverage that includes the initial and target states is also only possible in a white-box scenario.

t-way coverage groups the elements of the FSM in sets with exactly *t* consecutive elements, that is, elements such that there is a sequence of transitions of the FSM starting with one of them and ending with another one such that all the elements in between are in the set. We can check how many of these groups appear (without repetitions) in a test suite. We will use the percentage of the groups that appear in a test suite as the *t*-way coverage score, representing how much coverage such test suite provides.

Definition 28. *Given an FSM M , a grouping G (with $|G|$ elements) of its transition labels/transitions/states/actions in sets of t consecutive elements, and a test suite that traverses s of these sets (without repetitions), the t -way transition/extended transition/state/action coverage score is $\frac{s}{|G|}$.*

We call G set to the grouping G of the transition labels/transitions/states/actions in sets of t consecutive elements of an FSM.

In our work we have used values of *t* ranging from 1 to 3. In order to generate test suites using these coverage notions we used the same Grammar-Guided Genetic Programming Algorithm as in the previous section with BMI. The only difference is the change in the fitness function. In this case, we use one of the coverage criteria previously defined. Specifically, we define 12 fitness functions using *t*-way coverage criterion: we mixed $t \in \{1, 2, 3\}$ with transition, extended transition, state and action *t*-way coverage.

We performed multiple experiments to compare the measures between them and with using mutation score as a guide of the Genetic Algorithm. In

these comparisons, we found that 1-way transition coverage is the preferable choice when generating test suites if we are in a black-box scenario and 2-way state coverage if we are in a white-box scenario. However, some of our results raised some questions. Specifically, some coverage criteria had a very low computation time compared to the others. After a careful analysis we concluded that this was the effect of the difference in orders of magnitude between the G sets of different coverage criteria. These differences produce that some criteria have a G set so huge that the difference between two test suites is minimal, while others have a G set so small that all the test suites cover them. In either case, the lack of improvement from a test suite to another generates the stopping criterion to activate and thus the evolution is stopped sooner. Therefore, we can conclude that not all the coverage notions that we propose are useful as some will arise pointless results.

Regarding the choices performed during this research, there are some concerns. First, we choose to use coverage metrics that include all the elements that a test case traverses because using the ones that consider only the last element (or set of elements) of each test case would produce that all the test cases have the same score. Second, we compared with mutation score because it is the classical measure that is used to determine the fault finding capability of a test suite. Therefore, this looks like a useful baseline measure. However, for future work we would have to compare them to our previously proposed measure BMI. An extended discussion about these concerns can be found in [137].

6.5 Associated Papers

- **Alfredo Ibias, David Griñán and Manuel Núñez.** *GPTSG: A Genetic Programming test suite generator using Information Theory measures.* 15th International Work-Conference on Artificial Neural Networks, IWANN'19, LNCS 11506, pages: 716-728, Springer. ([128])
- **Alfredo Ibias, Manuel Núñez and Robert M. Hierons.** *Using mutual information to test from Finite State Machines: Test suite selection.* Information and Software Technology 132, pages: 106498. ([136])
- **Alfredo Ibias.** *Using mutual information to test from Finite State Machines: Test suite generation.* Unpublished. ([127])
- **Alfredo Ibias, Pablo Vazquez-Gomis and Miguel Benito-Parejo.** *Coverage-Based Grammar-Guided Genetic Programming Generation of Test Suites.* 2021 IEEE Congress on Evolutionary Computation, CEC'21, pages: 2411-2418, IEEE Computer Society. ([137])

Chapter 7

Integration Testing of Software Product Lines

*Most software today
is very much like an Egyptian pyramid
with millions of bricks piled
on top of each other,
with no structural integrity,
but just done by brute force
and thousands of slaves.*

Alan Kay

Addressing the Integration Testing of Software Product Lines (SPLs) is a research line focused on the quality of the interaction between multiple components. As such, it is an important problem when dealing with SPLs, because SPLs generate SUTs by composing features (that is, smaller programs or components). Therefore, testing how different features interact with each other is a classical case of Integration Testing. Improving and optimising this scenario is usually fulfilled by selecting a feature product with some specific characteristics. In order to produce such product, we proposed a special kind of Ant Colony Optimisation Algorithm (ACO).

In Section 7.1 we present the theoretical background common to this research line. Then, in Section 7.2 we propose a solution to the problem of selecting a feature combination with high probability and in Section 7.3 we propose a solution to the problem of selecting a feature combination with low testing cost. Finally, in Section 7.4 we list the papers related to the work presented in this chapter.

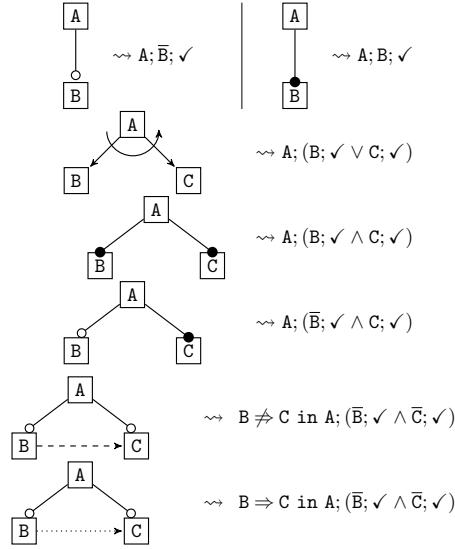


Figure 7.1: Examples of translation from FODA Diagrams into SPLA.

7.1 Theoretical Background

We decide to represent SPLs using a formal framework. There are multiple approaches to represent SPLs: FODA [149], RSEB [99], PLUS [79] and SPLA [11]. In our case, we are going to use the SPLA framework, as it represents SPLs as a process algebra. SPLA is a formal language capable to express FODA diagrams (Figure 7.1 shows some examples). For our two contributions in this area, we will use two of the SPLA extensions: one that has probabilities (SPLA^P [39]) whenever we find a choice in the representation of the SPL, and another one that has costs (SPLA-CRIS [38]) for the full product.

In order to work with the SPLA process algebra, a set of *features* will be considered, denoted by \mathcal{F} , and the elements A, B, C, \dots will stand for elements of \mathcal{F} . A special feature $\checkmark \notin \mathcal{F}$ will mark the end of a product.

We will use an Ant Colony Optimisation algorithm [71] (ACO). This algorithm finds solutions in a combinatorial optimisation problem.

Definition 29. A model $P = (\mathbf{S}, \Omega, f)$ of a combinatorial optimisation problem consists of:

- A search space \mathbf{S} defined over a finite set of discrete decision variables $X_i, i = 1, \dots, n$.
- A set Ω of constraints among the variables.
- An objective function $f : \mathbf{S} \rightarrow R_0^+$ to be minimised.

The generic variable X_i takes values in $D_i = v_i^1, \dots, v_i^{|D_i|}$. A feasible solution $s \in \mathbf{S}$ is a complete assignment of values to variables that satisfies all


```

Set parameters;
Initialise pheromone trails;
while termination criterion not reached do
  | Construct Ant Solutions;
  | Update Pheromones;
end

```

Algorithm 4: Ant Colony Optimisation algorithm: general scheme.

constraints in Ω . A solution $s^ \in \mathbf{S}$ is called a global optimum if and only if $f(s^*) \leq f(s) \forall s \in \mathbf{S}$.*

As explained in Chapter 4, in an ACO algorithm a set of agents (or ants) will explore a search space (defined as a graph) and will incrementally build a partial solution. Additionally, they will deposit a certain amount of pheromone on the edges of the graph, the amount depending on the quality of the solution found. In next iterations, the ants will use this pheromone information as a guide toward promising regions of the search space. The ACO general scheme is presented in Algorithm 4, and next we present a more detailed explanation of each step:

Construct Ant Solutions: In each iteration, a set of ants generates solutions exploring a path of the graph. They sequentially construct the path, adding one edge at a time, and selecting such edge using a stochastic mechanism biased by the pheromone associated with each of the possible edges.

Update Pheromones: After the ants have generated a solution, the pheromones of such solutions are modified, increasing those associated with good or promising solutions, and decreasing those that are associated with bad ones. Usually, all the pheromone values are decreased through pheromone evaporation. Later on, the pheromone levels associated with a chosen set of good solutions are increased.

7.2 Software Product Lines with Probabilities

We use the $\text{SPLA}^{\mathcal{P}}$ process algebra to represent SPLs with probabilities. In this formalism, all the considered probabilities are non-degenerated, that is, for all probability p we have $0 < p < 1$. Next, we define the $\text{SPLA}^{\mathcal{P}}$ syntax.

Definition 30. A probabilistic Software Product Line is a term generated by the following grammar:

$$\begin{aligned}
 P ::= & \checkmark \mid \text{nil} \mid \mathbf{A}; P \mid \bar{\mathbf{A}};_p P \mid P \vee_p Q \mid P \wedge Q \mid \\
 & \mathbf{A} \not\Rightarrow \mathbf{B} \text{ in } P \mid \mathbf{A} \Rightarrow \mathbf{B} \text{ in } P \mid P \setminus \mathbf{A} \mid P \Rightarrow \mathbf{A}
 \end{aligned}$$

where $\mathbf{A}, \mathbf{B} \in \mathcal{F}$, $\checkmark \notin \mathcal{F}$ y $p \in (0, 1)$. The set of all software product lines is denoted by $\text{SPLA}^{\mathcal{P}}$.

The probabilities of an $\text{SPLA}^{\mathcal{P}}$ expression usually represent the probability of each feature to be chosen. However, this does not need to be the case, as probabilities can represent any interesting and/or useful information. In our scenario we will assume that probabilities represent the likelihood of each feature of being chosen by a user, with the idea that looking for the feature combination with the higher probability will provide us with the one that needs more testing focus when testing the SPL.

$\text{SPLA}^{\mathcal{P}}$ has an operational semantics to guide how to interpret the expressions of the process algebra. This operational semantics produces a tree structure for each $\text{SPLA}^{\mathcal{P}}$ expression P . Traversing such tree, the set of products of P is computed. However, computing this tree is computationally expensive and can be infeasible in some cases.

Definition 31. Let $P, Q \in \text{SPLA}^{\mathcal{P}}$. We write $P \xrightarrow{s}_p Q$ if there exists a sequence of consecutive transitions

$$P = P_0 \xrightarrow{a_1}_{p_1} P_1 \xrightarrow{a_2}_{p_2} P_2 \cdots P_{n-1} \xrightarrow{a_n}_{p_n} P_n = Q$$

where $n \geq 0$, $s = a_1 a_2 \cdots a_n$ and $p = p_1 \cdot p_2 \cdots p_n$. We say that s is a trace of P .

Let $s \in \mathcal{F}^*$ be a trace of P . We define the product $[s] \subseteq \mathcal{F}$ as the set consisting of all features belonging to s .

Let $P \in \text{SPLA}^{\mathcal{P}}$. We define the set of probabilistic products of P , denoted by $\text{prod}^{\mathcal{P}}(P)$, as the set

$$\text{prod}^{\mathcal{P}}(P) = \{(pr, p) \mid p > 0 \wedge p = \sum \{q \mid P \xrightarrow{s\checkmark}_q Q \wedge [s] = pr\}\}$$

We propose a new ACO algorithm to explore this tree. The goal of this exploration is to find a combination of features that have a high enough probability for a given SPL. This algorithm needs three elements:

- A Software Product Line in the form of an $\text{SPLA}^{\mathcal{P}}$ expression. It will be the system that we are working with.
- An $\text{SPLA}^{\mathcal{P}}$ interpreter that allows us to explore the search space generated by the $\text{SPLA}^{\mathcal{P}}$ expression without fully computing it.
- An Ant Colony Optimisation algorithm. It leads the search for a feature combination with high probability.

The main point of this algorithm is that the $\text{SPLA}^{\mathcal{P}}$ interpreter will allow us to avoid the computation of the full $\text{SPLA}^{\mathcal{P}}$ expression tree. Therefore, this interpreter will return, for a given $\text{SPLA}^{\mathcal{P}}$ expression, its probability computed using the operational semantics of the $\text{SPLA}^{\mathcal{P}}$ algebra.

The ACO algorithm was developed to solve combinatorial optimisation problems. Therefore, we have to express our problem as a combinatorial optimisation one, using the following structure:

- Search space \mathbf{S} : the full $\text{SPLA}^{\mathcal{P}}$ tree, whose decision variables are the feature to choose next.
- Set of constraints Ω : It is composed by
 - a constraint stating that a valid path should end in a \checkmark feature and
 - a constraint stating that a valid path should fulfil the $\text{SPLA}^{\mathcal{P}}$ expression constraints.
- Objective Function f : the function assigning to each set of features their probability in the $\text{SPLA}^{\mathcal{P}}$ expression. In this case, we look to maximise it.

Having set the combinatorial optimisation problem, the ACO algorithm only needs to follow the general scheme presented in Algorithm 4. The only divergence with respect to the original algorithm is the fact that the ants generate the search space when exploring it. For that task, the $\text{SPLA}^{\mathcal{P}}$ interpreter will provide to the ACO algorithm the distances (in this case, the probabilities). Also, as the distances are probabilities, the total travelled distance is the product of the distance of each step (instead of its summation), so the final value is still a probability.

The proposal was evaluated by comparing the ACO approach with a brute force approach. Our experiments showed that we were able to save 67.33% of the computation time while obtaining products with 18.23% less probability than the optimal one. These results confirm that our approach could be very useful to work with SPLs with a huge number of features, that is, those for which a brute force algorithm could not be used due to the combinatorial explosion problem. Additionally, there are some interesting remarks about the results: first, there are some cases where the ACO algorithm saves more than 97% of the time while obtaining the optimal solution. Second, there is around 40% of the cases where the optimal solution is found and in these cases at least 30% of the execution time is saved.

7.3 Software Product Lines with Costs

When considering SPLs with costs, first we need to explain what we mean by *cost*. In our work, costs will represent the cost of testing a specific feature of the product, with the idea that reducing such cost will lead us to feature combinations that are easier to test. This situation is ideal when we are considering the addition of a feature to an existing SPL and we need to test whether this feature will not introduce new faults. In general, testing cost can refer to multiple different measures: monetary cost of testing such feature, time needed to test the product, or even the amount of resources

needed to perform such testing. In our work we will consider the testing cost in a broad sense and we will not fix which specific cost it is. Moreover, we assume that such costs are already included in the SPL, either produced by estimation, approximation or empirical methods. Finally, it is important to emphasise that in this work we are not focusing on finding a product that will arise a lot of faults, but instead we are looking for a product that will be easier to test. In other words, the testing cost is not a proxy for fault detection effectiveness.

We use the SPLA-CRIS process algebra to represent SPLs with costs. In this algebra, all the costs are positive.

Definition 32. *We will assume that we have a finite set of features \mathcal{F} and we will use $A, B, C \dots$ to denote single features. A Software Product Line is a term generated by the following Extended BNF-like expression:*

$$P ::= \surd \mid \mathbf{nil} \mid A; P \mid \bar{A}; P \mid P \vee Q \mid P \wedge Q \\ A \not\Rightarrow B \text{ in } P \mid A \Rightarrow B \text{ in } P$$

where $A, B \in \mathcal{F}$. We denote the set of terms of this algebra by SPLA.

Finally, given $P \in \text{SPLA}$, we define the products of P , denoted by $\text{prod}(P)$, as $\text{prod}(P) = \{[s] \mid s \in \text{traces}(P)\}$.

Our cost model is a cost function such that for each sequence of features (the already defined product) and a single feature (the new feature that we are adding) returns the cost of testing the addition of this new feature in the given product. In our case, we assume that costs can be represented by natural numbers. Note that we have to consider products without defined cost because they are incompatible with the definition of the SPL. For these products, we have to extend the set of costs with a new symbol \perp to represent *indefiniteness*.

Definition 33. *The set of costs is given by $\mathcal{N}_\perp = \mathcal{N} \cup \{\perp\}$. We extend arithmetic operations in the expected way: for any $x \in \mathcal{N}_\perp$ we have $x + \perp = \perp + x = \perp$ and $x \leq \perp$.*

A cost function is a function $c : \mathcal{F}^* \times \mathcal{F} \mapsto \mathcal{N}_\perp$.

Finally, it is important to remark that the position of the features in the trace will impact the cost of the product. In other words, the same product but with features chosen in different orders will have different testing costs. Therefore, we have to consider a set of costs for each product, one for each valid order of the product.

Definition 34. *Let c be a cost function. We consider the function $c_{\text{SPLA}} : \text{SPLA} \times \mathcal{P}(\mathcal{F}^*) \mapsto \mathcal{P}(\mathcal{N}_\perp)$ defined as follows:*

$$c_{\text{SPLA}}(P, p) = \{tc(P, s) \in \mathcal{N}_\perp \mid \exists s \text{ trace of } P : [s] = p\}$$

The exploration of an SPL, defined as an SPLA-CRIS expression, with the goal of finding a combination of features that have a low testing cost will be done with an ACO algorithm similar to one presented in the previous section. This algorithm needs three elements:

- A Software Product Line in the form of an SPLA-CRIS expression. It will be the system that we are working with.
- An SPLA-CRIS interpreter that allows us to explore the search space generated by the SPLA-CRIS expression without fully computing it.
- An Ant Colony Optimisation algorithm. It leads the search for a feature combination with high probability.

The main point of this algorithm is that the SPLA-CRIS interpreter will allow us to avoid the computation of the full SPLA-CRIS expression tree. Therefore, this interpreter will return, for a given SPLA-CRIS expression, its cost computed using the SPLA-CRIS operational semantics.

Same as before, the ACO algorithm needs that we express our problem as a combinatorial optimisation one. We have the following structure:

- Search space \mathbf{S} : the full SPLA-CRIS tree, whose decision variables are the feature to choose next.
- Set of constraints Ω : It is composed by:
 - a constraint stating that a valid path should end in a \checkmark feature,
 - a constraint stating that a valid feature combination should contain the previously selected feature and
 - a constraint stating that a valid path should fulfil the SPLA-CRIS expression constraints.
- Objective Function f : the function assigning to each set of features their cost in the SPLA-CRIS expression. In this case, we look to minimise it.

Having set the combinatorial optimisation problem, the ACO algorithm only needs to follow the general scheme presented in Algorithm 4. Again, the only divergence with respect to the original formulation is the fact that ants generate the search space by using distances (in this case, the costs) provided by the SPLA-CRIS interpreter. In this case, as the distances are costs, the total travelled distance is the sum of the distance of each step, as it is done in the original algorithm.

We compared the ACO approach with a brute force approach: we saved 99.14% of the computation time while obtaining products a 25.42% more expensive than the optimal one. These results confirm that our approach

could be very useful to work with SPLs with a huge number of features, that is, those for which a brute force algorithm could not be used due to the combinatorial explosion problem. However, in this case the brute force algorithm was only able to obtain the solution for part of the experimental subjects, as the combinatorial explosion problem made it impossible to be executed in the bigger SPLs. Therefore we also performed a comparison with a random selection algorithm. In this case our algorithm obtained, for the same execution time, solutions that are, on average, 14.87% cheaper. Moreover, in the worst cases the obtained solution is at least as expensive as the one obtained by the random selection algorithm.

Finally, it is worth to mention a recurrent question in this line of work: Is the ACO algorithm the best option to solve this problem? Although a thorough evaluation should be a matter of future work, there is some fundamental arguments to use ACO versus other Evolutionary Algorithms. First, it is important to note that we are searching in a search space that is an SPLA expression and such expression is easily transformed into a graph whose *final states* represent all the possible feature combinations that fulfil the expression restrictions. This implies that for applying algorithms like Particle Swarm Optimisation (PSO) we would need to transform such graph. This transformation is not trivial and would increase the complexity of the approach. Moreover, our graph can have cycles and, therefore, we would need to unfold such cycles if we want to apply a Genetic Programming Algorithm, that can deal only with acyclic graphs. This would increase the complexity of the approach too. Finally, our main task, after all, is to find a path in a graph and ACO was especially designed for this kind of problems. Therefore, ACO is easy to apply in this scenario, in contrast to the other main Evolutionary Algorithms, and it is optimised for such scenario. An extended discussion about this concern can be found in [131].

7.4 Associated Papers

- **Alfredo Ibias and Luis Llana.** *Feature Selection using Evolutionary Computation Techniques for Software Product Line Testing*. 2020 IEEE Congress on Evolutionary Computation, CEC'20, pages: 1-8, IEEE Computer Society. ([130])
- **Alfredo Ibias, Luis Llana and Manuel Núñez.** *Using Ant Colony Optimisation to Select Features having Associated Costs*. 33rd IFIP International Conference on Testing Software and Systems, ICTSS'21, pages: —, Lecture Notes in Computer Science, Springer (*to appear*). ([131])

Chapter 8

Detecting Hard-to-Kill Mutants

*There are two methods
in software design.
One is to make the program so simple,
there are obviously no errors.
The other is to make it so complicated,
there are no obvious errors.*

Tony Hoare

Mutation Testing helps to differentiate between test cases with higher fault finding capabilities and those with lower ones. To that end, it uses mutants, that is, modifications of the SUT, and the quality of such mutants can affect the classification of the test cases. Specifically, using the mutants that are harder-to-kill will allow for a finer grain classification of test cases, as less mutants will be killed by multiple test cases.

We proposed a swarm algorithm to detect hard-to-kill mutants. This kind of algorithms base their *intelligent* behaviour in their interactions as a swarm. They start with a swarm of agents (with little or not at all intelligence) that perform basic and repetitive tasks until the join work of all of them generates enough information to allow the swarm to perform intelligent decisions. This behaviour is the so called *emergency property*. We work in a scenario where we have m mutants and t test cases. The goal is to determine which mutants are hard to kill by this set of test cases. We have to take in account that our approach should keep a balance between computing time and results quality (in terms of the proportion of (un-)detected interesting mutants). Our swarm heuristic satisfies these requirements, as it avoids the application of the full set of test cases to all the mutants while, at the same time, gives more flexibility than a fixed cap (a fixed percentage of test cases that kill such mutant).

```

Set parameters;
Initialise kill matrix (all zeros);
Initialise iteration-hard-to-kill list (all mutants);
Initialise final-hard-to-kill list (empty);
while iteration-hard-to-kill list is not empty do
    | Assign a mutant and a set of test cases to each agent;
    | Each agent applies its set of test cases to its mutant;
    | Each agent updates the kill matrix;
    | Update iteration-hard-to-kill list;
end
Return final-hard-to-kill list;

```

Algorithm 5: Hard-to-kill mutants heuristic: general scheme.

This heuristic uses three elements:

- *Agents*: they conform the swarm that performs the evaluation.
- The *Kill Matrix*: it is the matrix where the agents store the information. It encodes for each pair mutant/test case if such test case killed the mutant, did not killed it, or it was not executed against it.
- The *hard-to-kill mutants lists*: they store the *promising* hard-to-kill mutants and are updated after each iteration of the algorithm. The algorithm needs two hard-to-kill mutants lists: one for storing the considered hard-to-kill mutants in the current iteration (to guide the algorithm) and one for storing the final solution.

Algorithm 5 presents a high-level view of the heuristic. For each iteration, our heuristic has four steps:

- For each agent, a mutant is chosen from the iteration-hard-to-kill list and $n \ll t$ test cases are chosen¹ to be applied to that mutant. Note that these test cases should not have been previously applied to this mutant. If a mutant cannot be removed from the iteration-hard-to-kill list with the remaining test cases, then it is added to the final-hard-to-kill list immediately and another mutant is taken.
- Each agent applies its set of n test cases to its mutant.
- Each agent updates the kill matrix using the following convention:
 - 1: the mutant has been killed by that test case.
 - 0: the test case has not been applied to the mutant.
 - -1: the mutant has not been killed by that test case.

¹Note that n is chosen by the user.

- The iteration-hard-to-kill list is updated.

Updating the iteration-hard-to-kill list is how the *emergency property* arises, as it is where the list of mutants considered hard-to-kill is updated. This step is performed after each iteration as a way to guide the development of the algorithm. After each iteration, the values of how many test cases killed each mutant are computed, storing the highest (max) and the lowest (min) ones. Then, the mutants whose value is less than or equal to $\min + \frac{\max - \min}{4}$ are marked as hard-to-kill. It is important to note that this bound is somewhat arbitrary. We selected this bound to try to get as hard-to-kill mutants those in the lower quarter of the obtained values, but it can be modified to any desired bound. Finally, the iteration-hard-to-kill list is purged of the mutants that are already in the final-hard-to-kill list and of those that have already been tested with all the test cases (and we add them to the final-hard-to-kill list).

Interestingly, our heuristic does not force the max value to be equal to the maximum number of test cases that kill a mutant because not all of these test cases will be executed on that mutant. For example, if a mutant is killed by all the test cases, then it will be frequently removed from the iteration-hard-to-kill list. Therefore, it will be hard that all the test cases will be executed over such mutant. Therefore, the max value will be lower than t (the total number of test cases) in most cases. Moreover, the difference between max and min will be lower or equal to $\frac{4}{3} \cdot n$ and, therefore, we can have $\max < \min + \frac{4}{3} \cdot n < t$ (and that will be usually the case).

Due to this behaviour, the heuristic is able to overcome the main problems when deciding which mutants are hard-to-kill. The first problem is the one regarding the application of all the test cases to all the mutants. Our heuristic does not necessarily apply all the test cases to all the mutants (this rarely happens). Therefore, it avoids the associated costs of other algorithms based on brute force. The second problem is the flexibility when deciding whether a mutant is hard-to-kill or not. Our heuristic is a more flexible approach than fixed percentages because it avoids extreme situations like empty hard-to-kill mutants sets or sets with all the mutants.

We performed several experiments where we compared our heuristic and a brute force approach, a cap (fixed percentage) approach, and a random approach. We found that our heuristic saves the 61.97% of the execution of test cases with respect to a brute force approach and needs a similar number of execution of test cases as a cap algorithm. Moreover, the sets produced by our heuristic have a good quality, far better than the quality of a random solution, and somewhat close to the quality of a cap approach. Finally, the last conclusion is that our heuristic is better than a cap approach because it avoids the extreme cases (from both sides) that the cap approach can produce, what makes our heuristic a more reliable solution.

8.1 Associated Papers

- **Alfredo Ibias and Manuel Núñez.** *Using a swarm to detect hard-to-kill mutants.* 2020 IEEE International Conference on Systems, Man, and Cybernetics, SMC'20, pages: 2190-2195, IEEE Computer Society. ([133])

Part IV

Conclusions

This part presents the thesis conclusions and lines of future work. Such conclusions include a review of the goals of the thesis and their level of achievement, a recapitulation of the research performed and their inclusion into the state-of-the-art (including a brief review of how the field has been improved), and a resume of the open lines that this thesis leaves as future work.

Chapter 9

Conclusions

*Every story has an end, but in life every
end is just a new beginning.*

Anonymous

This thesis has contributed to the Software Testing field, addressing four big problems and proposing new state-of-the-art solutions. Specifically, this thesis addresses the following problems: the Failed Error Propagation problem, the test case generation problem, the Integration Testing of Software Product Lines problem, and the detection of hard-to-kill mutants for Mutation Testing problem. The main goal of this thesis was to study how to use the available methods from Information Theory and Artificial Intelligence to solve Software Testing problems. This task needed of researching previous work, carefully studying alternatives and a several iterations when an alternative was considered to have potential to outperform current state-of-the-art. Following this procedure, after a lot of experiments, we produced the proposals presented in this thesis. Therefore, we can conclude that the main goal has been fulfilled substantially.

The secondary goal was to find synergies between Information Theory and Artificial Intelligence to better solve the problems addressed in this thesis. This goal has been fulfilled to some extent. Specifically, there are two problems for which some of the solutions mix tools coming from both fields: Failed Error Propagation and test case generation. In the Failed Error Propagation problem we found a problem generated by an Information Theory measure (Rényi's Squeeziness) that could be resolved thanks to the application of an Artificial Neural Network, a useful tool from the Artificial Intelligence field. In the test case generation problem the synergy is even more intricate, as we developed a Genetic Algorithm that needed a fitness function to guide its evolution, and we defined an Information Theory-based measure (TSDm) and an Information Theory-inspired measure (BMI) to be such fitness function. However, for the other problems we were not able to

find synergies and that would be matter of future work.

Regarding the Failed Error Propagation (FEP) problem, in the beginning of this thesis its state-of-the-art was a novel measure called Squeeziness whose application was limited to deterministic programs in a white-box scenario. In this thesis we took such measure and extended it to be able to deal with deterministic programs but in a black-box scenario. After that, we further develop it in two directions: improving its performance and improving its applicability. To improve its performance we focused on improving its suitability as a proxy for the likelihood of FEP by extending it to deal with different notions of entropy, including those that actually improve its performance. Additionally, we developed a tool to simplify its application to real programs. To improve its applicability, we relaxed its restrictions by developing a new conservative formulation that allows to apply it to non-deterministic systems. These new developments would be the new state-of-the-art for different situations, while its union in a concept of Non-Deterministic Rényi's Squeeziness would be matter of future work.

Regarding the test case generation problem, the state-of-the-art was a set of measures called Test Set Diameter (TSDm) that were used to generate a test suite from a predefined set of test cases. In this thesis we improved the applicability of such measures by developing a Grammar-Guided Genetic Programming Algorithm to automatically generate test suites guided by such measures. With this development, the test suites generated by the TSDm measures will not be limited to choose between the test cases present in the predefined set of test cases. The TSDm measures use Information Theory measures to increase diversity inside the test suite. In order to improve them, we developed a new measure that also looks to increase diversity inside the test suite. However, this new measure uses not only the information from the test suite, but also the one from the specification of the SUT. This way we created the concept of Biased Mutual Information (BMI). This measure outperformed the TSDm measures by a margin. Moreover, during its development we improved the Genetic Algorithm previously developed. As a side research in this line, and following some hints, we also explored the use of the Genetic Algorithm previously presented to generate test suites guided by coverage-based measures, but its comparison with BMI is left for future work.

Regarding the Integration Testing of Software Product Lines (SPLs) problem, we started with the formal definition of an SPL using a process algebra named SPLA. During the development of this thesis, we took this process algebra and used it (specifically, two of its derivations) to choose a feature combination useful to test the integration of the different features of the SPL. The first approach used the version with probabilities (SPLA^P) to select the feature combination that was more probable to be chosen by a user, with the goal of giving more testing time to such configuration. The second approach

used the version with costs (SPLA-CRIS) to select the feature combination that had a lower testing cost and a given feature, with the goal of being able to cheaply test the integration of a new feature that we want to include in the SPL. For both problems, we developed an Ant Colony Optimisation (ACO) algorithm that required small tweaks to properly work in these scenarios, in which the computation of all the possible feature combinations of an SPL falls quickly into a combinatorial explosion problem. Both solutions obtained good results, showing how they can be properly used by any practitioner.

Finally, regarding the detection of hard-to-kill mutants, the main method was to use hard cap values, setting that such mutants would be those killed by less than a fixed percentage of test cases. In this thesis we improved this approach by using a *soft* cap and developing a swarm algorithm to implement and compute such cap. This way, the mutants regarded as hard-to-kill would be the hardest to kill by the set of test cases given to the algorithm. Therefore, some extreme cases (like empty hard-to-kill mutant sets or those with all the mutants) are avoided. This new approach gave really good results.

This thesis leaves some open lines that can be matter of future work. Regarding the Failed Error Propagation (FEP) problem, additionally to the already mentioned lines, we would like to explore approximations to compute Squeeziness (and its improvements) more efficiently. We would also like to generalise Squeeziness (and its improvements) to deal with models of asynchronous [118, 183, 184] and distributed [120, 119] systems. We would like to integrate the tool developed to compute Rényi's Squeeziness with other tools that automatise testing, like those that incorporate the efficient and systematic generation and processing of mutants [65, 42, 90, 91, 104]. It would be desirable to refine the definition of Non-Deterministic Squeeziness to solve its problems in limit cases and implement Non-Deterministic Squeeziness in a tool that automatise its application (similar to the one we developed for Rényi's Squeeziness).

Regarding the test case generation problem, in addition to the already mentioned lines, we would like to extend our Genetic Algorithm to generate test suites for systems represented by Extended Finite State Machines (EFSMs) (that is, FSMs with data), or even for other kind of systems, like Binary Decision Diagrams [4]. Also, we would like to explore the application of BMI in situations with no specification but with the knowledge of the frequency of the input/output pairs, or even without it but using some mechanism (like random testing) to estimate such frequencies. A second line of work might consider to devise measures like BMI for non-deterministic specifications. It is worth to extend the comparison between BMI and the TSDm measures, and explore a wider range of coverage notions, specifically t -way coverage with $t > 3$ should be considered. We would like to explore the coverage notions of Petri Nets [3] too. We would like to explore the

significance of the relation between the size of the G set and the length of the test suites. Finally, we would like to use another kind of evolutionary computation algorithms, instead of a Genetic Programming one, like Tree Swarm Optimisation [98, 97].

Regarding the Integration Testing of Software Product Lines (SPLs), we would like to explore the use of our framework for SPLs with confidences instead of probabilities. A more practical line of work would analyse the performance of our proposals with bigger and more complex SPLs with the goal of checking whether our techniques scales well. We would like to compare our ACO approach with other evolutionary algorithms. We would like to consider SPLs with existing feature selections produced by an expert and compare their quality to the quality of the feature combinations generated by our approaches. Finally, we would like to integrate our proposals to existing tools such as ProFeat [53], to represent product lines, PRISM [158], to analyse probabilistic systems, and MEdit4CEP-CPN [32], to represent complex events.

Regarding the detection of hard-to-kill mutants, we would like to assess how the chosen bound modifies the hard-to-kill mutants set. We would like to explore the performance of our algorithm in a weak mutation scenario. Next, we would like to compare our swarm approach to other evolutionary algorithms and, finally, explore the relationship between the hard-to-kill mutants set produced by our algorithm and the set of fault revealing mutants [204].

Part V

Publications

This part presents the articles that were published during the development of this thesis.

Chapter 10

Publications

In this chapter I present the publications that conform the corpus of this thesis. All the publications are included with the approval of all the co-authors, as well as the authorisation to publish from the publishers. The papers presented in this chapter are (in order):

- **Alfredo Ibias, Robert M. Hierons and Manuel Núñez.** *Using Squeeziness to test component-based systems defined as Finite State Machines.* Information and Software Technology 112, pages: 132-147. ([129])
- **Alfredo Ibias and Manuel Núñez.** *Estimating fault masking using Squeeziness based on Rényi's entropy.* 35th ACM/SIGAPP Symposium on Applied Computing, SAC '20, pages: 1936-1943, ACM. ([132])
- **Alfredo Ibias and Manuel Núñez.** *SqSelect: Automatic assessment of Failed Error Propagation in state-based systems.* Expert Systems With Applications 174, pages: 114748. ([134])
- **Alfredo Ibias, David Griñán and Manuel Núñez.** *GPTSG: A Genetic Programming test suite generator using Information Theory measures.* 15th International Work-Conference on Artificial Neural Networks, IWANN'19, LNCS 11506, pages: 716-728, Springer. ([128])
- **Alfredo Ibias, Manuel Núñez and Robert M. Hierons.** *Using mutual information to test from Finite State Machines: Test suite selection.* Information and Software Technology 132, pages: 106498. ([136])
- **Alfredo Ibias, Pablo Vazquez-Gomis and Miguel Benito-Parejo.** *Coverage-Based Grammar-Guided Genetic Programming Generation of Test Suites.* 2021 IEEE Congress on Evolutionary Computation, CEC'21, pages: 2411-2418, IEEE Computer Society. ([137])

- **Alfredo Ibias and Luis Llana.** *Feature Selection using Evolutionary Computation Techniques for Software Product Line Testing.* 2020 IEEE Congress on Evolutionary Computation, CEC'20, pages: 1-8, IEEE Computer Society. ([130])
- **Alfredo Ibias, Luis Llana and Manuel Núñez.** *Using Ant Colony Optimisation to Select Features having Associated Costs.* 33rd IFIP International Conference on Testing Software and Systems, ICTSS'21, pages: —, Springer (*to appear*). ([131])
- **Alfredo Ibias and Manuel Núñez.** *Using a swarm to detect hard-to-kill mutants.* 2020 IEEE International Conference on Systems, Man, and Cybernetics, SMC'20, pages: 2190-2195, IEEE Computer Society. ([133])

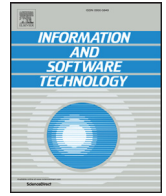
10.1 Using Squeeziness to test component-based systems defined as Finite State Machines

Authors	Alfredo Ibias, Robert M. Hierons and Manuel Núñez
Title	Using Squeeziness to test component-based systems defined as Finite State Machines
Publication Type	Journal
Venue	Information and Software Technology
Number	112
Year	2019
DOI/URL	https://doi.org/10.1016/j.infsof.2019.04.012
Pages	16
Authors' Contributions	Hierons and Núñez developed the theory. Ibias and Núñez designed the experiments. Ibias developed and executed the experiments. Ibias, Hierons and Núñez wrote the manuscript. Hierons and Núñez reviewed the manuscript.



Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Using Squeeziness to test component-based systems defined as Finite State Machines

Alfredo Ibias^a, Robert M. Hierons^b, Manuel Núñez^{a,*}^a Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Madrid, 28040, Spain^b Department of Computer Science, The University of Sheffield, Sheffield, S1 4DP, United Kingdom

A B S T R A C T

Context: Testing is the main validation technique used to increase the reliability of software systems. The effectiveness of testing can be strongly reduced by *Failed Error Propagation*. This situation happens when the System Under Test executes a faulty statement, the state of the system is affected by this fault, but the expected output is observed. Squeeziness is an information theoretic measure designed to quantify the likelihood of Failed Error Propagation and previous work has shown that Squeeziness correlates strongly with Failed Error Propagation in white-box scenarios. Despite its usefulness, this measure, in its current formulation, cannot be used in a black-box scenario where we do not have access to the source code of the components.

Objective: The main goal of this paper is to adapt Squeeziness to a black-box scenario and evaluate whether it can be used to estimate the likelihood that a component of a software system introduces Failed Error Propagation.

Method: First, we defined our black-box scenario. Specifically, we considered the Failed Error Propagation that a component introduces when it receives its input from another component. We were interested in this since such fault masking makes it more difficult to find faults in the *previous* component when testing. Second, we defined our notion of Squeeziness in this framework. Finally, we carried out experiments in order to evaluate our measure.

Results: Our experiments showed a strong correlation between the likelihood of Failed Error Propagation and Squeeziness.

Conclusion: We can conclude that our new notion of Squeeziness can be used as a measure that estimates the probability of Failed Error Propagation being introduced by a component. As a result, it has the potential to be used as a measure of testability, allowing testers to assess how easy it is to test either the whole system or a single component. We considered a simple model (Finite State Machines) but the notions and results can be extended/adapted to deal with more complex state-based models, in particular, those containing data.

1. Introduction

Software testing [3,35] is the main validation technique used to increase the reliability of complex software systems. Software testing has traditionally been considered to be an *informal* technique [18]. However, it is now known that testing activities can have a formal basis. Formal testing is an active research area [7,10,25] and the existence of several tools that support formal testing has led to the recognition that the combination of formal methods and testing facilitates test automation [42].

Failed Error Propagation (FEP) is a situation in which a faulty statement in the *System Under Test (SUT)* is executed during testing, the fault *corrupts* the internal state of the SUT, but the expected output is observed. Naturally, in order for a statement to be a fault there must be at least one input under which FEP does not occur. FEP is a form of fault masking and can reduce the effectiveness of testing: we might fail to find a fault despite executing the faulty statement in testing. Empirical studies have shown that many systems suffer from

FEP [4,33]. For example, Masri et al. [33] found that in 13% of the programs that they examined, a total of 60% or more of the tests suffered from FEP.

Recent work introduced the notion of Squeeziness [4,13] to capture FEP, with Squeeziness being a measure of the information (entropy) lost by a channel (the SUT) that takes input and returns output. The essential idea is that if the SUT maps two or more inputs to the same output then this channel (the SUT) can lead to a loss of information: if we know the program output then we may not know the program input that caused this (this is the loss of information). The motivation for looking at Squeeziness was that FEP can be caused by two program states, a *correct* program state and a *faulty* program state, being mapped to the same output, which is exactly this type of loss of information. In experiments, there was a rank correlation of close to 0.95 between measures of Squeeziness and the likelihood of FEP [4]. In addition, it has been found that the likelihood of FEP more strongly correlates with Squeeziness than with the Domain to Range Ratio [13].

* Corresponding author.

E-mail addresses: aibias@ucm.es (A. Ibias), r.hierons@sheffield.ac.uk (R.M. Hierons), manuelnu@ucm.es, mn@sip.ucm.es (M. Núñez).



Fig. 1. Representation of our testing scenario.

The goal of this paper is to adapt the notion of Squeeziness to a black box testing scenario in which a software system is composed of components and we have models of these components. We consider the situation in which we have a component C with model M and this component receives a sequence of inputs from another component C_p .¹ The sequence received by C is an input sequence for C and an output sequence for C_p and might result, for example, from communications through an internal network, a sequence of method/function calls, or shared storage/memory. We assume that these values (sent by C_p to C) are not directly observed by the tester. Further, C produces a sequence of outputs that are either observed during testing or are received by another component. A graphical representation of this type of systems can be found in Fig. 1. It is entirely possible that C_p produces an unexpected sequence but component C maps the expected and unexpected sequences to the same output sequence. If this occurs, C introduces a form of FEP that makes it more difficult to find faults in C_p .

In this paper we concentrate on a particular type of model, the Finite State Machine (FSM). An FSM has a finite set of states and transitions between the states, with each transition having a label: an input/output pair. The behaviour represented by an FSM is the set of input/output sequences that label paths from the initial state of the FSM. One of the main reasons for our interest in the FSM formalism is that it has been widely used as the basis for model-based testing (MBT). This line of work (MBT from FSM specifications) started in the 1950s with Moore’s seminal paper [34], with Hennie [23] later (in the 1960s) introducing the first FSM based test generation algorithm. Many FSM based test generation algorithms have since been devised (see, for example [12,30,31,39]). The initial work was largely in the context of testing hardware, since processors are typically specified as FSMs. Since the 1980s FSMs have also been used in the testing of communications protocols. More generally, FSMs are used as the basis for testing a wide variety of systems including embedded systems [9,36] and parts of operating systems [19,20]. Although FSMs do not directly model data, an FSM is typically extracted from a model (in a richer language) by either applying an abstraction or expanding out the data (possibly after applying an abstraction).

In this paper we assume that we have an FSMspecification of the component C being analysed. In this setting, component C can introduce FEP, and so potentially make testing more difficult, if there is a case where a component C_p should send sequence α to C , instead C_p sends $\alpha' \neq \alpha$ to C , and C produces the same output sequence β in response to α and α' . Since an FSM receives a sequence of inputs and produces a sequence of outputs, in our setting, α and α' will be potential inputs of C and β will be a potential output of C . Naturally, since α and α' are sent to C , and as we already mentioned, they are not directly observed by the tester. This situation is illustrated in Fig. 2. Assume that we want to implement the component C_p given in the upper part of Fig. 2 and that this component will be paired with component C . In this setting, it will be difficult to unmask a faulty implementation of C_p , such as the one shown in the lower part of Fig. 2, because C returns the same response, the sequence z_1z_1 , to the sequences y_1y_1 (produced by a correct implementation of C_p receiving x_1x_1) and y_2y_2 (produced by a faulty implementation of C_p also receiving x_1x_1). Note, as we already said, that a tester will not be able to observe whether the sequence provided to C is y_1y_1 or y_2y_2 .

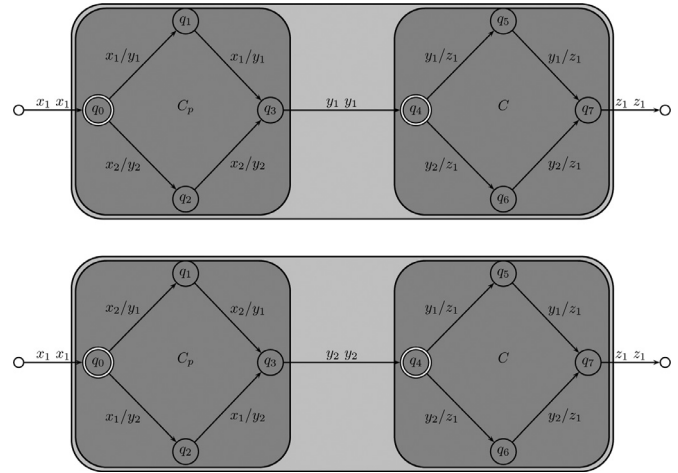


Fig. 2. A case of fault masking.

Unfortunately, we cannot simply reuse the previous approach [13] and results; we are considering a different scenario and also a different source of FEP. The following are the key differences.

1. We are interested in a *different type of FEP*. Previous work looked at the FEP within a program: the potential for a program to mask faults within itself. In contrast, we are interested in the potential for one program (component) C to mask faults in another component C_p .
2. We are interested in *programs that are state-based*. In contrast, previous work looked at programs that retain no state information (after processing an input). As a result, the input domain of an SUT is the set of all possible input sequences. This complicates the underlying theory since:
 - (a) The input domain is infinite (previous work assumed finite input domains [4]).
 - (b) We cannot consider arbitrary probability distributions over the set of input sequences.

Regarding the second point, we need to carefully consider how we can assign probabilities to input sequences since, for example, it may not make sense to have input sequences σ_1 and σ_2 such that σ_1 is a prefix of σ_2 and σ_2 is given a higher probability than σ_1 .

3. Previous work considered the source code of a program and we instead base the analysis on *models*. This brings a number of benefits, including the potential to apply the analysis at an earlier stage.

There were several reasons for us reconsidering the previous decision to base the analysis on the source code. A first practical concern is that approaches that analyse the source code are less likely to scale to situations in which there are multiple components. There is also the issue that for state-based systems there is a need to reason about the change in state caused when we execute the SUT with an input: state-based models make this explicit. Moreover, the source code of a component might not be available (e.g. if the development of a component has been outsourced). Finally, analysis based on models might be applied at an earlier stage of the development process. Note that detailed state-based models are used in a number of important application domains such as automotive and avionic systems; here the models are typically sufficiently detailed to be executable and also for code to be automatically generated from them² As a result, we chose to consider the case where we have *models* of the components and wish to analyse these models. Note that, as explained above, this means that our scenario (and source of FEP) differs from the previous work, as does the entity being analysed.

¹ C might actually receive input from multiple components; allowing this will not affect the underlying approach but complicates the exposition.

² There are a number of widely used tools such as STATEMATE and STATEFLOW that support this.

Although the FSM formalism is relatively simple, we establish the basis of a framework to test in more complex black-box contexts because the basis of testing is similar: we apply a sequence of inputs and decide whether the observed sequence of outputs is consistent with the specification of the system [20]. Further, an FSM might represent the semantics of a model written in a more expressive language. In addition to extending the notion of Squeeziness to a black box scenario, we evaluated this through two types of experiments. The first approach used was to model the component C in terms of the sizes of the inverse images of the possible output sequences; this was used to compare the probability of FEP with our proposed metrics and also the Domain To Range Ratio of C . A weakness of this approach, however, is that we cannot guarantee that the simulations correspond to potential FSM models. As a result, we also carried out experiments using randomly generated FSMs.

The overall results were encouraging, with there being a high correlation between our proposed measures and the probability of FEP. As a result, the proposed measures could act as testability measures for state-based testing and have the potential to help direct testing. There are two practical reasons for the interest in measures associated with FEP. First, there may be potential to generate test cases that achieve a given test purpose, such as testing a component C_p , and that have a low probability of FEP. Second, such measures might be used to estimate testability; we might expect it to be particularly difficult for testing to find a fault in a component C_p that sends its output to another component C that is likely to introduce FEP. Measures of testability might be used to direct additional testing towards difficult to test areas of a system.

As we have explained, there are several differences between the original scenario [13] and ours and these include the type of FEP considered and the entity being analysed. These differences introduced a number of technical challenges. First, we had to reshape the definition of Squeeziness because inputs and outputs have a different treatment in each scenario. In the previously considered white-box case, a program receives an input (a tuple of values) and returns an output (again, a tuple of values). Inputs and outputs were drawn from finite sets allowing, for example, the use of uniform distributions. In the scenario that we consider in this paper, an *input* is a sequence of input actions while an output is also a sequence, in this case of output actions. This leads to two issues, the first of which is that the ‘input set’ is infinite (it is the set of all input sequences), as is the ‘output set’. The second issue is that, even if we bound sequences to make the sets finite, we cannot define a uniform distribution over the sets of inputs and outputs because, for example, a prefix of a sequence should have a higher probability than the whole sequence.

There is a significant body of work on FEP and fault masking for white-box testing [6,33,44,46] and black-box testing [21,37,38,45]. As mentioned, previous work has also defined Squeeziness in a white-box scenario [4,13]. However, we are not aware of any work that uses an Information Theory foundation for addressing FEP in a black-box context. Naturally, there is work that looks at testing systems that are composed of components for which we have FSM models [5,15,41] but this previous work has considered rather different concerns. There has also been previous work that uses information theoretic measures in testing. For example, Information Theory has been used to devise new measures of *test diversity* [16,17], with the potential to either direct test generation towards diverse test suites or facilitate the development of new test criteria. Another line of work, though not one that uses Information Theory, aims to find diverse tests where diversity refers to the test outputs [2]. It is interesting to note that recent work found that white-box and black-box notions of diversity were both effective when searching for a good order of the test cases in a given test suite (the test prioritisation problem) [22].

The rest of the paper is organised as follows. In Section 2 we introduce concepts and terminology used in the rest of the paper. Section 3 develops our novel information theoretic measures for FSMs. Section 4 then describes the empirical evaluation carried out and the re-

sults of the different sets of experiments. Finally, in Section 5 we present our conclusions and some lines of future work.

2. Preliminaries

In this section we present the main definitions and concepts, regarding Finite State Machines (FSMs), that we use throughout this paper. The material presented in this section is based on classical work on testing from FSMs [32]. Most of the concepts are based on the original sources while some notation is adapted to facilitate the formulation of subsequent definitions.

Given a set A , we let A^* denote the set of finite sequences of elements of A ; $\epsilon \in A^*$ denotes the empty sequence. We let A^+ denote the set of non-empty sequences of elements of A . A^k denotes the set of sequences with length $k \geq 1$. We let $|A|$ denote the cardinal of set A . Given a sequence $\sigma \in A^*$, we have that $|\sigma|$ denotes its length. Given a sequence $\sigma \in A^*$ and $a \in A$, we have that σa denotes the sequence σ followed by a and $a\sigma$ denotes the sequence σ preceded by a .

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

A *Finite State Machine* is a (finite) labelled transition system in which transitions are labelled by an input/output pair. We use this formalism to define processes.

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

We say that M is *deterministic* if for all $q \in Q$ and $i \in I$ there exists at most one pair $(q', o) \in Q \times O$ such that $(q, i/o, q') \in T$. In this paper we consider deterministic FSMs.

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

We let $\text{FSM}(I, O)$ denote the set of FSMs with input set I and output set O .

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

As usual, we assume that the System Under Test (SUT) is input-enabled: the SUT should be able to react, somehow, to any external stimulus. In particular, if the tester applies an input action at a certain stage, then the system should be able to provide a response (that is, an output action). Actually, if an input cannot be applied in a state of the SUT, then we can assume that there is a response to the input that reports that this input is blocked and so an FSM that is not input-enabled can be converted into one that is. In addition, it has been shown that the problem of testing from an FSM that is not input-enabled can be mapped to the problem of testing from an input-enabled FSM [24,40]. As a result, the assumption that FSMs are input-enabled is not a significant restriction. However, we do not force specifications to be input-enabled. In particular, all the definitions and results concerning Squeeziness will not assume input-enableness. As stated in the previous definition, we consider the case where both specifications and SUTs are deterministic. This is similar to the previously explored white-box scenario that assumed that programs are deterministic.

Our main goal while testing is to decide whether the behaviour of an SUT conforms to the specification of the system that we would like

to build. In order to detect differences between specifications and SUTs, we need to compare the behaviours of specifications and SUTs and the main notion to define such behaviours is given by the concept of *trace*.

Definition 2. Let $M = (Q, q_{in}, I, O, T)$ be an FSM. We use the following notation.

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

$$\left\{ o_1 \dots o_k \in O^* \mid q \xrightarrow{(i_1, o_1) \dots (i_k, o_k)} \right\}$$

Note that if M is deterministic then this set is either empty or a singleton. In the last case we will sometimes write $\text{out}_M(q, \alpha) = o_1, \dots, o_k$.

3. Let $q \in Q$ be a state. We define $\text{dom}_M(q)$ as the set

$$\{\alpha \in I^* \mid \text{out}_M(q, \alpha) \neq \emptyset\}$$

If $q = q_{in}$ then we simply write dom_M . Similarly, we define $\text{image}_M(q)$ as the set

$$\left\{ o_1 \dots o_k \in O^* \mid \exists i_1 \dots i_k \in I^* : q \xrightarrow{(i_1, o_1) \dots (i_k, o_k)} \right\}$$

If $q = q_{in}$ then we simply write image_M . We denote by $\text{dom}_{M,k}$ the set $\text{dom}_M \cap I^k$. Similarly, We denote by $\text{image}_{M,k}$ the set $\text{image}_M \cap O^k$.

Note that if M is input-enabled then for all $k > 0$ we have that $\text{dom}_{M,k} = I^k$ and, therefore, for all $\alpha \in I^k$ we have that $\text{out}_M(q, \alpha) \neq \emptyset$.

3. Squeeziness for FSMs

In this section we show how the notion of Squeeziness can be adapted to the situation in which we would like to reason about the FEP introduced by a component C that has FSM specification M . As previously discussed, such FEP affects the testing of previous components (components that send output to C) since it might lead to a faulty sequence from a previous component C_p being mapped to the expected output sequence by C .

An FSM M can be seen as a function transforming sequences of input actions belonging to dom_M into sequences of output actions belonging to image_M . Therefore, we could say that M receives an *input* (an element of I^*) and returns an *output* (an element of O^* , with the same length as the input). We define *projections* of this function: for a natural number k , we restrict the function to the set of sequences of input actions that are of length k . In particular, these projections will allow us to consider finite sets of inputs (all the sequences of inputs of a certain length). We also introduce the notion of *collision*: two inputs collide if they produce the same output.

Definition 3. Let $M = (Q, q_{in}, I, O, T)$ be an FSM. We define $f_M : \text{dom}_M \rightarrow \text{image}_M$ as the function such that for all $\alpha \in \text{dom}_M$ we have $f_M(\alpha) = \beta$ for β such that $\text{out}_M(q_{in}, \alpha) = \{\beta\}$.

Let $k > 0$. We define $f_{M,k}$ to be the function $f_M \cap (I^k \times O^k)$, where we use the function f_M to denote the associated set of pairs. Let $\beta \in \text{image}_M$. We define $f_M^{-1}(\beta)$ to be the set $\{\alpha \in I^* \mid f_M(\alpha) = \beta\}$.

Let $\alpha_1, \alpha_2 \in I^*$. We say that α_1 and α_2 collide for M if $\alpha_1 \neq \alpha_2$ and $f_M(\alpha_1) = f_M(\alpha_2)$.

Note that if two sequences of input actions collide then they must have the same length (otherwise, the returned sequences of output actions would have different length and, therefore, cannot be equal). Next we introduce some notation for random variables and recall the concept of *entropy* [43] associated with a random variable and Squeeziness [13] of a function. The concept of entropy is a “measure of the average uncertainty in the random variable. It is the number of bits on average required to describe the random variable” [43]. In other words, entropy is a measure of the amount of information of a given set with a random variable ranging over it. The concept of Squeeziness then is defined as the amount of information lost after the application of a given function, that is, the difference between the amount of information (entropy) of the domain of the function and the amount of information (entropy) of the range of the function. In a broader sense, we can consider it to measure the difference between the amount of information that we have before applying the function and the amount of information that remains after applying the function. We are interested in total functions since we consider input-enabled FSMs³

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

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

Let $f : A \rightarrow B$ be a total function and consider two random variables ξ_A and ξ_B ranging, respectively, over A and B . The *Squeeziness* of f , denoted by $\text{Sq}(f)$, is defined as the loss of information after applying f to A , that is, $\mathcal{H}(\xi_A) - \mathcal{H}(\xi_B)$.

As we said, Squeeziness represents the amount of information lost by a given function. Since we have shown that FSMs can be seen as functions from a set of sequences of input actions to a set of sequences of output actions, we can adapt Squeeziness to deal with FSMs. First, we need to define how inputs are chosen and outputs are returned. We consider a probabilistic view where a random variable associated with each set of relevant inputs/outputs is taken into account. We studied two possible alternatives:

- We associate a random variable with the whole set of inputs/outputs (that is, a random variable induces a probability distribution over I^* and O^* , respectively).
- We associate a random variable with the set of inputs/outputs of a certain length (that is, there are different random variables associated with $I^1, I^2, \dots, O^1, O^2, \dots$).

In this paper we consider the second approach because it gives us an incremental procedure to compute a sequence of consecutive values of Squeeziness so that we can analyse how the series is *evolving*. Actually, the input sequence length used will depend on the amount of testing to be carried out since this will determine the lengths of the input sequences that a component is likely to receive. Note also that there is potential to use Squeeziness values, for different input sequence lengths, to inform the choice of test cases. In other words, we use these values with the aim of using test cases that minimise the likelihood of FEP occurring, that is, this approach provides a way to know, for a given length, if the probability of having FEP, once we have tested all the possible inputs with the given length, will be greater than 0 or not. Despite concentrating on the second approach, we believe that the first approach is also interesting. We consider the development of the first approach, and a comparison with the second approach, to be an interesting line of future work.

We have that $\text{dom}_{M,k}$ represents the possible inputs of length equal to k that M can perform (therefore, other elements of I^k have probability equal to zero) and $\text{image}_{M,k}$ represents the possible outputs

³ Recall that a partial FSM can be mapped to an input-enabled FSM.

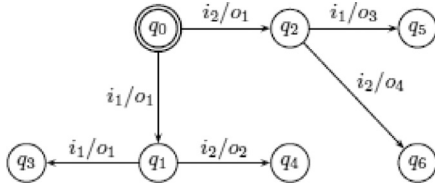
of length equal to k that M can produce after receiving an element of $\text{dom}_{M,k}$. Therefore, defining the random variables that range over each set as $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$, we have that the entropy will be the amount of information of each set, and the difference of entropy (that is, $H(\xi_{\text{dom}_{M,k}}) - H(\xi_{\text{image}_{M,k}})$) represents the amount of information destroyed by M . This is the notion of Squeeziness that we use in this paper.

Definition 5. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider two random variables $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$ ranging, respectively, over the domain and image of $f_{M,k}$. The Squeeziness of M at length k is defined as

$$\text{Sq}_k(M) = H(\xi_{\text{dom}_{M,k}}) - H(\xi_{\text{image}_{M,k}})$$

Squeeziness for FSMs is an interesting notion that has some unexpected properties. For example, it is not monotonic with respect to k . That is, there exist FSMs where using longer sequences can solve a loss of information produced by shorter sequences.

Example 1. Consider M , depicted below, where q_0 is the initial state.



We have that the value of Squeeziness for $k = 1$, assuming a uniform distribution of probabilities, is computed by the following expression

$$2 \cdot \left(-\frac{1}{2} \cdot \log_2\left(\frac{1}{2}\right)\right) - (-1 \cdot \log_2(1)) = \log_2(2) = 1$$

because we have $|\text{dom}_{M,1}| = 2$ and each input of $\text{dom}_{M,k}$ has probability $\frac{1}{2}$ while $|\text{image}_{M,1}| = 1$ and this output has probability 1. Meanwhile, for $k = 2$ we have

$$4 \cdot \left(-\frac{1}{4} \cdot \log_2\left(\frac{1}{4}\right)\right) - 4 \cdot \left(-\frac{1}{4} \cdot \log_2\left(\frac{1}{4}\right)\right) = 0$$

because we have $|\text{dom}_{M,2}| = 4$ and $|\text{image}_{M,2}| = 4$ and each input or output has probability $\frac{1}{4}$ due to the uniform distribution assumption.

Note that obtaining a value of Squeeziness equal to zero for a certain value of k does not imply that Squeeziness will be equal to zero for greater values of k . For example, if we add to q_3, q_4, q_5 and q_6 two outgoing transitions labelled, respectively, by i_1/o_1 and i_2/o_1 and reaching a new state q_7 , then we obtain a value of Squeeziness greater than 0 for $k = 3$.

An important remark concerning random variables associated with inputs and outputs is that given an FSM $M, k > 0$ and a random variable $\xi_{\text{dom}_{M,k}}$ we have that the probability distribution of the random variable $\xi_{\text{image}_{M,k}}$ is completely determined. This is because for each element $\beta \in \text{image}_{M,k}$ we have that

$$\sigma_{\xi_{\text{image}_{M,k}}}(\beta) = \sum_{\alpha \in f_M^{-1}(\beta)} \sigma_{\xi_{\text{dom}_{M,k}}}(\alpha)$$

The following result is immediate from the definition of entropy and the previous explanation concerning how the random variable associated with outputs is determined by the one corresponding to inputs.

Lemma 1. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. If $f_{M,k}$ is bijective then $\text{Sq}_k(M) = 0$.

Next, we present an alternative formulation of Squeeziness. The proof of the following result, given in the appendix, follows from the partition property of entropy [14] and the definition of $\sigma_{\xi_{\text{image}_{M,k}}}$ in terms of $\sigma_{\xi_{\text{dom}_{M,k}}}$. First, we give an auxiliary result concerning conditional distributions of random variables (the proof is also in the appendix). In the following, $\xi_1 | \xi_2$ denotes the conditional random variable ξ_1 given ξ_2 .

Lemma 2. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider two random variables $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$ ranging, respectively, over the domain and image of $f_{M,k}$. We have that $H(\xi_{\text{image}_{M,k}} | \xi_{\text{dom}_{M,k}}) = 0$.

Proposition 1. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider two random variables $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$ ranging, respectively, over the domain and image of $f_{M,k}$. We have that

$$H(\xi_{\text{dom}_{M,k}}) = H(\xi_{\text{image}_{M,k}}) - \mathcal{P}(M, \xi_{\text{image}_{M,k}})$$

where the term $\mathcal{P}(M, \xi_{\text{image}_{M,k}})$ is equal to

$$\sum_{\beta \in \text{image}_{M,k}} \sigma_{\xi_{\text{image}_{M,k}}}(\beta) \left(\sum_{\alpha \in f_M^{-1}(\beta)} \sigma_{\xi_{f_M^{-1}(\beta)}}(\alpha) \cdot \log_2(\sigma_{\xi_{f_M^{-1}(\beta)}}(\alpha)) \right)$$

A trivial corollary of the previous result provides an alternative definition of Squeeziness where the value is computed in terms of the inverse images partition of the input space taking into account, as previously explained, that we have

$$\sigma_{\xi_{\text{image}_{M,k}}}(\beta) = \sum_{\alpha \in f_M^{-1}(\beta)} \sigma_{\xi_{\text{dom}_{M,k}}}(\alpha)$$

Therefore, we only use the probability distribution on inputs given by $\xi_{\text{dom}_{M,k}}$.

Corollary 1. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider a random variable $\xi_{\text{dom}_{M,k}}$ ranging over the domain of $f_{M,k}$. We have that

$$\text{Sq}_k(M) = - \sum_{\beta \in \text{image}_{M,k}} \left(\sum_{\alpha \in f_M^{-1}(\beta)} \sigma_{\xi_{\text{dom}_{M,k}}}(\alpha) \right) \cdot \mathcal{R}_M(\beta)$$

where the term $\mathcal{R}_M(\beta)$ is equal to

$$\left(\sum_{\alpha \in f_M^{-1}(\beta)} \frac{\sigma_{\xi_{\text{dom}_{M,k}}}(\alpha)}{\sigma_{\xi_{\text{dom}_{M,k}}}(f_M^{-1}(\beta))} \cdot \log_2 \left(\frac{\sigma_{\xi_{\text{dom}_{M,k}}}(\alpha)}{\sigma_{\xi_{\text{dom}_{M,k}}}(f_M^{-1}(\beta))} \right) \right)$$

The above notion of Squeeziness is parameterised by the distribution over inputs to the function (and so input sequences). If we know the actual distribution then we can use this. If we do not know the distribution then there is a need to choose one and we now discuss two approaches to this.

3.1. Maximum entropy principle

In general, it is not possible to know the probability distribution that ranges over the inputs. Therefore, if we want to have an estimation of the different values of Squeeziness for a given FSM, then we need to make an assumption about this distribution. There are different possibilities. For example, we can assume *maximum entropy*, that is, we choose a probability distribution that maximises the entropy. If there are no further restrictions, then maximum entropy is obtained with a uniform distribution [14]. In this case, the weight of a single element of $\sigma_{\xi_{\text{dom}_{M,k}}}$ is $\frac{1}{|\text{dom}_{M,k}|}$. Thus, the weight of the inverse image of an output $\beta \in \text{image}_{M,k}$ is equal to $\frac{|f_M^{-1}(\beta)|}{|\text{dom}_{M,k}|}$. Finally, Squeeziness under the assumption of having

a uniform distribution over inputs is equal to

$$\begin{aligned} \text{Sq}_k(M) &= -\sum_{\beta \in \text{image}_{M,k}} \left(\sum_{\alpha \in f_M^{-1}(\beta)} \frac{1}{|\text{dom}_{M,k}|} \right) \\ &\quad \cdot \left(\sum_{\alpha \in f_M^{-1}(\beta)} \frac{1}{|\text{dom}_{M,k}|} \cdot \log_2 \left(\frac{1}{|\text{dom}_{M,k}|} \right) \right) \\ &= -\sum_{\beta \in \text{image}_{M,k}} \frac{|f_M^{-1}(\beta)|}{|\text{dom}_{M,k}|} \\ &\quad \cdot \left(\frac{|f_M^{-1}(\beta)|}{|f_M^{-1}(\beta)|} \cdot \log_2 \left(\frac{1}{|f_M^{-1}(\beta)|} \right) \right) \\ &= -\sum_{\beta \in \text{image}_{M,k}} \frac{|f_M^{-1}(\beta)|}{|\text{dom}_{M,k}|} \cdot \log_2 \left(\frac{1}{|f_M^{-1}(\beta)|} \right) \\ &= \frac{1}{|\text{dom}_{M,k}|} \cdot \sum_{\beta \in \text{image}_{M,k}} |f_M^{-1}(\beta)| \cdot \log_2(|f_M^{-1}(\beta)|) \end{aligned}$$

3.2. Maximum loss of information

Another strategy considers the worst case scenario, that is, we may suppose that the chosen probability distribution induces the maximum loss of information. In other words, we look for a probability distribution that maximises Squeeziness. This distribution is uniformly distributed in the largest inverse image of an element of the outputs and zero elsewhere [13]. Formally, consider $\beta' \in \text{image}_{M,k}$ such that for all $\beta \in \text{image}_{M,k}$ we have that $|f_M^{-1}(\beta')| \geq |f_M^{-1}(\beta)|$. Then,

$$\xi_{\text{dom}_{M,k}}(\alpha) = \begin{cases} \frac{1}{|f_M^{-1}(\beta')|} & \text{if } \alpha \in f_M^{-1}(\beta') \\ 0 & \text{otherwise} \end{cases}$$

Using this probability distribution, Squeeziness is defined as follows:

$$\begin{aligned} \text{Sq}_k(M) &= -\left(\sum_{\alpha \in f_M^{-1}(\beta')} \frac{1}{|f_M^{-1}(\beta')|} \right) \\ &\quad \cdot \left(\sum_{\alpha \in f_M^{-1}(\beta')} \frac{1}{|f_M^{-1}(\beta')|} \cdot \log_2 \left(\frac{1}{|f_M^{-1}(\beta')|} \right) \right) \\ [2em] &= -\frac{|f_M^{-1}(\beta')|}{|f_M^{-1}(\beta')|} \cdot \left(\frac{|f_M^{-1}(\beta')|}{|f_M^{-1}(\beta')|} \cdot \log_2 \left(\frac{1}{|f_M^{-1}(\beta')|} \right) \right) \\ [1.2em] &= -\log_2 \left(\frac{1}{|f_M^{-1}(\beta')|} \right) \\ [1.2em] &= \log_2(|f_M^{-1}(\beta')|) \end{aligned}$$

Let us remark that this probability distribution maximises Squeeziness because for any other possible distribution $\xi_{\text{dom}_{M,k}}$ we have $\text{Sq}_k(M) \leq \log_2(|f_M^{-1}(\beta')|)$. This result is an immediate consequence of the following result [13].

Lemma 3. *Let us consider $2 \cdot n$ non-negative real numbers $a_1, \dots, a_n, p_1, \dots, p_n \in \mathbb{R}^+$. If for all $1 \leq i \leq n$ we have that $a_1 \geq a_i$ and $\sum_i p_i \leq 1$, then $\sum_i (p_i \cdot a_i) \leq a_1$.*

An important consequence of this result is that it allows us to define a normalisation of the value of Squeeziness, if needed, so that we can have a concept of *normalised Squeeziness*. Later we will see that in the experiments we explored this normalised Squeeziness, which was obtained by dividing Squeeziness by the size of the maximum inverse domain of any output.

3.3. Domain to Range Ratio vs. Squeeziness

It is difficult to compare Squeeziness with other notions to compute fault masking because the literature is very scarce. One of the few notions in this line is the Domain to Range Ratio (DRR) [46]. In this section we explore how Squeeziness and DRR relate. In the next section we report on results of experiments that compared DRR with our notion of Squeeziness. First, we give the original definition.

Definition 6. Let $f : I \rightarrow O$ be a total and surjective function. We define the Domain to Range Ratio of f , denoted by $\text{DRR}(f)$, as $\frac{|I|}{|O|}$.

Next, we adapt this notion to our framework. Note that our functions are total and surjective because we restrict ourselves to their domains and ranges.

Definition 7. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider $f_{M,k} : \text{dom}_{M,k} \rightarrow \text{image}_{M,k}$. We define the Domain to Range Ratio for M and k , denoted by $\text{DRR}(f_{M,k})$, as $\frac{|\text{dom}_{M,k}|}{|\text{image}_{M,k}|}$.

The next result, whose proof is in the appendix, shows that this measure is inconsistent with Squeeziness.

Lemma 4. *There exist FSMs M_1 and M_2 and $k > 0$ such that $\text{DRR}(f_{M_1,k}) = \text{DRR}(f_{M_2,k})$ but $\text{Sq}_k(M_1) \neq \text{Sq}_k(M_2)$.*

There exist FSMs M_1 and M_2 and $k > 0$ such that $\text{DRR}(f_{M_1,k}) < \text{DRR}(f_{M_2,k})$ but $\text{Sq}_k(M_1) > \text{Sq}_k(M_2)$.

4. Empirical evaluation

In this section we outline the different experiments carried out to evaluate the proposed measure. First, we describe the experiments that used simulations, that is, instead of FSMs we consider sequences of input/output actions. Next we explain the experiments that used FSMs. We conclude the section with an evaluation of the threats to validity, and how they were addressed, and a discussion about the obtained results and some of their implications.

4.1. Evaluation via simulations

We outline an evaluation in which we simulated an FSM by randomly generating the sizes of the inverse images of the output sequences. We designed the simulation in this way since it represents the notion of FEP that we consider in this paper: one in which one component C masks a fault in another component C_p by mapping two potential output sequences α and α' for C_p (and so input sequences for C) to the same output sequence β . This scenario corresponds to FEP if, for example, α is an expected (correct) output sequence for C_p and α' is a possible faulty output sequence (for C_p). Observe that this type of FEP occurs if and only if α and α' are both in the inverse image of the same output sequence β . As a result, in order to reason about the probability of FEP it is sufficient to retain only the information about the sizes of the inverse images of output sequences and so we simulate these values (the sizes of the inverse images of output sequences).

In this section we first introduce a collision measure, which is the probability of FEP occurring. Although this collision measure could potentially be used to reason about FEP, as usual for this type of measure, it is computationally expensive to compute it. Therefore, it is important to study alternative measures, based on Information Theory, that are either less computationally expensive or that can be efficiently estimated. Having defined the collision measure, we then use experiments with randomly generated scenarios in order to compare this with our information theoretic measure and the Domain to Range Ratio.

Research Question 1. Is there a correlation between the measures defined in this paper and the probability of a component introducing FEP through masking incorrect output produced by earlier components?

4.1.1. Collisions and FEP

In our context, fault masking (FEP) happens when the expected and faulty input sequences, received from another component, produce the same sequence β of output actions. If given an FSM M and $k > 0$ we have that there exist $\beta \in \text{image}_{M,k}$ such that $\alpha, \alpha' \in f_{M,k}^{-1}(\beta)$, with $\alpha \neq \alpha'$, then there is a collision and this might hide a fault. Next we provide a notion to compute the probability of having a collision.

Definition 8. Let M be an FSM and $k > 0$. Let $\text{image}_{M,k} = \{\beta_1, \dots, \beta_n\}$ and for all $1 \leq i \leq n$ let $I_i = f_{M,k}^{-1}(\beta_i)$ and $m_i = |f_{M,k}^{-1}(\beta_i)|$. We have that $d = \sum_{i=1}^n m_i$ is the size of the input space.

Given a uniform distribution over the inputs, the probability of α and α' both being in the set I_i is equal to $p_i = \frac{m_i \cdot (m_i - 1)}{d \cdot (d - 1)}$. We have that the probability of having a collision in M for sequences of length k , denoted by $\text{PColl}_k(M)$, is given by

$$\text{PColl}_k(M) = \sum_{i=1}^n \frac{m_i \cdot (m_i - 1)}{d \cdot (d - 1)}$$

Observe that there is potential to use this measure, $\text{PColl}_k(M)$, instead of Squeeziness. The problem with using $\text{PColl}_k(M)$ is that it is computationally hard to compute. While this also applies to Squeeziness, it has the advantage of being an information theoretic measure. As a result, there is potential to draw on Information Theory research that has devised techniques that either estimate or bound measures [8,11]. Note that estimates and bounds will suffice as long as they are useful - they do not need to be precise. This is in contrast to some applications of Information Theory, such as security, in which we require guarantees. We therefore expect that much smaller samples should suffice.

Previous work [13] states that PColl can be seen as a probability of collisions when the probability distribution over the inputs is uniform. However, it is worth to mention that the relationship between $\text{PColl}_k(M)$ and $\text{Sq}_k(M)$ is not, in general, monotonic. The proof of the following result is given in the appendix.

Lemma 5. *There exist FSMs M_1 and M_2 and $k > 0$ such that $\text{Sq}_k(M_1) < \text{Sq}_k(M_2)$ but $\text{PColl}_k(M_1) > \text{PColl}_k(M_2)$.*

4.1.2. Experimental results

We now report on simulations that compared PColl , Sq and DRR . The three measures are defined (assuming uniform distributions over the inputs) in terms of the sizes of the subdomains ($f_{M,k}^{-1}(\beta)$). Our methodology to perform simulations followed the approach used in the original work on Squeeziness [13] but used a much wider range of scenarios.

First, we fixed the size of the input space (denoted by d) and a maximum subdomain size (denoted by m). Next, we generated random integers between 1 and m until the values summed to d ; if the sum of the values exceeded d then the last value was suitably reduced. Once we had these partitions, we computed the three measures. This way, d represents the number of different inputs of a fixed length k that the simulated FSM has and each partition (each random number) represents one output, whose value is the number of inputs that are in the inverse image of this output (i.e. the number of inputs that generate this output). This process was repeated 200 times for each pair (d, m) and we computed the Pearson correlation coefficient between these 200 values of PColl and the other two measures. We used 120 pairs with d ranging between 10^4 and $2 \cdot 10^9$ and m ranging between 10^2 and 10^4 . We also computed the Spearman Rank correlation coefficient, but the results were almost identical, so we will not discuss these correlation coefficients. For each pair (d, m) we performed the entire process twice.

The main result is that there is a strong correlation between PColl and Sq , with all of the values being greater than 0.96.

We obtained a not so strong correlation between PColl and DRR , with all correlations being between 0.86 and 0.67. Interestingly, the correlation between PColl and DRR , appears not to change as we increase the size of the input domain. This is in contrast to the previous white-box work, which found that increases in input-domain size led to a reduction

Table 1

Representative results from the simulation.

Input set size	Maximum size	Correlation of Sq	Correlation of DRR
10,000	100	0.968366	0.763623
10,000	100	0.973918	0.783759
10,000	200	0.973016	0.823959
10,000	200	0.967349	0.77492
10,000	10,000	0.967281	0.71496
10,000	10,000	0.966184	0.670497
100,000	500	0.980028	0.836659
100,000	500	0.972878	0.769055
500,000	5000	0.95885	0.743651
500,000	5000	0.969437	0.765643
2,000,000	5000	0.978818	0.810967
2,000,000	5000	0.964455	0.698505
200,000,000	2000	0.974498	0.799512
200,000,000	2000	0.980097	0.843219
1,000,000,000	200	0.978771	0.859822
1,000,000,000	200	0.968844	0.759952
2,000,000,000	5000	0.970575	0.807333
2,000,000,000	5000	0.965495	0.781112
2,000,000,000	10,000	0.969172	0.79843
2,000,000,000	10,000	0.972477	0.783512

in the effectiveness of all measures used [13]. This is promising since it suggests that effectiveness may be more robust in the context considered in this paper and so the measures may be effective in a wider range of scenarios.

A number of the most representative results can be found in Table 1 while the full set of results can be found in the appendix of the paper. Specifically, we have given the cases that obtain the highest Sq correlation, the highest DRR correlation, the lowest Sq correlation and the lowest DRR correlation. Also, we give the cases corresponding to the smallest scenario (that is, input set size of 10,000 and maximum subdomain size of 100) and the largest scenario (that is, input set size of 2,000,000,000 and maximum subdomain size of 10,000). Finally, we have given some cases in which the Sq and/or DRR values are around the mean of the values of each measure (Sq and DRR). It is important to note that when we show a case, we display the result of both runs, although the result of interest need not appear on both runs.

As a side note, we performed an additional experiment but the results were worse than expected. Specifically, we computed the results also for the normalised version of Squeeziness that we mentioned in Section 3.2, which is obtained by dividing Squeeziness by the size of the maximum inverse domain of any output. However, some of the correlations obtained were relatively small (see Table 2). Interestingly, we found very poor correlations for some of the small input sets, while the corresponding correlations for Squeeziness were good. Therefore, we decided to no longer consider this form of *normalised Squeeziness* during the rest of our experiments.

4.2. Empirical evaluation using FSMs

In the previous section we reported on the results of simulations that showed that Squeeziness is related to the probability of FEP. The simulations represented general functions, with finite input domains, by giving the sizes of the inverse images of outputs. However, it is unclear whether these simulations correspond to functions that can be described using FSMs and so in this section we report on the results of experiments that used FSM models. The experiments were driven by one research question that assessed whether the measure can be used as intended when we have FSMs.

Research Question 2. When using FSMs, is there a correlation between the measures defined in this paper and the probability of a component introducing FEP through masking incorrect output produced by earlier components?

Table 2

Representative results from the simulation with normalized Squeeziness.

Input set length	Maximum size	Correlation of Normalized Squeeziness
10,000	100	0.958346
10,000	100	0.961652
10,000	200	0.950883
10,000	200	0.926334
10,000	5000	0.469301
10,000	5000	0.471505
10,000	10,000	0.427412
10,000	10,000	0.470961
20,000	10,000	0.415143
20,000	10,000	0.515837
100,000	500	0.972534
100,000	500	0.96534
500,000	5000	0.926782
500,000	5000	0.949692
2,000,000	5000	0.971917
2,000,000	5000	0.958095
200,000,000	2000	0.974498
200,000,000	2000	0.980097
1,000,000,000	200	0.978771
1,000,000,000	200	0.968844
2,000,000,000	5000	0.970575
2,000,000,000	5000	0.965495
2,000,000,000	10,000	0.969172
2,000,000,000	10,000	0.972477

Next we report on the results of an experiment that assessed this research question. First, we briefly explain how we generated the (FSMs) used in our experiments.

4.2.1. FSM generator

In order to perform our experiments we need to generate FSMs. We developed an FSM generator that randomly generates FSMs given some parameters.⁴ The first issue we solved was to fix the internal representation of FSMs. Since our work is not the first one dealing with FSMs we decided to review the literature and found the OpenFST library [1]. This library is intended to work with Finite State Transducers (as its name indicates). These are a kind of FSMs with an input/output pair in each transition and a weight. Therefore, we simply ignored the weight. This library also provides shell commands that we can use, in particular, to generate the associated binary files and to generate the topological representation of each FSM as an image.

Once we had a proper representation for our FSMs, we developed the tool for generating those FSMs. The main reason for developing this tool was to generate a wide range of different FSMs that have some specific properties. In order to have a general tool that can be used in a range of experiments, we included some basic parameters:

- *#Rep*: the number of FSMs we want to generate.
- *Max_States*: the maximum number of states an FSM can have.
- *Min_States*: the minimum number of states an FSM must have.
- *Max_Transitions*: the maximum number of transitions each state of an FSM can have.
- *Min_Transitions*: the minimum number of transitions each state of an FSM must have.
- *#Inputs*: the number of inputs.
- *#Outputs*: the number of outputs.

⁴ All the tools developed to perform the experiments of this paper are freely available at <https://github.com/Colosu/FSTGenerator>.

After setting these basic parameters, the program can be executed. The execution flow for generating an FSM using *#Rep* is given in Algorithm 1. Note that, by construction, the tool returns connected (all states are reachable from the initial state) and deterministic FSMs. Also note that the algorithm allows the construction of FSMs that have loops.

Algorithm 1: FSM generation algorithm.

Result: *#Rep* FSMs.

machine = 0;

while *machine* < *#Rep* **do**

 Create a folder to save the FSM files;

 Set a random number *S* of states between

Min_States and *Max_States* for the FSM;

 Choose the state 0 as initial state;

for each state $0 \leq i < S - 1$ **of the machine do**

 Set a random number *T* of transitions

 between *Min_Transitions* and

Max_Transitions for the state;

for each transition $0 \leq j < T$ **of the state do**

if $j == 0$ **then**

 Set the state $i + 1$ as the end of the transition;

else

 Set a random state as the end of the transition;

end

 Set a random input label for the

 transition not previously used for another

 transition of the state (so FSMs are

 deterministic);

 Set a random output label for the

 transition;

 Save this transition to the FSM file;

end

end

 Create the binary file that the OpenFST library

 uses to interpret FSMs using the FSM file we

 created;

 Create a pdf image with the FSM topology;

machine + +;

end

In order to create input-enabled FSMs with our tool, as used in our experiment, we simply set *Min_Transitions* = *Max_Transitions* = *#Inputs*.

4.2.2. Experimental results

This section describes the results of experiments that addressed the research question. Similar to Section 4.1, we compared our measure with the probability of collision, but this time for the specific FSMs being considered. Recall that the probability of collision is given by the following expression:

$$PColl_k(M) = \sum_{j=1}^n \frac{m_j \cdot (m_j - 1)}{d \cdot (d - 1)}$$

where m_j is the cardinality of the inverse image of the j -th output (i.e. the number of inputs that lead to this output) and d is the cardinality of the inputs (i.e. the total number of inputs). Squeeziness was designed to compare models with the same input domains. In order to facilitate this task, we used input-enabled FSMs but the results are essentially the same if we use non input-enabled FSMs (as long as we consider the same number of input sequences in all the FSMs). We generated 500 machines with 25 states and 5 outgoing transitions from each state. We considered sets of 5 inputs and 5 outputs.

Having generated the FSMs, we computed Squeeziness and PColl for each FSM. The next step was to randomly partition the set of FSMs

Table 3
Results from the experiment with 500 FSMs with 25 states.

Run Number	Pearson Sq	Pearson DRR	Spearman Sq	Spearman DRR
1	0.878449	0.789925	0.915152	0.835599
2	0.769163	0.577342	0.709091	0.527583
3	0.926841	0.836864	0.939394	0.69347
4	0.919335	0.843178	0.890909	0.811444
5	0.888474	0.85478	0.733333	0.71462
6	0.779444	0.56354	0.842424	0.67769
7	0.899583	0.87125	0.927273	0.885083
8	0.895344	0.792316	0.854545	0.877186
9	0.683355	0.46901	0.842424	0.610832
10	0.906801	0.909021	0.830303	0.887425
11	0.56845	0.483205	0.721212	0.592422
12	0.838746	0.834886	0.672727	0.544839
13	0.630317	0.531773	0.793939	0.551174
14	0.410659	0.504509	0.272727	0.355335
15	0.640715	0.56302	0.151515	8.36862e-18
16	0.73553	0.601444	0.757576	0.549532
17	0.272227	0.160886	0.333333	0.113904
18	0.679269	0.577716	0.50303	0.449199
19	0.505532	0.276551	0.684848	0.334363
20	0.866044	0.856532	0.854545	0.877186
21	0.899159	0.832556	0.890909	0.830399
22	0.273041	0.0300172	0.224242	0.012975
23	0.635755	0.635614	0.830303	0.740844
24	0.907813	0.853587	0.854545	0.889898
25	0.804562	0.694005	0.660606	0.563845
26	0.438958	0.299874	0.6	0.375029
27	0.75262	0.577602	0.563636	0.394771
28	0.900993	0.911372	0.939394	0.885657
29	0.909105	0.863749	0.842424	0.805143
30	0.88053	0.818995	0.527273	0.644304
31	0.864043	0.782816	0.709091	0.664867
32	0.782251	0.763869	0.69697	0.846658
33	0.891232	0.815343	0.709091	0.660696
34	0.707623	0.522515	0.709091	0.486655
35	0.608514	0.549941	0.648485	0.589186
36	0.89894	0.824825	0.963636	0.806406
37	0.680069	0.520374	0.648485	0.555997
38	0.718919	0.528805	0.866667	0.761549
39	0.803944	0.857559	0.575758	0.71462
40	0.749198	0.46362	0.818182	0.635946
41	0.841066	0.416991	0.830303	0.341882
42	0.871523	0.699391	0.709091	0.661358
43	0.841827	0.606074	0.939394	0.905111
44	0.783823	0.706252	0.684848	0.552679
45	0.156226	-0.0418695	0.0545455	-0.12975
46	0.911163	0.819401	0.806061	0.793018
47	0.883863	0.780593	0.878788	0.774176
48	0.711095	0.516978	0.866667	0.603382
49	0.76034	0.434219	0.806061	0.568535
50	0.710906	0.75102	0.672727	0.742155

into groups of 10 and compute the (Pearson and Spearman) correlations (between Squeeziness and PCol1) for each group. We used multiple groups in order to obtain insights into the consistency of the results. Therefore, we obtained 50 values for each correlation coefficient. Note that the number of input sequences that we have to consider grows exponentially with the input sequence length. As a result of this exponential growth, and memory limits, we computed the measures for input sequences of length 10.

Similar to the simulations, we obtained positive experimental results. The results of this experiment can be found in Table 3. In most cases, the results show a high correlation between Squeeziness and PCol1, with a mean of 0.745468 for Pearson and 0.715152 for Spearman. This fact supports the results from the simulations. It is interesting to see that there were a few relatively small values but it seems likely that these were simply the result of the randomness in the experiments (we also observe some higher correlations, up to 0.96). Also, we can see that in most of the cases the correlation values of Squeeziness are higher than the ones of DRR, as we saw in the simulations. Specifically, the mean for

Table 4
Results from the experiment with 900 FSMs with 25 states.

Run Number	Pearson Sq	Pearson DRR	Spearman Sq	Spearman DRR
1	0.856654	0.759691	0.839822	0.718795
2	0.799091	0.691343	0.803782	0.710352
3	0.83536	0.720232	0.901224	0.809638
4	0.812958	0.809717	0.733037	0.667405
5	0.637571	0.565636	0.474972	0.439789
6	0.628766	0.550766	0.573304	0.468414
7	0.78118	0.662821	0.829143	0.722609
8	0.648335	0.563561	0.689433	0.607159
9	0.651849	0.52149	0.599555	0.452153
10	0.890877	0.84885	0.866518	0.83724
11	0.822498	0.745911	0.78109	0.776184
12	0.756156	0.648846	0.788654	0.674843
13	0.705051	0.628704	0.751724	0.733294
14	0.812525	0.539454	0.85673	0.580215
15	0.750044	0.522172	0.699221	0.548734
16	0.794857	0.677268	0.866963	0.742223
17	0.761287	0.696498	0.739711	0.695613
18	0.711764	0.700859	0.676085	0.626331
19	0.874628	0.85356	0.777976	0.775392
20	0.916858	0.897105	0.874527	0.792335
21	0.871061	0.902461	0.811791	0.814555
22	0.896291	0.867035	0.822469	0.733963
23	0.815214	0.79039	0.599555	0.478928
24	0.731722	0.642186	0.721913	0.593555
25	0.764644	0.677635	0.826029	0.77864
26	0.624603	0.626964	0.618687	0.585576
27	0.731404	0.558051	0.879422	0.838548
28	0.727717	0.659671	0.725918	0.601497
29	0.828107	0.716817	0.823359	0.673092
30	0.544259	0.319727	0.630256	0.430179

DRR is equal to 0.634677 for Pearson and equal to 0.611338 for Spearman. These values are noticeably lower than the ones corresponding to the correlations between Squeeziness and PCol1.

In order to check the flexibility of our results, we decide to repeat the experiment with a different configuration. We generated 900 machines with the same characteristics: 25 states, 5 outgoing transitions from each state and sets of inputs and outputs with 5 elements. We grouped the machines in 30 groups of 30 machines per group and repeated the experiment. Previously we used groups of 10 FSMs and using larger groups of FSMs allows us to check our intuition that there should be greater consistency in the results. The results are shown in Table 4. These results are slightly better than the previous ones, with a higher similarity between Pearson and Spearman correlations. In this case, the means of the correlations between Squeeziness and PCol1 are equal to 0.766111 for Pearson and equal to 0.752762 for Spearman; the ones corresponding to DRR are equal to 0.678847 for Pearson and equal to 0.663575 for Spearman. Therefore, the conclusions of the experiment are similar to the ones obtained in the previous experiment.

4.3. Threats to validity

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

First, we explore threats to internal validity, which consider uncontrolled factors that might be responsible for the obtained results. In our work, the main threat to internal validity is associated with the possible faults in the developed tools, which could lead to misleading results. In order to reduce the impact of this threat we tested our code with carefully constructed examples for which we could manually check the results. In addition, we repeated each experiment that used FSMs in order to check that the results were consistent and there was no randomisation involved.

Second, we consider threats to external validity, which concern conditions that allow us to generalise our findings to other situations. In our work, the main external threat is the different possible represen-

Table 5
Results from the experiment with FSMs between 10 and 25 states.

Run number	Pearson Sq	Spearman Sq
1	0.279452	0.333333
2	0.55035	0.309091
3	0.0716533	-0.151515
4	0.77656	0.890909
5	0.622	0.890909
6	0.66114	0.660606
7	0.655833	0.757576
8	0.317683	0.321212
9	0.87951	0.818182
10	0.798106	0.878788
11	0.834614	0.781818
12	0.732538	0.733333
13	0.27344	0.406061
14	0.583361	0.478788
15	0.580003	0.769697
16	0.892117	0.939394
17	0.166601	-0.0909091
18	0.455388	0.50303
19	0.843653	0.660606
20	0.551302	0.672727
21	0.901526	0.927273
22	0.89004	0.842424
23	0.442546	0.393939
24	0.683401	0.50303
25	0.931557	0.757576
26	0.52448	0.321212
27	0.600394	0.539394
28	0.323964	0.284848
29	0.698155	0.684848
30	0.61123	0.672727
31	0.559464	0.745455
32	0.765045	0.454545
33	0.570081	0.430303
34	0.859868	0.878788
35	0.917171	0.878788
36	0.837555	0.721212
37	0.539043	0.490909
38	0.704665	0.660606
39	0.740488	0.878788
40	0.552796	0.527273
41	0.717961	0.757576
42	0.634331	0.6
43	0.563094	0.672727
44	0.749326	0.587879
45	0.877225	0.866667
46	0.584465	0.587879
47	0.716488	0.10303
48	0.392777	0.563636
49	0.866184	0.890909
50	0.597951	0.672727

Table 6
Results from the experiment with FSMs between 25 and 50 states.

Run number	Pearson Sq	Spearman Sq
1	0.962624	0.951515
2	0.770315	0.624242
3	0.861972	0.769697
4	0.806692	0.660606
5	0.782762	0.745455
6	0.582544	0.842424
7	0.777772	0.781818
8	0.848743	0.793939
9	0.282844	0.309091
10	0.749289	0.660606
11	0.815693	0.587879
12	0.465784	0.527273
13	0.854386	0.866667
14	0.514125	0.478788
15	0.94217	0.90303
16	0.956262	0.90303
17	0.556821	0.539394
18	0.658927	0.478788
19	0.423597	0.478788
20	0.927198	0.866667
21	0.225621	0.515152
22	0.800149	0.769697
23	0.732404	0.769697
24	0.964573	0.90303
25	0.693287	0.563636
26	0.904747	0.612121
27	0.797333	0.684848
28	0.950163	0.842424
29	0.874851	0.793939
30	0.547064	0.709091
31	0.81926	0.866667
32	0.783165	0.878788
33	0.872504	0.866667
34	0.576504	0.50303
35	0.827418	0.915152
36	0.894726	0.781818
37	0.814328	0.406061
38	0.76672	0.890909
39	0.829572	0.648485
40	0.92247	0.951515
41	0.913127	0.806061
42	0.804393	0.781818
43	0.786996	0.866667
44	0.643162	0.612121
45	0.72758	0.721212
46	0.781083	0.757576
47	0.823419	0.915152
48	0.807975	0.818182
49	0.728113	0.793939
50	0.731225	0.769697

tations of a black-box component as an FSM. Such a threat cannot be entirely addressed since the population of such FSMs is unknown and it is not possible to sample from this (unknown) population. In order to reduce the impact of this threat we used both a large number of simulations and of randomly generated FSMs. Note also that the simulations provided significant diversity in terms of experimental subjects, with the role of the FSM-based experiments primarily being to check that the results extend to the class of functions that can be represented by FSMs.

Last, we consider threats to construct validity. This is related to the *reality* of our experiments, that is, whether our experiments reflect real-world situations. In our work, the main construct threat is whether the FSMs used in the experiments correspond to possible system components. In order to reduce the impact of this threat, we restricted our range of FSM samples to connected deterministic machines. In future work we intend to test with real-world cases and/or non-deterministic FSMs.

4.4. Discussion

The two sets of results presented in this section were encouraging. The simulations showed that there is a strong positive correlation between our notion of Squeeziness and a measure of the probability of collisions if we simulate the function computed by a component. As expected, this correlation was higher than the one that we obtained with DRR, with this being consistently seen across the 24,000 simulation experiments.

The simulations addressed the suitability of our measures for a general framework, in which we have a function that represents the (input/output) behaviour of the component of interest. Since we developed the details of the framework for FSM models we also had experiments that explored whether similar results hold for functions that can be represented by FSMs. The results of these experiments were similar to those previously observed, supporting the results that used simulations.

Interestingly, the correlations returned were slightly lower when using FSMs. There are at least three possible explanations for the differ-

Table 7
Results from the experiment with FSMs with 75 states.

Run number	Pearson Sq	Spearman Sq
1	0.872409	0.915152
2	0.744849	0.527273
3	0.910456	0.951515
4	0.943024	0.854545
5	0.828649	0.927273
6	0.837167	0.757576
7	0.797805	0.866667
8	0.89309	0.745455
9	0.46309	0.478788
10	0.94864	0.951515
11	0.951563	0.745455
12	0.973901	0.878788
13	0.973626	0.951515
14	0.991222	0.915152
15	0.862179	0.721212
16	0.92241	0.915152
17	0.796633	0.587879
18	0.991884	0.721212
19	0.649892	0.684848
20	0.897328	0.90303
21	0.783819	0.721212
22	0.653267	0.648485
23	0.865528	0.878788
24	0.819458	0.769697
25	0.829544	0.745455
26	0.800867	0.612121
27	0.764609	0.890909
28	0.896245	0.769697
29	0.738119	0.672727
30	0.893021	0.90303
31	0.91068	0.818182
32	0.937707	0.963636
33	0.834563	0.818182
34	0.750709	0.793939
35	0.564428	0.490909
36	0.937426	0.975758
37	0.936632	0.90303
38	0.89141	0.975758
39	0.745184	0.818182
40	0.835685	0.854545
41	0.847526	0.830303
42	0.961247	0.951515
43	0.890352	0.951515
44	0.951611	0.927273
45	0.944342	0.951515
46	0.870518	0.890909
47	0.913673	0.927273
48	0.545525	0.551515
49	0.741032	0.587879
50	0.896819	0.915152

ences. First, the experiments that used FSMs considered a smaller set of scenarios; it may be that we would observe results similar to those found in the simulations if we ran many more experiments with a wider range of FSMs. Second, the simulations may have used functions that are rather different from those found when using FSMs. If this is the case then the results might be better if we use more general types of models as specifications of components (rather than FSMs). Third, the differences may result from the simulations using larger sample sizes (sample size 200) than the experiments with FSMs (sample size 10). Note that the smaller sample size used in the FSM experiments (for practical reasons) could also explain the greater variability observed in the results.

In order to explore the two first possibilities, we repeated the FSM experiment with three new sets of FSMs. This allowed us to consider more scenarios (exactly, 1,500 additional ones) and to test if increasing or reducing the generality of the functions represented by the FSMs has any effect on the correlations.

The first set of samples included FSMs that had between 10 and 25 states, so we place even stronger limits on the generality of the functions

Table 8
Results from the experiment with sample size of 20 FSMs of 25 states.

Run number	Pearson Sq	Spearman Sq
1	0.827858	0.798496
2	0.895008	0.792481
3	0.58499	0.810526
4	0.707372	0.78797
5	0.837989	0.929323
6	0.885698	0.857143
7	0.76128	0.711278
8	0.84805	0.700752
9	0.767311	0.694737
10	0.880905	0.899248
11	0.722467	0.658647
12	0.834103	0.778947
13	0.880195	0.891729
14	0.874195	0.696241
15	0.57377	0.607519
16	0.736536	0.669173
17	0.733254	0.702256
18	0.846274	0.861654
19	0.765341	0.783459
20	0.784864	0.843609
21	0.678806	0.700752
22	0.914889	0.809023
23	0.813506	0.735338
24	0.742288	0.769925
25	0.668486	0.696241

represented. As expected, we got slightly worse correlations, with only 77% of samples having correlations greater than 0.5 instead of the 93% that we got with the initial experiment. That leads to a mean of 0.6376 for the Pearson correlation and of 0.6092 for the Spearman one. The full results can be found in [Table 5](#).

The second set of samples considered FSMs that had between 25 and 50 states, slightly increasing the generality of the functions represented. The results were reasonably similar to the initial FSM results, in that 91% of samples had correlations greater than 0.5. However, the results were arguably a little better since the lowest value was greater than 0.2 (instead of being negative as in the initial experiment). That leads to a mean of 0.7577 for the Pearson correlation and of 0.7297 for the Spearman one. The full results can be found in [Table 6](#).

In order to increase the confidence on the validity of our results when the number of states increase, we performed an additional experiment where all the FSMs have the same number of states (75). The values show that 98% of the results have correlations greater than 0.5, and the lowest correlation is greater than 0.4. That leads to a mean of 0.844027 for the Pearson correlation and of 0.810182 for the Spearman one. The full results are provided in [Table 7](#).

The results suggest that correlations will be better if we increase the generality of the functions represented by the FSMs, that is, as we consider bigger FSMs. This allows us to hypothesise that the results corresponding to FSMs will *tend towards* the results from the simulations. Also, as we considered many more scenarios, we can observe that even after taking into account the bad results of the experiment with FSMs between 10 and 25 states, 88% of the cases showed correlations greater than 0.5.

We also performed a small experiment to test what happens if we increase the sample size, that is, to explore the third possibility. We increased the sample size from 10 to 20 samples, using FSMs with 25 states from the initial (FSM) experiments. As expected, we got better correlations, with all the correlations being greater than 0.5. The full results can be found in [Table 8](#).

These results suggest that the correlations will improve if we use larger sample sizes and, in particular, this will reduce the variability of the results. So, again, we can expect that the results will *tend towards* the results from the simulations. Overall, our experiments with FSMs

Table 9
Results from the experiment with Maximum Loss of Information approach.

Run number	Pearson Sq	Spearman Sq
1	0.710801	0.791365
2	0.749008	0.701602
3	0.624626	0.713539
4	0.670041	0.646195
5	0.642364	0.496048
6	0.452779	0.525309
7	0.463446	0.646934
8	0.525565	0.613348
9	0.703414	0.589675
10	0.878989	0.792566
11	0.830456	0.75203
12	0.651943	0.772191
13	0.776044	0.582555
14	0.714373	0.733645
15	0.865138	0.738564
16	0.648421	0.828883
17	0.684968	0.647903
18	0.695542	0.57537
19	0.684215	0.679355
20	0.531623	0.502392
21	0.485815	0.505174
22	0.707509	0.637446
23	0.613067	0.495327
24	0.534643	0.478087
25	0.703643	0.75153
26	0.35483	0.373817
27	0.78047	0.66548
28	0.617858	0.523026
29	0.517789	0.567646
30	0.571259	0.560303

indicate that the measures perform well with FSMs and not just with simulated functions.

Finally, we did an experiment to show what happens if we do not assume a uniform distribution over the inputs of the FSM. In order to do so, we considered the same setup of our main experiment with 900 FSMs, but this time we used the Maximum Loss of Information approach explained in Section 3.2 for computing Squeeziness.

The results of this experiment showed lower correlations and can be found in Table 9. The results are unsurprising since PCol1 assumes a uniform distribution over the inputs of the FSM, that is, these two measures used different distributions. However, they are still relatively good, with a mean of 0.646355 for Pearson and of 0.629577 for Spearman. Note that the lower correlations suggest that techniques that use Squeeziness may be most effective when we know the true distribution of values.

To conclude, the results suggest that Squeeziness can be used to estimate the probability of the FEP introduced by a component. As a result, there is potential to use it to direct testing in order to avoid components that have a high probability of FEP. It might also be used as a measure of testability, with the tester potentially choosing to use more test cases in situations in which FEP is particularly likely. It would be interesting to explore this further through additional experiments.

5. Conclusions and future work

It is known that failed error propagation (FEP) can have a significant effect on testing. Recent work has shown that an information theoretic measure called Squeeziness strongly correlates with the likelihood of FEP [13]. However, this work only considered the white-box scenario in which the SUT simply receives input and returns output; there is no persistent state. In this paper we adapted the Squeeziness measure to work with situations in which we are interested in fault masking. Specifically, we adapted Squeeziness to the scenario in which we are interested in the FEP that a component C introduces when it receives its input from

another component C_p . We are interested in this since such FEP makes it more difficult to find faults in C_p when testing. The work also considered the black-box scenario, in which we base the computations on models. This has the advantage that the approach is applicable at an earlier stage (for example, as a notion of testability that can help inform test planning) and also that the approach can be used in situations in which the source code is not available (for example, when development has been outsourced).

It was not possible to directly reuse the previous notion of Squeeziness [13] since we considered a different scenario and also a different source of FEP. In addition, we argued that in our scenario it makes sense to base the analysis on models of components rather than the source code: this should aid scalability and also address the issue that we might not have access to the source code of a component. As a result, we addressed a different type of FEP and also used a different source of information (an FSM specification rather than the source code).

Having devised a new notion of Squeeziness, for black-box component-based systems, we carried out experiments in order to evaluate this measure. These experiments focused on the capability of the second component to hide faulty inputs from the first component by giving the expected outputs. In the experiments, we compared our measure with a measure of the probability of this hiding/FEP happening (PCol1). We used two types of experiments: simulations and experiments with FSMs. In both cases, we observed a strong correlation between the likelihood of FEP and our measure (Squeeziness). Interestingly, in the experiments with FSMs we observed a slight improvement when we increased the number of states. This supports our original hypothesis: our new notion of Squeeziness can be used as a measure that estimates the probability of FEP being introduced by a component.

The results in this paper have two potential uses. First, the measure defined might be used as a measure of testability, allowing one to assess how easy it is to test a system or part of a system. This might be used as part of the process of deciding how much testing is required. In addition, there is potential to use Squeeziness to direct testing. For example, we might want to execute a part of the system with a test case where the probability of FEP (introduced by another component) is relatively low.

We have several lines for future work. First, we will explore the previously mentioned potential uses, develop tools, and evaluate these on case studies. We plan to explore approximations, most likely based on sampling, and the trade-off between the cost of sampling (sample size) and the effectiveness of the estimates. We also intend to generalise the framework and measures to introduce data into the models. Finally, we would like to adapt Squeeziness to systems with other features. It is natural to consider how Squeeziness works in systems where decisions are probabilistically quantified and we will take as initial step our previous work on formally testing this kind of systems [28,29]. Similarly, we would like to consider distributed systems and how Squeeziness predicts FEP induced by different distributed components. Again, we will take as initial step our work on the distributed test architecture [26,27].

Conflict of Interest

No conflict of interest

Acknowledgements

We would like to thank the anonymous reviewers for the careful reading of the paper and the many constructive comments, which have helped us to further strengthen the paper.

This work has been supported by the Spanish MINECO-FEDER (grant number DARDOS, TIN2015-65845-C3-1-R); the Region of Madrid (grant

number FORTE-CM, S2018/TCS-4314); and the UK EPSRC (grant number InfoTestSS, EP/P006116/2).

Appendix A. Proofs of the results

Lemma 2 Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider two random variables $\xi_{dom_{M,k}}$ and $\xi_{image_{M,k}}$ ranging, respectively, over the domain and image of $f_{M,k}$. We have that $\mathcal{H}(\xi_{image_{M,k}} | \xi_{dom_{M,k}}) = 0$.

Proof. Consider the entropy of the conditional random variable $\xi_{image_{M,k}} | \xi_{dom_{M,k}}$. We have that $\mathcal{H}(\xi_{image_{M,k}} | \xi_{dom_{M,k}})$ is equal to

$$\sum_{\alpha \in \text{dom}_{M,k}} \sigma_{\xi_{dom_{M,k}}}(\alpha) \cdot \mathcal{H}(\xi_{image_{M,k}} | \xi_{dom_{M,k}} = \alpha)$$

If we unfold the second term of the sum we have that the previous expression is equal to

$$\sum_{\alpha \in \text{dom}_{M,k}} \sigma_{\xi_{dom_{M,k}}}(\alpha) \cdot \left(\sum_{\beta \in \text{image}_{M,k}} \gamma(\beta|\alpha) \cdot \log_2(\gamma(\beta|\alpha)) \right)$$

where $\gamma(\beta|\alpha) = \sigma_{(\xi_{image_{M,k}} | \xi_{dom_{M,k}})}(\beta|\alpha)$. We will prove that all the summands of the previous expression are equal to zero. Taking into account that M is deterministic we have that $\sigma_{(\xi_{image_{M,k}} | \xi_{dom_{M,k}})}$ can be either 0 or 1. Using this fact in the previous expression, we have two cases:

- If $\sigma_{(\xi_{image_{M,k}} | \xi_{dom_{M,k}})}(\beta|\alpha) = 0$ then the result obviously holds.
- Otherwise, that is, $\sigma_{(\xi_{image_{M,k}} | \xi_{dom_{M,k}})}(\beta|\alpha) = 1$, we have that $\log_2(\sigma_{(\xi_{image_{M,k}} | \xi_{dom_{M,k}})}(\beta|\alpha)) = 0$ and, again, the result holds.

We finally conclude that $\mathcal{H}(\xi_{image_{M,k}} | \xi_{dom_{M,k}}) = 0$. \square

Proposition 1 Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider two random variables $\xi_{dom_{M,k}}$ and $\xi_{image_{M,k}}$ ranging, respectively, over the domain and image of $f_{M,k}$. We have that

$$\mathcal{H}(\xi_{dom_{M,k}}) = \mathcal{H}(\xi_{image_{M,k}}) - \mathcal{P}(M, \xi_{image_{M,k}})$$

where the term $\mathcal{P}(M, \xi_{image_{M,k}})$ is equal to

$$\sum_{\beta \in \text{image}_{M,k}} \sigma_{\xi_{image_{M,k}}}(\beta) \cdot \left(\sum_{\alpha \in f_M^{-1}(\beta)} \sigma_{\xi_{f_M^{-1}(\beta)}}(\alpha) \cdot \log_2(\sigma_{\xi_{f_M^{-1}(\beta)}}(\alpha)) \right)$$

Proof. By the definition of conditional entropy [14] we have that $\mathcal{H}(\xi_{dom_{M,k}} | \xi_{image_{M,k}})$ is equal to

$$\sum_{\beta \in \text{image}_{M,k}} \sigma_{\xi_{image_{M,k}}}(\beta) \cdot \mathcal{H}(\xi_{dom_{M,k}} | \xi_{image_{M,k}} = \beta)$$

Next, we apply the notion of conditional probability and take into account that $\xi_{dom_{M,k}}$ restricted to $\xi_{image_{M,k}} = \beta$ is the random variable $\xi_{f_M^{-1}(\beta)}$ ranging over $f_M^{-1}(\beta)$ and whose probabilities are equal to

$$\frac{\sigma_{\xi_{dom_{M,k}}}(\beta)}{\sigma_{\xi_{dom_{M,k}}}(f_M^{-1}(\beta))}$$

Therefore, we we have that

$$\begin{aligned} \mathcal{H}(\xi_{dom_{M,k}} | \xi_{image_{M,k}} = \beta) &= \mathcal{H}(\xi_{f_M^{-1}(\beta)}) \\ &= -\sum_{\alpha \in f_M^{-1}(\beta)} \sigma_{\xi_{f_M^{-1}(\beta)}}(\alpha) \cdot \log_2(\sigma_{\xi_{f_M^{-1}(\beta)}}(\alpha)) \\ &= -\sum_{\alpha \in f_M^{-1}(\beta)} \frac{\sigma_{\xi_{dom_{M,k}}}(\alpha)}{\sigma_{\xi_{dom_{M,k}}}(f_M^{-1}(\beta))} \cdot \log_2 \left(\frac{\sigma_{\xi_{dom_{M,k}}}(\alpha)}{\sigma_{\xi_{dom_{M,k}}}(f_M^{-1}(\beta))} \right) \end{aligned}$$

Therefore, the term $\mathcal{H}(\xi_{dom_{M,k}} | \xi_{image_{M,k}})$ is equal to

$$-\sum_{\beta \in \text{image}_{M,k}} \sigma_{\xi_{image_{M,k}}}(\beta) \cdot \left(\sum_{\alpha \in f_M^{-1}(\beta)} \theta(\alpha) \cdot \log_2(\theta(\alpha)) \right) \quad (\text{A.1})$$

where $\theta(\alpha) = \sigma_{\xi_{f_M^{-1}(\beta)}}(\alpha)$. If we apply the *Chain rule* then we have

$$\mathcal{H}(\xi_{image_{M,k}}, \xi_{dom_{M,k}}) = \mathcal{H}(\xi_{image_{M,k}}) + \mathcal{H}(\xi_{dom_{M,k}} | \xi_{image_{M,k}})$$

where $\mathcal{H}(\xi_{image_{M,k}}, \xi_{dom_{M,k}})$ is the joint probability of the two random variables. Taking into account that, applying again the *Chain rule*, we also have

$$\mathcal{H}(\xi_{image_{M,k}}, \xi_{dom_{M,k}}) = \mathcal{H}(\xi_{dom_{M,k}}) + \mathcal{H}(\xi_{image_{M,k}} | \xi_{dom_{M,k}})$$

Combining the previous equalities we obtain

$$\begin{aligned} \mathcal{H}(\xi_{image_{M,k}}) + \mathcal{H}(\xi_{dom_{M,k}} | \xi_{image_{M,k}}) \\ \parallel \\ \mathcal{H}(\xi_{dom_{M,k}}) + \mathcal{H}(\xi_{image_{M,k}} | \xi_{dom_{M,k}}) \end{aligned}$$

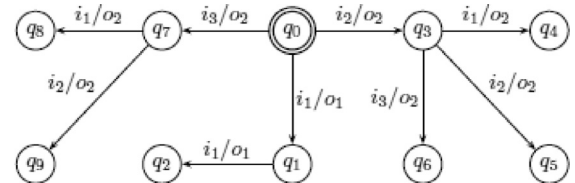
Finally, by Lemma 2, we have $\mathcal{H}(\xi_{image_{M,k}} | \xi_{dom_{M,k}}) = 0$ and taking into account the value of $\mathcal{H}(\xi_{dom_{M,k}} | \xi_{image_{M,k}})$, given in Eq. (A.1), we obtain the desired reformulation of $\mathcal{H}(\xi_{dom_{M,k}})$. \square

Lemma 4 There exist FSMs M_1 and M_2 and $k > 0$ such that $\text{DRR}(f_{M_1,k}) = \text{DRR}(f_{M_2,k})$ but $\text{Sq}_k(M_1) \neq \text{Sq}_k(M_2)$.

There exist FSMs M_1 and M_2 and $k > 0$ such that $\text{DRR}(f_{M_1,k}) < \text{DRR}(f_{M_2,k})$ but $\text{Sq}_k(M_1) > \text{Sq}_k(M_2)$.

Proof. First, let us note that in this proof we assume uniform distributions over inputs (and outputs) of the FSMs. However, the result holds for any probability distribution: we would only need to slightly modify the definition of the given machines.

In order to prove the first part of the result, we define two machines M_1 and M_2 , both with initial state q_0 , fulfilling the conditions. Let M_1 be the following FSM:

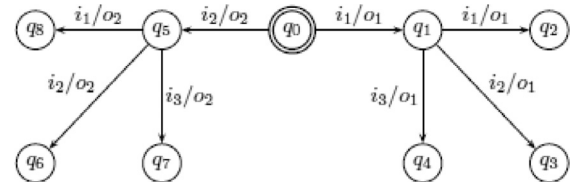


We have that $\text{dom}_{M_1,2}$ is equal to

$$\{(i_1, i_1), (i_2, i_1), (i_2, i_2), (i_2, i_3), (i_3, i_1), (i_3, i_2)\}$$

and $\text{image}_{M_1,2}$ is equal to $\{(o_1, o_1), (o_2, o_2)\}$. On the one hand we have $\text{DRR}(f_{M_1,2}) = 6/2 = 3$ while, on the other hand, we have $\text{Sq}_2(M_1) = \frac{5 \cdot \log_2(5) + 1 \cdot \log_2(1)}{6} \approx 1.9349$.

Now, let M_2 be the following FSM:

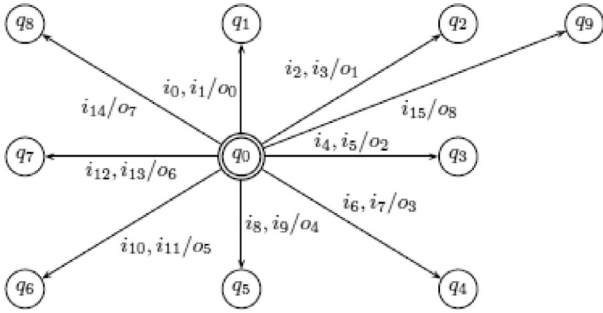


We have that $\text{dom}_{M_2,2}$ is equal to

$$\{(i_1, i_1), (i_1, i_2), (i_1, i_3), (i_2, i_1), (i_2, i_2), (i_2, i_3)\}$$

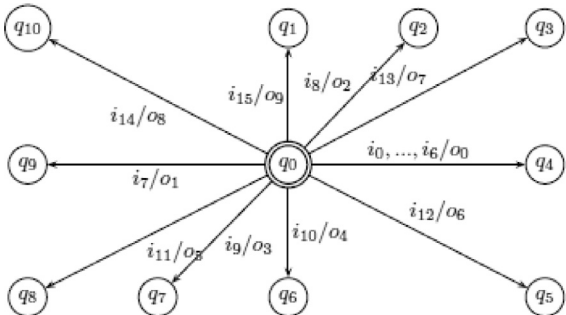
and $\text{image}_{M_2,2} = \{(o_1, o_1), (o_2, o_2)\}$. We have, on the one hand, that $\text{DRR}(f_{M_2,2}) = 6/2 = 3$ while, on the other hand, $\text{Sq}_2(M_2) = \frac{2 \cdot 3 \cdot \log_2(3)}{6} \approx 1.5849$.

In order to prove the second part of the result, let us consider again two machines M_1 and M_2 , with initial state q_0 , and we will show that they fulfill the required conditions. In these machines, we consider that $x_1, \dots, x_n/y$ is a shorthand for n different transitions labelled, respectively, by $x_1/y, x_2/y, \dots, x_n/y$. Let M_1 be:



We have that $\text{dom}_{M_1,1} = \{i_0, \dots, i_{15}\}$ and $\text{image}_{M_1,1} = \{o_0, \dots, o_8\}$. Therefore, $\text{DRR}(f_{M_1,1}) = 16/9 \approx 1.778$ while $\text{Sq}_1(M_1) = \frac{7 \cdot 2 \cdot \log_2(2) + 2 \cdot 1 \cdot \log_2(1)}{16} = 0.875$.

Finally, let M_2 be the FSM:

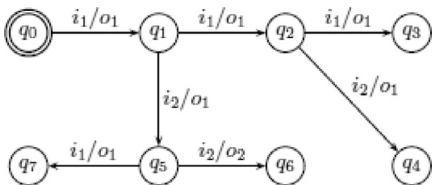


We have that $\text{dom}_{M_2,1} = \{i_0, \dots, i_{15}\}$ and $\text{image}_{M_2,1} = \{o_0, \dots, o_9\}$. Therefore, $\text{DRR}(f_{M_2,1}) = 16/10 = 1.6$ while $\text{Sq}_1(M_2) = \frac{1 \cdot 7 \cdot \log_2(7) + 9 \cdot 1 \cdot \log_2(1)}{16} \approx 1.2282$. \square

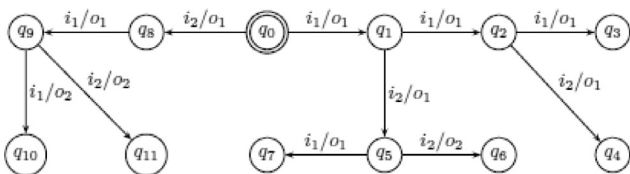
Lemma 5 There exist FSMs M_1 and M_2 and $k > 0$ such that $\text{Sq}_k(M_1) < \text{Sq}_k(M_2)$ but $\text{PColl}_k(M_1) > \text{PColl}_k(M_2)$.

Proof. First, let us note again that, similar to the proof of Lemma 4, in this proof we assume uniform distributions over inputs (and outputs) of the FSMs. Again, if we have a different probability distribution then we only need to adapt the definition of the machines so that the result still holds.

First, we consider M_1 with initial state q_0 :



Second, let M_2 , again with initial state q_0 , be:



On the one hand $\text{PColl}_3(M_1) = 0.5$ and $\text{PColl}_3(M_2) = 0.4$ while, on the other hand, we have $\text{Sq}_3(M_1) = 1.1887$ and $\text{Sq}_3(M_2) = 1.5849$. \square

Appendix B. Simulation Results

Here we show the results for the simulation performed on Section 4.1.

Table B1

First part of the results from the simulation.

Input set size	Maximum size	Correlation of Sq	Correlation of DRR
10,000	100	0.968366	0.763623
10,000	100	0.973918	0.783759
10,000	200	0.973016	0.823959
10,000	200	0.967349	0.77492
10,000	500	0.973849	0.828911
10,000	500	0.973267	0.803445
10,000	1000	0.963235	0.744021
10,000	1000	0.973658	0.804282
10,000	2000	0.969764	0.787121
10,000	2000	0.966409	0.753929
10,000	5000	0.968639	0.778538
10,000	5000	0.968937	0.768205
10,000	10,000	0.967281	0.71496
10,000	10,000	0.966184	0.670497
20,000	100	0.969669	0.780648
20,000	100	0.975364	0.824959
20,000	200	0.969587	0.778449
20,000	200	0.971707	0.771526
20,000	500	0.971942	0.798243
20,000	500	0.974174	0.792043
20,000	1000	0.971248	0.786153
20,000	1000	0.967574	0.769014
20,000	2000	0.967758	0.770978
20,000	2000	0.975119	0.819613
20,000	5000	0.972733	0.823052
20,000	5000	0.970411	0.780216
20,000	10,000	0.960576	0.728561
20,000	10,000	0.961688	0.724022
50,000	100	0.963278	0.74568
50,000	100	0.975731	0.817716
50,000	200	0.974623	0.795574
50,000	200	0.969418	0.746002
50,000	500	0.966153	0.777624
50,000	500	0.975947	0.84295
50,000	1000	0.967855	0.76079
50,000	1000	0.967894	0.789061
50,000	2000	0.96735	0.764992
50,000	2000	0.969433	0.804356
50,000	5000	0.97278	0.797072
50,000	5000	0.971647	0.792316
50,000	10,000	0.970928	0.779042
50,000	10,000	0.963673	0.723346
100,000	100	0.97475	0.797906
100,000	100	0.972203	0.799384
100,000	200	0.972457	0.788938
100,000	200	0.969988	0.78341
100,000	500	0.980028	0.836659
100,000	500	0.972878	0.769055
100,000	1000	0.976104	0.817482
100,000	1000	0.974571	0.820023
100,000	2000	0.971424	0.779667
100,000	2000	0.975182	0.787567
100,000	5000	0.96594	0.762143
100,000	5000	0.96303	0.73042
100,000	10,000	0.970134	0.757703
100,000	10,000	0.96836	0.778925
200,000	100	0.970841	0.801076
200,000	100	0.974049	0.798232
200,000	200	0.971829	0.776558
200,000	200	0.973847	0.79645
200,000	500	0.978293	0.822944
200,000	500	0.96523	0.748004
200,000	1000	0.968757	0.768184
200,000	1000	0.972733	0.808345
200,000	2000	0.971834	0.798966
200,000	2000	0.969003	0.749107
200,000	5000	0.970825	0.760313
200,000	5000	0.969484	0.76873
200,000	10,000	0.970044	0.792676
200,000	10,000	0.972554	0.788373

Table B2

Second part of the results from the simulation.

Input set size	Maximum size	Correlation of Sq	Correlation of DRR
500,000	100	0.97668	0.836037
500,000	100	0.977493	0.809851
500,000	200	0.963671	0.743951
500,000	200	0.974121	0.807426
500,000	500	0.971647	0.774395
500,000	500	0.973467	0.800447
500,000	1000	0.976121	0.820915
500,000	1000	0.97081	0.769445
500,000	2000	0.976695	0.803875
500,000	2000	0.973124	0.787502
500,000	5000	0.95885	0.743651
500,000	5000	0.969437	0.765643
500,000	10,000	0.971292	0.786862
500,000	10,000	0.975993	0.819747
1,000,000	100	0.976936	0.811867
1,000,000	100	0.971048	0.775681
1,000,000	200	0.970973	0.782711
1,000,000	200	0.977552	0.839242
1,000,000	500	0.972066	0.783899
1,000,000	500	0.974367	0.770392
1,000,000	1000	0.973926	0.79526
1,000,000	1000	0.974027	0.830407
1,000,000	2000	0.969736	0.780849
1,000,000	2000	0.97408	0.805192
1,000,000	5000	0.970854	0.809975
1,000,000	5000	0.970388	0.787131
1,000,000	10,000	0.967924	0.778203
1,000,000	10,000	0.970411	0.769844
2,000,000	100	0.975097	0.814434
2,000,000	100	0.968371	0.768775
2,000,000	200	0.974395	0.809679
2,000,000	200	0.97463	0.800698
2,000,000	500	0.97177	0.790358
2,000,000	500	0.970945	0.809109
2,000,000	1000	0.978102	0.826712
2,000,000	1000	0.971722	0.810432
2,000,000	2000	0.969418	0.755382
2,000,000	2000	0.970523	0.779241
2,000,000	5000	0.978818	0.810967
2,000,000	5000	0.964455	0.698505
2,000,000	10,000	0.96991	0.776906
2,000,000	10,000	0.963563	0.781282
5,000,000	100	0.971105	0.801428
5,000,000	100	0.975811	0.806359
5,000,000	200	0.965705	0.734183
5,000,000	200	0.975194	0.787636
5,000,000	500	0.965762	0.78538
5,000,000	500	0.977868	0.816896
5,000,000	1000	0.970797	0.782857
5,000,000	1000	0.974245	0.807752
5,000,000	2000	0.973636	0.783586
5,000,000	2000	0.972639	0.782383
5,000,000	5000	0.977712	0.793327
5,000,000	5000	0.963994	0.708333
5,000,000	10,000	0.972559	0.773815
5,000,000	10,000	0.975021	0.788634
10,000,000	100	0.972085	0.801643
10,000,000	100	0.96267	0.74051
10,000,000	200	0.973476	0.814127
10,000,000	200	0.978724	0.817254
10,000,000	500	0.968369	0.755809
10,000,000	500	0.976646	0.784194
10,000,000	1000	0.97411	0.792697
10,000,000	1000	0.970658	0.782375
10,000,000	2000	0.973856	0.793005
10,000,000	2000	0.974945	0.782697
10,000,000	5000	0.975649	0.814614
10,000,000	5000	0.9663	0.780145
10,000,000	10,000	0.974921	0.808942
10,000,000	10,000	0.974783	0.821714

Table B3

Third part of the results from the simulation.

Input set size	Maximum size	Correlation of Sq	Correlation of DRR
20,000,000	100	0.976361	0.816832
20,000,000	100	0.969996	0.785402
20,000,000	200	0.966911	0.773231
20,000,000	200	0.975891	0.830111
20,000,000	500	0.975834	0.80509
20,000,000	500	0.971753	0.761665
20,000,000	1000	0.970692	0.800126
20,000,000	1000	0.972765	0.780929
20,000,000	2000	0.975548	0.79739
20,000,000	2000	0.97661	0.790627
20,000,000	5000	0.975512	0.81321
20,000,000	5000	0.969801	0.778989
20,000,000	10,000	0.97061	0.79285
20,000,000	10,000	0.974807	0.823849
50,000,000	100	0.972157	0.775908
50,000,000	100	0.97394	0.744055
50,000,000	200	0.977712	0.825954
50,000,000	200	0.964124	0.754767
50,000,000	500	0.976058	0.824369
50,000,000	500	0.971696	0.792425
50,000,000	1000	0.968602	0.773925
50,000,000	1000	0.975643	0.813831
50,000,000	2000	0.972101	0.80533
50,000,000	2000	0.96896	0.763188
50,000,000	5000	0.967312	0.733459
50,000,000	5000	0.970914	0.792814
50,000,000	10,000	0.974186	0.831489
50,000,000	10,000	0.97075	0.794533
100,000,000	100	0.967785	0.791843
100,000,000	100	0.973939	0.79906
100,000,000	200	0.970936	0.797435
100,000,000	200	0.971179	0.792618
100,000,000	500	0.965457	0.764338
100,000,000	500	0.967388	0.749111
100,000,000	1000	0.967278	0.762974
100,000,000	1000	0.975128	0.816993
100,000,000	2000	0.976852	0.809661
100,000,000	2000	0.973916	0.811798
100,000,000	5000	0.964856	0.752126
100,000,000	5000	0.975177	0.804654
100,000,000	10,000	0.97333	0.797859
100,000,000	10,000	0.979012	0.839706
200,000,000	100	0.974298	0.793441
200,000,000	100	0.974201	0.817327
200,000,000	200	0.973198	0.79773
200,000,000	200	0.969628	0.752662
200,000,000	500	0.979169	0.843415
200,000,000	500	0.975039	0.830218
200,000,000	1000	0.975452	0.842656
200,000,000	1000	0.973656	0.81612
200,000,000	2000	0.974498	0.799512
200,000,000	2000	0.980097	0.843219
200,000,000	5000	0.97596	0.81765
200,000,000	5000	0.973072	0.794025
200,000,000	10,000	0.972525	0.790124
200,000,000	10,000	0.975228	0.812101
500,000,000	100	0.97099	0.788888
500,000,000	100	0.971083	0.798639
500,000,000	200	0.967438	0.779869
500,000,000	200	0.977179	0.832308
500,000,000	500	0.965965	0.778361
500,000,000	500	0.968144	0.764191
500,000,000	1000	0.974112	0.800833
500,000,000	1000	0.973997	0.779971
500,000,000	2000	0.971501	0.782711
500,000,000	2000	0.970228	0.743784
500,000,000	5000	0.976165	0.825479
500,000,000	5000	0.973031	0.779755
500,000,000	10,000	0.969547	0.772517
500,000,000	10,000	0.966348	0.773234

Table B4

Last part of the results from the simulation.

Input set size	Maximum size	Correlation of Sq	Correlation of DRR
1,000,000,000	100	0.96974	0.779927
1,000,000,000	100	0.974667	0.824957
1,000,000,000	200	0.978771	0.859822
1,000,000,000	200	0.968844	0.759952
1,000,000,000	500	0.975528	0.799788
1,000,000,000	500	0.972865	0.806221
1,000,000,000	1000	0.966998	0.742382
1,000,000,000	1000	0.970395	0.795114
1,000,000,000	2000	0.96474	0.784384
1,000,000,000	2000	0.966843	0.768588
1,000,000,000	5000	0.966975	0.753142
1,000,000,000	5000	0.969392	0.777797
1,000,000,000	10,000	0.970387	0.78255
1,000,000,000	10,000	0.966483	0.741448
2,000,000,000	100	0.968286	0.797514
2,000,000,000	100	0.974423	0.78976
2,000,000,000	200	0.97463	0.779878
2,000,000,000	200	0.969308	0.776731
2,000,000,000	500	0.97068	0.77233
2,000,000,000	500	0.964814	0.741365
2,000,000,000	1000	0.977148	0.802956
2,000,000,000	1000	0.972999	0.824011
2,000,000,000	2000	0.966897	0.756296
2,000,000,000	2000	0.967144	0.731439
2,000,000,000	5000	0.970575	0.807333
2,000,000,000	5000	0.965495	0.781112
2,000,000,000	10,000	0.969172	0.79843
2,000,000,000	10,000	0.972477	0.783512

References

- [1] 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, volume 4783, Springer, 2007, pp. 11–23.
- [2] N. Alshahwan, M. Harman, Coverage and fault detection of the output-uniqueness test selection criteria, in: 24th ACM SIGSOFT Int. Symposium on Software Testing and Analysis, ISSTA'14, ACM Press, 2014, pp. 181–192.
- [3] P. Ammann, J. Offutt, Introduction to Software Testing, 2nd, Cambridge University Press, New York, NY, USA, 2017.
- [4] K. Androustopoulos, D. Clark, H. Dan, R.M. 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.
- [5] R. Anido, A.R. Cavalli, L.A. Paula Lima Jr., N. Yevtushenko, Test suite minimization for testing in context, *Softw. Test. Verif. Reliab.* 13 (3) (2003) 141–155.
- [6] T. Apiwatanapong, R.A. Santelices, P.K. Chittimalli, A. Orso, M.J. Harrold, MA-TRIX: maintenance-oriented testing requirements identifier and examiner, in: 1st Testing: Academia and Industry Conference - Practice And Research Techniques, TAIC PART'06, IEEE Computer Society, 2006, pp. 137–146.
- [7] R.V. Binder, B. Legeard, A. Kramer, Model-based testing: where does it stand? *Commun. ACM* 58 (2) (2015) 52–56.
- [8] M. Boreale, M. Paolini, On formally bounding information leakage by statistical estimation, in: 17th Int. Conf. on Information Security, ISC'14, LNCS 8783, Springer, 2014, pp. 216–236.
- [9] C. Braunstein, A.E. Haxthausen, W.L. Huang, F. Hübner, J. Peleska, U. Schulze, L.V. Hong, Complete model-based equivalence class testing for the ETCS ceiling speed monitor, in: 16th Int. Conf. on Formal Engineering Methods, ICFEM'14, LNCS 8829, Springer, 2014, pp. 380–395.
- [10] A.R. Cavalli, T. Higashino, M. Núñez, A survey on formal active and passive testing with applications to the cloud, *Annal. Telecommun.* 70 (3–4) (2015) 85–93.
- [11] T. Chothia, Y. Kawamoto, C. Novakovic, Leakwatch: estimating information leakage from java programs, in: 19th European Symposium on Research in Computer Security, ESORICS'14, LNCS 8713, Springer, 2014, pp. 219–236.
- [12] T.S. Chow, Testing software design modeled by finite state machines, *IEEE Trans. Softw. Eng.* 4 (1978) 178–187.
- [13] D. Clark, R.M. Hierons, Squeeziness: an information theoretic measure for avoiding fault masking, *Inf. Process. Lett.* 112 (8–9) (2012) 335–340.
- [14] T.M. Cover, J.A. Thomas, Elements of Information Theory, Wiley Interscience, Hoboken, NJ, USA, 1991.
- [15] K. El-Fakih, A. Petrenko, N. Yevtushenko, FSM test translation through context, in: 18th Int. Conf. on Testing Communicating Systems, TestCom'06, LNCS 3964, Springer, 2006, pp. 245–258.
- [16] 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.
- [17] 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.
- [18] M.C. Gaudel, Testing can be formal, too!, in: 6th Int. Joint Conf. CAAP/FASE, Theory and Practice of Software Development, TAPSOFT'95, LNCS 915, Springer, 1995, pp. 82–96.
- [19] W. Grieskamp, Y. Gurevich, W. Schulte, M. Veanes, Generating finite state machines from abstract state machines, in: ACM SIGSOFT Symposium on Software Testing and Analysis, ISSTA'02, ACM Press, 2002, pp. 112–122.
- [20] W. Grieskamp, N. Kicillof, K. Stobie, V. Braberman, Model-based quality assurance of protocol documentation: tools and methodology, *Softw. Test. Verif. Reliab.* 21 (1) (2011) 55–71.
- [21] Q. Guo, R.M. Hierons, M. Harman, K. Derderian, Improving test quality using robust unique input/output circuit sequences (UIOCs), *Inf. Softw. Technol.* 48 (8) (2006) 696–707.
- [22] 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'14, ACM Press, 2016, pp. 523–534.
- [23] 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.
- [24] R.M. Hierons, Testing from partial finite state machines without harmonised traces, *IEEE Trans. Softw. Eng.* 43 (11) (2017) 1033–1043.
- [25] R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A.J.H. Simons, S. Vilkomir, M.R. Woodward, H. Zedan, Using formal specifications to support testing, *ACM Comput. Surv.* 41 (2) (2009). 9:1–9:76
- [26] R.M. Hierons, M.G. Merayo, M. Núñez, Implementation relations and test generation for systems with distributed interfaces, *Distribut. Comput.* 25 (1) (2012) 35–62.
- [27] R.M. Hierons, M.G. Merayo, M. Núñez, Bounded reordering in the distributed test architecture, *IEEE Trans. Reliab.* 67 (2) (2018) 522–537.
- [28] R.M. Hierons, M. Núñez, Using schedulers to test probabilistic distributed systems, *Formal Aspect. Comput.* 24 (4–6) (2012) 679–699.
- [29] R.M. Hierons, M. Núñez, Implementation relations and probabilistic schedulers in the distributed test architecture, *J. Syst. Softw.* 132 (2017) 319–335.
- [30] I. Hwang, A.R. Cavalli, Testing a probabilistic FSM using interval estimation, *Comput. Netw.* 54 (7) (2010) 1108–1125.
- [31] Z. Kohavi, Switching and Finite State Automata Theory, McGraw-Hill, 1978.
- [32] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines: a survey, *Proc. IEEE* 84 (8) (1996) 1090–1123.
- [33] W. Masri, R. Abou-Assi, M. El-Ghali, N. Al-Fatairi, An empirical study of the factors that reduce the effectiveness of coverage-based fault localization, in: 2nd Int. Workshop on Defects in Large Software Systems, DEFECTS'09, ACM Press, 2009, pp. 1–5.
- [34] E.P. Moore, Gedanken experiments on sequential machines, in: C. Shannon, J. McCarthy (Eds.), Automata Studies, Princeton University Press, 1956.
- [35] G.J. Myers, C. Sandler, T. Badgett, The Art of Software Testing, 3rd, John Wiley & Sons, Hoboken, NJ, USA, 2011.
- [36] J. Peleska, Model-based avionic systems testing for the airbus family, in: 23rd IEEE European Test Symposium, ETS'18, IEEE Computer Society, 2018, pp. 1–10.
- [37] A. Petrenko, Fault model-driven test derivation from finite state models: Annotated bibliography, in: 4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067, Springer, 2001, pp. 196–205.
- [38] A. Petrenko, S. Boroday, R. Groz, Confirming configurations in EFSM testing, *IEEE Transactions on Software Engineering* 30 (1) (2004) 29–42.
- [39] A. Petrenko, N. Yevtushenko, Testing from partial deterministic FSM specifications, *IEEE Transactions on Computers* 54 (9) (2005) 1154–1165.
- [40] A. Petrenko, N. Yevtushenko, G. von Bochmann, Testing deterministic implementations from their nondeterministic FSM specifications, in: 9th IFIP Workshop on Testing of Communicating Systems, IWTC'S'96, Chapman & Hall, 1996, pp. 125–140.
- [41] A. Petrenko, N. Yevtushenko, G. von Bochmann, R. Dssouli, Testing in context: framework and test derivation, *Computer Communications* 19 (1996) 1236–1249.
- [42] M. Shafique, Y. Labiche, A systematic review of state-based test tools, *Int. J. Softw. Tool. Technol. Transf.* 17 (1) (2015) 59–76.
- [43] C.E. Shannon, A mathematical theory of communication, *Bell Syst. Tech. J.* 27 (1948) 623–656. 379–423
- [44] X. Wang, S.C. Cheung, W.K. Chan, Z. Zhang, Taming coincidental correctness: coverage refinement with context patterns to improve fault localization, in: 31st Int. Conf. on Software Engineering, ICSE'09, IEEE Computer Society, 2009, pp. 45–55.
- [45] Y. Wang, M.U. Uyar, S.S. Batth, M.A. Fecko, Fault masking by multiple timing faults in timed EFSM models, *Comput. Netw.* 53 (5) (2009) 596–612.
- [46] M.R. Woodward, Z.A. Al-Khanjari, Testability, fault size and the domain-to-range ratio: an eternal triangle, in: 12th Int. Symposium on Software Testing and Analysis, ISSTA'00, ACM Press, 2000, pp. 168–172.

10.2 Estimating fault masking using Squeeziness based on Rényi's entropy

Authors	Alfredo Ibias and Manuel Núñez
Title	Estimating fault masking using Squeeziness based on Rényi's entropy
Publication Type	Conference
Venue	35th ACM/SIGAPP Symposium on Applied Computing
Year	2020
DOI/URL	https://doi.org/10.1145/3341105.3373920
Pages	8
Authors' Contributions	Ibias and Núñez developed the theory. Ibias and Núñez designed the experiments. Ibias developed and executed the experiments. Ibias and Núñez wrote the manuscript. Núñez reviewed the manuscript.

Estimating fault masking using Squeeziness based on Rényi's entropy

Alfredo Ibias

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid
Madrid, Spain
aibias@ucm.es

Manuel Núñez

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid
Madrid, Spain
mn@sip.ucm.es

ABSTRACT

Squeeziness is an Information Theory notion that has been proven to strongly correlate with the likelihood of Failed Error Propagation (FEP). This allows us to estimate the FEP of a certain system by computing its Squeeziness. The original notion of Squeeziness is based on the classical notion of entropy defined by Shannon. In this paper we study alternative notions of Squeeziness based on a more general notion of entropy introduced by Rényi. In contrast to Shannon's entropy, which is univocally defined, Rényi's entropy depends on a parameter α . We define Squeeziness by using Rényi's entropy and analyse the correlation of the different notions of Squeeziness with the likelihood of FEP. Our experiments showed that although $\alpha = 1$, corresponding to Shannon's entropy, induces good correlations, there are values of α showing better correlations.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Mathematics of computing** → *Information theory*; • **Theory of computation** → *Abstract machines*;

KEYWORDS

Formal approaches to testing; Information Theory; Failed Error Propagation; Rényi's entropy

ACM Reference Format:

Alfredo Ibias and Manuel Núñez. 2020. Estimating fault masking using Squeeziness based on Rényi's entropy. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20)*, March 30–April 3, 2020, Brno, Czech Republic. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3341105.3373920>

1 INTRODUCTION

Software Testing [2, 26] is the main validation technique to detect faults in software systems. Traditionally, Software Testing was a repertory of *informal* techniques. However, it has been shown that

it is possible to *formalise* it [9, 16] and there are many tools supporting the theoretical frameworks [23, 29]. One of the main scenarios where formal methods for testing are fundamental is black-box testing. In this scenario, the tester observes the reaction of the System Under Test (SUT) to the provided inputs without having access to the internal structure of the SUT. Many formal approaches have been developed for black-box testing but there are some shortcomings difficult to overcome. Among them, we can stand out Failed Error Propagation (FEP).

In terms of the RIP model [2] (stating that three conditions must be present for a failure to be observed: Reachability, Infection and Propagation), we might *reach* a fault such that the *infection* is not *propagated* to the final (observable) state. The lack of access to the internal structure of the SUT negates the tester the possibility to detect these faults in a black-box scenario.¹

It might be thought that if the previous faults do not alter the outputs, then we should not worry about them. However, more complex forms of FEP consist in faults whose errors do not propagate to the outputs in some cases, but they generate wrong outputs in other cases. These forms of FEP are specially dangerous because their detection depends on executing the right test, the one that propagates the error to the output. However, if the right test is not in the selected test suite (maybe because it will only detect this fault and the other tests could detect more than one fault at the same time), then the error will remain undetected. Finally, one last dangerous form of FEP appears in systems where the first execution of the faulty code does not generate the wrong output, but it still corrupts the internal state of the system. This can lead to a wrong output after some time.

An example of FEP is illustrated in Figure 1 (this is the scenario that we consider in this paper). We have a component C receiving a sequence of inputs from another component C_P . C and C_P can be modelled as Finite State Machines (FSMs). We assume that these values (sent by C_P to C) are not directly observed by the tester and that C produces a sequence of outputs that are either observed during testing or are received by another component. In this context, C_P could produce an unexpected sequence but component C could map the expected and unexpected sequences to the same output sequence: C would introduce a form of FEP that makes it more difficult to find faults in C_P . These faults could be unleashed if we compose C_P with a different component C' . Assume that we want to implement the component C_P given in the middle part of Figure 1 and that this component will be paired with component C . In this setting, it will be difficult to unmask a faulty implementation

¹These faults can be detected in a white-box scenario because the code is available and then we can *follow* the produced error.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SAC '20, March 30–April 3, 2020, Brno, Czech Republic

© 2020 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.

ACM ISBN 978-1-4503-6866-7/20/03...\$15.00

<https://doi.org/10.1145/3341105.3373920>

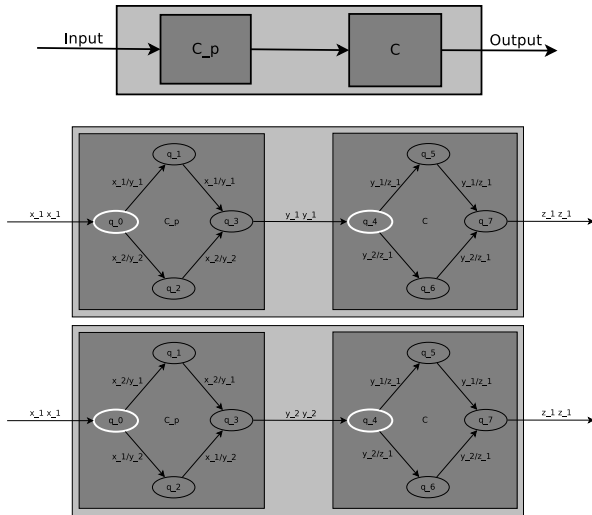


Figure 1: Representation of our testing scenario.

of C_p , such as the one shown in the lower part of Figure 1, because C returns the same response, the sequence z_1z_1 , to the sequences y_1y_1 (produced by a correct implementation of C_p receiving x_1x_1) and y_2y_2 (produced by a faulty implementation of C_p also receiving x_1x_1). Note, as we already said, that a tester will not be able to observe whether the sequence provided to C is y_1y_1 or y_2y_2 .

Since FEP cannot be directly detected, it is important to estimate its likelihood. Squeeziness [3, 11] was used to estimate FEP in a white-box scenario because there is a strong rank correlation between Squeeziness and FEP. Specifically, Squeeziness is a measure of the loss of information (entropy) that happens in a channel (in this case, the SUT) that takes inputs and return outputs. The idea behind it is that if the SUT maps two or more inputs to the same output then this channel (the SUT) can lead to a loss of information: if we know the program output then we may not know the program input that caused this. In recent work, and this is the notion that we consider in this paper, Squeeziness has been adapted to a black-box scenario [21], where the specification of the SUT is given as a FSM. The observable behaviour of an FSM is given by the set of input/output sequences, usually called *traces*, that label paths from the initial state of the system.

Squeeziness was always defined using Shannon's entropy [30], although there are many alternative notions to define what is intended as entropy. Rényi's entropy [28] provides an infinite family of *entropies* because its definition is parameterised by a positive real value α . A good property of this general notion is that Shannon's entropy, as well as other notions appearing in the literature, are specific cases of this generalisation (Shannon's entropy corresponds to $\alpha = 1$). The main goal of this paper is to generalise Squeeziness to deal with Rényi's entropy and explore how it improves the performance of Squeeziness. We compute the correlations between Squeeziness based on Rényi's entropy and the likelihood of FEP, for FSMs with different number of states. First, we consider the values for the extreme cases ($\alpha \in \{0, 1, \infty\}$). Afterwards, we use uniformly distributed values in the ranges $[0, 1]$, $[1, 10]$ and

$[10, 100]$. The obtained results were very promising. There is a correlation between FEP and Rényi's Squeeziness that ranges between 0.5 and 0.9. Furthermore, the best correlations are obtained with $\alpha \in (2, 3)$, where the correlations range between 0.75 and 0.9.

The rest of the paper is organised as follows. In Section 2 we explain some basic concepts on FSMs and present the main definitions of previous work. In Section 3 we develop the adaptation of the previously explained theory to the new concept of entropy. In Section 4 we explain the experiments performed to evaluate our newly developed theory. In Section 5 we present our conclusions and some lines of future work.

2 PRELIMINARIES

In this section we will present some concepts that are required to understand the work presented in this paper. These concepts are standard in classical work on testing from FSMs [22]. Most of them are based on the original sources, while some notation is adapted to facilitate the formulation of subsequent definitions.

2.1 Basic concepts

Given a set A , we let A^* denote the set of finite sequences of elements of A . $\epsilon \in A^*$ denotes the empty sequence. We let A^+ denote the set of non-empty sequences of elements of A . A^k denotes the set of sequences with length $k \geq 1$. We let $|A|$ denote the cardinal of set A . Given a sequence $\sigma \in A^*$, we have that $|\sigma|$ denotes its length. Given a sequence $\sigma \in A^*$ and $a \in A$, we have that σa denotes the sequence σ followed by a and $a\sigma$ denotes the sequence σ preceded by a .

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

A *Finite State Machine* is a (finite) labelled transition system in which transitions are labelled by an input/output pair. We use this formalism to define processes.

Definition 2.1. We say that $M = (Q, q_{in}, I, O, T)$ is a *Finite State Machine* (FSM), where Q is a finite set of states, $q_{in} \in Q$ is the initial state, I is a finite set of input actions, O is a finite set of output actions, and $T \subseteq Q \times (I \times O) \times Q$ is the transition relation. A transition $(q, (i, o), q') \in T$, also denoted by $q \xrightarrow{i/o} q'$ or by $(q, i/o, q')$, means that from state q after receiving input i it is possible to move to state q' and produce output o .

We say that M is *deterministic* if for all $q \in Q$ and $i \in I$ there exists at most one pair $(q', o) \in Q \times O$ such that $(q, i/o, q') \in T$. In this paper we consider deterministic FSMs.

An FSM can be represented by a diagram in which nodes represent states of the FSM and transitions are represented by arcs between the nodes. We use a double circle to denote the initial state.

As stated in the previous definition, we consider that FSMs are deterministic. This restriction is taken to mimic the white-box scenario where Squeeziness was originally introduced and considered, as usual, that programs are deterministic.

Definition 2.2. Let $M = (Q, q_{in}, I, O, T)$ be an FSM. We say that $(i_1, o_1) \dots (i_k, o_k) \in (I \times O)^*$ is a *trace* of M 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 $q_0 = q_{in}$. Let $s = i_1 \dots i_k \in I^*$ be a sequence of input actions. We define $\text{out}_M(s)$ as the set

$$\{o_1 \dots o_k \in O^* \mid (i_1/o_1) \dots (i_k/o_k) \text{ trace of } M\}$$

Note that if M is deterministic, then this set is either empty or a singleton. In the last case we will sometimes write $\text{out}_M(s) = o_1, \dots, o_k$.

We define dom_M as the set $\{s \in I^* \mid \text{out}_M(s) \neq \emptyset\}$. Similarly, we define image_M as the set

$$\{o_1 \dots o_k \in O^* \mid \exists s \in I^* : o_1 \dots o_k \in \text{out}_M(s)\}$$

We denote by $\text{dom}_{M,k}$ the set $\text{dom}_M \cap I^k$. Similarly, we denote by $\text{image}_{M,k}$ the set $\text{image}_M \cap O^k$.

2.2 Shannon-based Squeeziness in a black-box setting

Squeeziness has been used to estimate the existence of FEP in a black-box scenario [21]. In order to do that, FSMs represent specifications as functions that transform sequences of input actions into sequences of output actions. Those inputs will belong to $\text{dom}_M \subseteq I^*$, while outputs will belong to $\text{image}_M \subseteq O^*$. Projections of these functions restrict the function to sequences of input actions of length $k > 0$. Finally, we review the notion of *collision*, which happens when two different inputs produce the same output.

Definition 2.3. Let $M = (Q, q_{in}, I, O, T)$ be an FSM. We define $f_M : \text{dom}_M \rightarrow \text{image}_M$ as the function such that for all $s \in \text{dom}_M$ we have $f_M(s) = \text{out}_M(s)$.

Let $k > 0$. We define $f_{M,k}$ to be the function $f_M \cap (I^k \times O^k)$, where we use the function f_M to denote the associated set of pairs. Let $t \in \text{image}_M$. We define $f_M^{-1}(t)$ to be the set $\{s \in I^* \mid f_M(s) = t\}$.

Let $s_1, s_2 \in I^*$. We say that s_1 and s_2 collide for M if $s_1 \neq s_2$ and $f_M(s_1) = f_M(s_2)$.

Squeeziness represents the amount of information lost by a function. Thus, Squeeziness for an FSM was defined as the Squeeziness of the function that represents this FSM. In order to properly compute it, it was necessary to define how inputs are chosen and outputs are returned. A probabilistic view, where a random variable is associated with each set of relevant inputs/outputs, was considered. Specifically, a random variable was associated with the set of inputs/outputs of a certain length (that is, there are different random variables associated with $I^1, I^2, \dots; O^1, O^2, \dots$). Since $\text{dom}_{M,k}$ includes the inputs of length equal to k that M can perform and $\text{image}_{M,k}$ includes the outputs of length equal to k that M can produce after receiving an element of $\text{dom}_{M,k}$, random variables ranging over each set are defined as $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$, respectively. With these random variables, the concept of Squeeziness for FSMs was defined.

Definition 2.4. Let S be a set and ξ_S be a random variable over S . We denote by σ_{ξ_S} the probability distribution induced by ξ_S .

Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider two random variables $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$ ranging, respectively, over the domain and image of $f_{M,k}$. The Squeeziness of M at length k is defined as

$$\text{Sq}_k(M) = \mathcal{H}(\xi_{\text{dom}_{M,k}}) - \mathcal{H}(\xi_{\text{image}_{M,k}})$$

where $\mathcal{H}(\xi_S)$ denotes the (Shannon's) entropy of the random variable ξ_S that ranges over the set S , which is defined as

$$\mathcal{H}(\xi_S) = - \sum_{s \in S} \sigma_{\xi_S}(s) \cdot \log_2(\sigma_{\xi_S}(s))$$

There is an important remark concerning random variables associated with inputs and outputs: given an FSM M , $k > 0$ and a random variable $\xi_{\text{dom}_{M,k}}$, we have that the probability distribution of the random variable $\xi_{\text{image}_{M,k}}$ is completely determined. This is because for each element $t \in \text{image}_{M,k}$ we have that

$$\sigma_{\xi_{\text{image}_{M,k}}}(t) = \sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s)$$

Therefore, the formulation of Squeeziness is

$$\text{Sq}_k(M) = - \sum_{t \in \text{image}_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s) \right) \cdot \mathcal{R}_M(t)$$

where the term $\mathcal{R}_M(t)$ is equal to

$$\sum_{s \in f_M^{-1}(t)} \frac{\sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\sigma_{\xi_{\text{dom}_{M,k}}}(f_M^{-1}(t))} \cdot \log_2 \left(\frac{\sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\sigma_{\xi_{\text{dom}_{M,k}}}(f_M^{-1}(t))} \right)$$

Finally, the last concept that we will recall from previous work is *probability of collisions* (PColl [11]). In our context, fault masking (FEP) happens when the expected and faulty input sequences, received from another component, produce the same sequence t of output actions. If given an FSM M and $k > 0$ we have that there exists $t \in \text{image}_{M,k}$ such that $s, s' \in f_M^{-1}(t)$, with $s \neq s'$, then there is a collision. Note that collisions are a precondition of FEP.

Definition 2.5. Let M be an FSM and $k > 0$. Let $\text{image}_{M,k} = \{t_1, \dots, t_n\}$ and for all $1 \leq i \leq n$ let $I_i = f_{M,k}^{-1}(t_i)$ and $m_i = |f_{M,k}^{-1}(t_i)|$. We have that $d = \sum_{i=1}^n m_i$ is the size of the input space.

Given a uniform distribution over the inputs, the probability of s and s' both being in the set I_i is equal to $p_i = \frac{m_i \cdot (m_i - 1)}{d \cdot (d - 1)}$. We have that the probability of having a collision in M for sequences of length k , denoted by $\text{PColl}_k(M)$, is given by

$$\text{PColl}_k(M) = \sum_{i=1}^n \frac{m_i \cdot (m_i - 1)}{d \cdot (d - 1)}$$

With regard to this definition, a topic that has been already addressed is the potential to use $\text{PColl}_k(M)$ instead of Squeeziness. The problem with using $\text{PColl}_k(M)$ is that it is hard to compute. While this also applies to Squeeziness, the latter has the advantage of being an information theoretic measure. As a result, we can use Information Theory to either estimate or bound measures [6, 10], what will suffice for our task.

3 RÉNYI'S ENTROPY AND SQUEEZINESS

Previous work on Squeeziness used Shannon's entropy, but there exist alternative definitions of entropy that are worth exploring. In fact, there exists a general definition of entropy, dependent on a parameter α , called *Rényi's entropy* [28].

Definition 3.1. Let S be a set and ξ_S be a random variable over S . Let $\alpha \in \mathbf{R}_+ \setminus \{1\}$. The Rényi's entropy of the random variable ξ_S with respect to α , denoted by $\mathcal{H}_\alpha(\xi_S)$, is defined as:

$$\mathcal{H}_\alpha(\xi_S) = \frac{1}{1-\alpha} \cdot \log_2 \left(\sum_{s \in S} \sigma_{\xi_S}(s)^\alpha \right)$$

Let S and T be sets and $f : S \rightarrow T$ be a total function. Let us consider two random variables ξ_S and ξ_T ranging, respectively, over S and T , and $\alpha \in \mathbf{R}_+ \setminus \{1\}$. The Rényi's Squeeziness of f with respect to α , denoted by $\text{Sq}_\alpha(f)$, is defined as the loss of information after applying f to S taking into account α , that is, $\mathcal{H}_\alpha(\xi_S) - \mathcal{H}_\alpha(\xi_T)$.

It is well-known that when α tends to 1, Rényi's entropy becomes Shannon's entropy, that is,

$$\lim_{\alpha \rightarrow 1} \mathcal{H}_\alpha(\xi_S) = \mathcal{H}(\xi_S) = - \sum_{s \in S} \sigma_{\xi_S}(s) \cdot \log_2(\sigma_{\xi_S}(s))$$

Next, we can define the Squeeziness of an FSM using Rényi's entropy in the same way as it was defined in Definition 2.4.

Definition 3.2. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider two random variables $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$ ranging, respectively, over the domain and image of $f_{M,k}$. Let $\alpha \in \mathbf{R}_+ \setminus \{1\}$. Rényi's Squeeziness of M at length k with respect to α is defined as

$$\text{Sq}_{\alpha,k}(M) = \mathcal{H}_\alpha(\xi_{\text{dom}_{M,k}}) - \mathcal{H}_\alpha(\xi_{\text{image}_{M,k}})$$

We can provide an alternative definition of Rényi's Squeeziness taking into account, as previously explained, that we have

$$\sigma_{\xi_{\text{image}_{M,k}}}(t) = \sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s)$$

Therefore, we only need to use the probability distribution on inputs given by $\xi_{\text{dom}_{M,k}}$. The proof of the following result is straightforward.

LEMMA 3.3. Let $M = (Q, q_{in}, I, O, T)$ be an FSM, $k > 0$ and $\alpha \in \mathbf{R}_+ \setminus \{1\}$. Let us consider a random variable $\xi_{\text{dom}_{M,k}}$ ranging over the domain of $f_{M,k}$. We have that

$$\text{Sq}_{\alpha,k}(M) = \frac{1}{1-\alpha} \cdot \log_2 \left(\frac{\sum_{s \in \text{dom}_{M,k}} (\sigma_{\xi_{\text{dom}_{M,k}}}(s))^\alpha}{\sum_{t \in \text{image}_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s) \right)^\alpha} \right)$$

If α tends to 1 then we obtain Shannon's entropy [28] and we have

$$\text{Sq}_{1,k}(M) = - \sum_{t \in \text{image}_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s) \right) \cdot \mathcal{R}_M(t)$$

where the term $\mathcal{R}_M(t)$ is equal to

$$\sum_{s \in f_M^{-1}(t)} \frac{\sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\sigma_{\xi_{\text{dom}_{M,k}}}(f_M^{-1}(t))} \cdot \log_2 \left(\frac{\sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\sigma_{\xi_{\text{dom}_{M,k}}}(f_M^{-1}(t))} \right)$$

If α tends to ∞ then we obtain min-entropy [28] (that is, $\mathcal{H}_\infty(X) = -\log_2(\max_i p_i)$) and we have

$$\text{Sq}_{\infty,k}(M) = \log_2 \left(\frac{\max_{t \in \text{image}_{M,k}} \sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\max_{s \in \text{dom}_{M,k}} \sigma_{\xi_{\text{dom}_{M,k}}}(s)} \right)$$

The proof of the previous result when α tends to 1 uses the formulation of Squeeziness given in previous work [21].

The definition of Rényi's Squeeziness keeps some of the interesting properties of the notion of Squeeziness based on Shannon's entropy [21]. The first result corresponds to the relation of the bijectivity of a function and the nullity of its Squeeziness.

LEMMA 3.4. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. If $f_{M,k}$ is bijective then $\text{Sq}_{\alpha,k}(M) = 0$.

The second result corresponds to the non-monotonicity of the relationship between Squeeziness and PColl.

LEMMA 3.5. There exist FSMs M_1 and M_2 and $k > 0$ such that, for all $\alpha \in \mathbf{R}_+ \setminus \{1\}$, $\text{Sq}_{\alpha,k}(M_1) \leq \text{Sq}_{\alpha,k}(M_2)$ but $\text{PColl}_k(M_1) > \text{PColl}_k(M_2)$. In fact, the result also holds when α tends to 1 and when it tends to ∞ .

The previously defined notion of Squeeziness is parameterised by the distribution over the inputs of the function (that is, over the input sequences that the FSM can perform). If we know the actual distribution, then we can use this. If we do not know the distribution, then there is a need to choose one and we now discuss two approaches to do this.

3.1 Maximum entropy principle

We can select the distribution that maximises the entropy. If there are no further restrictions, maximum entropy is obtained with a uniform distribution [1, 12]. Then, under this distribution, the weight of a single element of $\text{dom}_{M,k}$ is $\frac{1}{|\text{dom}_{M,k}|}$ and the weight of the

inverse image of an output $t \in \text{image}_{M,k}$ is equal to $\frac{|f_M^{-1}(t)|}{|\text{dom}_{M,k}|}$.

Under these assumptions, and after some algebraic manipulations, the formula for Rényi's Squeeziness becomes:

$$\text{Sq}_{\alpha,k}(M) = \frac{1}{1-\alpha} \cdot \log_2 \left(\frac{|\text{dom}_{M,k}|}{\sum_{t \in \text{image}_{M,k}} (|f_M^{-1}(t)|)^\alpha} \right)$$

As usual, we have two special cases: α tending to 1 or to ∞ . If α tends to 1, then we are using Shannon's entropy and we have the following simplified formulation [21]:

$$\text{Sq}_{1,k}(M) = \frac{1}{|\text{dom}_{M,k}|} \cdot \sum_{t \in \text{image}_{M,k}} |f_M^{-1}(t)| \cdot \log_2(|f_M^{-1}(t)|)$$

If α tends to ∞ , then we are using min-entropy and, after some algebraic manipulations, we obtain the following formulation:

$$\text{Sq}_{\infty,k}(M) = \log_2 \left(\max_{t \in \text{image}_{M,k}} |f_M^{-1}(t)| \right)$$

3.2 Maximum loss of information

Another option is to consider the worst case scenario, that is, the scenario where the probability distribution induces the maximum loss of information. In order to maximize the loss of information, we need to maximize Squeeziness. Then, the probability distribution will be the one that is uniformly distributed in the largest inverse image of an element of the outputs and zero elsewhere [11]. Formally, consider $t' \in \text{image}_{M,k}$ such that for all $t \in \text{image}_{M,k}$ we have that $|f_M^{-1}(t')| \geq |f_M^{-1}(t)|$. Then,

$$\sigma_{\xi_{\text{dom}_{M,k}}}^{\xi}(s) = \begin{cases} \frac{1}{|f_M^{-1}(t')|} & \text{if } s \in f_M^{-1}(t') \\ 0 & \text{otherwise} \end{cases}$$

Using this probability distribution, the formulation of Rényi's Squeeziness can be transformed into the following one:

$$\text{Sq}_{\alpha,k}(M) = \log_2 \left(|f_M^{-1}(t')| \right)$$

In this case, unlike the previous ones, Squeeziness does not depend on the value of α . In particular, the two special cases (α tending to 1 and α tending to ∞) have the same formulation.

4 EMPIRICAL EVALUATION

In order to explore the convenience of Rényi's Squeeziness, we will use the same reference measure that has been used in previous work [11, 21]: the probability of collisions (PColl as introduced in Definition 2.5). Then, our experiments will essentially compute the correlation between Rényi's Squeeziness, for different values of α , and the corresponding values of PColl.

With this methodology in mind, we asked ourselves the following research questions.

4.1 Research Questions

In order to decide whether a notion of Squeeziness based on Rényi's entropy has some scientific interest, our first research question considers whether we obtain an improvement with respect to the framework where Shannon's entropy is used.

RESEARCH QUESTION 1. *Does there exist $\alpha \in \mathbf{R}_+ \setminus \{1\}$ whose corresponding Squeeziness correlates better with FEP than $\alpha = 1$? Is it unique?*

Then, in order to evaluate how the size of the FSM affects the capability of Squeeziness to detect FEP, we propose the following research question.

RESEARCH QUESTION 2. *Is there an improvement in the capability of Squeeziness to detect cases of FEP when the size of the FSM increases?*

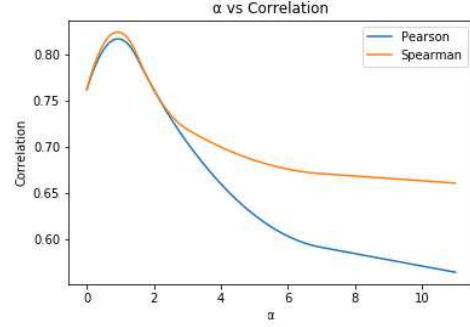


Figure 2: Initial hypothesis of the Pearson and Spearman correlations.

4.2 Experiments

In order to answer the research questions, we performed several experiments. We used 3 different sets of experimental subjects:

- Set1: 500 randomly generated FSMs with 50 states, 5 outgoing transitions from each state, and input and output alphabets of size 5.
- Set2: 3500 randomly generated FSMs with 5 outgoing transitions from each state, and input and output alphabets of size 5. This set is divided in 7 subsets, each one with 500 FSMs with the same number of states: 10, 20, 30, 40, 50, 60 and 70 states respectively.
- Set3: 241 deterministic FSMs coming from a recently collected benchmark [27], which represent real systems.

The code developed to perform these experiments can be found at the repositories <https://github.com/Colosu/RenyiSqueeziness> and <https://github.com/Colosu/RenyiSqueezinessReal>.

Our initial hypothesis was that $\alpha = 1$ could be the best possible value to use in the computation of Squeeziness based on Rényi's entropy. If we were able to show evidence of this hypothesis, then we could discard Rényi's entropy and stick to the original work on Squeeziness where Shannon's entropy was used. Therefore, we did a preliminary experiment where we computed the values of Squeeziness for Set1 and the extreme cases: $\alpha \in \{0, 1, \infty\}$. The best correlation between $\text{Sq}_{\alpha,10}(M)$ and $\text{PColl}_{10}(M)$ was obtained when $\alpha = 1$. Thus, we hypothesised that the curves showing correlation values versus α will be like the ones given in Figure 2.

In order to explore if our initial hypothesis was correct, we had to explore how the correlations perform for more values of $\alpha \in \mathbf{R}_+ \setminus \{1\}$ (specifically, we considered values of α uniformly distributed in the ranges $[0, 1]$, $[1, 10]$ and $[10, 100]$). In addition, we varied the number of states of the FSMs so that the results did not depend on a specific structure of the considered systems. In order to do that we set the following experiment. We decided to explore the correlations for FSMs with 10, 20, 30, 40, 50, 60 and 70 states (that is, Set2). Then, for each number of states we used 500 FSMs with the selected number of states, 5 outgoing transitions from each state, and input and output alphabets of size 5. Those parameters were

FSM size	10	20	30	40	50	60	70
$\alpha = 0$	0.486431	0.648953	0.701338	0.769901	0.762446	0.796774	0.795541
$\alpha = 0.1$	0.494168	0.657132	0.707142	0.776037	0.768796	0.802761	0.801591
$\alpha = 0.2$	0.502264	0.665466	0.712918	0.782091	0.775042	0.808578	0.807555
$\alpha = 0.3$	0.510699	0.673913	0.718639	0.788036	0.781146	0.814196	0.813402
$\alpha = 0.4$	0.51945	0.682431	0.724282	0.793846	0.787073	0.819592	0.819105
$\alpha = 0.5$	0.528487	0.690979	0.729829	0.7995	0.792791	0.824745	0.82464
$\alpha = 0.6$	0.537774	0.699512	0.735264	0.804978	0.798272	0.829641	0.829985
$\alpha = 0.7$	0.547273	0.70799	0.740575	0.810265	0.803492	0.834269	0.835126
$\alpha = 0.8$	0.556945	0.71637	0.745755	0.815351	0.808431	0.838623	0.840049
$\alpha = 0.9$	0.566748	0.724615	0.750799	0.820227	0.813072	0.8427	0.844744
$\alpha \rightarrow 1$	0.576644	0.732685	0.755706	0.824886	0.817402	0.8465	0.849204
$\alpha = 2$	0.666551	0.796947	0.79693	0.858781	0.842322	0.86988	0.879998
$\alpha = 3$	0.6879	0.814853	0.80419	0.856621	0.827749	0.85593	0.867284
$\alpha = 4$	0.668366	0.796306	0.769726	0.816314	0.768392	0.8129	0.81342
$\alpha = 5$	0.646528	0.771592	0.734235	0.778553	0.702129	0.768647	0.768431
$\alpha = 6$	0.629883	0.751551	0.706872	0.751479	0.652932	0.729861	0.734089
$\alpha = 7$	0.617827	0.736731	0.686756	0.732532	0.619978	0.699701	0.707358
$\alpha = 8$	0.609014	0.725885	0.671986	0.719035	0.59767	0.677549	0.686398
$\alpha = 9$	0.602426	0.717853	0.660975	0.709193	0.581985	0.661496	0.66999
$\alpha = 10$	0.597385	0.711794	0.652614	0.701865	0.570544	0.649774	0.657148
$\alpha = 20$	0.578372	0.690007	0.62309	0.677375	0.532961	0.612142	0.61089
$\alpha = 30$	0.573687	0.68491	0.617039	0.672633	0.525918	0.605177	0.601954
$\alpha = 40$	0.571616	0.682788	0.614691	0.670875	0.523325	0.602808	0.598745
$\alpha = 50$	0.570441	0.681663	0.613478	0.670023	0.522039	0.601727	0.597174
$\alpha = 60$	0.569685	0.680978	0.612748	0.669539	0.521291	0.601131	0.596266
$\alpha = 70$	0.569159	0.680523	0.612263	0.669233	0.520809	0.600758	0.595689
$\alpha = 80$	-nan	0.6802	0.611919	0.669024	0.520477	0.600504	0.595297
$\alpha = 90$	-nan	0.679959	0.611664	0.668872	0.520235	0.600319	0.595018
$\alpha = 100$	-nan	-nan	-nan	-nan	0.520052	0.600178	-nan
$\alpha \rightarrow \infty$	0.566197	0.678277	0.609975	0.667912	0.518761	0.599138	0.593596

Table 1: Pearson correlations between Rényi's Squeeziness and PColl.

FSM size	10	20	30	40	50	60	70
$\alpha = 0$	0.665932	0.726231	0.755532	0.750096	0.762307	0.777202	0.788155
$\alpha = 0.1$	0.671902	0.732722	0.762912	0.756609	0.769409	0.782764	0.793699
$\alpha = 0.2$	0.678459	0.739066	0.769426	0.762733	0.776362	0.789153	0.798919
$\alpha = 0.3$	0.684373	0.744736	0.776493	0.769181	0.783407	0.794884	0.804108
$\alpha = 0.4$	0.691055	0.751285	0.782856	0.775335	0.790109	0.800489	0.809323
$\alpha = 0.5$	0.696784	0.756789	0.789381	0.781042	0.797739	0.805876	0.813798
$\alpha = 0.6$	0.702745	0.762747	0.79492	0.786539	0.804071	0.811034	0.818569
$\alpha = 0.7$	0.708673	0.769094	0.799407	0.792022	0.8099	0.816104	0.822752
$\alpha = 0.8$	0.714102	0.774626	0.804144	0.797834	0.81562	0.820327	0.826531
$\alpha = 0.9$	0.720631	0.780229	0.809282	0.80226	0.820262	0.824522	0.830042
$\alpha \rightarrow 1$	0.726016	0.785447	0.812757	0.806145	0.824706	0.828592	0.833766
$\alpha = 2$	0.764498	0.823818	0.842291	0.837653	0.853455	0.851616	0.857426
$\alpha = 3$	0.764927	0.830047	0.837677	0.844951	0.846256	0.850812	0.852832
$\alpha = 4$	0.746544	0.818795	0.813882	0.827714	0.816204	0.833841	0.829668
$\alpha = 5$	0.730784	0.799161	0.783527	0.801646	0.776347	0.802588	0.79919
$\alpha = 6$	0.717461	0.779128	0.753262	0.777588	0.742974	0.771497	0.765878
$\alpha = 7$	0.707339	0.762354	0.728659	0.754509	0.715904	0.744383	0.734198
$\alpha = 8$	0.699611	0.748681	0.710361	0.737043	0.693262	0.724578	0.706547
$\alpha = 9$	0.693184	0.737617	0.696699	0.723698	0.675814	0.709165	0.682964
$\alpha = 10$	0.688319	0.729412	0.685111	0.713579	0.663306	0.696883	0.664693
$\alpha = 20$	0.668653	0.700691	0.648802	0.677524	0.614724	0.650798	0.595545
$\alpha = 30$	0.663651	0.693924	0.640996	0.670478	0.605531	0.639959	0.583053
$\alpha = 40$	0.660772	0.691046	0.637872	0.667504	0.602163	0.636674	0.578645
$\alpha = 50$	0.65943	0.689444	0.636246	0.666404	0.60056	0.635238	0.576823
$\alpha = 60$	0.658432	0.688849	0.635403	0.66581	0.599142	0.634257	0.575432
$\alpha = 70$	0.657831	0.688425	0.634893	0.665391	0.598497	0.633749	0.57483
$\alpha = 80$	0.657283	0.6882	0.634572	0.665172	0.59808	0.633143	0.574501
$\alpha = 90$	0.656874	0.687997	0.634294	0.664936	0.597839	0.632637	0.574192
$\alpha = 100$	0.655963	0.687767	0.634214	0.664901	0.597719	0.632636	0.573859
$\alpha \rightarrow \infty$	0.652587	0.685688	0.63173	0.662879	0.595149	0.629928	0.571167

Table 2: Spearman correlations between Rényi's Squeeziness and PColl.

selected in the same way as the ones used in previous work [21] so that we could properly compare the results.²

Then, we took each set of 500 FSMs with the same number of states and computed, for each $\alpha \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, \infty\}$, the values of $Sq_{\alpha,10}(M)$ and $PColl_{10}(M)$, that is, we considered sequences of 10 inputs for each FSM M . Afterwards, we computed the Pearson and Spearman correlations between these values. The full results are displayed in Tables 1 and 2. Interestingly, but somehow expected, although the Pearson and Spearman correlations are different in all the cases, the difference between both correlations strongly decreases when the size of the FSMs increases.

In order to analyse all these values, we performed, for each set of FSMs with the same number of states, the cubic interpolation of the correlations that those FSMs will have for an $\alpha \in [0, 101]$. The interpolation of the Pearson correlations is displayed in Figure 3 and the one corresponding to the Spearman correlations is displayed in Figure 4, with each curve corresponding to the different number of states of the FSMs.

All the curves have a peak in the range (2, 4). Therefore, we decided to reproduce the experiment but using $\alpha \in \{2, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, e, 2.8, 2.9, 3, 3.1, \pi, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4\}$. Then, we obtained the interpolations presented in Figures 5 and 6. From the plots we can observe that, for each number of states, we have a different α that gives the highest correlation but we can bound these values in the interval (2, 3).

²We performed some experiments with different sizes of inputs and outputs alphabets and the results were essentially the same.

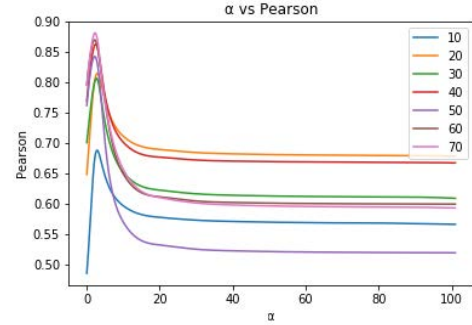


Figure 3: Interpolation of the Pearson correlations.

As a safety check, we decided to explore if the results are similar in a *real* scenario. In order to do that, we performed our experiments over a recently collected benchmark [27]. This benchmark has 241 deterministic FSMs, which represent real systems (the previously mentioned Set3). We took those FSMs and repeated the experiment: we computed, for each $\alpha \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, \infty\}$, the values of $Sq_{\alpha,3}(M)$ and $PColl_3(M)$ for each FSM M . Then, we computed the Pearson and Spearman correlations between these values. With these correlations, we interpolated the correlations for $\alpha \in [0, 101]$. Due to space limitations, we do not present the results in the paper, but the conclusion is that the interpolation curves behave similarly to the curves that we obtained with the randomly generated FSMs, that is, we obtain again peaks

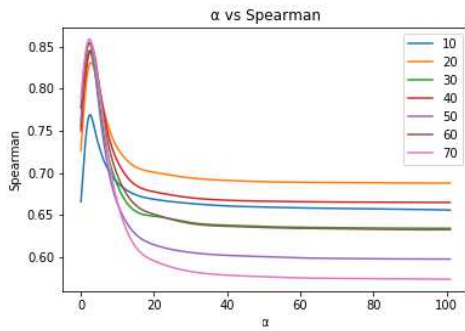


Figure 4: Interpolation of the Spearman correlations.

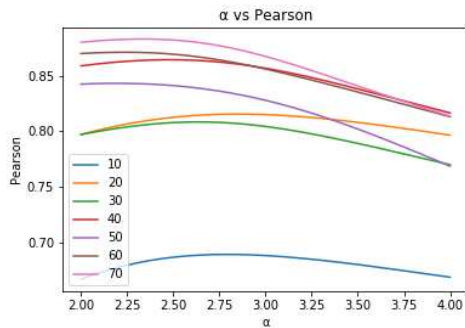


Figure 5: Interpolation of the Pearson correlations at their peak.

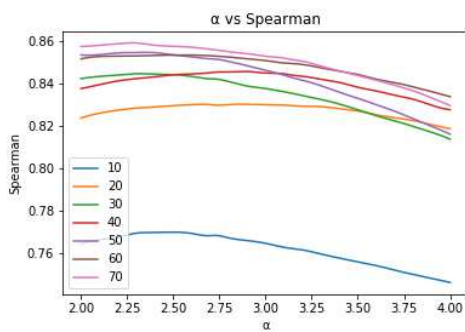


Figure 6: Interpolation of the Spearman correlations at their peak.

in the values of correlation for values of α belonging to the interval $(2, 3)$. Therefore, we have an empirical confirmation that the results obtained for randomly generated FSMs can be extrapolated to real FSMs.

4.3 Answers to the research questions

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

RESEARCH QUESTION 1. *Does there exist $\alpha \in \mathbb{R}_+ \setminus \{1\}$ whose corresponding Squeeziness correlates better with FEP than $\alpha = 1$? Is it unique?*

The answer to this question is positive. There exist values of α whose corresponding Squeeziness are better suited to detect FEP than $\alpha = 1$. However, this value is not unique. Although in all our experiments we can bound this α in the interval $(2, 3)$, the actual peak depends on the specific FSM. In any case, we conclude that using Renyi's entropy with values of α in this interval produces better notions of Squeeziness than the original notion where Shannon's entropy was used.

RESEARCH QUESTION 2. *Is there an improvement in the capability of Squeeziness to detect cases of FEP when the size of the FSM increases?*

The answer to this question is that, in general, there is an improvement. However, this improvement is not continuous and sometimes there is a deterioration of the results.

4.4 Threats to validity

We have to explore the threats to internal (related to uncontrolled factors), external (related to generalisation factors) and construct (related to reality factors) validity.

The main threat to the internal validity of our work is associated with the possible faults in the developed tools, which could lead to misleading results. In order to reduce the impact of this threat we tested our code with carefully constructed examples for which we could manually check the results. Luckily, once the FSMs have been (randomly) generated, our experiments had no randomisation factor involved. Therefore, there is no need to repeat the experiments.

The main external validity threat is the different representations of black-box components FSMs. Such a threat cannot be entirely addressed since this population is unknown and it is not possible to sample from an unknown population. In order to reduce the impact of this threat, we used a large number of randomly generated FSMs and checked our results with the results of repeating the experiment in a set of benchmark FSMs that represent real systems.

The main threat to the construct validity of our work is whether the FSMs used in the experiments correspond to possible system components. In order to reduce the impact of this threat, we restricted our range of FSM samples to connected deterministic machines. Also, we checked our results with the results of repeating the experiment in the set of already mentioned benchmark FSMs.

Finally, we have computed (many) results for $\alpha \in [0, 100]$ and have concluded that the peak of the correlations always belongs to the interval $(2, 3)$. Actually, all the curves were strictly decreasing from $\alpha = 3$. However, and this is an important threat to our results, we cannot claim that for a certain size of the analysed FSMs, there do not exist $\alpha \in (100, \infty)$ producing better correlations. We were sampling different values of α in the interval $(100, \infty)$ and confirmed that the correlations were decreasing. We have a strong confidence in this trend but it is not possible to prove that there does

not exist a better correlation for a value (or values) of $\alpha \in (100, \infty)$. Note that even if this value exists, but we claim again that this is very unlikely, our experimental results show that it will be difficult to compute sensible correlations. In fact, the results of the Pearson correlation for $\alpha = 100$ already show that five out of seven correlations could not be computed (see penultimate row of Table 1). Therefore, the potential small gain would be mitigated by the problems associated with the computation of the measure.

5 CONCLUSIONS AND FUTURE WORK

It is known that FEP can have a significant effect on testing. Recent work has shown that an information theoretic measure called Squeeziness strongly correlates with the likelihood of FEP both in white-box [11] and black-box [21] scenarios. However, this work only considered Squeeziness based on Shannon's entropy. In this paper we adapted the Squeeziness measure to be based in a more general notion: Rényi's entropy.

Once we defined our new notion of Squeeziness, we carried out experiments in order to evaluate this measure. In the experiments, we compared our measure with PCol1, a measure that has been shown to be very good in estimating the likelihood of FEP. We observed a strong correlation between PCol1 and our notion(s) of Squeeziness (formally, one notion for each $\alpha \in \mathbf{R}_+ \cup \{\infty\}$). Also, we observed better results when we chose values of α belonging to (2, 3). In particular, all these values return better correlations than $\alpha = 1$. Interestingly, our experiments also showed an improvement of the correlations when we were increasing the number of states of the generated FSMs and kept the value of α constant.

For future work, we have several lines of research. We plan to explore approximations, most likely based on sampling, and the trade-off between the cost of sampling (sample size) and the effectiveness of the estimates. We will extend our tool GPTSG [20] with the different pieces of software that we have developed to perform our experiments. Among other features, the tool will automatically choose an a priori *very good* value of α by taking into account the characteristics of the models. The new tool will also generate and process big amounts of mutants [8, 13–15]. Finally, we would like to take previous research as initial step to generalise the framework and measures to deal with asynchronous [17, 24, 25], distributed [7, 18, 19] and cloud [4, 5] systems.

ACKNOWLEDGMENTS

This work is supported by the MINECO-FEDER under Grant No.: RTI2018-093608-B-C31 and by the Region of Madrid under Grant No.: S2018/TCS-4314 co-funded by EIE Funds of the European Union.

REFERENCES

- [1] J. Acharya, A. Orlitsky, A. T. Suresh, and H. Tyagi. 2017. Estimating Rényi Entropy of Discrete Distributions. *IEEE Transactions on Information Theory* 63, 1 (2017), 38–56.
- [2] P. Ammann and J. Offutt. 2017. *Introduction to Software Testing* (2nd ed.). Cambridge University Press.
- [3] K. Androutsopoulos, D. Clark, H. Dan, R.M. Hierons, and M. Harman. 2014. 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, 573–583.
- [4] A. Bernal, M. E. Cambronoero, A. Núñez, P. C. Cañizares, and V. Valero. 2019. Improving cloud architectures using UML profiles and M2T transformation techniques. *The Journal of Supercomputing* 75, 12 (2019), 8012–8058.
- [5] A. Bernal, M. E. Cambronoero, V. Valero, A. Núñez, and P. C. Cañizares. 2019. A Framework for Modeling Cloud Infrastructures and User Interactions. *IEEE Access* 7 (2019), 43269–43285.
- [6] M. Boreale and M. Paolini. 2014. On Formally Bounding Information Leakage by Statistical Estimation. In *17th Int. Conf. on Information Security, ISC'14, LNCS 8783*. Springer, 216–236.
- [7] J. Boubeta-Puig, Gregorio Díaz, H. Macià, V. Valero, and G. Ortiz. 2019. MEdit4CEP-CPN: An approach for complex event processing modeling by prioritized colored Petri nets. *Information Systems* 81 (2019), 267–289.
- [8] P. C. Cañizares, A. Núñez, and M. G. Merayo. 2018. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software* 143 (2018), 187–207.
- [9] A. R. Cavalli, T. Higashino, and M. Núñez. 2015. A survey on formal active and passive testing with applications to the cloud. *Annales of Telecommunications* 70, 3–4 (2015), 85–93.
- [10] T. Chothia, Y. Kawamoto, and C. Novakovic. 2014. LeakWatch: Estimating Information Leakage from Java Programs. In *19th European Symposium on Research in Computer Security, ESORICS'14, LNCS 8713*. Springer, 219–236.
- [11] D. Clark and R. M. Hierons. 2012. Squeeziness: An information theoretic measure for avoiding fault masking. *Inform. Process. Lett.* 112, 8–9 (2012), 335–340.
- [12] T. M. Cover and J. A. Thomas. 1991. *Elements of Information Theory*. Wiley Interscience.
- [13] P. Delgado-Pérez, Louis M. Rose, and I. Medina-Bulo. 2019. Coverage-based quality metric of mutation operators for test suite improvement. *Software Quality Journal* 27, 2 (2019), 823–859.
- [14] P. Gómez-Abajo, E. Guerra, Juan de Lara, and M. G. Merayo. 2018. A tool for domain-independent model mutation. *Science of Computer Programming* 163 (2018), 85–92.
- [15] L. Gutiérrez-Madroñal, A. García-Domínguez, and I. Medina-Bulo. 2019. Evolutionary mutation testing for IoT with recorded and generated events. *Software - Practice & Experience* 49, 4 (2019), 640–672.
- [16] R. M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luetgten, A.J.H. Simons, S. Vilkomir, M.R. Woodward, and H. Zedan. 2009. Using formal specifications to support testing. *Comput. Surveys* 41, 2 (2009), 9:1–9:76.
- [17] R. M. Hierons, M. G. Merayo, and M. Núñez. 2017. An extended framework for passive asynchronous testing. *Journal of Logical and Algebraic Methods in Programming* 86, 1 (2017), 408–424.
- [18] R. M. Hierons, M. G. Merayo, and M. Núñez. 2018. Bounded reordering in the distributed test architecture. *IEEE Transactions on Reliability* 67, 2 (2018), 522–537.
- [19] R. M. Hierons and M. Núñez. 2017. Implementation relations and probabilistic schedulers in the distributed test architecture. *Journal of Systems and Software* 132 (2017), 319–335.
- [20] A. Ibas, D. Griñán, and M. Núñez. 2019. 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, 716–728.
- [21] A. Ibas, R. M. Hierons, and M. Núñez. 2019. Using Squeeziness to test component-based systems defined as Finite State Machines. *Information & Software Technology* 112 (2019), 132–147.
- [22] D. Lee and M. Yannakakis. 1996. Principles and methods of testing finite state machines: A survey. *Proc. IEEE* 84, 8 (1996), 1090–1123.
- [23] R. Marinescu, C. Seceleanu, H. Le Guen, and P. Pettersson. 2015. *A Research Overview of Tool-Supported Model-based Testing of Requirements-based Designs*. Advances in Computers, Vol. 98. Elsevier, Chapter 3, 89–140.
- [24] M. G. Merayo, R. M. Hierons, and M. Núñez. 2018. Passive Testing with Asynchronous Communications and Timestamps. *Distributed Computing* 31, 5 (2018), 327–342.
- [25] M. G. Merayo, R. M. Hierons, and M. Núñez. 2018. A tool supported methodology to passively test asynchronous systems with multiple users. *Information & Software Technology* 104 (2018), 162–178.
- [26] G. J. Myers, C. Sandler, and T. Badgett. 2011. *The Art of Software Testing* (3rd ed.). John Wiley & Sons.
- [27] D. Neider, R. Smetsers, F. W. Vaandrager, and H. Kuppens. 2019. Benchmarks for Automata Learning and Conformance Testing. In *Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, T. Margaria, S. Graf, and K. G. Larsen (Eds.). Springer, 390–416.
- [28] A. Rényi. 1961. On Measures of Entropy and Information. In *4th Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*. University of California Press, 547–561.
- [29] M. Shafique and Y. Labiche. 2015. A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer* 17, 1 (2015), 59–76.
- [30] C. E. Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27 (1948), 379–423, 623–656.

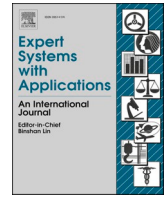
10.3 SqSelect: Automatic assessment of Failed Error Propagation in state-based systems

Authors	Alfredo Ibias and Manuel Núñez
Title	SqSelect: Automatic assessment of Failed Error Propagation in state-based systems
Publication Type	Journal
Venue	Expert Systems With Applications
Number	174
Year	2021
DOI/URL	https://doi.org/10.1016/j.eswa.2021.114748
Pages	17
Authors' Contributions	Ibias and Núñez developed the theory. Ibias and Núñez designed the experiments. Ibias developed and executed the experiments. Ibias and Núñez wrote the manuscript. Núñez reviewed the manuscript.



Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

SqSelect: Automatic assessment of Failed Error Propagation in state-based systems

Alfredo Ibias^a, Manuel Núñez^{a,b,*}

^a Design and Testing of Reliable Systems research group, Universidad Complutense de Madrid, Madrid, Spain

^b Institute of Knowledge Technology, Universidad Complutense de Madrid, Madrid, Spain

ARTICLE INFO

Keywords:

Software testing
Failed error propagation
Expert systems
Information theory

ABSTRACT

Current software systems are inherently complex and this fact strongly complicates, and makes more expensive, to validate them. Therefore, it is a must to provide methodologies, supported by tools, that can direct validation activities so that they focus on specific aspects of the system (e.g. its critical parts, common errors produced by developers, components that are expensive to fix after deployment, etc). Among the different validation techniques, testing is the most widely used. In this paper we focus on one of the main problems when testing systems with many components: the likelihood of Failed Error Propagation (FEP). FEP appears when we have faulty components such that their wrong behaviour is not revealed when isolatedly testing them but that might produce an error when they are combined with other components. Given a component, it is not possible to automatically assess the likelihood of FEP. However, previous work has shown that there is a strong correlation between the likelihood of FEP and an Information Theory notion called Squeeziness. Recent work has shown that it is possible to compute different values of Squeeziness (essentially, Squeeziness depends on a positive real value) and some of them are more suitable to estimate FEP. In this paper we present our tool SqSelect. Our tool receives either a specific system or its more important characteristics (number of states, maximum and minimum number of outgoing transitions from a state, size of the input and output alphabets) and returns interesting data that can help the tester to estimate the presence of FEP. In particular, our tool provides the most promising value (s) of the parameter associated with Squeeziness so that the likelihood of FEP can be more accurately estimated. In order to compute these values, our tool relies on an artificial neural network that has been extensively trained (compressing the information from around 250,000 systems and around 1,500,000 executions).

1. Introduction

The main goal of an expert system is to replace a human expert in the process of providing a decision or verdict. Usually, expert systems have to deal with big amounts of information so that they are indeed more effective than the replaced human experts because they will not be able to process all the needed information. Expert systems currently deal with very heterogeneous systems: they can assess the risks of cost overruns in real power plants (Islam, Nepal, Skitmore, & Kabir, 2019), rank hockey players according to difference performance metrics (Gu, Foster, Shang, & Wei, 2019), deal with different environments in the Health domain (Díaz, Macià, Valero, Boubeta-Puig, & Cuartero, 2020;

García de Prado, Ortiz, & Boubeta-Puig, 2017), or detect security attacks (Roldán, Boubeta-Puig, Martínez, & Ortiz, 2020), among many others. The goal of this paper is to introduce an expert system such that given a state-based system, and using an artificial neural network (ANN) previously trained, provides relevant information, in a structured and comprehensive way, that will help testers by guiding some of the testing activities.

Software Testing (Ammann & Offutt, 2017; Myers, Sandler, & Badgett, 2011) is the main validation technique to detect faults in software systems. Although Software Testing was traditionally considered to be mainly *informal*, this situation has changed. Actually, from the 1990s it is well-known and understood that testing can be formalised

This work has been supported by the Spanish MCIU-FEDER (grant number FAME, RTI2018-093608-B-C31), the Region of Madrid (grant number FORTE-CM, S2018/TCS-4314), the Region of Madrid – Complutense University of Madrid (grant number PR65/19-22452) and the Santander – Complutense University of Madrid (grant number CT63/19-CT64/19).

* Corresponding author.

E-mail addresses: aibias@ucm.es (A. Ibias), mn@sip.ucm.es (M. Núñez).

<https://doi.org/10.1016/j.eswa.2021.114748>

Received 21 February 2020; Received in revised form 17 June 2020; Accepted 16 February 2021

Available online 26 February 2021

0957-4174/© 2021 Elsevier Ltd. All rights reserved.

(Gaudel, 1995). Thus, formal approaches to testing is a flourishing field of study (Cavalli, Higashino, & Núñez, 2015; Hierons et al., 2009) and there are many tools supporting the theoretical frameworks (Marinescu, Seceleanu, Guen, & Pettersson, 2015; Shafique & Labiche, 2015). One of the main scenarios where formal methods for testing are fundamental is black-box testing: the tester provides inputs to the system under test (SUT), without having access to its internal structure, observe its reaction (outputs) and analyse whether these reactions are expected or not.

Testing a complex system consists of many strands. For example, depending on the amount of resources available for testing, we can perform activities such as analyse the isolated behaviour of different components, assess the robustness of a system or check whether a system is using more resources than expected. However, one facet is unavoidable: current software systems are composed of many components and an appropriate testing plan must assess how different units/components work together. One of the main problems that can appear when testing components that are working together is Failed Error Propagation (FEP). Suppose that we execute a faulty statement of a component, so that the resulting internal state becomes faulty, but the differences between the faulty and correct state are not reflected in the output provided by the component. If this component works alone, then FEP does not represent a problem but if we transfer the (faulty) state to a different component, then an unforeseen error might appear.

In Fig. 1 we graphically show the scenario where FEP can appear. We use Finite State Machines (FSMs) to represent components but any other state-based formalism could be used and the adaption of our framework would be straightforward. We have two components, C_p and C . The former receives a sequence of inputs. In particular, this sequence can be

provided by the tester. Then, C_p performs some computations and sends a sequence of inputs to C . In order to not violate the assumption concerning a black-box framework, we assume that the actions sent from C_p to C cannot be observed by the tester. C will produce a sequence (they will represent outputs of the whole system) that can be observed by the tester. In this context, the component C_p could produce an unexpected sequence, this is the one received by C , but C might produce the same result from the expected and unexpected sequences: C would introduce a form of FEP that makes it more difficult to find faults in C_p . These faults could be revealed if C_p is combined with another component C' .

Assume that we want to implement the component C_p given in the middle part of Fig. 1 and that this component will be paired with component C . In this setting, it will be difficult to unmask a faulty implementation of C_p , such as the one shown in the lower part of Fig. 1, because C returns the same response, the sequence $z_1 z_1$, to the sequences $y_1 y_1$ (produced by a correct implementation of C_p receiving $x_1 x_1$) and $y_2 y_2$ (produced by a faulty implementation of C_p also receiving $x_1 x_1$). Note, as we already said, that a tester will not be able to observe whether the sequence provided to C is $y_1 y_1$ or $y_2 y_2$.

Several empirical studies have shown that FEP hampers testing (Masri, Abou-Assi, El-Ghali, & Al-Fatairi, 2009; Santelices & Harrold, 2011; Wang, Cheung, Chan, & Zhang, 2009): in 13% of the examined programs, a total of 60% or more of the tests suffered from FEP (Masri et al., 2009). Although it is not clear how Software Testing could be better designed to ameliorate the problems caused by FEP, it is important to reduce the probability that test cases will suffer from FEP and, therefore, it is useful to have different metrics that can help us to identify parts of a system that are more likely to lead to FEP (Androutsopoulos, Clark, Dan, Hierons, & Harman, 2014). There is a line of research

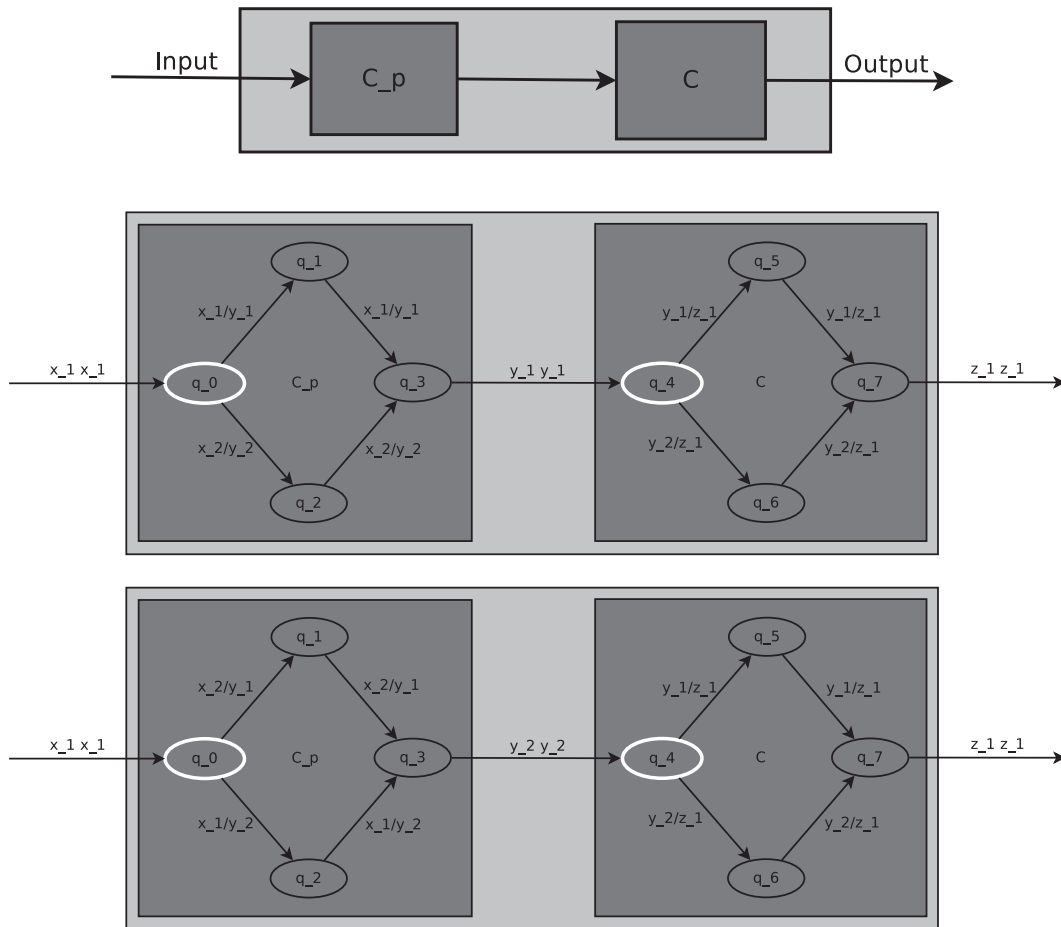


Fig. 1. Representation of our testing scenario.

looking for measures to indirectly estimate FEP and it seems that an Information Theory measure called Squeeziness (Clark & Hierons, 2012; Clark, Hierons, & Patel, 2019) is a good candidate. Actually, an empirical study (Androutopoulos et al., 2014) of 30 programs and more than $7 \cdot 10^6$ tests showed that the Spearman rank correlation of Squeeziness with FEP is close to 0.95. Squeeziness was adapted to a black-box testing framework (Ibias, Hierons, & Núñez, 2019) and the correlation with FEP was also very high (around 0.80 for both Pearson and Spearman correlations).

The original notion of Squeeziness relies on Shannon's entropy (Shannon, 1948). Although this is a *standard* notion, it is not the only possibility. Actually, it is possible to define an infinite number of notions of entropy, one for each $\alpha \in \mathbb{R}_+$: Rényi's entropy (Rényi, 1961). Shannon's entropy corresponds to Rényi's entropy when $\alpha = 1$. Recent work (Ibias & Núñez, 2020) has studied the different notions of Squeeziness induced by each value of α and, interestingly enough, has shown that for each system there is a different *best value* of α . By best value we mean the value such that the corresponding notion of Squeeziness has the highest rank correlation with FEP. In particular, even though $\alpha = 1$ always provides good correlations, all the experiments revealed that the best value was never reached with this value. Therefore, given a system, it would be interesting to know the optimum value of α so that the FEP of the system can be estimated with a higher precision. In order to compute this value, there are two important facts that must be taken into account. First, the computation of this value for a specific system needs a non-negligible amount of computing power. Second, it has been observed (Ibias & Núñez, 2020) that *similar* systems obtain *similar* values. Thus, it would be enough to have an approximate method to compute a good enough approximation of the desired value. We have developed a tool, called SqSelect, that receives either a specific system or its defining characteristics (among others, number of states, maximum and minimum number of outgoing transitions, size of the input and output alphabets) and provides several interesting data that can help the tester to decide the amount of testing that should be used to evaluate FEP. Specifically, SqSelect provides the value of Squeeziness that more accurately estimates the likelihood of having cases of FEP in the system. In order to compute our solution, integrated in SqSelect, we rely on an ANN that we have extensively trained using 249,000 different systems and a total of 1,494,000 executions, obtaining a mean loss of only 1.280935.

Concerning related work, there is plenty of work on Information Theory and testing (Androutopoulos et al., 2014; Clark, Feldt, Poulding, & Yoo, 2015; Clark & Hierons, 2012; Feldt, Poulding, Clark, & Yoo, 2016; Feldt, Torkar, Gorschek, & Afzal, 2008; Ibias et al., 2019; Ibias, Núñez, & Hierons, 2021; Miransky, Davison, Reesor, & Murtaza, 2012; Pattipati & Alexandridis, 1990; Pattipati, Deb, Dontamsetty, & Maitra, 1990; Sagarna, Arcuri, & Yao, 2007; Yoo, Harman, & Clark, 2013) where theoretical frameworks, sometimes supported by empirical evidence, are presented and discussed. However, to the best of our knowledge, there do not exist frameworks where relevant Information Theory measures are automatically (and using a small computing power) estimated. There is a recent proposal to use expert systems in testing (Cañizares, Núñez, & Lara, 2019) but their focus is on memory systems and the underlying testing approach is Metamorphic Testing. It is natural that expert and recommender systems use artificial neural networks, and we may mention some recent work (de Mesquita Sá et al., 2019; Ayala, Borrego, Hernández, & Ruiz, 2020), but their goal is not related to testing and their field of application are very far from ours. Finally, it is worth mentioning that artificial neural networks are a good tool to confront testing activities (Serna et al., 2019) but they have never been used to estimate Information Theory measures. There is some work in the field of *profiling* tools for error propagation but our work diverges in several aspects and, therefore, it is difficult to compare it with them. PROPANE (Hiller, Jhumka, & Suri, 2002) needs the source code to work, that is, it can be classified as a white-box approach. EPIC (Hiller,

Jhumka, & Suri, 2004) considers a running system. Therefore, it can be considered to be a black-box approach. A similar consideration applies to later work dealing with the analysis of operating systems (Coppik, Schwahn, Winter, & Suri, 2017; Johansson & Suri, 2005), multi-threaded programs (Chan, Winter, Saissi, Pattabiraman, & Suri, 2017), automotive systems (Piper, Winter, Schwahn, Bidarahalli, & Suri, 2015) and software architecture (Abdelmoez et al., 2004). Although SqSelect also works with a black-box, the abstract representation of the systems that we consider, represented as FSMs, is very far from the systems studied in these profiling approaches. However, the most important difference between the previously mentioned approaches and ours concerns how error propagation is studied. The former ones analyse error propagation of the system with the goal of detecting dependencies between components and determine which components can have the most harmful errors for the whole system (because their errors propagate more than others). In our case, our goal is to determine which components can hide errors from being detected, that is, which components do not propagate errors. In Fig. 2 we sketch the main features of our tool and of the tools and studies more related to ours. Specifically, we mention whether the study uses Information Theory, is tool-supported, works with a white/black box or considers error propagation or failed error propagation.

The rest of the paper is structured as follows. In Section 2 we introduce the notations and concepts that we will use. Section 3 provides a high level description of the behaviour of SqSelect and explains how our ANN was designed, trained and tested. In Section 4 we present the different modes that are available in our tool. In Section 5 we discuss some design decisions and justify the chosen options. In Section 6 we give some representative case studies to show the behaviour of our tool. Finally, in Section 7 we present our conclusions and some lines for future work.

2. Background

In this section we present the basic concepts and notions that are required to understand the work presented in this paper.

2.1. Basic concepts

Given a set A , we let A^* denote the set of finite sequences composed from elements of A ; as usual, we consider that $\epsilon \in A^*$ denotes the empty sequence. We let A^+ denote the set of non-empty sequences of elements of A . A^k denotes the set of sequences with length $k \geq 1$. We let $|A|$ denote the cardinal of set A . Given a sequence $\sigma \in A^*$, we have that $|\sigma|$ denotes its length. Given a sequence $\sigma \in A^*$ and $a \in A$, we have that σa denotes the sequence σ followed by a and $a\sigma$ denotes the sequence σ preceded by a .

Throughout this paper we let I be the set of input actions and O be the set of output actions. In our context, it is important to distinguish between input actions and *inputs* of the system. Specifically, an input of a system is a non-empty sequence of input actions, that is, an element of I^+ . We have a similar situation concerning output actions and *outputs* of the system. A *Finite State Machine* is a (finite) labelled transition system in which transitions are labelled by an input/output pair.

Definition 1. A tuple $M = (Q, q_{in}, I, O, T)$ is a *Finite State Machine* (FSM), where Q is a finite set of states, $q_{in} \in Q$ is the initial state, I is a finite set of input actions, O is a finite set of output actions, and $T \subseteq Q \times (I \times O) \times Q$ is the transition relation. A transition $(q, (i, o), q') \in T$, also denoted by $(q, i/o, q')$, means that from state q after receiving the input action i it is possible to move to state q' and produce the output action o . We say that M is *deterministic* if for all $q \in Q$ and $i \in I$ there exists at most one pair $(q', o) \in Q \times O$ such that $(q, i/o, q') \in T$. In this paper we consider deterministic FSMs.

In this paper we use this simple formalism to define processes but our

Tool or framework	Information Theory	Tool supp.	Black box	White box	FEP	Error Prop.	Field of application
SqSelect	General Squeeziness	Yes	Yes	No	Yes	No	FSMs
(Ibiás et al., 2019)	Squeeziness	Yes	Yes	No	Yes	No	FSMs
TrEKer (Coppik et al., 2017)	No	Yes	Yes	No	No	Yes	Operating Systems Kernels
IPA (Chan et al., 2017)	No	Yes	No	Yes	No	Yes	Analysis of Multithreads
PBM (Piper et al., 2015)	No	Yes	Yes	No	No	Yes	Automotive Systems
(Androutsopoulos et al., 2014)	Squeeziness	No	No	Yes	Yes	No	Source Code
(Clark and Hierons, 2012)	Squeeziness	No	No	Yes	Yes	No	Source Code
(Johansson and Suri, 2005)	No	No	Yes	No	No	Yes	Operating Systems
EPIC (Hiller et al., 2004)	No	Yes	Yes	No	No	Yes	Data Errors in Software
(Abdelmoez et al., 2004)	No	No	No	Yes	No	Yes	Software Architecture
PROPANE (Hiller et al., 2002)	No	Yes	No	Yes	No	Yes	Source Code

Fig. 2. Comparison of SqSelect with other (failed) error propagation tools and frameworks.

methodology can be easily adapted to deal with any other state-based formalism as long as it provides notions of inputs and outputs. Note that FSMs, despite their simplicity, are frequently used to represent a wide variety of systems: state-based software systems, logic circuits, communication protocols, etc. An FSM can be seen as a diagram where nodes denote states of the FSM and transitions are represented by arcs between the nodes. We use a double circle to denote the initial state.

As stated in the previous definition, we consider that FSMs are deterministic. This restriction is taken to mimic the white-box scenario where Squeeziness was originally introduced and considered, as usual, that programs are deterministic.

Definition 2. Let $M = (Q, q_{in}, I, O, T)$ be an FSM. We say that $(i_1, o_1) \dots (i_k, o_k) \in (I \times O)^*$ is a trace of M 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 $q_0 = q_{in}$. We denote by $traces(M)$ the set of traces of M . Note that $\epsilon \in traces(M)$. Let $s = i_1 \dots i_k \in I^*$ be a sequence of input actions. We define $out_M(s)$ as the set

$$\{o_1 \dots o_k \in O^* \mid (i_1/o_1) \dots (i_k/o_k) \text{ trace of } M\}$$

Note that if M is deterministic, then this set is either empty or a singleton. In the last case we will simply write $out_M(s) = o_1, \dots, o_k$.

We define dom_M as the set $\{s \in I^* \mid out_M(s) \neq \emptyset\}$. Similarly, we define $image_M$ as the set

$$\{o_1 \dots o_k \in O^* \mid \exists s \in I^* : o_1 \dots o_k \in out_M(s)\}$$

We denote by $dom_{M,k}$ the set $dom_M \cap I^k$. Similarly, we denote by $image_{M,k}$ the set $image_M \cap O^k$. Next we present a simple example to illustrate the previous concepts.

Example 1. In Fig. 3 we have an FSM with 10 states, S_0 is its initial state (the double circle), a set of inputs $I = \{a, b, c\}$ and set of outputs $O = \{x, y, z\}$. For example, $(a, z)(a, x)$ is a trace that takes the system from state S_0 to state S_4 ; $out_M(aa) = zx$ while $out_M(ba) = \emptyset$. We have

$$dom_M = \{a, b, c, aa, ab, bb, ca, cb, cc\}$$

$$image_M = \{z, y, zx, zz, yz\}$$

Moreover, $dom_{M,1} = \{a, b, c\}$ and $image_{M,2} = \{zx, zz, yz\}$. Finally, we review the concept of Failed Error Propagation. This kind of error appears when we have a fault in the program but the error associated with this fault does not have an effect in the output of the program. Using the RIP model (Ammann & Offutt, 2017) (stating that three conditions must be present for a failure to be observed: Reachability, Infection and Propagation), we might reach a fault such that the infection is not propagated to

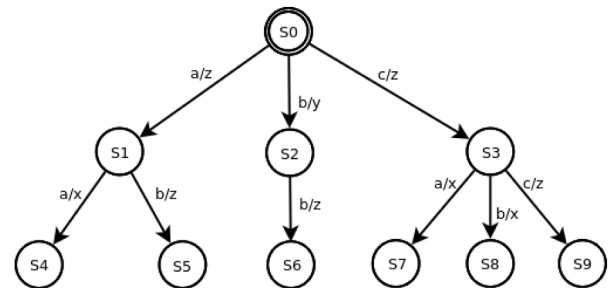


Fig. 3. FSM example.

the final (observable) state. The lack of access to the internal structure of the SUT negates the tester the possibility to detect these faults in a black-box scenario.¹ As we have already mentioned in the introduction of this paper, empirical studies have shown that many systems suffer from FEP (Masri et al., 2009): in 13% of the analysed programs, a total of 60% or more of the tests suffered from FEP.

It might be thought that if the previous faults do not alter the outputs, then we should not worry about them. However, complex forms of FEP include faults whose errors do not propagate to the outputs in some cases, but they generate wrong outputs in other cases. These forms of FEP are especially dangerous because their detection depends on executing the right test, the one that propagates the error to the output. Due to resources, time and budget restrictions this could be a hard work, because the testing process has to be restricted to the application of some selected cases: the ones that are more promising at detecting faults. However, if the right test is not in the selected test suite (maybe because it will only detect this fault and the other tests could detect more than one fault at the same time), then the error will remain undetected.

2.2. Shannon-based Squeeziness in a black-box setting

Before we introduce the notion of Squeeziness, we define some auxiliary concepts. First, note that FSMs can be seen as functions that transform inputs into outputs. Projections of these functions restrict the function to inputs of a certain length. Finally, we review the notion of collision, which happens when two different inputs produce the same

¹ Note that these faults can be observed in a white-box scenario because the tester has access to the code and, therefore, it is possible to follow the produced error.

output.

Definition 3. Let $M = (Q, q_{in}, I, O, T)$ be an FSM. We define $f_M : \text{dom}_M \rightarrow \text{image}_M$ as the function such that for all $s \in \text{dom}_M$ we have $f_M(s) = \text{out}_M(s)$. Given $k > 0$, we define $f_{M,k}$ as the function $f_M \cap (I^k \times O^k)$, where f_M denotes the associated set of pairs. Let $t \in \text{image}_M$. We define $f_M^{-1}(t)$ to be the set $\{s \in I^* | f_M(s) = t\}$. Let $s_1, s_2 \in I^*$. We say that s_1 and s_2 collide for M if $s_1 \neq s_2$ and $f_M(s_1) = f_M(s_2)$.

Example 2. Consider again the FSM depicted in Fig. 3. For example, f_M maps $aa \rightarrow zx$, $ab \rightarrow zz$ and $bb \rightarrow yz$, among others. The inputs aa , ca and cb collide; the inputs a and c also collide.

Squeezeziness has been successfully used to estimate the existence of FEP in white-box (Clark & Hierons, 2012; Clark et al., 2019) and black-box (Ibias et al., 2019) scenarios. It represents the amount of information lost by a function. Thus, Squeezeziness for an FSM can be defined as the Squeezeziness of the function that represents this FSM. In order to properly compute it, it was necessary to define how inputs are chosen and outputs are returned. A probabilistic view, where a random variable is associated with each set of relevant inputs/outputs, can be considered. Specifically, a random variable can be associated with the set of inputs/outputs of a certain length (that is, there are different random variables associated with $I^1, I^2, \dots; O^1, O^2, \dots$). Since $\text{dom}_{M,k}$ includes the inputs of length equal to k that M can perform and $\text{image}_{M,k}$ includes the outputs of length equal to k that M can produce after receiving an element of $\text{dom}_{M,k}$, random variables ranging over each set are defined as $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$, respectively.

Definition 4. Let S be a set and ξ_S be a random variable over S . We denote by σ_{ξ_S} the probability distribution induced by ξ_S .

Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider two random variables $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$ ranging, respectively, over the domain and image of $f_{M,k}$. The Squeezeziness of M at length k is defined as

$$Sq_k(M) = \mathcal{H}(\xi_{\text{dom}_{M,k}}) - \mathcal{H}(\xi_{\text{image}_{M,k}})$$

where $\mathcal{H}(\xi_S)$ denotes the (Shannon's) entropy of the random variable ξ_S that ranges over the set S , which is defined as

$$\mathcal{H}(\xi_S) = - \sum_{s \in S} \sigma_{\xi_S}(s) \cdot \log_2(\sigma_{\xi_S}(s))$$

There is an important remark concerning random variables associated with inputs and outputs: given an FSM $M, k > 0$ and a random variable $\xi_{\text{dom}_{M,k}}$, we have that the probability distribution of the random variable $\xi_{\text{image}_{M,k}}$ is completely determined. This is because for each element $t \in \text{image}_{M,k}$ we have that

$$\sigma_{\xi_{\text{image}_{M,k}}}(t) = \sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s)$$

Therefore, the formulation of Squeezeziness is

$$Sq_k(M) = - \sum_{t \in \text{image}_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s) \right) \cdot \mathcal{R}_M(t)$$

where the term $\mathcal{R}_M(t)$ is equal to

$$\sum_{s \in f_M^{-1}(t)} \frac{\sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\sigma_{\xi_{\text{dom}_{M,k}}}(f_M^{-1}(t))} \cdot \log_2 \left(\frac{\sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\sigma_{\xi_{\text{dom}_{M,k}}}(f_M^{-1}(t))} \right)$$

Example 3. Consider again the FSM given in Fig. 3. If we assume that inputs are uniformly distributed (later we will explain why this is a reasonable assumption), then we have that

$$Sq_1(M) = 3 \cdot \left(-\frac{1}{3} \log_2 \left(\frac{1}{3} \right) \right) - \left(-\frac{1}{3} \log_2 \left(\frac{1}{3} \right) - \frac{2}{3} \log_2 \left(\frac{2}{3} \right) \right) \approx 1.585 - 0.918 = 0.667$$

Using similar computations we have that

$$Sq_2(M) \approx 2.585 - 1.459 = 1.126$$

2.3. Rényi's-based Squeezeziness in a black-box setting

The original work on Squeezeziness used Shannon's entropy, but there exists a general definition of entropy, depending on a parameter α , called Rényi's entropy (Rényi, 1961).

Definition 5. Let S be a set and ξ_S be a random variable over S . Let $\alpha \in \mathbf{R}_+ \setminus \{1\}$. The Rényi's entropy of the random variable ξ_S with respect to α , denoted by $\mathcal{H}_\alpha(\xi_S)$, is defined as:

$$\mathcal{H}_\alpha(\xi_S) = \frac{1}{1-\alpha} \cdot \log_2 \left(\sum_{s \in S} \sigma_{\xi_S}(s)^\alpha \right)$$

We have that when α tends to 1, Rényi's entropy becomes Shannon's entropy (Rényi, 1961), that is,

$$\lim_{\alpha \rightarrow 1} \mathcal{H}_\alpha(\xi_S) = \mathcal{H}(\xi_S) = - \sum_{s \in S} \sigma_{\xi_S}(s) \cdot \log_2(\sigma_{\xi_S}(s))$$

Squeezeziness of an FSM using Rényi's entropy has been recently defined (Ibias & Núñez, 2020) in the same way as (Shannon's) Squeezeziness was defined in Definition 4.

Definition 6. Let $M = (Q, q_{in}, I, O, T)$ be an FSM and $k > 0$. Let us consider two random variables $\xi_{\text{dom}_{M,k}}$ and $\xi_{\text{image}_{M,k}}$ ranging, respectively, over the domain and image of $f_{M,k}$. Let $\alpha \in \mathbf{R}_+ \setminus \{1\}$. Rényi's Squeezeziness of M at length k with respect to α is defined as

$$Sq_{\alpha,k}(M) = \mathcal{H}_\alpha(\xi_{\text{dom}_{M,k}}) - \mathcal{H}_\alpha(\xi_{\text{image}_{M,k}})$$

In the Appendix I of the paper we provide an equivalent, but simpler to compute, formulation of Rényi's Squeezeziness (see Lemma 1).

The previously defined notion of Squeezeziness is parameterised by the distribution over the inputs of the function (that is, over the input sequences that the FSM can perform). If we know the actual distribution, then we can use this. If we do not know the distribution, then there is a need to choose one and we now discuss two approaches to do this.

2.3.1. Maximum entropy principle

We can select the distribution that maximises entropy. If there are no further restrictions, maximum entropy is obtained with a uniform distribution (Acharya, Orlitsky, Suresh, & Tyagi, 2017; Cover & Thomas, 1991). Then, under this distribution, the weight of a single element of $\text{dom}_{M,k}$ is $\frac{1}{|\text{dom}_{M,k}|}$ and the weight of the inverse image of an output $t \in \text{image}_{M,k}$ is equal to $\frac{|f_M^{-1}(t)|}{|\text{dom}_{M,k}|}$.

Under these assumptions, and after some algebraic manipulations, the formula for Rényi's Squeezeziness becomes:

$$Sq_{\alpha,k}(M) = \frac{1}{1-\alpha} \cdot \log_2 \left(\frac{|\text{dom}_{M,k}|}{\sum_{t \in \text{image}_{M,k}} (|f_M^{-1}(t)|)^\alpha} \right)$$

As usual, we have two special cases: α tending to 1 and tending to ∞ . If α tends to 1, then we are using Shannon's entropy and we have the following simplified formulation (Ibias et al., 2019):

$$Sq_{1,k}(M) = \frac{1}{|\text{dom}_{M,k}|} \cdot \sum_{t \in \text{image}_{M,k}} |f_M^{-1}(t)| \cdot \log_2(|f_M^{-1}(t)|)$$

If α tends to ∞ , then we are using min-entropy and, after some algebraic manipulations, we obtain the following formulation:

$$Sq_{\infty,k}(M) = \log_2 \left(\max_{t \in \text{image}_{M,k}} |f_M^{-1}(t)| \right)$$

2.3.2. Maximum loss of information

Another option is to consider the worst case scenario, that is, the scenario where the probability distribution induces the maximum loss of information. In order to maximise the loss of information, we need to maximise Squeeziness. In this case, the probability distribution is the one that is uniformly distributed in the largest inverse image of an element of the outputs and zero elsewhere (Clark & Hierons, 2012). Formally, consider $t' \in \text{image}_{M,k}$ such that for all $t \in \text{image}_{M,k}$ we have that $|f_M^{-1}(t')| \geq |f_M^{-1}(t)|$. Then,

$$\sigma_{\text{image}_{M,k}}(s) = \begin{cases} \frac{1}{|f_M^{-1}(t')|} & \text{if } s \in f_M^{-1}(t') \\ 0 & \text{otherwise} \end{cases}$$

Using this probability distribution, Rényi's Squeeziness becomes:

$$Sq_{\alpha,k}(M) = \log_2(|f_M^{-1}(t')|)$$

In this case, unlike the previous ones, Squeeziness does not depend on the value of α . In particular, the two special cases (α tending to 1 and α tending to ∞) have the same formulation.

2.4. Probability of collisions

FEP happens when the expected and faulty inputs, received from another component, produce the same output. Therefore, a collision (see Definition 3) is an indication of FEP and it is useful to compute the probability of having collisions in the FSM under study. This probability is given by PColl (Clark & Hierons, 2012).

Definition 7. Let M be an FSM and $k > 0$. Let $\text{image}_{M,k} = \{t_1, \dots, t_n\}$ and for all $1 \leq i \leq n$ let $I_i = f_{M,k}^{-1}(t_i)$ and $m_i = |f_{M,k}^{-1}(t_i)|$. We have that $d = \sum_{i=1}^n m_i$ is the size of the input space.

Given a uniform distribution over the inputs, the probability of s and s' both being in the same set I_i is equal to $p_i = \frac{m_i \cdot (m_i - 1)}{d \cdot (d - 1)}$. We have that the probability of having a collision in M for sequences of length k , denoted by $\text{PColl}_k(M)$, is given by

$$\text{PColl}_k(M) = \sum_{i=1}^n \frac{m_i \cdot (m_i - 1)}{d \cdot (d - 1)}$$

With regard to this definition, a topic that has been already addressed is the potential to use $\text{PColl}_k(M)$ instead of Squeeziness. The problem with using $\text{PColl}_k(M)$ is that it is hard to compute. While this also applies to Squeeziness, the latter has the advantage of being an information theoretic measure. As a result, we can use Information Theory to either estimate or bound measures (Boreale & Paolini, 2014; Chothia, Kawamoto, & Novakovic, 2014), what will suffice for our final goal.

2.5. Artificial Neural Networks

We review the main concepts around Artificial Neural Networks (ANN). They try to mimic the brain behaviour. We use them to infer the value of α that more appropriately assesses the likelihood of the presence of cases of FEP. There are two types of ANNs: *classification* and *regression*. The former ones classify the input in one of many predefined classes

while the latter ones return a real number that represents the value associated to the input in a one dimensional space.

An ANN is composed of different layers, where each layer has an associated activation function and a set of neurons. The layers of an ANN are commonly grouped in three types:

- Input layer: the layer that receives the input. It has as many neurons as the dimension of the input.
- Hidden layers: the layers inside the ANN that are not accessible from outside. They have various sizes and activation functions.
- Output layer: the last layer of the ANN. It has as many neurons as classes for classification ANNs, or one neuron in the case of regression ANNs.

A neuron is a simple agent that performs the following steps:

- Compute the weighted sum of the outputs of the previous layer. For each neuron of the previous layer, the agent has an associated weight that multiplies the output value of that neuron. Then, the agent sums all the weighted output values.
- Apply the activation function. The agent applies the activation function associated to its layer to the result of the previous step. These activation functions are non-linear.

Activation functions are non-linear because if they were linear, then the ANN would be equivalent to a single neuron with the proper weights. There are many activation functions (the interested reader is referred to classical material on ANNs (Goodfellow, Bengio, & Courville, 2016; Jain, Mao, & Mohiuddin, 1996)). In this paper we consider the linear activation function for the output layer (because we use a regression ANN) and the leaky ReLU activation function, that we will explain later, for the hidden layers. Note that using a linear activation function is equivalent to not using any activation function at all.

Finally, ANNs use the *back-propagation algorithm* (Rumelhart, Hinton, & Williams, 1986) to learn the values of the weights of each neuron. This learning method needs a set of examples given as input/output pairs (do not confuse with our inputs and outputs), which we call the *training set*. This algorithm performs, for each element of the training set, the following steps:

- Compute the result of the ANN for the given input.
- Compute the loss of the ANN result with respect to the expected result. The loss function usually is the mean squared error (squared L2 norm) between both values.
- Compute the gradient of the loss function with respect to each weight by the chain rule, trying to minimise the loss. This computation is done from the last layer to the previous ones, in an incremental way.

Using this method the ANN updates its weights, efficiently learning the hidden function that associates the inputs to the outputs of each element of the training set.

After the learning, a common practice is to test how well the ANN performs on previously unseen examples. In order to perform this check, it is necessary to have another set of examples which we call the test set. With this test set, the *accuracy* of the ANN is computed. For classification ANNs the accuracy is defined as the ratio between the number of elements of the test set that have been well classified and the cardinal of the tests set. For regression ANNs there is no such concept of accuracy; the performance of the ANN is given by the mean loss of the examples of the test set. This concept of accuracy/mean loss can be extended to the performance of the ANN on the training set, although in this case it only measures how well the ANN learned the elements of the training set.

3. Methodology

In this section we review the main concepts behind the construction

of our tool `SqSelect`.² In particular, we will explain how the ANN driving the behaviour of `SqSelect` was designed, trained and tested.

Squeeziness is useful to assess the likelihood of FEP in a system. Besides, PColl (see Definition 7) is a reference measure of FEP in a system. Therefore, for a given system, we would like to know the *best* value of α to compute Rényi's Squeeziness, that is, the value of α that better assess the likelihood of having cases of FEP. In other words, the value whose associated Rényi's Squeeziness has a higher correlation with PColl. In order to compute these correlations (one for each α) we consider *families of systems*, that is, we grouped the FSMs according to their characteristics. Then, we compute the different Rényi's Squeeziness values (for different values of α) for all the systems of a family and measure the correlation, for each α , between those values of Rényi's Squeeziness and the values of PColl.

There are two standard options to compute correlations: *Pearson* correlation, which focuses on the proportionality between the variables, and *Spearman* correlation, which focuses on the monotony between the variables. We will focus on the first option and during the rest of the paper when we simply say correlation we are referring to Pearson correlation (we will sometimes use Spearman correlation and we will clearly identify it). A discussion about this decision and its implications is given in Section 5.

If we are able to compute the best value of α for a family of systems, then each time that our tool receives a system belonging to the family we only need to compute its Rényi's Squeeziness using this α . However, the number of different system families is infinite, even if we use a few parameters to define each family. Therefore, it is impossible to compute this *best notion* for each potential family. In order to circumvent this restriction, `SqSelect` implements an approximation method to compute a *good enough* value of α for any system provided to the tool, even if its family has not been analysed before. `SqSelect` uses an ANN as approximation method. We decided to rely on this machine learning technique because our preliminary experiments showed that the *best* values of α , for different families, have low correlation to the parameters that codify those families. Therefore, we need an approximation technique that could deal with this *lack of correlation*. In Fig. 4 we give this distribution for the entries of the dataset that we used to train our ANN. Each point corresponds to a different family of systems and depth: the points to the right of each value in the x -axis represent different search depths. We can observe how, for the same parameter, the distribution of the points along the possible values of α is almost uniform for each of its values, showing no correlation between α and the considered parameter.

After training the ANN using a huge number of cases, we can use it to approximate the value of α that will be better for a given system. In order to do so, we need this system to be modelled as an FSM, from where we can obtain their characteristic parameters. These parameters will be the inputs of the ANN underlying the behaviour of `SqSelect`.

Finally, knowing the value of α whose Rényi's Squeeziness will approximate better the likelihood of having a case of FEP in a system, the only remaining step is to compute the actual Squeeziness of the system. In order to do so we implement the methodology defined in our previous work (Ibiás & Núñez, 2020). First of all, we assume that the probability distribution of the inputs follows a uniform distribution. With this choice we know that we are maximising entropy. In addition, we can use the simplified formulation given in Lemma 1. Second, we need to fix a *search depth* k . This value will indicate the maximum length of the input sequences that we will be used for testing (and therefore, the expected output sequences). Once we have set the search depth, we must obtain two values in order to compute Rényi's Squeeziness:

- Number of inputs: the number of input sequences of length k of the FSM. This number is stored in a variable *inputs*.

- Inverse image of the outputs: the number of input sequences that lead to the same output sequence. These values are stored in a dictionary *mapOtoI* that keeps, for each output sequence of length k , the number of inputs that lead to it.

In order to compute these values, we have to collect the traces of the FSM until the desired depth is reached. If a trace reaches the desired depth (or a deadlock), then the *inputs* count is increased by 1 and the output obtained is searched in the *mapOtoI* dictionary to increase by 1 the count of inputs that lead to that output. Once we processed all these values, we only need to apply the formula given in Lemma 1.

3.1. The ANN underlying `SqSelect`

In order to infer the α value that will return the better approximation of the likelihood of the presence of cases of FEP in the system, we have developed an ANN using PyTorch (Paszke et al., 2019). We may consider that this ANN is the *core* of `SqSelect`. During the rest of this section we review the different phases involved in the creation of the ANN.

3.1.1. The dataset

First of all, we need a dataset to train the ANN. In order to obtain this dataset, we developed a program such that given a family of FSMs and a search depth, it computes the value of α such that Rényi's Squeeziness values have a higher correlation with PColl values. We execute this program for different families and different depths. For each family and depth we added a row to the dataset with the family parameters, the depth and the best value of α .

In order to get examples of each family, we developed a java script that generates, using the parameters of the family, a given number of FSMs of each of them. This script uses the *automatalib* (Isberner, Howar, & Steffen, 2015) library to represent and manipulate FSMs and saves the newly randomly generated FSMs in dot files. With this script, we generated 100 examples of each family. The different families were defined by combining the following parameter values:

- Number of states: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100.
- Maximum number of transitions: 5, 10, 15, 20.
- Minimum number of transitions: 1, 5, 10.
- Input alphabet size: 10, 20, 30, 40, 50.
- Output alphabet size: 10, 20, 30, 40, 50.

Let us remind that, as we said before, our experiments showed that there is no correlation between each single parameter and the computed best α . This is the reason why we *combine* them into an ANN.

3.1.2. The structure of the ANN

Our ANN implemented a regression ANN because we want to obtain an approximation of the best α without restraining the results to some predefined classes.

Since we are dealing with a problem of *dimensionality*, versus *complexity*, our ANN is quite simple while being very effective. It has 4 layers:

- An *input layer* of size 6.
- A *hidden layer* of size 12, with a leaky ReLU activation function.
- A *hidden layer* of size 3, with a leaky ReLU activation function.
- An *output layer* of size 1, with a linear activation function.

Our input layer has six neurons because it receives the five parameters that determine the family of FSMs and a sixth parameter: the *search depth*. We consider this additional parameter because our first experiments showed that the *best* α strongly depends on the established search depth.

We use leaky ReLU activation functions because they improve the actual ReLU activation function by avoiding the so called *dying ReLU*

² Our tool is freely available at <https://github.com/Colosu/SqSelect/>.

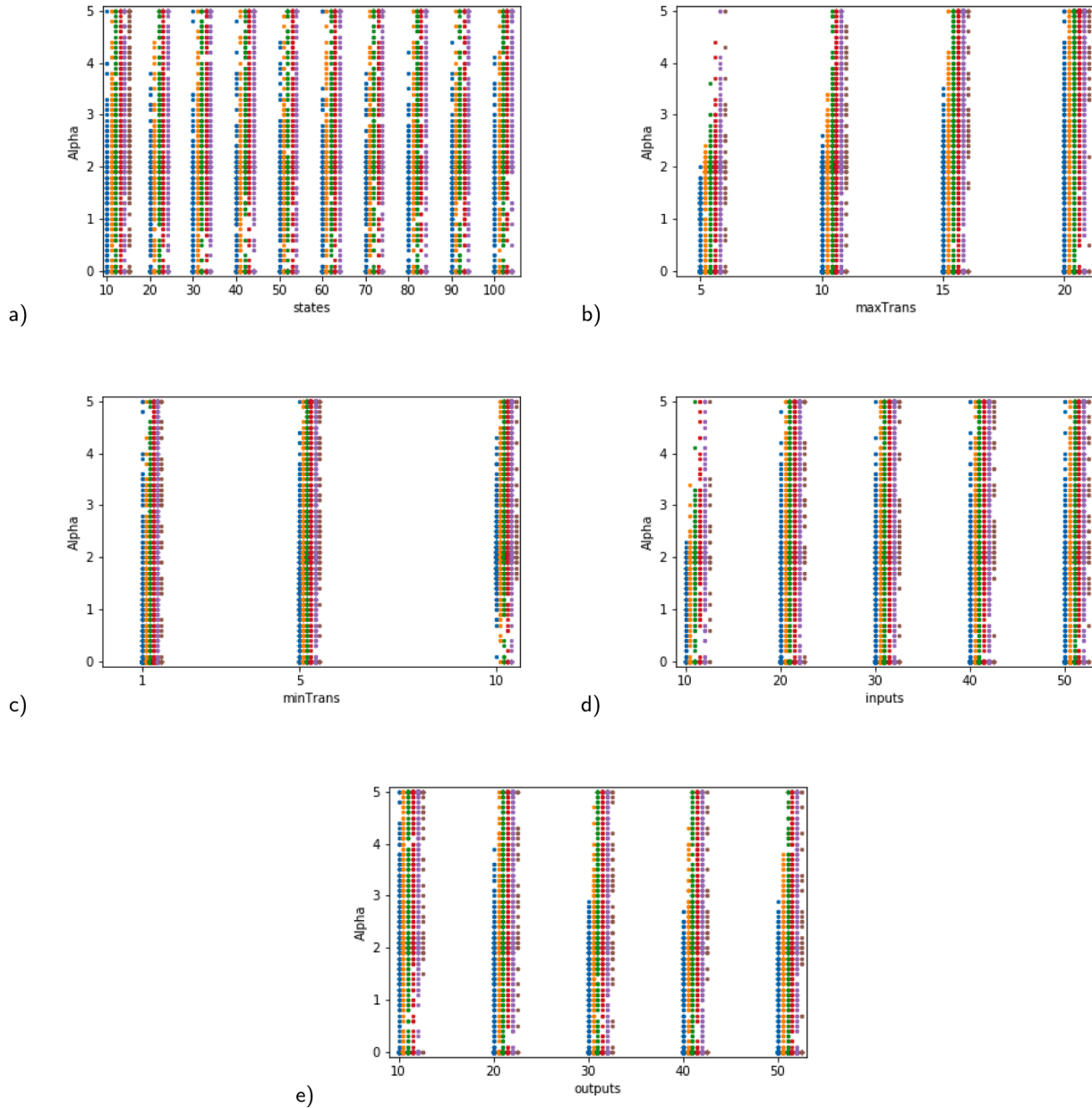


Fig. 4. Correspondence between parameters and α of elements of the dataset: states (a), maximum transitions (b), minimum transitions (c), input alphabet size (d) and output alphabet size (e).

problem and speeding up the learning (He, Zhang, Ren, & Sun, 2015). We can compare both activation functions in Fig. 5.

The *dying ReLU* problem happens because the ReLU activation function returns zero for all negative inputs. This evolves into some neurons *dying* because they always return a zero value and, therefore, they do not help to discriminate the input. Also, it is unlikely that the neuron will ever output again non-zero values. The leaky ReLU activation function solves this problem by returning a small negative value, instead of zero, what allows the neuron to add something to the discrimination of the input. In addition, it will be able in the future to output non-zero values.

3.1.3. The learning process

Having the ANN and the dataset, we normalised the data in the dataset, we shuffled the entries of the dataset and divided it into two sets: a set containing 70% of the entries, for training, and a set with 30% of the entries, for testing. Following this procedure we ensure that in the

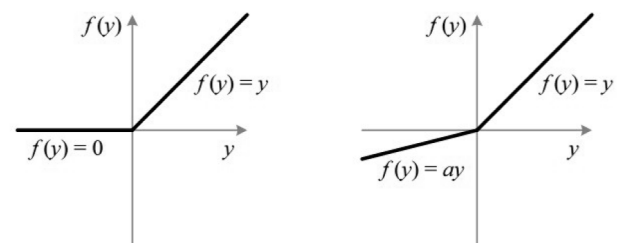


Fig. 5. ReLU vs leaky ReLU.

test set there is a representative of each FSM class that conforms the dataset, and of each search deep. This will ensure that we can assess the generalisation capability of our ANN with the test set. We set the loss function to the mean square error and the optimisation algorithm to the standard gradient descend with learning rate 0.01. With this setting, we

trained our ANN to approximate the α values with the training set, obtaining a mean loss of 1.280935 (1.333986 for Spearman). This loss is small enough to claim that our ANN learned really well the training dataset.

We used the test set to assess how good the training has been. In this case, we obtained a mean loss of 1.326940 (1.368981 for Spearman).

4. The SqSelect tool

In this section we present our tool to facilitate the assessment of the FEP of a system by computing relevant values associated with the Squeeziness of the system. SqSelect has four different modes, depending on what input parameters are available and what results are requested. These modes are:

- Alpha mode. The tool receives a set of parameters (see below) profiling the system of interest, and the search depth, and returns the α value providing a Squeeziness that is closer to the true likelihood of the presence of cases of FEP.
- Alpha file mode. The tool receives a search depth and a system and computes the corresponding α value.
- Squeeziness file mode. SqSelect receives a search depth and a system and computes the best α value and reports the likelihood of the presence of cases of FEP in the form of a Squeeziness value.
- Squeeziness range mode. The tool receives a search depth and a system and computes the Squeeziness of the system for different values of α . Then, it gives back a graph with the likelihood of the presence of cases of FEP in the system associated to each value of α .

The tool also provides the best value of α for smaller search depths in alpha and alpha file modes. In Squeeziness file mode, SqSelect provides Rényi's Squeeziness for smaller search depths. These values might be useful if the tester, in view of additional information, decides to reduce the search depth associated with testing.

Note that in the first two modes we only compute the value of α . If the tester wants to obtain the value of Rényi's Squeeziness for that α , then it is enough to use the third mode. The rationale is that computing Squeeziness, even for medium-size search depths, needs an important amount of computing time. If a tester either only needs the value of α or desires to use an alternative measure (for this α), then he does not need to consume those computing resources.

The set of parameters used in the *Alpha mode* is given by:

- Number of states of the FSM.
- Maximum number of transitions outgoing from a state of the FSM.
- Minimum number of transitions outgoing from a state of the FSM.
- Input alphabet size of the FSM.
- Output alphabet size of the FSM.

In the second and third modes, SqSelect computes these values from the actual system.

The system received in the *Alpha file mode*, the *Squeeziness file mode* and the *Squeeziness range mode* should be a dot file (the standard for representing graphs in plain text), where the first node is marked with the red colour (Isberner et al., 2015).

In each mode (except in the Squeeziness range mode) there is an option called *Use Spearman correlation* that tells the tool to use the best value of α according to the Spearman correlation instead of the default Pearson correlation.

Finally, all modes have an option to save the generated plot to a file, in case it is needed.

Next, we analyse the main characteristics of each mode.

4.1. Alpha mode

SqSelect receives a set of parameters corresponding to the system

and the search depth. Then, SqSelect calls the ANN that computes the best α for assessing the likelihood of the presence of cases of FEP in the system. The tool shows this value in its results panel. An example of execution is shown in Fig. 6.

This mode is intended for users that do not have the description of their systems in the format used by our tool, so that they cannot use other modes. Therefore, with this mode they can still get a generic best α for their system. Afterwards, they can compute Rényi's Squeeziness, for that α , of their system using their own algorithm.

4.2. Alpha file mode

SqSelect receives an actual FSM and the search depth. Then, SqSelect computes the same parameters as the ones that are needed for the Alpha mode and works as in that mode. An example of execution is shown in Fig. 7.

This mode is intended for users that, although having the system in a valid format, do not want our tool to compute Rényi's Squeeziness. This is the case when they have a faster, cheaper or in general better implementation of the formula from Lemma 1. SqSelect will obtain the best α for their system and then the users will compute its Rényi's Squeeziness using either another tool or their own algorithm.

4.3. Squeeziness file mode

SqSelect receives a system and the search depth. Then, SqSelect computes the same parameters as the ones that are needed for the Alpha mode. After the best α is obtained, the tool proceeds to compute Rényi's Squeeziness for the selected search depth and this α . As an additional information, SqSelect computes values of α for smaller depths and the corresponding Rényi's Squeeziness. All the values are shown as a graph, while the value corresponding to the selected search depth appears in the results panel. An example of execution is shown in Fig. 8.

This is the main mode of SqSelect. We expect that most users will have their system in the valid format, they will give it to the tool and they will get the value of Rényi's Squeeziness that better assess the likelihood of having cases of FEP for that system. If the users develop different versions of their system, then they can compare the Rényi's Squeeziness values obtained for each version and select the one having a lower Rényi's Squeeziness, because it is the one with smaller likelihood to suffer cases of FEP. If the users only have one version of the system, and similar to the previous mode, they still can use the results displayed in the graph to decide the search depth that they want to use for testing.

4.4. Squeeziness range mode

SqSelect receives a system and the search depth. In contrast to the previous modes, SqSelect does not compute the best α . Instead, it computes all the values of Rényi's Squeeziness for a selected number of values $\alpha \in [0, 5]$. This interval is predefined and was chosen because in our experiments we never obtained a best value of α greater than 5 and the selected values are representative of the best α values for the families of systems appearing in the dataset. The tool shows these values in its graph panel. An example of execution is shown in Fig. 9.

This mode is intended for users that do not want to know only Rényi's Squeeziness for the best α , but that want to know it for a huge range of values of α .

5. Discussion

In this section we discuss some decisions we took along the development of SqSelect.

The first decision was which correlation to use as a reference for the selection of α . Pearson correlation focuses on the linearity of the points that are being correlated. Therefore, higher (in absolute value) correlations imply that the points maintain a proportionality while correla-

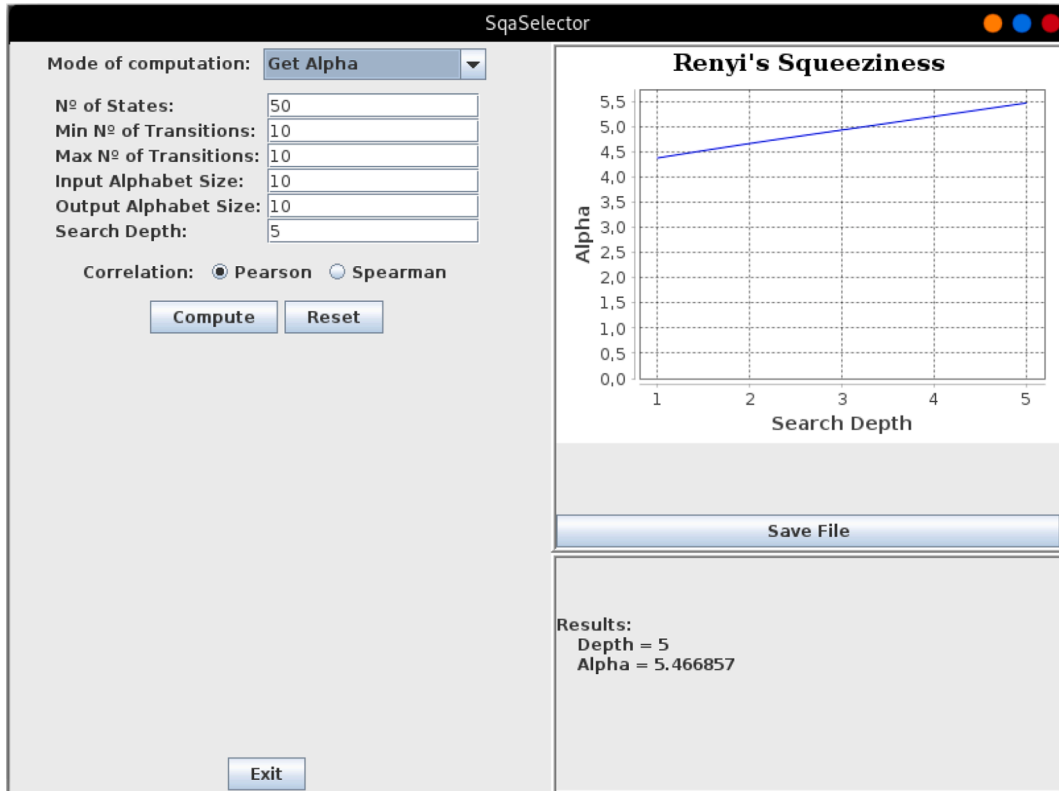


Fig. 6. Alpha mode example: Coffee machine case study.

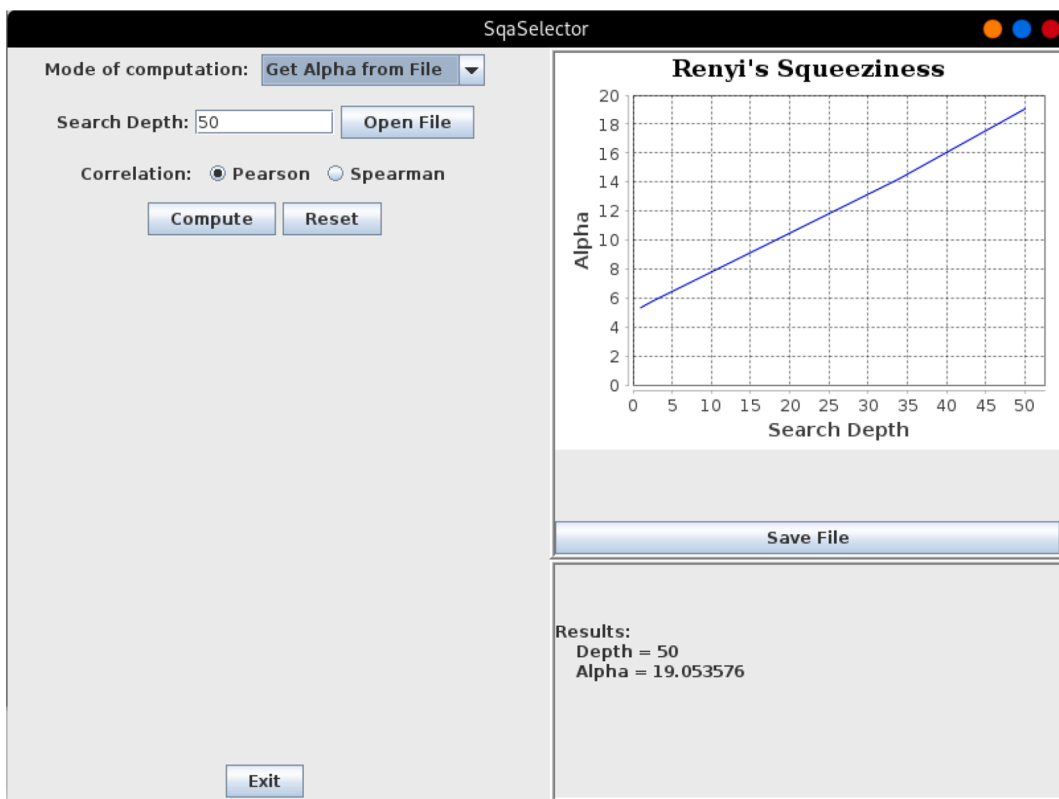


Fig. 7. Alpha file mode example: TCP case study.

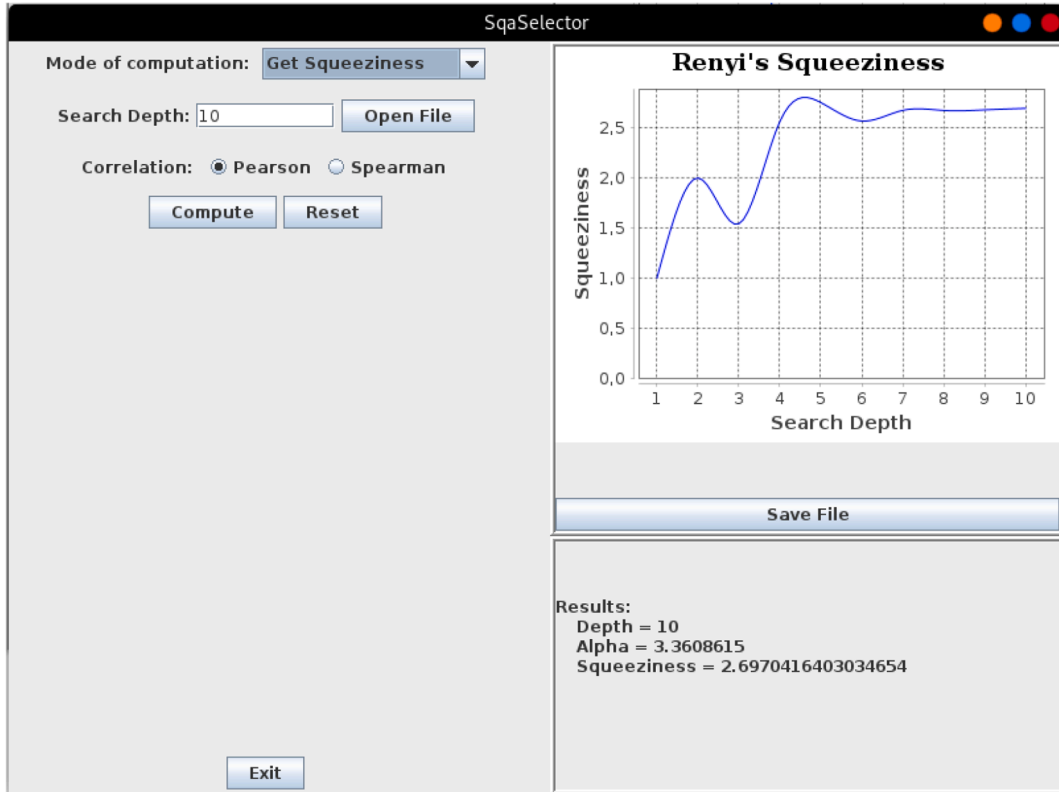


Fig. 8. Squeeziness file mode example: Logic circuit case study.

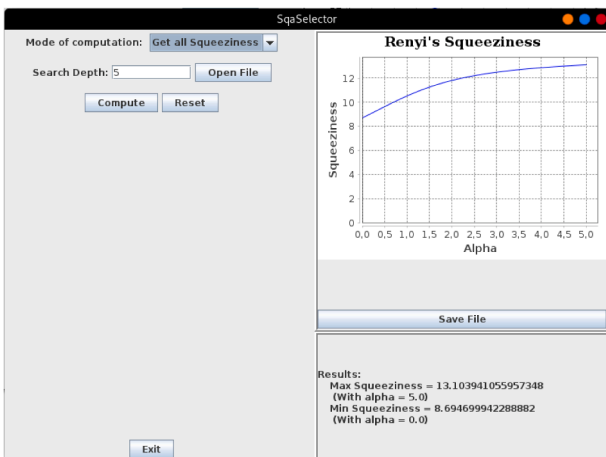


Fig. 9. Squeeziness range mode example: Bankard case study.

tions near 0 imply that the points are distributed more in a point cloud way. Meanwhile, Spearman correlation focuses on the monotony of the points that are being correlated. Therefore, higher (in absolute value) correlations imply that the points maintain monotony while correlations near 0 imply that many points break the monotony. The main goal of our research is to find values of α whose Rényi's Squeeziness has a higher similarity to the likelihood of the appearance of FEP. In order to have higher similarity, it is not enough to keep monotony; it is also necessary to have some kind of proportionality. Therefore, we think that it is better to stick to Pearson but if users prefer to use Spearman, then we allow them to choose it. In order to implement this duality, we obtained two datasets, one for the best values of α according to Pearson correlation and another one according to Spearman correlation. We used the same ANN structure with both datasets, obtaining one ANN that learned the

Pearson dataset and another one that learned the Spearman dataset. In any case, our experiments showed that the difference between the Pearson and Spearman correlations is minimal.

The second decision was which machine learning technique to use. Currently, there are a myriad of options, from a classical decision tree algorithm to complex deep learning algorithms or even reinforcement learning algorithms. However, not all these techniques would perform well in our scenario. For example, reinforcement learning techniques require that the problem can be expressed as a Markov Decision Process (MDP), what in fact is a $1\frac{1}{2}$ game (Chatterjee, de Alfaro, & Henzinger, 2004). However, our problem cannot be redefined (at least, in an easy and intuitive way) as an MDP. Decision trees and the linear regression techniques cannot be applied to our problem due to the small correlation between the elements of the dataset, that we explained before. Therefore, we were left with methods that were complex enough to manage the apparent non-correlation between the elements of the dataset. From this set of methods, we chose a simple one, because we wanted to avoid using approaches that were overpowered for the task at hand. That is why we decided to use artificial neural networks. However, we would like to justify with empirical evidence that this was the right choice.

During the development of SqaSelect we considered the use of a more complex (deep) neural network. However, our preliminary experiments did not show any improvement over the shallow ANN that we are using. Actually, we usually obtained cases of overfitting while experimenting with deep neural networks. Next we give the details of one of the (representative) settings that we used. We considered a deep neural network with six hidden layers and with the following neurons per layer: 6 neurons for the input layer, 100 for the first hidden layer, 75, 50, 25, 12 and 3 for the subsequent hidden layers, and 1 neuron for the output layer (remind we are training a regression ANN). With this deep ANN we obtained the following results for the Pearson dataset: *training loss* equal to 1.236317 and *test loss* equal to 1.372902. The results for the Spearman dataset were considerably worse: 2.396616 and 2.387157, respectively. Another example is a neural network with eight hidden

layers and with the following neurons per layer: 6 neurons for the input layer, 12 for the first hidden layer, 48, 128, 75, 50, 25, 12 and 3 for the subsequent hidden layers, and 1 neuron for the output layer. With this deep ANN we obtained the following results for the Pearson dataset: *training loss* equal to 1.284181 and *test loss* equal to 1.381884. The results for the Spearman dataset were also considerably worse: 2.400117 and 2.383044, respectively. Therefore, the results are slightly worse than the ones corresponding to our shallow ANN (in both Pearson and Spearman datasets and in both training and test sets). In addition, this kind of networks needs extra computation resources and this extra complexity does not pay back with an increase on performance. After this comparison we considered that our shallow neural network was a better option.

A second experiment allowed us to compare the results provided by our ANN and a multi-variable linear regression model (Kutner, Nachtsheim, Wasserman, & Neter, 2003). We trained a sklearn (Pedregosa et al., 2011) linear regression model, which automatically adapts to the multivariable case, with our datasets. Then, we computed model loss using the same method as we used for our ANN network, the mean square error, and obtained the following results for the Pearson dataset: *training loss* equal to 1.523924 and *test loss* equal to 1.575762. The results for the Spearman dataset were very similar: 1.547626 and 1.527375, respectively. Therefore, the results show that our ANN outperforms the LNR that we implemented. We also trained 4 different Ridge regression models and 4 different Lasso regression models. In all the cases we computed the model loss as the mean square error between the predictions and the real values. The results corresponding to all these experiments are displayed in Fig. 10.

As a further exploration of these alternatives, we computed the confidence boundaries for each one, using Fixed-Width bands (Macskassy & Provost, 2004) (see Figs. 11 and 12). We decided to compute these confidence boundaries for the first 100 elements of the test dataset because, although including more elements would affect the shape of the confidence bands, we want to address the differences between different models instead of generate good bands for a given model (Macskassy & Provost, 2004). The red line presents the real value of α ; the blue line presents the value of α obtained using the regression model; and the light blue zone presents the confidence interval in which we can assure, from the answer of our regression model, that the true value of α will be with a 95% of confidence.

These confidence boundaries show that our shallow ANN gets fitter confidence boundaries and that the obtained values of α fit better the real values. Specifically, there are some alternatives like Ridge Regression (alpha = 5) and Lasso Regression (alpha = 0.01) that are considerably worse, but the other alternatives are not better than ours.

Regression Model	Pearson Training Loss	Pearson Test Loss	Spearman Training Loss	Spearman Test Loss
Linear Regression	1.523924	1.575762	1.547626	1.527375
Ridge Regression (alpha = 5)	2.153214	2.182028	2.110532	2.103499
Ridge Regression (alpha = 1)	1.734578	1.775401	1.737016	1.728423
Ridge Regression (alpha = 0.01)	1.524000	1.575669	1.547695	1.527758
Ridge Regression (alpha = 0.00001)	1.523924	1.575762	1.547626	1.527375
Lasso Regression (alpha = 0.01)	2.469327	2.489932	2.392343	2.383564
Lasso Regression (alpha = 0.001)	1.556424	1.609071	1.580521	1.564440
Lasso Regression (alpha = 0.00001)	1.523928	1.575770	1.547630	1.527369
Lasso Regression (alpha = 1e-10)	1.523924	1.575762	1.547626	1.527375
Shallow ANN (2 hidden layers)	1.280935	1.326940	1.333986	1.368981
Deep ANN (6 hidden layers)	1.236317	1.372902	2.396616	2.387157
Deep ANN (8 hidden layers)	1.284181	1.381884	2.400117	2.383044

Fig. 10. Comparison of regression models (sorted by model complexity).

6. Case studies

In this section we present some case studies where we use our tool in order to assess the likelihood of FEP in some FSMs. In order to ensure that these FSMs are representative, we used a recently collected benchmark (Neider, Smetsers, Vaandrager, & Kuppens, 2019) of FSMs modelling real-world systems. We explore how to apply our tool to 4 FSMs: the classical coffee machine, the TCP protocol, a logic circuit, and a bankcard system. We hope that this section will be a useful reference for anyone that wants to use SqSelect.

6.1. The coffee machine

This system represents the classical coffee machine that has been extensively used as an example of FSM. We use this system to show how to use the *alpha mode*. In this case, we need some assumptions: we only want to get the best α and we do not have the FSM modelled in an appropriate format for the tool. Therefore, we have to use the system to obtain the different parameters that SqSelect uses for computing the best α . We obtained the following parameters:

- Number of states: 6.
- Maximum number of transitions: 4.
- Minimum number of transitions: 4.
- Input alphabet size: 4.
- Output alphabet size: 3.

Finally, setting that we want to explore the FSM up to a depth of 5, we have to introduce these values in the tool. We obtained a value of α equal to 0.106380135, as it is displayed in Fig. 6.

6.2. The TCP protocol

This system represents the widely known TCP protocol, massively used in the communications through internet. Specifically, it represents the TCP protocol from the perspective of the server, what is an FSM with 55 states. We use this system to show how to use the *Alpha file mode*. In this case, we assume that we only want to get the best α and that we have the protocol modelled as an FSM in dot format. We also set that we want to explore the FSM up to a depth of 50.

Therefore, we load the file in SqSelect and set the search depth to 50. The tool computes a value of α equal to 0.346357, as it can be seen in Fig. 7.

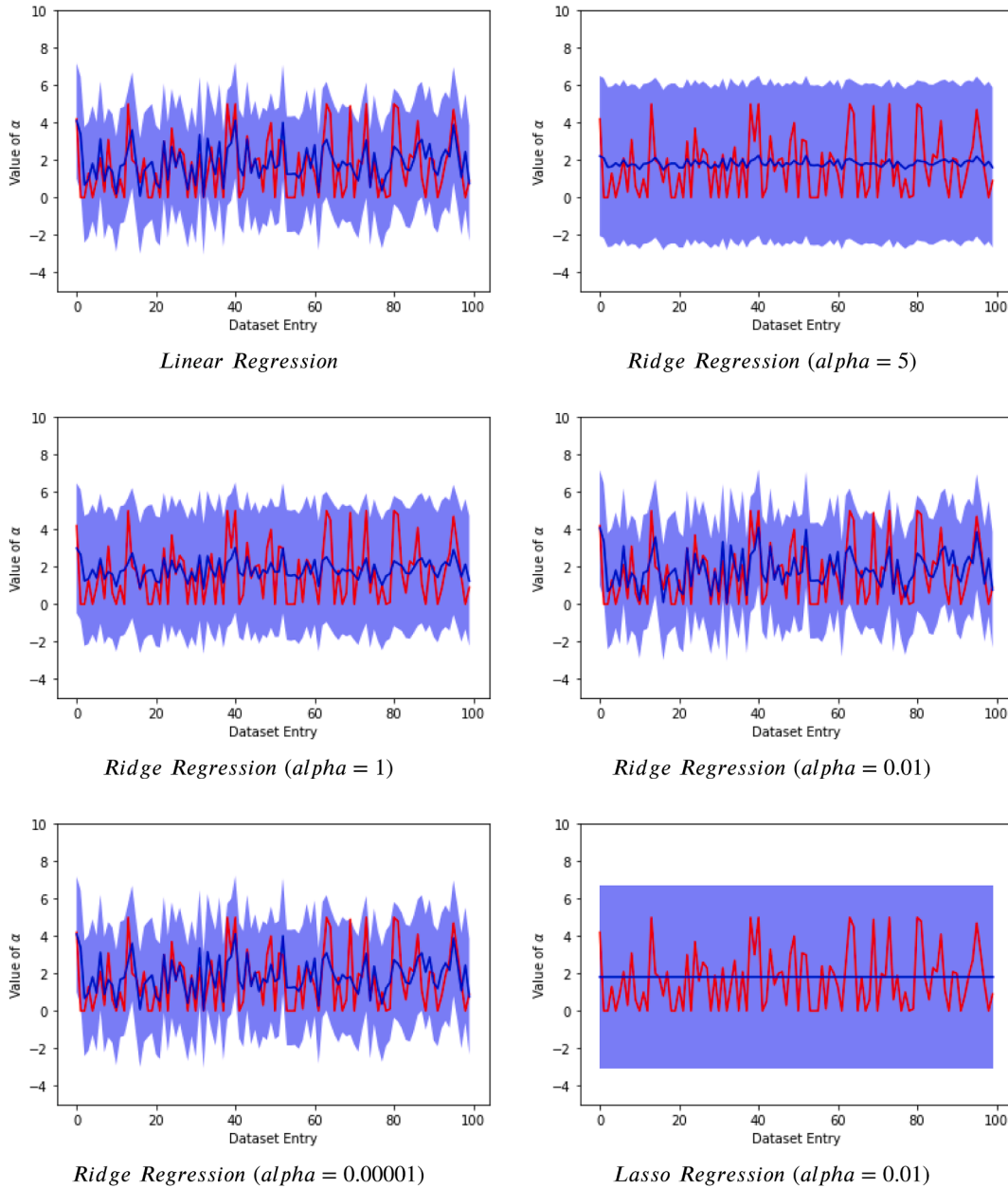


Fig. 11. Confidence boundaries plots (part 1).

6.3. A logic circuit

This system represents a dk27 logic circuit (Yang, 1991). We use this system to show how to use the *Squeeziness file mode*. We assume that we have the system in the appropriate format and that we set a search depth equal to 10. This system has 7 states and 14 transitions and SqSelect returns the results displayed in Fig. 8.

From these results, we get that the best α for a search depth of 10 was $\alpha = 0.045744844$ and it gives a Rényi's Squeeziness value equal to 2.4884. However, if we carefully analyse the plot, then we observe that using inputs of length 10 is not the optimal option concerning a trade-off between amount of testing and likelihood of suffering from FEP. As seen in the plot, inputs of length greater than 6 will suffer more FEP than inputs of the said length. Inputs of lengths 4 and 5 will also suffer more FEP than inputs of length 6. In contrast, inputs of length smaller than 4 will be less prone to have FEP than inputs of length 6. However, these last inputs will hardly explore more than half of the system (at least, in this case). Therefore, if resources are scarce and the tester might have to

reconsider whether to explore the SUT up to a smaller depth, then a better option is to test using inputs of length 6.

6.4. The bankcard

This system represents the behaviour of an ATM when authenticating a debit or credit card. We use this system to show how to use the *Squeeziness range mode*. In order to do so, we assume that we want to know how Squeeziness evolves for different alphas, so that we can have a global vision of how likely is that our system is affected by cases of FEP without having to stick to only one α . Since this system has only 7 states, we set the search depth to 5. We obtained the result displayed in Fig. 9.

7. Conclusions

We have developed the tool SqSelect that can be used to assess the likelihood of the presence of FEP in a system. It relies in the concept of Squeeziness, which has been previously proven to be useful for this task.

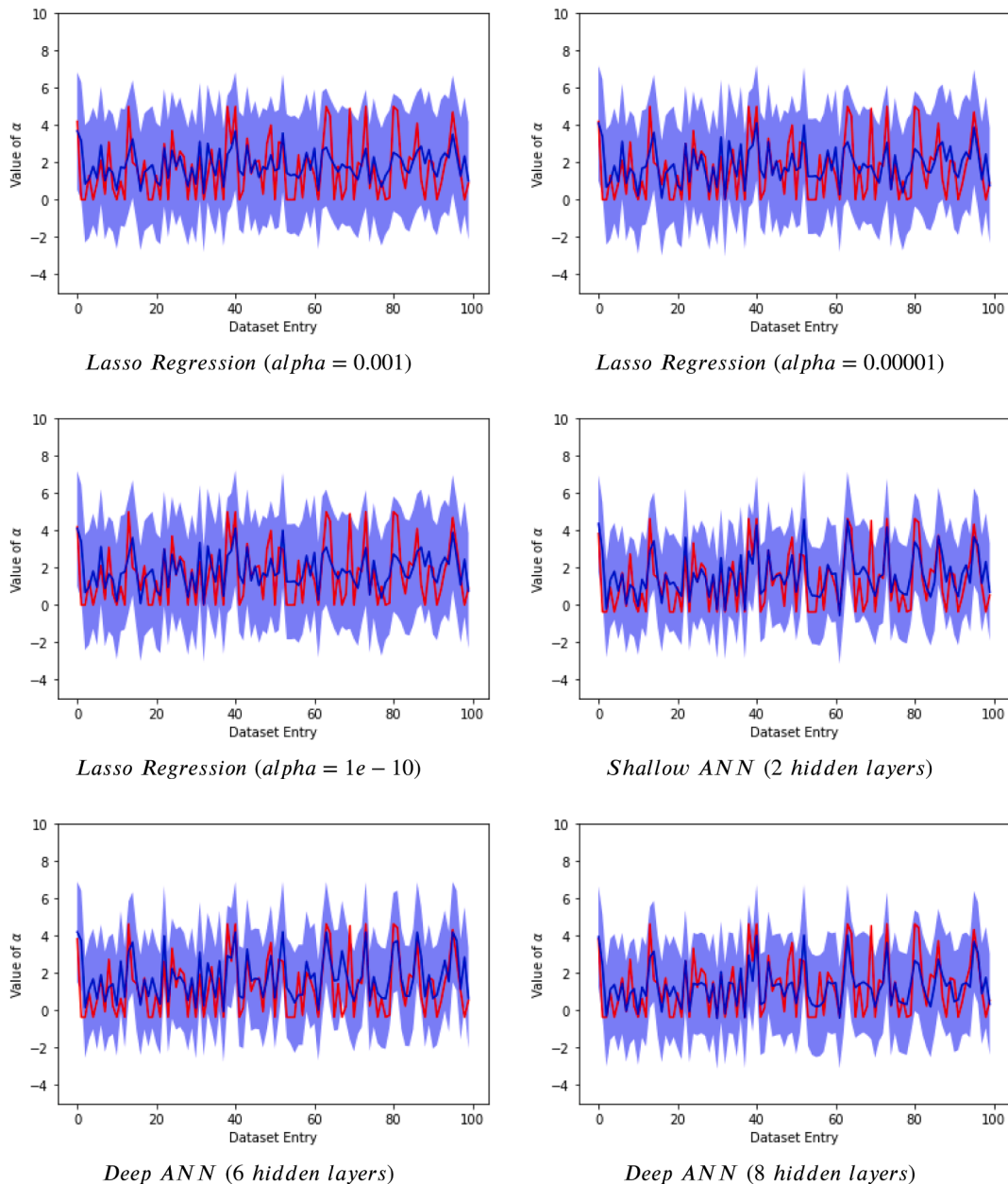


Fig. 12. Confidence boundaries plots (part 2).

Moreover, `SqSelect` implements an extended version of Squeeziness based on Rényi's notion of entropy, instead of the classical Shannon's entropy notion. This general version relies on a parameter, named α , to perform the computation. In previous work we empirically showed that the election of the α parameter is not an easy task. This is due to the fact that the reliability of the assessment of the likelihood of the presence of FEP strongly varies among different values of α . Therefore, we have to develop a method to obtain the best value of α to assess the likelihood of the presence of FEP in the system. In `SqSelect` we used this method to get data for training an artificial neural network that infers the best α for a given system. Then, the tool uses this α to compute Rényi's Squeeziness, so that the user can have an idea of how prone the system is to have cases of FEP. Moreover, `SqSelect` implements another three modes, so it can be used by a wider kind of users.

As future work, there are some open research lines that can improve the efficiency and usefulness of `SqSelect`. One line of future work is the improvement of the computation of Rényi's Squeeziness, so that `SqSelect` can be more efficient and quick. Second, we would like to

integrate `SqSelect` with other tools that automatise testing, so that it can be used as a previous step to assess how much testing should be performed. In particular, we would like to incorporate the efficient and systematic generation and processing of mutants (Cañizares, Núñez, & Merayo, 2018; Delgado-Pérez, Rose, & Medina-Bulo, 2019; Gómez-Abajo, Guerra, de Lara, & Merayo, 2018; Gómez-Abajo, Guerra, de Lara, & Merayo, 2021; Gutiérrez-Madroñal, García-Domínguez, & Medina-Bulo, 2019) so that we can offer mutation testing features. Finally, we would like to extend our tool to deal with other FSM-based formalism. We are particularly interested in distributed systems where communications can be asynchronous (Hierons, Merayo, & Núñez, 2017; Hierons, Merayo, & Núñez, 2018; Merayo, Hierons, & Núñez, 2018; Merayo, Hierons, & Núñez, 2018).

CRediT authorship contribution statement

Alfredo Ibias: Conceptualization, Software, Validation, Formal analysis, Data curation, Writing - original draft, Writing - review &

editing, Visualization. Manuel Núñez: Conceptualization, Methodology, Writing - original draft, Writing - review & editing, Supervision, Funding acquisition.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Alternative definition of Rényi’s Squeeziness

Lemma 1. Let $M = (Q, q_{in}, I, O, T)$ be an FSM, $k > 0$ and $\alpha \in \mathbf{R}_+ \setminus \{1\}$. Let us consider a random variable $\xi_{dom_{M,k}}$ ranging over the domain of $f_{M,k}$. We have that

$$Sq_{\alpha,k}(M) = \frac{1}{1-\alpha} \log_2 \left(\frac{\sum_{s \in dom_{M,k}} (\sigma_{\xi_{dom_{M,k}}}(s))^\alpha}{\sum_{t \in image_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{dom_{M,k}}}(s) \right)^\alpha} \right)$$

If α tends to 1 then we obtain Shannon’s entropy (Rényi, 1961) and we have

$$Sq_{1,k}(M) = - \sum_{t \in image_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{dom_{M,k}}}(s) \right) \cdot \mathcal{R}_M(t)$$

where the term $\mathcal{R}_M(t)$ is equal to

$$\sum_{s \in f_M^{-1}(t)} \frac{\sigma_{\xi_{dom_{M,k}}}(s)}{\sigma_{\xi_{dom_{M,k}}}(f_M^{-1}(t))} \cdot \log_2 \left(\frac{\sigma_{\xi_{dom_{M,k}}}(s)}{\sigma_{\xi_{dom_{M,k}}}(f_M^{-1}(t))} \right)$$

If α tends to ∞ then we obtain min-entropy (Rényi, 1961) (that is, $\mathcal{H}_\infty(X) = -\log_2(\max_i p_i)$) and we have

$$Sq_{\infty,k}(M) = \log_2 \left(\frac{\max_{t \in image_{M,k}} \sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{dom_{M,k}}}(s)}{\max_{s \in dom_{M,k}} \sigma_{\xi_{dom_{M,k}}}(s)} \right)$$

Proof.

$$\begin{aligned} Sq_{\alpha,k}(M) &= \mathcal{H}_\alpha(\xi_{dom_{M,k}}) - \mathcal{H}_\alpha(\xi_{image_{M,k}}) = \\ &= \frac{1}{1-\alpha} \log_2 \left(\sum_{s \in dom_{M,k}} \sigma_{\xi_{dom_{M,k}}}(s)^\alpha \right) \\ &\quad - \frac{1}{1-\alpha} \log_2 \left(\sum_{t \in image_{M,k}} \sigma_{\xi_{image_{M,k}}}(t)^\alpha \right) = \\ &= \frac{1}{1-\alpha} \log_2 \left(\frac{\sum_{s \in dom_{M,k}} \sigma_{\xi_{dom_{M,k}}}(s)^\alpha}{\sum_{t \in image_{M,k}} \sigma_{\xi_{image_{M,k}}}(t)^\alpha} \right) = \\ &= \frac{1}{1-\alpha} \log_2 \left(\frac{\sum_{s \in dom_{M,k}} (\sigma_{\xi_{dom_{M,k}}}(s))^\alpha}{\sum_{t \in image_{M,k}} \left(\sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{dom_{M,k}}}(s) \right)^\alpha} \right) \end{aligned}$$

When $\alpha \rightarrow 1$, the result has been proven in previous work (Ibias et al., 2019). Finally, if $\alpha \rightarrow \infty$, then the proof is the following:

$$\begin{aligned}
\text{Sq}_{\infty,k}(M) &= \mathcal{H}_{\infty}(\xi_{\text{dom}_{M,k}}) - \mathcal{H}_{\infty}(\xi_{\text{image}_{M,k}}) \\
&= -\log_2 \left(\max_{s \in \text{dom}_{M,k}} \sigma_{\xi_{\text{dom}_{M,k}}}(s) \right) \\
&\quad + \log_2 \left(\max_{t \in \text{image}_{M,k}} \sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s) \right) \\
&= \log_2 \left(\frac{\max_{t \in \text{image}_{M,k}} \sum_{s \in f_M^{-1}(t)} \sigma_{\xi_{\text{dom}_{M,k}}}(s)}{\max_{s \in \text{dom}_{M,k}} \sigma_{\xi_{\text{dom}_{M,k}}}(s)} \right)
\end{aligned}$$

Appendix B. Deployment of the tool

In this appendix we briefly analyse some issues related to the deployment of `SqSelect`. Specifically, we will discuss the inclusion of our ANN, developed using PyTorch (therefore, written in python), in `SqSelect`, a tool that has been developed using java. Although we found some libraries that claim to (easily) perform this integration, we were unable to make them work. We contemplated several alternatives: deploy the ANN in a controlled server, execute the python code from java, integrate the python code in java and use another library for the ANN. The second and third options, in addition to being more complex than the first one, have the problem of dealing with the communication between the java virtual machine and the python libraries, what is a really complex task. The last option has the problem of having to refactor the entire ANN, without being sure that the new library will be able to being properly integrated into java. Finally, the first option, despite of being not so complex, has the security risks associated to having a server running in the background. Luckily, those risks could be easily overcome using some *tricks*: we execute the server only in localhost, avoiding the possibility of accessing it from outside the computer; we use an uncommon port in order to both avoid collision with other server utilities and make it difficult to find the correct port; and we use a specific kind of message so that the server does not read any other type of message. Therefore, we decided to perform a small workaround: we set up a Flask server in python that receives the ANN input parameters by using a JSON form and returns the ANN output (the best α value) in a JSON response. `SqSelect` has to set up this server in localhost, avoiding any call from outside the computer.

This option arises a potential security concern: whether deploying a server each time that `SqSelect` is initialised is secure. In other words, we have to evaluate whether this action will be a backdoor to execute python code in the host computer. We are aware of this concern but discard it because the server is deployed in localhost, without access to internet. So, it will not be an open port for potential external attackers. Also, it does not use a common port, so it will be difficult to find the open port from inside the own computer. Finally, the Flask application only accepts one type of messages, rejecting any other input.

References

- Abdelmoez, W., Nassar, D.E.M., Shereshevsky, M., Gradetsky, N., Gunnalan, R., Ammar, H. H., Yu, B. & Mili, A. (2004). Error propagation in software architectures. In 10th IEEE International software metrics symposium, METRICS'04 (pp. 384–393). IEEE Computer Society.
- Acharya, J., Orlitsky, A., Suresh, A. T., & Tyagi, H. (2017). Estimating Renyi entropy of discrete distributions. *IEEE Transactions on Information Theory*, 63(1), 38–56.
- Ammann, P., & Offutt, J. (2017). *Introduction to software testing* (2nd Ed.). Cambridge University Press.
- Androutopoulos, K., Clark, D., Dan, H., Hierons, R. & Harman, M. (2014). An analysis of the relationship between conditional entropy and failed error propagation in software testing. In Int 36th Int. Conf. on Software Engineering, ICSE'14 (pp. 573–583). ACM Press.
- Ayala, D., Borrego, A., Hernández, I., & Ruiz, D. (2020). A neural network for semantic labelling of structured information. *Expert Systems with Applications*, 143, Article 113053.
- Boreale, M. & Paolini, M. (2014). On formally bounding information leakage by statistical estimation. In 17th Int. Conf. on Information Security, ISC'14, LNCS 8783 (pp. 216–236). Springer.
- Cañizares, P. C., Núñez, A., & Lara, J. (2019). An expert system for checking the correctness of memory systems using simulation and metamorphic testing. *Expert Systems with Applications*, 132, 44–62.
- Cañizares, P. C., Núñez, A., & Merayo, M. G. (2018). Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software*, 143, 187–207.
- Cavalli, A. R., Higashino, T., & Núñez, M. (2015). A survey on formal active and passive testing with applications to the cloud. *Annales of Telecommunications*, 70(3–4), 85–93.
- Chan, A., Winter, S., Saissi, H., Pattabiraman, K. & Suri, N. (2017). IPA: Error propagation analysis of multi-threaded programs using likely invariants. In 10th Int. Conf. on Software Testing, Verification and Validation, ICST'17 (pp. 184–195). IEEE Computer Society.
- Chatterjee, K., de Alfaro, L. & Henzinger, T. A. (2004). Trading memory for randomness. In 1st Int. Conf. on Quantitative Evaluation of Systems, QEST'04 (pp. 206–217).
- Chothia, T., Kawamoto, Y., & Novakovic, C. (2014). Leakwatch: Estimating information leakage from java programs. In 19th European symposium on research in computer security (pp. 219–236). Springer.
- Clark, D., Feldt, R., Poulding, S. M. & Yoo, S. (2015). Information transformation: An underpinning theory for software engineering. In 37th IEEE/ACM international conference on software engineering, ICSE'15 (pp. 599–602).
- Clark, D., & Hierons, R. M. (2012). Squeeziness: An information theoretic measure for avoiding fault masking. *Information Processing Letters*, 112(8–9), 335–340.
- Clark, D., Hierons, R. M., & Patel, K. (2019). Normalised squeeziness and failed error propagation. *Information Processing Letters*, 149, 6–9.
- Coppik, N., Schwahn, O., Winter, S. & Suri, N. (2017). TrEKer: tracing error propagation in operating system kernels. In 32nd IEEE/ACM Int. Conf. on Automated Software Engineering, ASE'17 (pp. 377–387). IEEE Computer Society.
- Cover, T. M., & Thomas, J. A. (1991). *Elements of information theory*. Wiley Interscience.
- de Mesquita Sá Junior, J. J., Correia Ribas, L. & Martínez Bruno O. (2019). Randomized neural network based signature for dynamic texture classification. *Expert Systems with Applications* 135, 194–200.
- Delgado-Pérez, P., Rose, L. M., & Medina-Bulo, I. (2019). Coverage-based quality metric of mutation operators for test suite improvement. *Software Quality Journal*, 27(2), 823–859.
- Díaz, G., Macià, H., Valero, V., Boubeta-Puig, J., & Cuartero, F. (2020). An intelligent transportation system to control air pollution and road traffic in cities integrating CEP and colored petri nets. *Neural Computing and Applications*, 32(2), 405–426.
- Feldt, R., Poulding, S. M., Clark, D. & Yoo, S. (2016). Test set diameter: Quantifying the diversity of sets of test cases. In 9th IEEE Int. Conf. on Software Testing, Verification and Validation, ICST'16 (pp. 223–233). IEEE Computer Society.
- Feldt, R., Torkar, R., Gorschek, T., & Afzal, W. (2008). Searching for cognitively diverse tests: Towards universal test diversity metrics. In 1st IEEE Int. Conf. on Software Testing Verification and Validation Workshops (pp. 178–186). IEEE Computer Society.
- García de Prado, A., Ortiz, G., & Boubeta-Puig, J. (2017). COLLECT: COLlaborativE ConText-aware service oriented architecture for intelligent decision-making in the Internet of Things. *Expert Systems with Applications*, 85, 231–248.
- Gaudel, M. -C. (1995). Testing can be formal, too! In 6th Int. Joint Conf. CAAP/FASE, Theory and Practice of Software Development, TAPSOFT'95, LNCS 915 (pp. 82–96). Springer.
- Gómez-Abajo, P., Guerra, E., de Lara, J., & Merayo, M. G. (2018). A tool for domain-independent model mutation. *Science of Computer Programming*, 163, 85–92.

- Gómez-Abajo, P., Guerra, E., de Lara, J., & Merayo, M. G. (2021). Wodel-Test: A model-based framework for language-independent mutation testing. *Software and Systems Modeling*. in press.
- Goodfellow, I. J., Bengio, Y., & Courville, A. C. (2016). *Deep Learning*. Adaptive computation and machine learning. MIT Press.
- Gu, W., Foster, K., Shang, J., & Wei, L. (2019). A game-predicting expert system using big data and machine learning. *Expert Systems with Applications*, 130, 293–305.
- Gutiérrez-Madroñal, L., García-Domínguez, A., & Medina-Bulo, I. (2019). Evolutionary mutation testing for IoT with recorded and generated events. *Software – Practice & Experience*, 49(4), 640–672.
- He, K., Zhang, X., Ren, S. & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In 15th IEEE Int. Conf. on Computer Vision, ICCV'15 (pp. 1026–1034). IEEE Computer Society.
- Hierons, R. M., Bogdanov, K., Bowen, J., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Luettgen, G., Simons, A., Vilkomir, S., Woodward, M., & Zedan, H. (2009). Using formal specifications to support testing. *ACM Computing Surveys*, 41(2), 9:1–9:76.
- Hierons, R. M., Merayo, M. G., & Núñez, M. (2017). An extended framework for passive asynchronous testing. *Journal of Logical and Algebraic Methods in Programming*, 86(1), 408–424.
- Hierons, R. M., Merayo, M. G., & Núñez, M. (2018). Bounded reordering in the distributed test architecture. *IEEE Transactions on Reliability*, 67(2), 522–537.
- Hiller, M., Jhumka, A., & Suri, N. (2002). PROPANE: An environment for examining the propagation of errors in software. In 13th Int. Symposium on Software Testing and Analysis (pp. 81–85). ACM Press.
- Hiller, M., Jhumka, A., & Suri, N. (2004). EPIC: Profiling the propagation and effect of data errors in software. *IEEE Transactions on Computers*, 53(5), 512–530.
- Ibias, A., Hierons, R. M., & Núñez, M. (2019). Using squeeziness to test component-based systems defined as Finite State Machines. *Information & Software Technology*, 112, 132–147.
- Ibias, A., & Núñez, M. (2020). Estimating fault masking using Squeeziness based on Rényi's entropy. In 35th ACM symposium on applied computing (pp. 1936–1943). ACM Press.
- Ibias, A., Núñez, M., & Hierons, R. M. (2021). Using mutual information to test from Finite State Machines: Test suite selection. *Information & Software Technology*, 132, Article 106498.
- Isberner, M., Howar, F. & Steffen, B. (2015). The open-source learnlib. In: 27th Int. Conf. on Computer Aided Verification, CAV'15, LNCS 9206 (pp. 487–495). Springer.
- Islam, M. S., Nepal, M. P., Skitmore, R. M., & Kabir, G. (2019). A knowledge-based expert system to assess power plant project cost overrun risks. *Expert Systems with Applications*, 136, 12–32.
- Jain, A. K., Mao, J., & Mohiuddin, K. M. (1996). Artificial neural networks: A tutorial. *IEEE Computer*, 29(3), 31–44.
- Johansson, A., & Suri, N. (2005). Error propagation profiling of operating systems. In 35th Int. Conf. on Dependable Systems and Networks (pp. 86–95). IEEE Computer Society.
- Kutner, M. H., Nachtsheim, C. J., Wasserman, W., & Neter, J. (2003). *Applied linear regression models* (4th Ed.). McGraw-Hill.
- Laski, J. W., Szermer, W., & Luczycki, P. (1995). Error masking in computer programs. *Software Testing, Verification and Reliability*, 5(2), 81–105.
- Macaskassy, S. A. & Provost, F. J. (2004). Confidence bands for ROC curves: Methods and an empirical study. In 1st Int. Workshop on ROC analysis in Artificial Intelligence, ROCAI'04 (pp. 61–70).
- Marinescu, R., Seceleanu, C., Guen, H. L. & Pettersson, P. (2015). A Research Overview of Tool-Supported Model-based Testing of Requirements-based Designs. Vol. 98 of *Advances in Computers*. Elsevier, Ch. 3, pp. 89–140.
- Masri, W., Abou-Assi, R., El-Ghali, M., & Al-Fatairi, N. (2009). An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In 2nd Int. Workshop on Defects in Large Software Systems (pp. 1–5). ACM Press.
- Merayo, M. G., Hierons, R. M., & Núñez, M. (2018). Passive testing with asynchronous communications and timestamps. *Distributed Computing*, 31(5), 327–342.
- Merayo, M. G., Hierons, R. M., & Núñez, M. (2018). A tool supported methodology to passively test asynchronous systems with multiple users. *Information & Software Technology*, 104, 162–178.
- Miranskyy, A. V., Davison, M., Reesor, R. M., & Murtaza, S. S. (2012). Using entropy measures for comparison of software traces. *Information Sciences*, 203, 59–72.
- Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing* (3rd Ed.). John Wiley & Sons.
- Neider, D., Smetsters, R., Vaandrager, F. W., & Kuppens, H. (2019). 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* (pp. 390–416). Springer.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S., 2019. PyTorch: An imperative style, high-performance deep learning library. In 32nd Annual Conf. on Neural Information Processing Systems, NeurIPS'19 (pp. 8024–8035). Curran Associates Inc.
- Pattipati, K. R., & Alexandridis, M. G. (1990). Application of heuristic search and information theory to sequential fault diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(4), 872–887.
- Pattipati, K. R., Deb, S., Dontamsetty, M., & Maitra, A. (1990). START: System testability analysis and research tool. In *IEEE Conference on Systems Readiness Technology 'Advancing Mission Accomplishment'* (pp. 395–402).
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.
- Piper, T., Winter, S., Schwahn, O., Bidarahalli, S., & Suri, N. (2015). Mitigating timing error propagation in mixed-criticality automotive systems. In *Int. Symposium on Real-Time Distributed Computing, ISORC'15* (pp. 102–109). IEEE Computer Society.
- Rényi, A. (1961). On measures of entropy and information. In *4th Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics* (pp. 547–561). University of California Press.
- Roldán, J., Boubeta-Puig, J., Martínez, J. L., & Ortiz, G. (2020). Integrating complex event processing and machine learning: An intelligent architecture for detecting iot security attacks. *Expert Systems with Applications*, 149(113251), 1–22.
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323, 533–536.
- Sagarna, R., Arcuri, A. & Yao, X. (2007). Estimation of distribution algorithms for testing object oriented software. In 9th IEEE Congress on Evolutionary Computation, CEC'07 (pp. 438–444). IEEE Computer Society.
- Santelices, R. A. & Harrold, M. J. (2011). Applying aggressive propagation-based strategies for testing changes. In 4th Int. Conf. on Software Testing, Verification and Validation, ICST'11 (pp. 11–20). IEEE Computer Society Press.
- Serna M. E., Acevedo M. E. & Serna A. A. (2019). Integration of properties of virtual reality, artificial neural networks, and artificial intelligence in the automation of software tests: A review. *Journal of Software: Evolution and Process* 31 (7), 2159.
- Shafique, M., & Labiche, Y. (2015). A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer*, 17(1), 59–76.
- Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27(379–423), 623–656.
- Wang, X., Cheung, S. -C., Chan, W. K. & Zhang, Z. (2009). Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In 31st Int. Conf. on Software Engineering, ICSE'09 (pp. 45–55). IEEE Computer Society.
- Woodward, M. R., & Al-Khanjari, Z. A. (2000). Testability, fault size and the domain-to-range ratio: An eternal triangle. In 12th Int. Symposium on Software Testing and Analysis (pp. 168–172). ACM Press.
- Yang, S. (1991). *Logic synthesis and optimization benchmarks user guide: Version 3.0*. Tech. rep. Microelectronics Center of North Carolina
- Yoo, S., Harman, M., & Clark, D. (2013). Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transactions on Software Engineering and Methodology*, 22(3), 19: 1–29.



Alfredo Ibias received B.A. degrees in Computer Science and in Mathematics from Complutense University of Madrid, Spain, and an M.A. degree in Formal Methods in Computer Science from the same university. He is currently working on a Ph.D. degree in Computer Science at the same university.




Manuel Núñez received a Ph.D. degree in Mathematics and an M.S. degree in Economics. He is a Professor of Computer Science with the Complutense University of Madrid, Spain. He belongs to the IEEE SMC Technical Committee on Computational Collective Intelligence, he is a member of several Editorial Boards and has served on more than 130 Program Committees of international events in Computer Science.

10.4 GPTSG: A Genetic Programming test suite generator using Information Theory measures

Authors	Alfredo Ibias, David Griñán and Manuel Núñez
Title	GPTSG: A Genetic Programming test suite generator using Information Theory measures
Publication Type	Conference
Venue	15th International Work-Conference on Artificial Neural Networks
Year	2019
DOI/URL	https://doi.org/10.1007/978-3-030-20521-8_59
Pages	13
Authors' Contributions	Ibias and Núñez developed the theory. Ibias, Griñán and Núñez designed the experiments. Ibias and Griñán developed and executed the experiments. Ibias and Núñez wrote the manuscript. Núñez reviewed the manuscript.



GPTSG: A Genetic Programming Test Suite Generator Using Information Theory Measures

Alfredo Ibias¹, David Griñán², and Manuel Núñez¹ 

¹ Complutense University of Madrid, 28040 Madrid, Spain
{aibias,manuelnu}@ucm.es

² Polytechnic University of Madrid, 28223 Madrid, Spain
david.grinanm@alumnos.upm.es

Abstract. The automatic generation of test suites that get the best score with respect to a given measure is costly in terms of computational power. In this paper we present a genetic programming approach for generating test suites that get a good enough score for a given measure. We consider a black-box scenario and include different Information Theory measures. Our approach is supported by a tool that will actually generate test suites according to different parameters. We present the results of a small experiment where we used our tool to compare the goodness of different measures.

Keywords: Testing · Genetic programming · Test generation · Information Theory

1 Introduction

Software testing [2, 20] is an important research topic because it helps to ensure that the systems work as expected. Specially important have been the efforts to define testing in a formal way, which is still an active research area [3]. There are two orthogonal approaches to test a system: Consider that the system is a white-box or consider that it is a black-box. In this paper, we will focus on the latter as it poses a higher challenge because we have to rely on a model of the system, but we cannot see how the implementation of the system works. One of the main components of black-box testing consists in generating test suites that find the maximum number of faults. Actually, this is a critical part when we are talking about systems that require a lot of time to process an input or that have a small time window where you can test them. As it is a critical task, a lot of work around generating and selecting test suites and constructing tools supporting the theoretical frameworks has been performed [23].

Research partially supported by the Spanish project DArDOS (TIN2015-65845-C3-1-R) and the Comunidad de Madrid project FORTE-CM (S2018/TCS-4314).

Genetic programming has been a successful technique to find *good* enough solutions to NP-hard problems. Specifically, genetic programming was developed to solve the restrictions of genetic algorithms [15]. Instead of representing each solution as a vector, each element of the solutions space is coded as a tree, with no size limitations. In order to ensure that these trees always represent feasible solutions, the *grammar-based genetic programming* paradigm was proposed [18]. Trees according to this paradigm are produced as derivations from a grammar that is specifically designed so that every solution is feasible. This approach has been used to search for neural net structures [4], Bayesian network structures [21] and rule-based systems [17]. Another field that has been applied to find *good* test suites is Information Theory [6]. There are several approaches proposing different measures to select the *better* test suites from a given set and comparing them [9, 11] and the ones that show better performance are measures based on Information Theory.

In order to find the best test suite, up to a given size, to test a certain system we confront the classical combinatorial explosion: we have to check how good are all the subsets (up to a given size) of the set of available test cases. Previous work has automatically generated test suites (or test cases) using genetic algorithms [7, 8, 16, 22]. Most of these approaches usually consider the basic conception of a genetic algorithm, with the risk of losing the correctness of the test suite, or needing to introduce some special items (as a *do not care element* [10]) in order to preserve it. Finally, although there are some work using genetic programming, we are not aware of any work that uses genetic programming ensuring at the same time the correctness and length of the test suite.

In this paper we propose a genetic programming algorithm to generate test suites, guided by a grammar that ensures their correctness. The algorithm generates test suites that get a *good* score for a given measure, and with a fixed length, in order to avoid the generation of extremely long and computationally heavy (or short and useless) test suites. This algorithm is supported by a tool that implements it, using already proved Information Theory based measures. This tool also allows users to compare the test suites generated by the algorithm using two different measures and see how well each of them performs. Finally, the tool allows to include measures defined by the user.

The rest of the paper is organized as follows. In Sect. 2 we introduce the main concepts that we will use along the paper. In Sect. 3 we introduce the core of our genetic programming algorithm. In Sect. 4 we present the main features of our tool and the results of an experiment. Finally, in Sect. 5 we give the conclusions of our work.

2 Preliminaries

In this section we present the main definitions and concepts that we use throughout this paper. Most of the concepts are based on the classical notions while some notation is adapted to facilitate the formulation of subsequent definitions.

Given a set A , we let:

- A^* denote the set of finite sequences of elements of A .
- $\epsilon \in A^*$ denote the empty sequence.
- A^+ denote the set of non-empty sequences of elements of A .
- $|A|$ denote the cardinal of set A .

Given a sequence $\sigma \in A^*$, we have that $|\sigma|$ denotes its length. Given a sequence $\sigma \in A^*$ and $a \in A$, we have that σa denotes the sequence σ followed by a and $a\sigma$ denotes the sequence σ preceded by a .

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

A *Finite State Machine* is a (finite) labelled transition system in which transitions are labelled by an input/output pair. We use this formalism to define specifications.

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

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

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

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

In this paper we assume that FSMs are deterministic. We make this assumption because most Information Theory measures are applied to code and code is usually deterministic. We do not impose that FSMs are input-enabled. We will assume the *test hypothesis* [14]: the *System Under Test* (SUT) can be modelled as an object described in the same formalism as the specification (in our case, an FSM). Note that we do not need to have access to this description; we are indeed in a black-box testing framework because we only assume the existence of such FSM. Actually, it would be enough to assume that each time that the SUT receives a sequence of input actions, it returns a sequence of output actions. As usual, we do need access to the specification.

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

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

Using the notion of trace, we can introduce the notion of test: a test is a sequence of (input action, output action) pairs. A test suite will be a set of tests.

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

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

Intuitively, a test (α, β) for M denotes that the application of the sequence of input actions α to a correct system (with respect to M) should show the sequence of output actions β . Note that if we would allow non-determinism, then the previous inequality must be appropriately replaced to express that the behaviours of the SUT must be a subset of those of the specification. For now, we will assume the determinism of the FSMs.

In order to select the tests that can detect the higher amount of fails in the program, it is useful to have a *measure* on the goodness of a test suite. Let us emphasize that measures will be, in general, heuristics to find good solutions and that each measure should be validated with experiments. Usually, higher values of a measure will be associated with better solutions, but this relation need not be monotonic. The measures that we use in this paper have been introduced in previous work and it has been shown that they are useful to find good test suites. We introduce a general notion of measure.

Definition 4. A measure is a function

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

Intuitively, a measure is a function that receives an FSM and a test suite and returns a real number representing how good the measure considers that this test suite is to detect fails in an SUT. This notion of measure allows us to use information both from the specification and the test suite that we are

evaluating, although it not necessarily has to use information from both, that is, a measure could work only with the information from the test suite and not use the specification at all. Finding the best test suite according to a measure (that is, the test suite that gets the best score) is usually an NP-hard problem (due to the combinatorial explosion). Therefore, we decided to rely on genetic programming in order to obtain *relatively good* test suites. A genetic algorithm is composed by:

```

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

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

The structure of a genetic algorithm is given in Algorithm 1. A basic genetic programming algorithm is a genetic algorithm where the codification of the population in genes does not use a linear structure (as a vector) but a tree-like structure [15]. Most of the work using genetic algorithms to generate test suites rely on a linear structure to represent the test suite. Specially, they use to rely on a vector of the inputs of the test suite [7, 8, 16, 22]. This encoding of a test suite presents a problem: if the FSM is not input-enabled, then the algorithm could generate invalid tests that will always fail when applied to the SUT, even if this is totally equivalent to the FSM. As we are working with deterministic but not necessarily input-enabled FSMs, we have to face this problem and using a grammar-guided genetic programming algorithm allows us to ensure the correctness of the generated test suites. This approach also allows us to use the information from the output that each input generates in each state of the FSM (as the inputs do not have to generate the same output in all the states).

3 The Genetic Programming Algorithm

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

3.1 Encoding

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

- A start non-terminal symbol S that starts the grammar.
- A non-terminal symbol T that introduces each test of the test suite.
- A non-terminal symbol N for each state, where $N \in \mathbb{N}$ 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 $S \rightarrow T$ to generate the initial test.
- A production rule $T \rightarrow T + T$ to introduce a new test.
- A production rule $T \rightarrow 0$ to start each test in the FSM initial state.
- A production rule $N \rightarrow 'a/b' + M$ for each transition from the state N to a state M with input/output pair (a, b) .
- A production rule $N \rightarrow 'null'$ for each state N to a terminal to represent the end of the test.

Given an FSM, the generation of the associated grammar is automatic (and it has been implemented as part of our tool).

3.2 Initial Population

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

3.3 Fitness Function

The fitness function of our genetic programming algorithm will be the available measures. As previously defined, they will receive the test suite and the FSM and will return a real value that represents how *good* is this test suite according to the measure. An important remark about fitness functions is that they should

be easy to compute, as they will be invoked many times during the execution of the algorithm. Therefore, fitness functions with high computational cost will lead to a higher computational cost of the algorithm.

We decided to give the users the capability to select the fitness function that better suites their problem, along with the decision on whether the score should be maximized or minimized. As we explained before, fitness functions should have similar performances and this is the case for the measures based on Information Theory that we include in our tool. Among them, we can mention, due to the big improvement with respect to previous measures, the *Test Set Diameter* (TSDm) based measures [9]. We implemented the Input-TSDm, the Output-TSDm and the InputOutput-TSDm. Also, we implemented a measure that we have developed in our research group and that it is called *Biased Mutual Information*. Note that users of our tool can add their own measures. So, our tool can be use to evaluate the usefulness of new proposals because they can be compared with existing ones.

3.4 Stopping Criterion

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

3.5 Selection Method

We use a variant of elitist reduction [19]. First, the test suites that got a fitness score over the mean (or under the mean if we want to minimize) go directly to the next epoch. In addition, the ones that are under the mean can pass to the next epoch if their score is higher than the mean minus a random number modulo the distance between the mean and the best score.

3.6 Crossover Method

The choice of crossover method depends on our encoding and the characteristics we want the produced test suites to have. As we use a grammatical encoding, we need to use a grammatical crossover. We have considered a mixture between the Whigham crossover [18] and the standard grammatical crossover [5]. Also, as we want all our test suites to have the same length (as previously defined), we need to slightly modify crossovers in order to achieve a crossover that keeps the length fixed. Algorithm 2 shows how crossover is performed.

Finally, we need to set the probability of producing the crossover. In our case, giving how hard is to perform a crossover, we decided to set this probability to 90%, so that we favour the mixture between test suites.

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

Algorithm 2. Crossover algorithm

3.7 Mutation Method

A mutation consists in generating a new test with the same length. The probability of performing a mutation will be, as usual [19], equal to 5% for each test of each test suite of the population.

4 GPTSG

We have implemented a tool¹ supporting our framework. The tool has two main uses: generate a test suite with a given length according to a selected measure and compare different measures. In order to develop the tool, we looked for libraries dealing with FSMs and we decided to use the OpenFST library [1]. Therefore, input files must be in OpenFST format, with the .fst extension. The tool will have two kind of calls *generate* and *compare*. The syntax of the two calls is:

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

¹ The tool can be downloaded from <https://github.com/Colosu/gptsg>.

and two examples of calls are:

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

Currently, our tool supports the following fitness functions:

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

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

4.1 Test Suite Generation

In order to generate a *good* enough test suite, we implemented the genetic programming algorithm explained in the previous section, giving some configuration to the user. The tool needs that the input FSM is in OpenFST format (in a `.fst` file). This format is easy to use and can be learned quickly. Also, the tool needs to know the length of the expected test suite, in terms of input actions, and the measure to use as a fitness function. Then, the user will receive a `.txt` file with the generated test suite, with each test conformed by a succession of input/output pairs.

4.2 Test Suite Comparison

The tool allows users to compare two measures. It needs to know the length of the desired test suite, the two measures to be compared and if it should maximize or minimize each measure. Essentially, the tool takes the set of 100 FSMs that are shipped with the tool, representing different and diverse scenarios and characteristics, and for each one of them it generates two test suites according to the corresponding measures. Then, the tool produces 1000 mutants of the corresponding FSM and checks which test suite kills more mutants. With the results for each FSM, the tool produces an output telling the percentage of cases where each test suite has killed more mutants, along with a percentage of how many mutants were killed by each test suite. This process is repeated 50 times, getting 50 results, and at the end, the program gives a mean of all the results obtained for the 50 repetitions. This process is given in Algorithm 3.

```

Data: length, measure1, measure2
Result: .txt file with the values
REP = 50;
FSM = 100;
for each REP do
  Set control values to 0;
  for each FSM F do
    Generate TS1 genetic test suite using measure measure1;
    Generate TS2 genetic test suite using measure measure2;
    Generate 1000 mutants of F;
    Check which test suite kills more mutants;
  end
  Output the percentage of runs TS1 killed more mutants;
  Output the percentage of runs TS2 killed more mutants;
  Output the percentage of mutants killed by TS1;
  Output the percentage of mutants killed by TS2;
end
Output the average percentage of runs TS1 killed more mutants;
Output the average percentage of runs TS2 killed more mutants;
Output the average percentage of mutants killed by TS1;
Output the average percentage of mutants killed by TS2;

```

Algorithm 3. Test suite comparison algorithm

4.3 Experiment

Next we show the results of a small experiment to evaluate our genetic programming algorithm and tool. First, we compared the Input Test Set Diameter measure, used as fitness function, and a random test suite generation. We observed that the genetically generated test suite killed more mutants than the randomly generated test suite in a 75.3% of the cases, killing an average of 47.1% of the mutants, while the randomly generated test suite killed more mutants the 24.7% of the cases, killing an average of 43.9% of the mutants. We can see the full comparison in Fig. 1 (left), where each of the first 50 rows shows the result of an iteration of the experiment. In order to see how two measures are compared, we rerun the comparison algorithm to compare the maximization of the Input Test Set Diameter and the maximization of the Output Test Set Diameter. The results can be seen in Fig. 1 (right). As expected, they obtain similar results, getting better results the Output TSDm due to the randomization involved in the genetic algorithm. On average, the Input TSDm killed more mutants the 49% of the cases, killing 47.3% of the mutants, while Output TSDm killed more mutants the 51% of the cases, killing 47.5% of the mutants.

Iteration Number	% wins ITSDm	% wins random	% mutants killed by ITSDm	% mutants killed by random
1	0.757576	0.242424	0.475081	0.439909
2	0.744898	0.255102	0.476306	0.443786
3	0.75	0.25	0.46496	0.43275
4	0.8	0.2	0.47095	0.43229
5	0.75	0.25	0.46347	0.43407
6	0.69	0.31	0.46947	0.4419
7	0.72449	0.27551	0.475051	0.447755
8	0.72449	0.27551	0.475051	0.447857
9	0.74	0.26	0.46035	0.43142
10	0.767677	0.232323	0.471525	0.441646
11	0.72449	0.27551	0.473204	0.443663
12	0.767677	0.232323	0.470525	0.441
13	0.646465	0.353535	0.465929	0.440162
14	0.717172	0.282828	0.468394	0.437384
15	0.81	0.19	0.47143	0.43202
16	0.686869	0.313131	0.469606	0.441101
17	0.76	0.24	0.46867	0.43234
18	0.81	0.19	0.47033	0.43285
19	0.79798	0.20202	0.468172	0.438222
20	0.83	0.17	0.46732	0.43214
21	0.77	0.23	0.46903	0.43535
22	0.757576	0.242424	0.473949	0.440232
23	0.78	0.22	0.46716	0.43258
24	0.744898	0.255102	0.475673	0.442582
25	0.79798	0.20202	0.475343	0.446455
26	0.79	0.21	0.4678	0.43391
27	0.68	0.32	0.46184	0.43662
28	0.79	0.21	0.46923	0.43317
29	0.75	0.25	0.4663	0.43457
30	0.79	0.21	0.46861	0.43607
31	0.747475	0.252525	0.474848	0.436535
32	0.663265	0.336735	0.469214	0.445235
33	0.73	0.27	0.46731	0.43186
34	0.721649	0.278351	0.478278	0.449763
35	0.72	0.28	0.46624	0.43619
36	0.79798	0.20202	0.472141	0.439152
37	0.783505	0.216495	0.48668	0.450495
38	0.75	0.25	0.46744	0.43613
39	0.767677	0.232323	0.465687	0.432909
40	0.676768	0.323232	0.469	0.443343
41	0.742268	0.257732	0.484216	0.450979
42	0.75	0.25	0.46626	0.43473
43	0.77551	0.22449	0.477082	0.443643
44	0.77	0.23	0.46771	0.43251
45	0.76	0.24	0.47459	0.43538
46	0.757576	0.242424	0.471333	0.439051
47	0.744898	0.255102	0.479857	0.444378
48	0.806122	0.193878	0.478143	0.437531
49	0.777778	0.222222	0.473505	0.440222
50	0.74	0.26	0.46422	0.43529
Mean	0.752723	0.247277	0.470851	0.438578

Iteration Number	% wins ITSDm	% wins OTSDm	% mutants killed by ITSDm	% mutants killed by OTSDm
1	0.494845	0.505155	0.484216	0.48366
2	0.546392	0.453608	0.481206	0.479309
3	0.44	0.56	0.46812	0.47277
4	0.525253	0.474747	0.473485	0.468192
5	0.443299	0.556701	0.481979	0.489619
6	0.515152	0.484848	0.474323	0.473404
7	0.525253	0.474747	0.477051	0.474596
8	0.56	0.44	0.4725	0.4662
9	0.48	0.52	0.46631	0.47339
10	0.45	0.55	0.46801	0.47551
11	0.515152	0.484848	0.469535	0.472646
12	0.489796	0.510204	0.476612	0.478694
13	0.51	0.49	0.46966	0.46947
14	0.40404	0.59596	0.468949	0.483323
15	0.47	0.53	0.46272	0.46797
16	0.52	0.48	0.46723	0.46533
17	0.367347	0.632653	0.468765	0.480704
18	0.535354	0.464646	0.471131	0.471232
19	0.428571	0.571429	0.47351	0.48201
20	0.51	0.49	0.46843	0.47081
21	0.561224	0.438776	0.483551	0.479612
22	0.43	0.57	0.46509	0.46947
23	0.414141	0.585859	0.470253	0.475808
24	0.555556	0.444444	0.473596	0.469566
25	0.494949	0.505051	0.474384	0.475111
26	0.52	0.48	0.46671	0.46436
27	0.505155	0.494845	0.480866	0.481639
28	0.515152	0.484848	0.472889	0.473586
29	0.469388	0.530612	0.474306	0.476204
30	0.545455	0.454545	0.477	0.473242
31	0.510204	0.489796	0.476571	0.477184
32	0.51	0.49	0.46423	0.46837
33	0.469388	0.530612	0.477316	0.479204
34	0.408163	0.591837	0.475296	0.484643
35	0.545455	0.454545	0.468455	0.471949
36	0.494949	0.505051	0.475091	0.475495
37	0.5	0.5	0.46762	0.46903
38	0.46	0.54	0.46477	0.47045
39	0.474227	0.525773	0.48034	0.484113
40	0.45	0.55	0.46602	0.47369
41	0.489796	0.510204	0.480306	0.480286
42	0.546392	0.453608	0.484423	0.484804
43	0.51	0.49	0.46951	0.46428
44	0.47	0.53	0.46927	0.47261
45	0.469388	0.530612	0.478684	0.477918
46	0.43	0.57	0.46678	0.47126
47	0.438776	0.561224	0.479082	0.480398
48	0.515464	0.484536	0.479969	0.483546
49	0.56	0.44	0.4709	0.46138
50	0.484848	0.515152	0.472162	0.471455
Mean	0.489581	0.510419	0.47293	0.474633

Fig. 1. Results of ITSDm vs random (left) and ITSDm vs OTSDm (right).

5 Conclusions

The automatic generation of good test suites is a fundamental task when limitations in testing complex systems come into play. In this paper we have presented a genetic programming algorithm to generate these test suites. We have implemented a tool to support our algorithm so that any potential user can apply it. The tool allows users to compare genetically generated test suites that outperform different measures, so that we can compare the performance of each measure. We have relied on Information Theory to define the measures that will work as fitness functions of our genetic programming algorithm. Finally, we have performed several experiments with our tool and report on two of them. There are some lines for future work.

We are considering several lines of future work. First, it would be interesting to find and define new measures so that we can extend the catalogue of our tool. Second, we are working on extending our algorithm to deal with the generation of test suites to test systems with distributed interfaces [12, 13].

References

1. Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., Mohri, M.: OpenFst: a general and efficient weighted finite-state transducer library. In: Holub, J., Žd'árek, J. (eds.) CIAA 2007. LNCS, vol. 4783, pp. 11–23. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76336-9_3
2. Ammann, P., Offutt, J.: Introduction to Software Testing, 2nd edn. Cambridge University Press, Cambridge (2017)
3. Cavalli, A.R., Higashino, T., Núñez, M.: A survey on formal active and passive testing with applications to the cloud. *Ann. Telecommun.* **70**(3–4), 85–93 (2015)
4. Couchet, J., Manrique, D., Porras, L.: Grammar-guided neural architecture evolution. In: Mira, J., Álvarez, J.R. (eds.) IWINAC 2007. LNCS, vol. 4527, pp. 437–446. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73053-8_44
5. Couchet, J., Manrique, D., Rios, J., Rodríguez-Patón, A.: Crossover and mutation operators for grammar-guided genetic programming. *Soft Comput.* **11**(10), 943–955 (2007)
6. Cover, T.M., Thomas, J.A.: Elements of Information Theory, 2nd edn. Wiley, Hoboken (2006)
7. Derderian, K., Merayo, M.G., Hierons, R.M., Núñez, M.: Aiding test case generation in temporally constrained state based systems using genetic algorithms. In: Cabestany, J., Sandoval, F., Prieto, A., Corchado, J.M. (eds.) IWANN 2009. LNCS, vol. 5517, pp. 327–334. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02478-8_41
8. Derderian, K., Merayo, M.G., Hierons, R.M., Núñez, M.: A case study on the use of genetic algorithms to generate test cases for temporal systems. In: Cabestany, J., Rojas, I., Joya, G. (eds.) IWANN 2011. LNCS, vol. 6692, pp. 396–403. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21498-1_50
9. Feldt, R., Poulding, S.M., Clark, D., Yoo, S.: Test set diameter: quantifying the diversity of sets of test cases. In: 9th IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, pp. 223–233. IEEE Computer Society (2016)
10. Guo, Q., Hierons, R.M., Harman, M., Derderian, K.: Computing unique input/output sequences using genetic algorithms. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 164–177. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24617-6_12
11. Henard, C., Papadakis, M., Harman, M., Jia, Y., Traon, Y.L.: Comparing white-box and black-box test prioritization. In: 38th International Conference on Software Engineering, ICSE 2014, pp. 523–534. ACM Press (2016)
12. Hierons, R.M., Merayo, M.G., Núñez, M.: Bounded reordering in the distributed test architecture. *IEEE Trans. Reliab.* **67**(2), 522–537 (2018)
13. Hierons, R.M., Núñez, M.: Implementation relations and probabilistic schedulers in the distributed test architecture. *J. Syst. Softw.* **132**, 319–335 (2017)

14. ISO/IEC JTC1/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)
15. Koza, J.R.: Genetic Programming. MIT Press, Cambridge (1993)
16. Lefticaru, R., Ipate, F.: Automatic state-based test generation using genetic algorithms. In: 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2007, pp. 188–195. IEEE Computer Society (2007)
17. Luna, J.M., Romero, J.R., Ventura, S.: Design and behavior study of a grammar-guided genetic programming algorithm for mining association rules. *Knowl. Inf. Syst.* **32**(1), 53–76 (2012)
18. McKay, R.I., Hoai, N.X., Whigham, P.A., Shan, Y., O’Neill, M.: Grammar-based genetic programming: a survey. *Genet. Program. Evolvable Mach.* **11**(3–4), 365–396 (2010)
19. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Cambridge (1998)
20. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing, 3rd edn. Wiley, Hoboken (2011)
21. Regolin, E.N., Pozo, A.T.R.: Bayesian automatic programming. In: Keijzer, M., Tettamanzi, A., Collet, P., van Hemert, J., Tomassini, M. (eds.) EuroGP 2005. LNCS, vol. 3447, pp. 38–49. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31989-4_4
22. Samarah, A., Habibi, A., Tahar, S., Kharma, N.N.: Automated coverage directed test generation using a cell-based genetic algorithm. In: 11th Annual IEEE International High-Level Design Validation and Test Workshop, pp. 19–26. IEEE Computer Society (2006)
23. Shafique, M., Labiche, Y.: A systematic review of state-based test tools. *Int. J. Softw. Tools Technol. Transf.* **17**(1), 59–76 (2015)

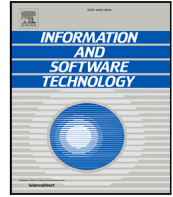
10.5 Using mutual information to test from Finite State Machines: Test suite selection

Authors	Alfredo Ibias, Manuel Núñez and Robert M. Hierons
Title	Using mutual information to test from Finite State Machines: Test suite selection
Publication Type	Journal
Venue	Information and Software Technology
Number	132
Year	2021
DOI/URL	https://doi.org/10.1016/j.infsof.2020.106498
Pages	21
Authors' Contributions	Ibias, Núñez and Hierons developed the theory. Ibias and Núñez designed the experiments. Ibias developed and executed the experiments. Ibias and Núñez wrote the manuscript. Núñez and Hierons reviewed the manuscript.



Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsofUsing mutual information to test from Finite State Machines: Test suite selection[☆]Alfredo Ibias^a, Manuel Núñez^{a,*}, Robert M. Hierons^b^a Instituto de Tecnología del Conocimiento, Universidad Complutense de Madrid, Madrid, Spain^b Department of Computer Science, The University of Sheffield, Sheffield, United Kingdom

ARTICLE INFO

Keywords:

Formal approaches to testing
Information Theory
Mutual information
Finite State Machines

ABSTRACT

Context: Mutual Information is an information theoretic measure designed to quantify the amount of similarity between two random variables ranging over two sets. In this paper, we adapt this concept and show how it can be used to select a *good* test suite to test from a Finite State Machine (FSM) based on a *maximise diversity* approach.

Objective: The main goal of this paper is to use Mutual Information in order to select test suites to test from FSMs and evaluate whether we obtain better results, concerning the quality of the selected test suite, than current state-of-the-art measures.

Method: First, we defined our scenario. We considered the case where we receive two (or more) test suites and we have to choose between them. We were interested in this scenario because it is a recurrent case in regression testing. Second, we defined our notion based on Mutual Information: Biased Mutual Information. Finally, we carried out experiments in order to evaluate the measure.

Results: We obtained experimental evidence that demonstrates the potential value of the measure. We also showed that the time needed to compute the measure is negligible when compare to the time needed to apply extra testing. We compared our measure with a state-of-the-art test selection measure and showed that our proposal outperforms it. Finally, we have compared our measure with a notion of transition coverage. Our experiments showed that our measure is slightly worse than transition coverage, as expected, but its computation is 10 times faster.

Conclusion: Our experiments showed that Biased Mutual Information is a good measure for selecting test suites, outperforming the current state-of-the-art measure, and having a (negative) correlation to fault coverage. Therefore, we can conclude that our new measure can be used to select the test suite that is likely to find more faults. As a result, it has the potential to be used to automate test generation.

1. Introduction

Software testing [1,2] is the main technique to validate complex systems with the goal of increasing their reliability. Testing is a time consuming part of software development; it has been observed that testing can cost more than 50% of the development budget [2]. Therefore, it is important to devise methodologies that reduce the time taken without notably decreasing effectiveness. A good starting point is to reduce the number of tests (the size of the *test suite*) that we apply to

the System Under Test (SUT).¹ Therefore, we require approaches that *select* a test suite that is small enough to be used in practice and is likely to be effective in finding faults. We can rephrase this as the problem of maximising the expected test effectiveness for a given cost/time. Since the cost of test execution typically depends on the test suite size, we reduce this to the problem of choosing amongst different test suites, that have the same size, the one that is most likely to find faults.

The main aim of the work presented in this paper is to devise measures that can help testers, or testing tools, to choose between

[☆] This work has been supported by the Spanish MINECO/FEDER (grant FAME, RTI2018-093608-B-C31); the Region of Madrid, Spain (grant FORTE-CM, S2018/TCS-4314) co-funded by EIE Funds of the European Union; the Region of Madrid - Complutense University of Madrid, Spain (grant PR65/19-22452); and the UK EPSRC (grant InfoTestSS, EP/P006116/2).

* Corresponding author.

E-mail addresses: aibias@ucm.es (A. Ibias), mn@sip.ucm.es (M. Núñez), r.hierons@sheffield.ac.uk (R.M. Hierons).

URLs: <https://alfredoibias.com/> (A. Ibias), <http://antares.sip.ucm.es/manolo/> (M. Núñez), <https://robhieronsgithub.io/> (R.M. Hierons).

¹ The number of tests needed to exhaustively test even the simplest systems is exorbitant. For example, exhaustive testing of a black-box implementation of a method adding two numbers on a 32-bit machine needs around $8 \cdot 10^{28}$ tests.

alternative test suites. We focus on the case where we want to choose between different finite test suites that have the same number of inputs (and so, most likely, the same execution cost). The problem studied is relevant in a number of contexts. For example, we might build a test suite in an incremental manner. When choosing a next test case to add we will be comparing test suites of the same size (the current test suite extended by the different test cases that could be added) and will choose the test suite that we expect to be most effective. Thus, incremental approaches to building a test suite involve the comparison of test suites of the same size. Measures that compare test suites might also be used to help guide test suite generation since, for example, the measures could form fitness functions to be used within a search-based approach. The problem is also relevant in the context of regression testing, where we typically have to repeatedly use a test suite. Ideally, one executes the entire test suite in regression testing but often this is too expensive, with this having led to a significant body of work regarding the problem of selecting a ‘best’ subset (see, for example, [3–5]).

This paper considers the situation in which the SUT is a black-box; we know its input and output alphabets but have no additional information. As a result, in choosing a test suite we cannot use information about the internal structure of the SUT. A side-effect of this is that we will not have access to information about the coverage of the SUT achieved by a test suite. However, we assume that we do have a specification of the system that we want to build. In order to simplify the presentation, we assume that the specification is given by a Finite State Machine (FSM) but the approach can be adapted to deal with other state-based formalisms, in particular, those containing data.

The choice of FSMs as a formalisation was motivated by the fact that they have been used in a number of areas. The early work largely concerned protocol conformance testing [6,7] and hardware (processor) testing, since processor designs are FSMs (see, for example, [8]). FSM-based techniques have also been used in testing web-services and web-based applications [9,10]. It is important to observe that it is not necessary for the user to produce an FSM specification; the specification may be in some other state-based formalism, such as state-charts, with a model being mapped to an FSM that represents its semantics (possibly after some abstraction). This makes FSM-based approaches applicable to a wide range of state-based specifications, such as those used in the embedded systems industry. The value of such an approach was also demonstrated when used to test a number of Microsoft Windows protocols [11]. Recent work has shown that FSM-based test generation techniques can be applied when testing from a class of rather more expressive models (reactive I/O-state-transition systems, RIOSTS) [12]. The benefit is that RIOSTS can also be used with a range of embedded systems [12], with the approach having been evaluated on part of the European Train Control System and also an airbag controller [13]. Finally, it is worth mentioning that there are several tools that can be used to specify and analyse FSMs (for example, fsm-lib-cpp,² automatalib [14] and OpenFST [15]).

This paper describes a novel *information theoretic measure* and proposes its use to inform the choice between test suites. Specifically, we define a notion, *Biased Mutual Information (BMI)*, inspired by Mutual Information [16], and use it to compare test suites of similar length. The idea behind the definition of BMI is that two tests that share a large amount of information will be more likely to explore the same paths of an FSM, applying the same inputs and expecting the same outputs. The intended goal of BMI is thus to *indirectly* maximise diversity and is motivated by it having been widely recognised that diversity has a strong impact on test quality [17–20]. Our hypothesis was that a test suite with lower BMI will tend to be more effective (be more likely to find faults) because lower BMI implies that the tests in the test suite share less information and, therefore, they explore more behaviours of the FSM. In order to evaluate this hypothesis, we used mutants, that

simulate faults. Specifically, given an FSM and a pair of test suites, we checked whether the test suite with lower BMI killed more mutants. Our experiments revealed that minimising BMI led to test suites that kill more mutants most of the time. Moreover, we obtained a (negative) correlation between BMI and mutation score. We also found that the time needed to compute BMI is negligible when compared to the time typically needed to apply a single test suite. The consequence of this fact is that it will often be worth *spending* some time to decide between two test suites, instead of applying both of them, if, as usual, resources and time are scarce. These results suggest that BMI can be used to guide testing towards test suites that are likely to be more effective.

Interestingly, in additional experiments it was found that BMI outperformed a previous information theoretic approach, the test set diameter (TSDm) measure [21], when selecting between two randomly generated test suites. We also compared our measure with a notion of transition coverage. Specifically, we explored the question of which measure was most effective when used to choose between two test suites: choosing the test suite with smaller BMI or the test suite with higher transition coverage. Since transition coverage has more information available (the states from which inputs are applied), it was not surprising that transition coverage was slightly more effective. However, BMI was found to be considerably faster. As a result, in the context of testing from an FSM, there is a trade-off between effectiveness and speed. Note that, although the comparison with transition coverage was interesting, the aim of the work described in this paper was to produce a general method that can be used in a range of scenarios. This includes scenarios in which, for example, we do not have a complete specification but we can estimate the frequency of input/output pairs, which is the only information required to compute BMI. There is potential to use (random) sampling to estimate such frequencies.

Although it will become clearer when we give the formal definition of BMI, we will briefly explain why we need a *bias* in our notion. Essentially, in order to compute our measure we need to know the frequency of occurrence of each input/output pair in the underlying FSM. Since the formulation of Mutual Information applies a logarithm over that frequency, if we have only one occurrence of an input/output pair (i, o) in the FSM, then we obtain $\log(1) = 0$. This produces two undesirable consequences. First, we obtain the smallest possible value for (i, o) while we want to give it the highest weight. Second, when we combine this weight with other values by multiplying them, we obtain 0, which leads to final values that are uninformative. The introduction of a *bias* solves this problem.

The rest of the paper is structured as follows. In Section 2 we review basic concepts and notation used in the paper. In Section 3 we present related work. In Section 4 we formally define our measure and explore some of its properties. Section 5 reports on the experiments and Section 6 discusses threats to validity. In Section 7 we discuss some decisions that we took during the research presented in this paper. Finally, in Section 8 we provide conclusions and discuss future work.

2. Preliminaries

In this paper, systems will be modelled as *Finite State Machines* (FSMs). In order to define an FSM, we first introduce some 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 .

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

² <https://github.com/agbs-uni-bremen/fsm-lib-cpp>.

An FSM is a (finite) labelled transition system in which every transitions has a label in the form of an *input/output pair* (a pair containing an input action and an output action). We use this formalism to define specifications.

Definition 1. A *Finite State Machine* (FSM) is represented by a tuple $M = (Q, q_{in}, I, \mathcal{O}, T)$ in which Q is a finite set of states, $q_{in} \in Q$ is the initial state, I is a finite set of input actions, \mathcal{O} is a finite set of output actions, and $T \subseteq Q \times (I \times \mathcal{O}) \times Q$ 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 $i \in I$ there exists at most one pair $(q', o) \in Q \times \mathcal{O}$ such that $(q, i/o, q') \in T$.

We assume that FSMs are deterministic; this makes the work compatible with the previously devised information theoretic (white-box) TSDm measure [21]. In particular, it allows us to run experiments that compare the proposed approach with TSDm.

An FSM can be represented by a diagram in which nodes represent states of the FSM and transitions are represented by arcs between the nodes. We use an incoming edge with no source to denote the initial state. In our case, all states are final as long as they are reachable from the initial state.

We will assume the *minimal test hypothesis* [22]: the SUT can be modelled as an (unknown) object described in the same formalism as the specification (here, an FSM). Note that we do not need to have access to this description; we are in a black-box testing framework and only assume the existence of such an FSM. In principle, we could weaken this hypothesis to simply assume that each time the SUT receives a sequence of input actions, it returns a sequence of output actions.

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

Definition 2. Let $M = (Q, q_{in}, I, \mathcal{O}, T)$ be an FSM, $\sigma = (i_1, o_1) \dots (i_k, o_k) \in (I \times \mathcal{O})^k$ 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 $\text{traces}(M)$ the set of traces of M . Note that $\epsilon \in \text{traces}(M)$ for every FSM M .

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

Definition 3. Let $M = (Q, q_{in}, I, \mathcal{O}, T)$ be an FSM. We say that $t = (i_1, o_1) \dots (i_k, o_k) \in (I \times \mathcal{O})^k$ is a *test* for M if $t \in \text{traces}(M)$. The *length* 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 $t = (\lambda, \mu) \in (I^+ \times \mathcal{O}^+)$. 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 *length* of the test suite is the sum of the lengths 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 $(\lambda, \mu') \in \text{traces}(M')$ 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.

Intuitively, a test (λ, μ) for M denotes that the application of the sequence of input actions λ to a correct system (with respect to M) should lead to the sequence of output actions μ . Note that if we allowed non-determinism, then the previous inequality must be appropriately replaced to express that a behaviour of the SUT must be one of those of the specification, and we will have a notion of conformance similar to ioco [23].

The concept of Test Set Diameter (TSDm) [21], which will be used as the state-of-the-art measure to compare with, is derived from Kolmogorov complexity [24]. The *Kolmogorov complexity* of a string is the length of the shortest program that produces that string. It has been shown that Kolmogorov complexity can be approximated using Normalised Compression Distance [25].

Definition 4. 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 $\text{ncd}(x, y)$ 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 5. 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 $\text{NCD}(X)$ the Normalised Compression Distance of X and we define it as

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

where

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

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 Test Set Diameter of a test suite is defined in terms of the NCD of a (multi-)set of tests. We can consider different *multisets* for computing this metric. For example, we could use the multiset of test inputs (Input-TSDm), the multiset of test outputs (Output-TSDm) or even the multi-set of execution traces (Trace-TSDm). In this paper, as recommended in the original work [21], we used ITSDm to drive test selection.

It is important to note that Test Set Diameter is the current baseline for the use of Information Theory to direct the generation and selection of test suites in black-box testing. An extensive study [26] found that Test Set Diameter outperformed many alternatives. Specifically, the study shows that, for 5 well-known programs (Grep, Sed, Flex, Make and GZip), ITSDm is the best alternative, obtaining fault detection rates between 85% and 95%. Moreover, they observed that ITSDm is able to compete with white-box techniques, obtaining differences in fault detection rates of less than 2%, which is remarkable given the fact that ITSDm, as a black-box technique, works with less information than white-box techniques.

Finally, we recall the classical definition of mutual information [16], which we use as an inspiration for our measure.

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

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

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

3. Related work

There are many techniques for generating a test suite from an FSM specification. Possibly the first work in this area was the seminal paper by Moore, published in 1956, that described an overall framework [27]. Later, in 1964, Hennie [28] published a test generation algorithm. Hennie’s algorithm required the specification to be a deterministic, completely-specified FSM that has a special type of sequence, called a distinguishing sequence.³ Later, a more general test generation algorithm was published [29,30], with this only requiring that the specification is a deterministic, completely-specified FSM.⁴

The area of FSM-based test generation has been extended in a number of directions. The techniques mentioned above are *complete* in the sense that they return test suites that determine correctness as long as the SUT is equivalent to an (unknown) FSM in some well-defined fault domain; the fault domain typically places an upper bound on the number of states of the minimal FSM that represents the behaviour of the SUT. A number of other complete test generation algorithms that have been devised, typically with a focus on producing relatively small complete test suites (see, for example, [31–33]). There are also complete techniques for testing from other classes of FSM such as (possibly non-deterministic) partial FSMs (see, for example, [34–36]), FSMs representing distributed systems (see, for example, [37,38]), probabilistic FSMs (see, for example, [39,40]), and stochastic FSMs (see, for example, [41]). It is not always feasible to produce and use complete test suites and so there are also approaches that aim to cover the FSM specification in some way (see, for example, [42]). The interest in FSM-based testing is motivated by the ability of FSMs to suitably model many classes of system, possibly after an abstraction has been applied. Sometimes, however, it does not make sense to abstract out data (for example, where this data determines which transitions can be executed). There has thus been interest in testing from extended FSMs (EFSMs), which allow data (see, for example, [43–47]), including work on testing from Stream X-machines [48].

Interestingly, although there has been much work on testing from both FSMs and EFSMs, it appears that notions of diversity have not been utilised in these areas. A potential advantage of diversity-based approaches is that they can be used with any test budget: given a bound on the overall test execution cost/time available, one can aim to find the most diverse test suite whose cost does not exceed this bound. Diversity-based approaches are thus potentially extremely flexible and it should be possible to apply them with FSMs, EFSMs or FSMs extended with other features such as probability and time. However, as a first step this paper restricts attention to FSMs.

Information Theory has already been used in testing [49–58]. In particular, the problem of choosing among different test suites has been addressed before [17,21,59]. However, as far as we are aware, this is the first work in which a measure inspired by Information Theory has been used to choose between test suites in a black-box testing framework, in particular, in testing from FSMs.

4. Biased mutual information

As previously explained, our goal is to increase test suite diversity with the hope that this will be reflected in the capability of the test suite to detect faults. Lower values of mutual information should be associated with higher diversity. We will consider both the input part of the test and the expected output with the goal of also increasing output diversity [60]. We can utilise output as well as input since we have a specification.

³ A distinguishing sequence is an input sequence that produces different output sequences from the different states of the FSM.

⁴ It also requires the specification to be minimal but any deterministic, completely-specified FSM can be converted into an equivalent minimal deterministic, completely-specified FSM.

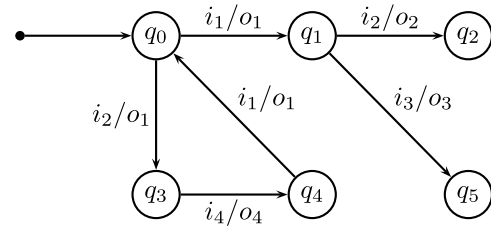


Fig. 1. Example of FSM.

The intuition behind maximising diversity as a goal is very simple. Assume that we have an FSM with a set of input/output pairs labelling its transitions. If we select two different input/output pairs, and observe no failure, then we know that this selection traversed two different transitions of the FSM that represents the SUT. However, selecting one input/output pair twice leads to a scenario where we might traverse the same transition of the FSM twice (in particular, FSMs can have loops and so can return to a state met earlier). One might argue that this happens with probability $\frac{1}{m}$, where m is the number of times that the input/output pair labels a transition of the FSM; even for large m , we have a non-zero probability of traversing a transition more than once. This scenario also shows that we have to be careful when looking for measures of diversity, as the probability decreases when m increases. For example, the cases where $m = 2$ should be rather different to the case where $m = 200$. Therefore, one should not automatically discard test suites where a pair appears more than once. Instead, we should take into account the number of times a label appears in the FSM specification. We illustrate this with a simple example.

Example 1. Consider the FSM given in Fig. 1 and its test suites

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

and

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

On the one hand, \mathcal{T}_1 has a mutual information of 0. Even though the input action i_2 appears twice in the test suite, we know that the pairs (i_2, o_1) and (i_2, o_2) represent different behaviours. On the other hand, \mathcal{T}_2 has a non-zero mutual information (applying the classical formula given in Definition 6 we have that this value is equal to 0.53). Therefore, a measure based on mutual information should choose the first test suite.

Initially, we would like to compute the mutual information of two tests. Each test is a sequence of input/output pairs. If we abstract out the position of the pairs in the sequence, we obtain a set of pairs. Given two tests t_1 and t_2 , in order to compute the mutual information $I(\xi_{t_1}; \xi_{t_2})$ we need a definition of the probability distribution $\sigma_{\xi_i}(x)$ (see Definition 6). The first attempt was to give an *intuitive* definition of σ in which we used the uniform distribution as the probability distribution in the mutual information formula. That is, if a label appears in m transitions of the machine, then the probability of this label will be $\frac{1}{m}$; the probability that the pair x corresponds to a specific transition of specification M . Also, we replaced the notion of joint probability with a notion of *composition*. In this, if we have two transitions with the same input/output pair (i, o) , then the *composition* will return the common weight, that is, the weight of (i, o) ; otherwise, we multiply the weights. Alternatively, this composition can be seen as the product of the weights of each element reweighted by the number of repetitions of the input/output pair in the FSM.

Unfortunately, this choice does not induce a probability distribution over the pairs in the tests and so it is not a mathematically valid formulation of the mutual information between two tests. As a result, there is a need to explore alternatives. After several possibilities were

considered (discussed in Section 7), we found that none of them consistently gave better results, so we kept the initial intuition to not restrict ourselves to random variables. Then, in the next definition the different occurrences of σ do not refer to probability distribution functions, but we keep this notation to retain the structure of the original formulation. As a result of the above, we will use the term $\sigma_t(x)$, even though t does not explicitly appear in the right hand side of the following formulae, because t is an implicit parameter; $\sigma_t(x)$ is only defined for x if $x \in t$. In the rest of this section, recall that 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 7. Let $M = (Q, q_{in}, I, \mathcal{O}, T)$ be an FSM, t be a test for M and $x \in I \times \mathcal{O}$ be an input/output pair such that $x \in t$. We let:

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

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:

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

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:

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

Note that the intuition assumes a uniform distribution over the set of transitions of M with the same label. We could choose another distribution for those probabilities by, for example, increasing the probability associated with transitions that are reached from the initial state after fewer transitions. However, this would complicate the computation of the measure and preliminary experiments did not show a significant improvement. Therefore, if the *real* distribution is not known then we use a uniform distribution in order to aid simplicity. Note that uniform distributions also have desirable properties (in particular, this distribution maximises entropy [61]). Naturally, we should not use uniform distributions if we have evidence that they are inappropriate (i.e. because using the true distribution is known to lead to better results). For example, this is the case if we have probabilistic user models indicating the probabilities with which users choose inputs [62]. Next we give a result allowing us to simplify the formulation.

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 $x \in_{n_x} t_1$.

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

$$\sigma_{t_1, t_2}(x_1, x_2) \cdot \log_2 \frac{\sigma_{t_1, t_2}(x_1, x_2)}{\sigma_{t_1}(x_1) \cdot \sigma_{t_2}(x_2)} \quad (1)$$

we will distinguish two cases: $x_1 = x_2$ and $x_1 \neq x_2$. First, let us consider $x_1 = x_2$. Note that $\sigma_{t_1}(x_1) = \sigma_{t_2}(x_2)$, because these values depend only on M and x . In addition, the composition of an element of a test and itself is the probability of the element (as we stated in Definition 7). Therefore, $\sigma_{t_1, t_2}(x, x) = \sigma_{t_1}(x)$. Now, taking into account Definition 7

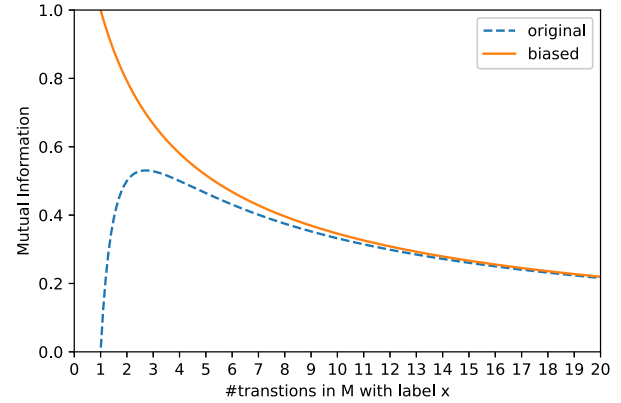


Fig. 2. Measure comparison plot.

we have that if $x_1 = x_2$ then the previous term is equal to

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

and, simplifying, we conclude that if $x_1 = x_2$, then the term given in Eq. (1) is equal to

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

Now, let us consider $x_1 \neq x_2$. In this case, $\sigma_{t_1, t_2}(x_1, x_2) = \sigma_{t_1}(x_1) \cdot \sigma_{t_2}(x_2)$ and, therefore, the term given in Eq. (1) becomes equal to 0 because we have $\log_2(1)$ as one of the factors.

Putting together these two cases in the Mutual Information formula given in Definition 7, we obtain the following expression:

$$I(t_1; t_2) = \sum_{x_2 \in t_2} \sum_{\substack{x_1 \in t_1 \\ x_1 = x_2}} \frac{\log_2(m_x)}{m_x}$$

Note that in the inner sum, for a given x_2 , we are always adding the same value. Therefore, we can simplify it as a multiplication of that value times the number of times it is added. This last factor corresponds to the number of times $x_1 = x_2$ appears in the test t_1 , that is, the value n_x such that $x_2 \in_{n_x} t_1$. This results in the following expression:

$$I(t_1; t_2) = \sum_{x_2 \in t_2} n_x \cdot \frac{\log_2(m_x)}{m_x}$$

that can easily be rewritten as:

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

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

An important remark about this formula is that it is not monotonic and it is equal to 0 if all the transitions of the specification have different input/output pairs. Since we are interested in values that are useful when comparing test suites (therefore, we need monotonicity and we should avoid “division by zero”), we solve this problem with a simple transformation. The dashed curve in Fig. 2 shows the behaviour of the previous formula. We make a small translation in the X axis of the logarithm of the formula, so that its behaviour is the one given by the solid curve.

Definition 8. 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 $x \in_{n_x} t_1$.

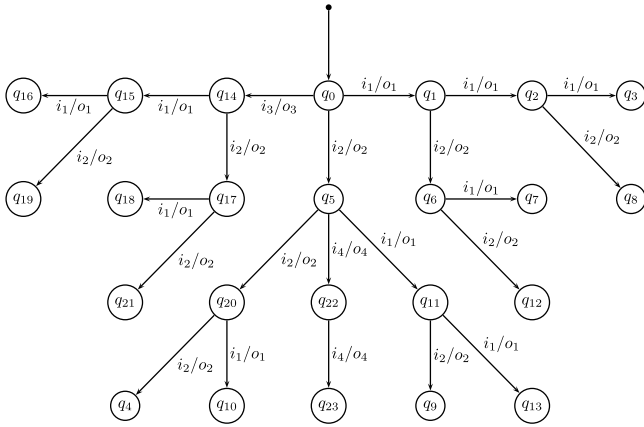


Fig. 3. Another example of FSM.

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

Example 2. Consider the FSM M depicted in Fig. 3 and the test suites

$$\mathcal{T}_1 = \{t_1 = (i_2 i_4 i_4, o_2 o_4 o_4), t_2 = (i_3 i_2 i_1, o_3 o_2 o_1)\}$$

and

$$\mathcal{T}_2 = \{t_3 = (i_3 i_1 i_1, o_3 o_1 o_1), t_4 = (i_3 i_2 i_2, o_3 o_2 o_2)\}$$

Note that the only pair appearing in both t_1 and t_2 is (i_2, o_2) (this input/output pair appears 9 times in M); similarly, the only common pair for t_3 and t_4 is (i_3, o_3) (this appears once in M). The (biased) mutual information of each test suite can be computed as follows:

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

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

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

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

Therefore, the first test suite would be *better* if we consider biased mutual information, but would be *worse* if we consider mutual information. In principle, we should prefer \mathcal{T}_1 because it is more likely that it will check more transitions than \mathcal{T}_2 . In fact, in this example we know that the second test suite will traverse the same transition twice.

The biased mutual information between two tests will be used as the basis of a measure for a test suite. The idea is that we need to compute the cumulative amount of biased mutual information between all the pairs of tests. 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 $\mathcal{T} = \{t_1, \dots, t_k\}$, then we apply Definition 8 to all the pairs of tests included in \mathcal{T} :

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

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

In addition, we need to take into account repetitions in each test belonging to the test suite. Intuitively, we should penalise test suites that have tests with many repeated input/output pairs, even if these pairs do not appear in other tests of the suite. We present a simple example to motivate why we need to take this into account.

Example 3. Consider the test suites

$$\mathcal{T}_1 = \{t_1 = (i_1 i_1 i_1, o_1 o_1 o_1), t_2 = (i_2 i_3 i_4, o_2 o_3 o_4)\}$$

and

$$\mathcal{T}_2 = \{t_3 = (i_1 i_7 i_9, o_1 o_7 o_9), t_4 = (i_2 i_3 i_4, o_2 o_3 o_4)\}$$

There is no mutual information between t_1 and t_2 (similarly between t_3 and t_4). However, we should take into account the repetition in t_1 and we should prefer \mathcal{T}_2 .

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 our formula for the *Biased Mutual Information* (BMI) of a test suite.

Definition 9. Let M be an FSM and $\mathcal{T} = \{t_1, \dots, t_k\}$ be a test suite for M . We have

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

In the next section we describe the experiments used to evaluate BMI.

5. Empirical evaluation

In this section we describe the experiments used to evaluate the proposed measure (BMI) and assess its ability to help the tester to choose *good* test suites. We performed several experiments with different models and in each we:

- *Derived test suites* by randomly traversing the specification.
- *Generated mutants* of the specification. The rationale is that test suite \mathcal{T}_1 is better than test suite \mathcal{T}_2 if \mathcal{T}_1 kills⁵ more mutants than \mathcal{T}_2 .

In the experiments we used 241 real FSMs from a benchmark [63]. We also used randomly generated FSMs, with this providing us with a larger set of subjects and also the ability to explore how performance changes as we vary FSM properties such as the alphabet size. There were two main reasons for us using randomly generated mutants in the experiments. First, the benchmark FSMs do not come with faulty versions and, indeed, we are not aware of any FSM benchmark that includes faulty programs or models. Second, we do not have a general *fault model* representing potential (faulty) implementations of a specification. If we had such a fault model, then we could have used it instead of the randomly generated mutants. Observe that since we used FSMs it is possible to identify equivalent mutants in low-order polynomial time and so we were able to remove them. Therefore, we have an implicit fault model, which is all non-equivalent mutants that can be generated using our mutation operators.

In the rest of this section, first, we state the research questions. We then explain the experimental design before discussing the results of the experiments and what these tell us about the research questions. All the code, benchmarks and results from the experiments are available at <https://github.com/Colosu/BMI-test-selection>.

⁵ A test suite kills a mutant if the test suite contains a test case such that the specification and mutant produce different outputs in response to this test case.

Table 1
Properties of the FSM sets.

Set number	Set size	Range # states	Range outgoing transitions	Range input alphabet size	Range output alphabet size
1	241	[3, 156]	[0, 130]	[2, 130]	[2, 65]
2	100	50	[2, 5]	5	5
3	100	50	[2, 5]	25	25

5.1. Research questions

In order to *evaluate* BMI, we first checked how *well* it works.

The main motivation for the work described in this paper is that we would like to be able to choose between alternative test suites. Thus, as a first step we assessed whether BMI is effective in guiding such a choice.

Research Question 1. *Given a pair of test suites with the same length, will the one with lower Biased Mutual Information tend to have higher fault detection ability?*

We also wanted to address the related question of whether lower levels of Biased Mutual Information are associated with higher fault coverage; whether a correlation exists.

Research Question 2. *Are lower levels of Biased Mutual Information associated with higher fault coverage?*

If we have a positive answer (with statistical significance) to the above questions, then we would like to see how BMI compares to the currently proposed Information Theory approach, the test set diameter (TSDm) measure.⁶

Research Question 3. *Do test suites selected by BMI have higher fault coverage than those selected by test set diameter (ITSDm)?*

Finally, we wanted to check whether the time taken to compute BMI might be better used in executing additional tests.

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

By the length of a test suite we mean the sum of the lengths of the tests (sequences).

5.2. Experimental subjects

We designed a series of experiments in order to address the research questions. We used three sets of FSMs.

Set1 A benchmark of 335 FSMs recently collected [63]. From this, we selected the 241 deterministic FSMs. These FSMs represent real-world systems, ranging from the classical *coffee machine* to more complex systems such as ATMs, circuits, network protocols and X-ray systems.

Set2 A set of 100 randomly generated FSMs with 50 states and input and output alphabets with 5 elements. Each state was given a random number, between 2 and 5, of outgoing transitions (except state 50, which had no outgoing transitions).

Set3 Another set of 100 randomly generated FSMs with 50 states and input and output alphabets of size 25. Again, each state was given a random number, between 2 and 5, of outgoing transitions (except state 50, which had no outgoing transitions). These were used in order to explore the effect of having a larger input alphabet.

Table 2
Summary of the results of the experiment with real FSMs.

Value of A	# runs [0.5, 0.6]	# runs [0.6, 0.7]	# runs [0.7, 0.8]	Min value	Max value	% success (mean)
1/5	41	9	0	0.536232	0.616319	58.4039%
2/5	12	38	0	0.556391	0.649412	61.6790%
3/5	1	48	1	0.597360	0.703226	65.1793%
4/5	3	44	3	0.588235	0.712062	64.0050%

In **Table 1** we summarise the main parameters of the FSMs in these three sets.

We used the first set (Set1) in order to evaluate BMI on real FSMs. This provided FSMs with varying numbers of states and varying alphabet sizes. While this is useful, there is only a limited number of FSMs and the use of these also limits the control we have (e.g. if we want to vary the size of the alphabets). This motivated the use of the other two sets of FSMs.

In order to assess the fault detection ability of a test suite, we generated mutants of the specification. We only use the mutation operator that modifies the target state of a transition because a preliminary experiment showed that faults produced by mutations on the labels of the transitions are much easier to detect during testing. The use of mutants is a standard approach to assessing fault detection effectiveness; see the discussion in the introduction. 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 $t = t' i/o t''$ for an input action i , output action o and prefix t' such that t' takes N to a state from which there is no transition with input action i .

For an FSM specification M , we generated test suites by randomly traversing M . The FSMs typically had at least one sink state, with no outgoing transitions: if such a sink state was reached then a new test was started (at the initial state).

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

5.3. Using BMI to help guide test suite choice

These experiments explored whether BMI is valid for our purposes. Initially, we used the real FSMs (Set1). For each FSM, we generated two test suites, from the FSM, of length $num_states \times alphabet_size \times A$ where we used the following values for A : $\{\frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}\}$. Recall that we generated a test suite by randomly traversing the FSM, starting a new test (sequence) whenever a sink state is met; the length of a test suite is thus the sum of the lengths of the individual tests (sequences). We let the test suite length depend on the number of states because the number of states of the FSMs in the benchmark varied, from 3 to 156 states; we used a length proportional to the potential maximum number of transitions of the FSM.

For each pair of test suites, we computed BMI values to determine which test suite would be selected if we chose the one with lower BMI. We also produced 100 mutants of the original FSM; recall that the mutation operator modifies the target state of a transition. We applied each test suite to all of the mutants to determine how many mutants were killed by a test suite. We repeated this procedure 10 times for each FSM, obtaining 10 pairs of test suites, and then computed how many times the test suite selected using BMI was the one that killed more mutants (the *percentage of success*). Where two test suites in a pair had the same mutation score or the same BMI, we replaced this pair of test suites with another randomly generated pair. We repeated this process 50 times for each value of A .

Table 2 summarises the results (the full results are given in **Tables 3–6**). We always had a percentage of success that was higher than 58%, and in most cases it was higher than 60%. These results suggest that BMI performs better than random selection and that the difference increases for larger test suites.

⁶ In Section 5.5 we describe how TSDm has been used in test selection.

Table 3
Percentages of success of our selected test suite in the experiment with real FSMs and $A = 1/5$.

Run #	Percentage of success
1	0.5761467889908257
2	0.5682242990654206
3	0.5600739371534196
4	0.5918727915194346
5	0.5875
6	0.6102941176470589
7	0.5652173913043478
8	0.5804701627486437
9	0.5652985074626866
10	0.5958254269449715
11	0.5952813067150635
12	0.5882352941176471
13	0.5892547660311959
14	0.5863309352517986
15	0.5678571428571428
16	0.5948905109489051
17	0.569620253164557
18	0.563963963963964
19	0.6133333333333333
20	0.5893805309734513
21	0.6077348066298343
22	0.5871212121212122
23	0.6007604562737643
24	0.5868372943327239
25	0.5979202772963604
26	0.5769944341372912
27	0.5989010989010989
28	0.5362318840579711
29	0.6040145985401459
30	0.5471014492753623
31	0.5886939571150097
32	0.585278276481149
33	0.6046099290780141
34	0.6163194444444444
35	0.5594795539033457
36	0.5659050966608085
37	0.5981981981981982
38	0.5730129390018485
39	0.5801801801801801
40	0.5884543761638734
41	0.5748613678373382
42	0.5545774647887324
43	0.5893805309734513
44	0.5981818181818181
45	0.5822550831792976
46	0.5801801801801801
47	0.5640569395017794
48	0.5786713286713286
49	0.6106870229007634
50	0.6083032490974729

Table 4
Percentages of success of our selected test suite in the experiment with real FSMs and $A = 2/5$.

Run #	Percentage of success
1	0.6290726817042607
2	0.5892857142857143
3	0.6421319796954315
4	0.6048192771084338
5	0.6456310679611651
6	0.6059113300492611
7	0.625
8	0.592964824120603
9	0.6132075471698113
10	0.6363636363636364
11	0.6122931442080378
12	0.6222222222222222
13	0.6144278606965174
14	0.6123456790123457
15	0.6327014218009479
16	0.6015037593984962
17	0.6374407582938388
18	0.5949367088607594
19	0.6186666666666667
20	0.6363636363636364
21	0.6155717761557178
22	0.6425
23	0.6289156626506024
24	0.6491228070175439
25	0.5924932975871313
26	0.6428571428571429
27	0.6109785202863962
28	0.5772946859903382
29	0.6169665809768637
30	0.5815602836879432
31	0.6388140161725068
32	0.6341463414634146
33	0.6486486486486487
34	0.6275
35	0.645933014354067
36	0.6265356265356266
37	0.5944584382871536
38	0.6268656716417911
39	0.5965770171149144
40	0.6216867469879518
41	0.6035353535353535
42	0.6191646191646192
43	0.5891089108910891
44	0.5758293838862559
45	0.628140703517588
46	0.6116504854368932
47	0.6494117647058824
48	0.556390977443609
49	0.596401028277635
50	0.6218274111675127

We then carried out experiments to assess BMI in a more controlled scenario, using the following approach. We started with the 100 randomly generated FSMs in Set2 (50 states, between 2 and 5 outgoing transitions for each state, and alphabets of size 5). We used the process described above but this time generating 1000 mutants and fixing the test suite length to 100, because now we have a fixed number of states. We repeated the whole process 50 times, obtaining 50 different percentages of success. Since each percentage of success was obtained from 100 different FSMs, the calculation of a mean percentage of success required 5000 repetitions.

One issue in the design of the experiments was whether we should check for redundancy in a test suite, where *redundancy* corresponds to the case where one test (sequence) is a prefix of another. We carried out experiments to explore the effect of this choice on our results. In these experiments, we had two scenarios: (1) randomly generated test suites without checking for prefixes; and (2) discarding prefixes when randomly generating test suites. Table 7 provides a summary of the results (full results can be found in Table 8), where we can see that very similar results were obtained in the two scenarios. Also, we can

see that in both cases, all the values are in the range [0.5, 0.8), with the majority of them belonging to the range [0.6, 0.7). These observations suggest that the effectiveness of BMI is not affected by whether we allow the test suite to have redundancy. As a result, the remaining experiments consider the case where we do not allow the test suite to have redundant tests (in practice, this will typically be the case).

We repeated the experiments using the randomly generated FSMs with input alphabet of size 25 (Set3), using test suites without redundancy. The mean percentage of success was 75.0605% (full results can be found in Table 9). These results are much better than the results from the previous experiments and this suggests that BMI works better in FSMs with larger input alphabets.

Over all the obtained results, we performed an homogeneity of variance check and a statistical hypothesis test whose null hypothesis was that random selection and ordering on BMI give similar results. The homogeneity check showed that there is no homogeneity, so we cannot use the ANOVA test. Instead, we applied a Kruskal–Wallis H-test where we tested whether the results of the experiment are distributed as the distribution that would arise from random selection. We assumed that

Table 5
Percentages of success of our selected test suite in the experiment with real FSMs and $A = 3/5$.

Run #	Percentage of success
1	0.6346153846153846
2	0.6633663366336634
3	0.6482758620689655
4	0.6782334384858044
5	0.5973597359735974
6	0.6382252559726962
7	0.6054421768707483
8	0.6611295681063123
9	0.6331168831168831
10	0.6820987654320988
11	0.6254071661237784
12	0.6452702702702703
13	0.6528662420382165
14	0.6807817589576547
15	0.6666666666666666
16	0.6511627906976745
17	0.6524590163934426
18	0.6372881355932203
19	0.6910828025477707
20	0.6461038961038961
21	0.6918032786885245
22	0.625
23	0.6587837837837838
24	0.6
25	0.6221498371335505
26	0.7032258064516129
27	0.6710963455149501
28	0.6514084507042254
29	0.6478405315614618
30	0.6366666666666667
31	0.6577181208053692
32	0.6741935483870968
33	0.6806451612903226
34	0.6326530612244898
35	0.6697530864197531
36	0.6825396825396826
37	0.6317567567567568
38	0.6419354838709678
39	0.6489028213166145
40	0.6477987421383647
41	0.6390728476821192
42	0.6861538461538461
43	0.6515151515151515
44	0.6495176848874598
45	0.6513157894736842
46	0.6389776357827476
47	0.6457564575645757
48	0.6332288401253918
49	0.6261980830670927
50	0.6914498141263941

Table 6
Percentages of success of our selected test suite in the experiment with real FSMs and $A = 4/5$.

Run #	Percentage of success
1	0.6359832635983264
2	0.622568093385214
3	0.6475409836065574
4	0.7021276595744681
5	0.6509803921568628
6	0.7120622568093385
7	0.636734693877551
8	0.61003861003861
9	0.6126126126126126
10	0.673469387755102
11	0.6204081632653061
12	0.6653061224489796
13	0.6282051282051282
14	0.664
15	0.6613545816733067
16	0.5933609958506224
17	0.6363636363636364
18	0.6428571428571429
19	0.6336206896551724
20	0.6170212765957447
21	0.6223175965665236
22	0.6386554621848739
23	0.6788617886178862
24	0.5882352941176471
25	0.648068669527897
26	0.628318584070964
27	0.64
28	0.6529680365296804
29	0.6567796610169492
30	0.6491228070175439
31	0.6533333333333333
32	0.6074380165289256
33	0.6260504201680672
34	0.6582278481012658
35	0.6443514644351465
36	0.6317991631799164
37	0.6125
38	0.6313725490196078
39	0.7086614173228346
40	0.6440677966101694
41	0.6173913043478261
42	0.6592920353982301
43	0.6390041493775933
44	0.6942148760330579
45	0.6220472440944882
46	0.6473029045643154
47	0.5966386554621849
48	0.6091954022988506
49	0.6115702479338843
50	0.6111111111111112

the distribution that arises from random selection would be a Poisson distribution with $\lambda = 50$.⁷ Then, we computed the p-values and, in all cases, the null hypothesis was rejected with a p-value extremely close to zero. We also performed an effect size measure.⁸ We computed Cliff's delta statistic and we obtained large effect sizes (all higher than 1). This reinforces the conclusions derived from the previously computed p-values.

5.4. Assessing correlation

The results described above provide evidence that BMI works when choosing between two test suites. Ideally, we would also like a measure that correlates with the fault detection ability of a test suite.

⁷ In each selection, we have a binomial distribution (50% probability of choosing the better test suite), and the repetition of this binomial distribution produces a Poisson distribution of $\lambda = 50$.

⁸ <https://github.com/txt/ase16/blob/master/doc/stats.md>.

We repeated the experiment described in the previous section using the same FSMs (Set2) but different test suites and mutants. For each FSM, we then computed the correlation between BMI and the mutation score (i.e. the number of killed mutants). We obtained a mean Pearson correlation of -0.369134 and a mean Spearman correlation of -0.356978 . The fact that the correlations are negative implies that lower BMI is associated with higher mutation scores, addressing RQ2. These correlations are consistent with the results from the previous experiment. The full results are displayed in Table 10.

We repeated the experiment with the FSMs with alphabet size 25 (Set3). The results show a stronger (negative) correlation. Specifically, we obtained a mean Pearson correlation of -0.650256 and a mean Spearman correlation of -0.634711 . The full results can be found in Table 11.

Again, we checked whether the results are statistically significant. Here our null hypothesis is that there is no correlation (that is, the correlation is equal to 0). First, we performed an homogeneity of variance check that told us that there is no homogeneity, and then we applied a Kruskal–Wallis H-test where we tested whether the results of

Table 7
Summary of the results of the experiment with controlled FSMs with alphabet of size 5.

Type of test suite	# runs [0.5, 0.6)	# runs [0.6, 0.7)	# runs [0.7, 0.8)	Min value	Max value	% success (mean)
<i>with repetitions</i>	15	32	3	0.540816	0.714286	62.2402%
<i>without repetitions</i>	13	34	3	0.520408	0.737374	62.4924%

Table 8
Percentages of success of our selected test suite in the experiment with controlled FSMs with alphabet of size 5.

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

Table 9
Percentages of success of our selected test suite in the experiment with controlled FSMs with alphabet of size 5, and associated time costs.

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

the experiment are distributed as the distribution that would arise from random selection. We assumed that the distribution that arises from random selection would be a normal distribution of $\mu = 0$ and $\sigma = 0.1$. Then, we computed the final p-values that state the significance of our results. Again, the null hypothesis was rejected, with the returned p-values being very close to zero. We also performed an effect size measure, computing the Cliff's delta statistic. We obtained large effect sizes (all higher than 0.8). This reinforces the conclusions derived from the previously computed p-values.

5.5. Comparison with TSDm

We performed additional experiments to compare BMI with a previous information theoretic proposal: the Test Set Diameter (TSDm) measure.

In order to compare the measures, we started by using Set2 (100 FSMs with alphabets of size 5). For each FSM, we randomly generated two test suites, as in previous experiments. We then computed the ITSDm and BMI values of both test suites and recorded which one was better for each measure. We then produced 1000 mutants of the original FSM. As before, mutations changed the target state of a

Table 10
Correlation between BMI and mutation score with alphabet size of 5.

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

Table 11
Correlation between BMI and mutation score with alphabet size of 25.

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

transition. Finally, we computed the mutation score of each test suite.⁹ We performed this procedure for each FSM and then computed how many times the test suite selected using BMI had the higher mutation score, how many times it was the one selected using ITSDm, and how many times both measures selected the same test suite. Additionally, in this last case we checked how many times the test suite selected was the one with the higher mutation score, and how many times it was the one with lower mutation score. We presented the results as percentages. We repeated the whole process until we obtained 50 different sets of the corresponding four percentages (each percentage was obtained from 100 different FSMs and 1000 mutants per FSM).

Additionally, we computed the mean execution time for the computation of each measure, in order to compare the differences in performance with the differences in execution time.

In the experiments, on average, 55.84% of the time BMI selected the best test suite, while ITSDm selected the best test suite only 46.72%

⁹ A mutant is killed if an input action is received in a state where it is not defined or the wrong output action is observed.

of the time. Moreover, in 30.16% of cases, BMI chose the best test suite and ITSDm chose the worst one, while the other way around only occurred 21.04% of the time. Finally, 23.12% of the time both measures failed to select the best test suite. Full results are displayed in [Table 12](#).

Regarding execution time, we found that the mean time taken to compute BMI was 0.043 s, while the corresponding value for ITSDm was 0.174 s. Thus, the proposed measure required 75.29% less time than ITSDm, while at the same time obtaining better results. These results are promising: the proposed measure tended to be a better guide, than ITSDm, when choosing between alternative test suites and took less time to compute.

We repeated this experiment using Set3 (100 FSMs with alphabets of size 25) and we again obtained better results for BMI. On average, 75.78% of the time BMI selected the best test suite, while for ITSDm this happened only 46.66% of the time. Moreover, 40% of the time BMI chose the best test suite while ITSDm chose the worst one, while the other way around only happened 10.88% of the time. Finally, 13.34% of the time both measures failed to select the best test suite. Full results are displayed in [Table 13](#). With respect to time, the execution time

Table 12
Percentages from the comparison of BMI with ITSDm (Set2).

Run #	BMI wins	ITSDm wins	Draw winning	Draw losing
1	0.29	0.26	0.18	0.27
2	0.3	0.17	0.24	0.29
3	0.39	0.18	0.2	0.23
4	0.32	0.18	0.28	0.22
5	0.24	0.21	0.27	0.28
6	0.36	0.15	0.25	0.24
7	0.33	0.22	0.24	0.21
8	0.28	0.23	0.25	0.24
9	0.31	0.17	0.26	0.26
10	0.31	0.24	0.26	0.19
11	0.33	0.22	0.25	0.2
12	0.29	0.19	0.23	0.29
13	0.33	0.27	0.16	0.24
14	0.27	0.23	0.23	0.27
15	0.36	0.24	0.23	0.17
16	0.32	0.24	0.23	0.21
17	0.26	0.22	0.3	0.22
18	0.31	0.23	0.26	0.2
19	0.26	0.16	0.29	0.29
20	0.25	0.23	0.23	0.29
21	0.31	0.29	0.23	0.17
22	0.27	0.24	0.26	0.23
23	0.27	0.24	0.29	0.2
24	0.32	0.25	0.26	0.17
25	0.29	0.21	0.28	0.22
26	0.35	0.24	0.2	0.21
27	0.31	0.22	0.19	0.28
28	0.31	0.25	0.23	0.21
29	0.28	0.23	0.29	0.2
30	0.22	0.21	0.35	0.22
31	0.3	0.17	0.29	0.24
32	0.35	0.18	0.26	0.21
33	0.29	0.21	0.22	0.28
34	0.34	0.16	0.27	0.23
35	0.32	0.15	0.3	0.23
36	0.36	0.14	0.24	0.26
37	0.31	0.19	0.3	0.2
38	0.35	0.18	0.29	0.18
39	0.34	0.16	0.22	0.28
40	0.32	0.23	0.26	0.19
41	0.23	0.21	0.35	0.21
42	0.23	0.3	0.17	0.3
43	0.25	0.24	0.27	0.24
44	0.27	0.2	0.29	0.24
45	0.31	0.19	0.3	0.2
46	0.29	0.18	0.32	0.21
47	0.21	0.22	0.33	0.24
48	0.28	0.24	0.27	0.21
49	0.37	0.19	0.21	0.23
50	0.32	0.16	0.26	0.26

Table 13
Percentages from the comparison of BMI with ITSDm (Set3).

Run #	BMI wins	ITSDm wins	Draw winning	Draw losing
1	0.38	0.11	0.38	0.13
2	0.43	0.15	0.31	0.11
3	0.38	0.12	0.38	0.12
4	0.36	0.1	0.4	0.14
5	0.41	0.06	0.39	0.14
6	0.33	0.2	0.34	0.13
7	0.5	0.1	0.29	0.11
8	0.39	0.14	0.35	0.12
9	0.39	0.07	0.37	0.17
10	0.36	0.12	0.38	0.14
11	0.42	0.13	0.32	0.13
12	0.41	0.05	0.4	0.14
13	0.44	0.1	0.33	0.13
14	0.37	0.1	0.38	0.15
15	0.38	0.07	0.37	0.18
16	0.36	0.13	0.37	0.14
17	0.51	0.09	0.31	0.09
18	0.43	0.09	0.38	0.1
19	0.36	0.16	0.37	0.11
20	0.35	0.1	0.4	0.15
21	0.4	0.1	0.35	0.15
22	0.44	0.13	0.32	0.11
23	0.41	0.06	0.41	0.12
24	0.44	0.11	0.29	0.16
25	0.39	0.1	0.38	0.13
26	0.4	0.11	0.34	0.15
27	0.45	0.07	0.31	0.17
28	0.34	0.14	0.41	0.11
29	0.45	0.14	0.31	0.1
30	0.41	0.06	0.36	0.17
31	0.38	0.12	0.36	0.14
32	0.42	0.12	0.32	0.14
33	0.38	0.07	0.42	0.13
34	0.4	0.1	0.32	0.18
35	0.43	0.09	0.28	0.2
36	0.4	0.13	0.35	0.12
37	0.26	0.12	0.47	0.15
38	0.43	0.09	0.34	0.14
39	0.42	0.15	0.34	0.09
40	0.5	0.1	0.28	0.12
41	0.41	0.15	0.34	0.1
42	0.44	0.08	0.4	0.08
43	0.42	0.06	0.4	0.12
44	0.31	0.09	0.45	0.15
45	0.46	0.09	0.31	0.14
46	0.42	0.17	0.31	0.1
47	0.43	0.1	0.32	0.15
48	0.31	0.13	0.42	0.14
49	0.37	0.14	0.37	0.12
50	0.32	0.13	0.39	0.16

results are better too, with the computation of BMI taking a mean time of 0.045 s, while the corresponding time for ITSDm was 0.32 s. This corresponds to a 85.94% saving in execution time.

Similar to the comparison with random ordering, we analysed the statistical significance of the results of the experiments, with the null hypothesis being that the two approaches (using ITSDm and BMI) give similar results. Again, we performed an homogeneity of variance check that told us that for the experiment with Set2 there is homogeneity, but for the experiment with Set3 there is none. Therefore, we applied an upper-tailed ANOVA test to the results for the experiments with Set2 (where we tested whether the results concerning BMI and the ones concerning ITSDm come from the same distribution) and we applied a Kruskal–Wallis H-test to the results for the experiment with Set3 (where we tested if the BMI results come from the same distribution as the ITSDm results). We computed the p-values, obtaining p-values close to zero, so we reject the hypothesis. We performed two effect size measures (one for each experiment), computing the Cliff's delta statistic. We obtained large effect sizes (greater than 0.89).

5.6. Execution time

On average we needed 0.00083786 s to compute BMI for test suites of length 100.¹⁰ In order to simulate the use of BMI, we extended the experiment for RQ1 as follows. First, we recorded the (mean) time needed to compute BMI for test suites of length 100, 200, 300, 400, 500, 600, 700, 800, 900 and 1000. We also determined the mean time needed to apply these test suites to mutants (averaged over 1000 mutants) assuming that the time needed to execute a transition is 0, 0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1 and 1 seconds (test execution terminated once a failure occurred). We compared the mean time to compute BMI with the mean time needed to apply a test suite, obtaining the results in Fig. 4.

¹⁰ The experiments were run on a GNU/Linux machine with an AMD[®] Ryzen threadripper 1920X at 3.50 GHz \times 12 cores and with 32 GB of RAM (although only one core was running at a time and we did not use more than 4Gb of RAM).

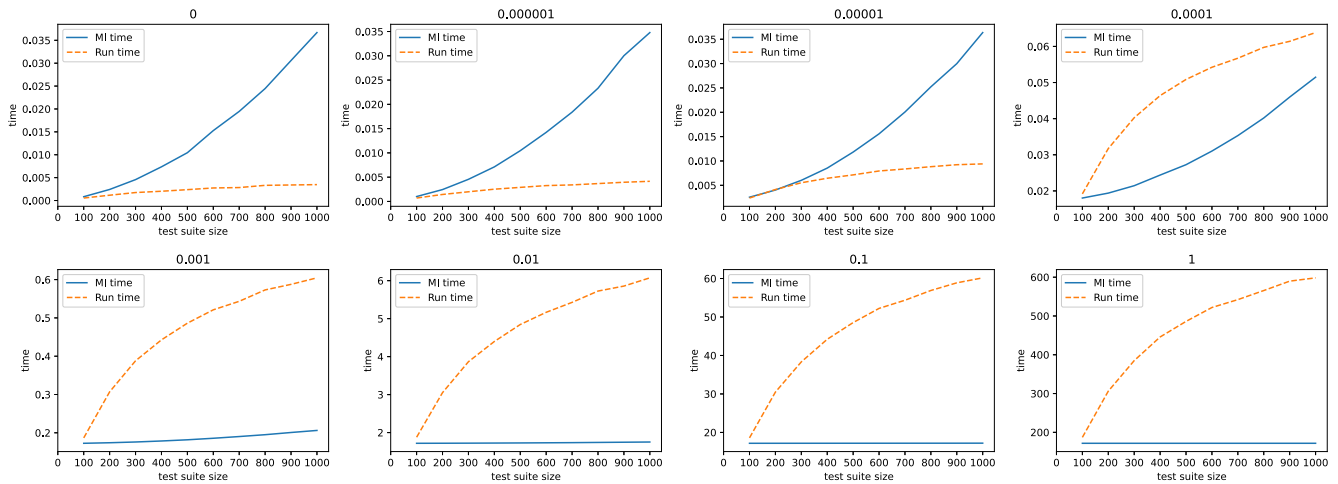


Fig. 4. Time comparison plots (left to right, from top to bottom, with transition time of 0, 0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1, 1 seconds).

Note that the mean time needed to compute BMI includes the time needed to determine how many times each input/output pair appears in the specification. This is computed only once for a given FSM. Therefore, although we included this in the time needed to compute BMI, when comparing test suites we need to carry out this computation only once.

As we can see, as long as the time needed to execute a transition is greater than 0.001 s, computing BMI is much faster than applying test suites. If this time is higher than 0.1 s then we need minutes (or even hours if it is higher than 1 second) to apply a test suite. As an additional result of the experiments, we validated that the time needed to compute BMI scales (approximately) quadratically with respect to the test suite length.

5.7. Assessing fault complexity and test suites coverage

We decided to assess the fault complexity and test suite coverage of the randomised elements we use in our experiments. In order to do so, we performed a new experiment. For the FSMs in Set2 and Set3, we randomly generated 100 test suites and 1000 mutants. We then computed, for each mutant, how many test suites killed this mutant and the coverage of each test suite, both in terms of transitions traversed and states visited, with respect to each original FSM.

For Set2, each mutant was killed by 52.78% and 96.01% of the test suites, with a mean value of 73.45%. Full results are displayed in Tables 14 and 15. For Set3, the results range from a minimum of 0% of test suites killing the mutant to a maximum of 90.21%, with a mean of 43.63%. Full results are displayed in Tables 16 and 17 (minimum results are not displayed this time because there was always a mutant that is not killed by any test suite).

Regarding coverage, we first computed the state coverage and transition coverage of each test suite. In addition, in order to normalise the values we required an upper bound on the *maximum achievable coverage* since this would allow us to compute the ratio between obtained coverage and an upper bound. Let us illustrate these concepts with a simple example.

Example 4. Consider an FSM with 50 states and 150 transitions and a test suite of length 75. An upper bound of the maximum transition coverage that we can reach is 50% (the test suite will traverse at most 75 of the 150 different transitions) while the maximum state coverage that we can reach is 100%. Note that these values are upper bounds but need not be least upper bounds because the structure of the FSM might induce smaller *real* maxima. For example, if we consider the FSM depicted in Fig. 1, then any test suite with 6 inputs will have a

maximum transition coverage of $\frac{6}{7} \cdot 100\%$ although our estimate will be 100% because the FSM has 6 transitions.

Let us suppose that the test suite has a 60% state coverage and 40% transition coverage. Then we know that our test suite covers at least 60% of the maximum number of states that can be covered by any test suite of its size (as already said, the real value might be higher). Similarly, our test suite covers at least 80% of the maximum number of transitions that can be covered with a test suite of its size.

We did not compute a true maximum value of coverage because of the time required. For example, it is straightforward to prove that the problem of computing the maximum state coverage for a given FSM and test suite length is NP-hard (by reduction from the Hamiltonian path problem [64]).

For Set2 we obtained an average state coverage of 77.42%, while the estimate of maximum achievable state coverage was always 100%. For transition coverage, Set2 obtained an average of 40.39% total coverage. If we *normalise* this number with respect to the estimate of the maximum achievable transition coverage we have 69.49%. Regarding Set3, we have 78.54% state coverage (again, the estimate of the maximum achievable state coverage was 100%), 41.16% total transition coverage, and 69.91% transition coverage with respect to the estimate of the maximum achievable coverage. The full results can be found at Tables 18 and 19 for Set2, and Tables 20 and 21 for Set3.

5.8. Comparison with coverage guided selection

During the analysis of coverage, a concern was raised: whether our BMI measure was simply a proxy for transition coverage. In order to determine whether this was the case, we developed a new experiment. This experiment was essentially the same as the experiment where we compared BMI with TSDm, but this time we compared BMI with an approach in which we select the test suite with higher transition coverage (over the specification) instead of the one with a higher ITSDm value.

We took Set2 and Set3. For each FSM in one of these sets, we generated two test suites of length 100, we computed both BMI and the transition coverage of each test suite, and marked the one chosen by each method. Then, we generated 1000 mutants of the FSM, computed how many mutants were killed by each test suite, and compared them. Then, we computed how many times both methods select the same test suite and how many times they did not. When they coincide, we consider two cases: they selected the test suite that killed more mutants or they selected the test suite that killed fewer mutants. When they differed, we determined which test suite killed more mutants and which one was the method that choose this test suite. We repeated this full experiment 50 times.

Table 14
Percentage of test suites that kill the mutants (Set2 Part I).

FSM #	Min.	Max.	Average
1	0.41	0.99	0.66156
2	1.0	1.0	1.0
3	0.28	0.97	0.59814
4	0.65	0.98	0.79892
5	0.58	0.95	0.76067
6	0.81	0.99	0.91935
7	0.91	1.0	0.94743
8	0.53	0.97	0.77722
9	0.49	0.98	0.74142
10	0.56	0.98	0.74353
11	0.95	1.0	0.97229
12	0.87	1.0	0.93984
13	0.58	0.98	0.74424
14	0.75	0.98	0.85946
15	0.77	1.0	0.87572
16	0.54	0.93	0.73653
17	0.0	0.79	0.42637
18	0.76	1.0	0.86361
19	0.51	1.0	0.72173
20	0.0	0.89	0.40575
21	0.34	1.0	0.63675
22	0.97	1.0	0.98332
23	0.95	1.0	0.97183
24	0.51	0.97	0.71475
25	0.19	0.9	0.53744
26	0.74	0.98	0.85649
27	0.47	0.95	0.69854
28	0.98	1.0	0.9874
29	0.46	0.95	0.69807
30	0.84	0.99	0.91555
31	0.76	1.0	0.87161
32	0.0	0.92	0.43817
33	0.17	0.94	0.51964
34	0.26	0.97	0.57787
35	0.0	0.98	0.41146
36	0.51	0.94	0.71073
37	0.53	0.99	0.7301
38	0.71	0.96	0.83734
39	0.62	0.94	0.77565
40	0.0	0.82	0.42639
41	0.94	1.0	0.96297
42	0.88	1.0	0.93942
43	0.0	0.92	0.44462
44	0.0	0.9	0.42206
45	0.0	0.88	0.43675
46	0.45	0.96	0.67961
47	0.59	0.95	0.78192
48	0.46	0.98	0.69809
49	0.47	0.96	0.71049
50	0.62	0.99	0.78636

Table 15
Percentage of test suites that kill the mutants (Set2 Part II).

FSM #	Min.	Max.	Average
51	0.66	0.99	0.80505
52	0.4	0.97	0.65833
53	0.86	1.0	0.92803
54	0.49	0.91	0.71166
55	0.59	1.0	0.79225
56	0.2	0.97	0.5383
57	0.48	0.94	0.69313
58	0.65	1.0	0.80878
59	0.78	0.99	0.87713
60	0.64	0.96	0.79091
61	0.85	1.0	0.93616
62	0.42	0.94	0.68205
63	0.9	1.0	0.94805
64	0.64	0.98	0.80052
65	0.35	0.96	0.63074
66	0.08	0.94	0.48562
67	0.26	0.86	0.6002
68	0.47	0.95	0.71338
69	0.37	0.96	0.69181
70	0.34	0.93	0.61436
71	0.17	0.91	0.5281
72	0.62	0.94	0.77926
73	0.0	0.89	0.45093
74	0.92	1.0	0.95179
75	0.24	0.85	0.53192
76	0.79	0.98	0.87988
77	0.92	1.0	0.96011
78	0.78	1.0	0.8672
79	0.25	0.92	0.56085
80	0.92	1.0	0.95496
81	0.48	0.98	0.70715
82	0.57	0.98	0.76196
83	0.75	0.98	0.8651
84	0.45	0.99	0.70628
85	0.77	1.0	0.89163
86	0.76	0.98	0.86909
87	0.0	0.9	0.44222
88	0.58	0.95	0.75736
89	0.61	0.96	0.77888
90	0.32	0.91	0.59404
91	0.81	0.99	0.88447
92	0.41	0.95	0.66994
93	0.51	0.95	0.74146
94	0.15	0.91	0.52903
95	0.3	0.94	0.60817
96	0.63	0.95	0.79545
97	0.74	0.98	0.84585
98	0.65	0.95	0.79298
99	0.3	0.94	0.606
100	0.58	0.96	0.77294

The results for Set2 are interesting: in 65.5% of the cases, BMI selected the same test suite as coverage. In 54.78% of the cases BMI selected the test suite that killed more mutants while coverage selected that test suite 62.16% of the time. In 13.56% of the experiments BMI selected the test suite that killed more mutants while coverage selected the other test suite. In 20.94% of the experiments the situation was the other way round. Finally, both measures failed to select the best test suite 24.28% of the time.

Regarding computation time, we have that BMI needed a mean of 0.00295048 s while coverage required a mean of 0.0583516 s. This constitutes a 94.94% saving.

The results for Set3 are not so different: BMI and coverage selected the same test suite 85.78% of the time. In addition, in 75.96% of the experiments BMI selected the test suite that killed more mutants, while coverage selected this test suite 81.78% of the time. In 4.2% of cases, BMI selected the test suite that killed more mutants and coverage selected the other test suite; the situation was the other way round 10.02% of the time. Finally, in 14.02% of the experiments, both measures failed to select the test suite that killed more mutants.

Regarding computation time, BMI took 0.00137012 s on average and coverage took 0.016663 s on average. This is a saving of 91.78%.

Similar to the other comparisons, we analysed the statistical significance of the results of the experiments assuming as the null hypothesis that the two approaches (using transition coverage and BMI) give similar results. Again, we performed an homogeneity of variance check that told us that there is no homogeneity for our results. Therefore, we applied a Kruskal–Wallis H-test to the results, where we tested if the BMI results come from the same distribution as the transition coverage results. We computed the p-values, obtaining p-values close to zero, so we reject the hypothesis that the results are statistically equivalent. We also computed the effect-size using Cliff’s delta statistic. We obtained large effect sizes (greater than 0.82).

Based on the above, we can conclude that BMI is not a proxy for transition coverage. In addition, although BMI is not as effective as coverage, it takes less time to compute, which can be a critical fact in some industrial size cases. There is therefore a trade-off between the effectiveness of the approaches and the time taken.

Table 16

Percentage of test suites that kill the mutants (Set3 Part I). Minimum is always 0.

FSM #	Max.	Average
1	0.91	0.44829
2	0.9	0.44203
3	0.86	0.43297
4	0.87	0.44348
5	0.88	0.41344
6	0.95	0.43757
7	0.93	0.43475
8	0.89	0.40389
9	0.96	0.43645
10	0.84	0.42938
11	0.91	0.42769
12	0.94	0.45951
13	0.86	0.43165
14	0.83	0.42816
15	0.92	0.4193
16	0.83	0.44864
17	0.89	0.48418
18	0.88	0.43903
19	0.95	0.43351
20	0.92	0.4278
21	0.92	0.41964
22	0.9	0.44669
23	0.84	0.42021
24	0.83	0.43035
25	0.87	0.41209
26	0.91	0.42674
27	0.84	0.45367
28	0.96	0.40972
29	0.89	0.41803
30	0.96	0.43839
31	0.89	0.4641
32	0.97	0.46382
33	0.9	0.44115
34	0.91	0.44495
35	0.91	0.43584
36	0.8	0.4282
37	0.92	0.44832
38	0.9	0.44736
39	0.91	0.45373
40	0.88	0.40431
41	0.87	0.41552
42	0.95	0.43048
43	0.8	0.41188
44	0.88	0.47413
45	0.96	0.43364
46	0.91	0.41166
47	0.9	0.45991
48	0.94	0.45115
49	0.97	0.39983
50	0.92	0.45365

Table 17

Percentage of test suites that kill the mutants (Set3 Part II). Minimum is always 0.

FSM #	Max.	Average
51	0.96	0.44439
52	0.84	0.39549
53	0.96	0.43365
54	0.97	0.42208
55	0.95	0.46957
56	0.96	0.4291
57	0.94	0.42368
58	0.9	0.41643
59	0.85	0.45532
60	0.88	0.44115
61	0.92	0.43872
62	0.95	0.44664
63	0.88	0.48217
64	0.87	0.45015
65	0.93	0.42311
66	0.88	0.44841
67	0.93	0.44696
68	0.91	0.42443
69	0.83	0.44163
70	0.82	0.41825
71	0.88	0.44828
72	0.9	0.43722
73	0.96	0.43312
74	0.85	0.4523
75	0.85	0.4469
76	0.84	0.41888
77	0.87	0.43075
78	0.93	0.42524
79	0.98	0.41455
80	0.92	0.40092
81	0.99	0.43639
82	0.9	0.44402
83	0.92	0.45994
84	0.89	0.44517
85	0.97	0.46387
86	0.87	0.41392
87	0.9	0.43428
88	0.99	0.4524
89	0.9	0.41782
90	0.87	0.4276
91	0.94	0.44664
92	0.87	0.42702
93	0.95	0.40269
94	0.84	0.42797
95	0.89	0.46605
96	0.91	0.48529
97	0.91	0.43881
98	0.92	0.43231
99	0.88	0.44156
100	0.86	0.43291

In some ways it is not too surprising that transition coverage can be more effective than BMI since transition coverage uses more information about the FSM: it uses information about the states associated with each input/output pair rather than just input/output pair frequency. It is therefore, for example, able to identify cases where the same input/output pair appears but they represent different parts (transitions) of the specification.

In principle, it should be possible to extend the definition of BMI to use information about the transitions executed instead of input/output pair frequency. However, this would have at least two disadvantages. The first disadvantage is simply that it would be necessary to traverse the FSM specification and this would increase the computation time. The second, and rather more important, disadvantage is that such a revised definition of BMI would only be applicable in situations in which we have an FSM specification. This would go against the aim of developing a measure that can be used in a range of scenarios. For example, in principle it should be possible to use BMI without having a specification, as long as it is possible to include some initial random

testing in order to provide estimates of input/output pair frequency; clearly, transition coverage cannot be applied in such situations.

5.9. Summary

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

Research Question 1. *Given a pair of test suites with the same length, will the one with lower Biased Mutation Information tend to have higher fault detection ability?*

The answer to this question is affirmative: BMI tended to select test suites with higher fault coverage than random selection. In the experiments BMI selected the best test suite 62.4924% of the time when we used FSMs with an alphabet of size 5 (Tables 7 and 8) and 75.0605% of the times when we used FSMs with an input alphabet of size 25 (Table 9). BMI also selected test suites with higher fault coverage in the scenario with real FSMs.

Table 18

Test suite coverage evaluation results in percentages (Set2 Part I). We include Percentages with respect to Maximal Achievable Transition Coverage.

FSM #	State coverage	Transition coverage	PMATC
1	0.7452	0.390893	0.6567
2	0.7672	0.404909	0.6681
3	0.766	0.38388	0.7025
4	0.7568	0.399706	0.6795
5	0.7912	0.409429	0.716499
6	0.7974	0.412286	0.721499
7	0.761	0.408863	0.682801
8	0.7236	0.402134	0.6595
9	0.7736	0.408647	0.6947
10	0.7458	0.377457	0.653
11	0.788	0.420706	0.7152
12	0.763	0.405353	0.6891
13	0.7338	0.382743	0.6698
14	0.803	0.404309	0.7318
15	0.7208	0.403291	0.6372
16	0.79	0.409829	0.717199
17	0.7866	0.399945	0.727899
18	0.7304	0.421753	0.649499
19	0.724	0.398503	0.665501
20	0.8022	0.375026	0.7238
21	0.7558	0.405522	0.661
22	0.7714	0.390471	0.6638
23	0.7632	0.385706	0.6827
24	0.7878	0.4148	0.725899
25	0.7706	0.410542	0.6815
26	0.814	0.388229	0.745401
27	0.7474	0.382102	0.6725
28	0.7958	0.402944	0.725299
29	0.775	0.409415	0.700101
30	0.794	0.399011	0.726199
31	0.8078	0.427134	0.7005
32	0.7938	0.427824	0.7273
33	0.7868	0.392528	0.698699
34	0.7786	0.403121	0.6974
35	0.7692	0.401861	0.691201
36	0.7616	0.390387	0.7066
37	0.7714	0.401221	0.690101
38	0.7874	0.431205	0.7158
39	0.7882	0.394365	0.7138
40	0.7884	0.406514	0.7114
41	0.7444	0.378352	0.688599
42	0.763	0.401198	0.670001
43	0.7324	0.403187	0.6451
44	0.7898	0.396448	0.7255
45	0.796	0.399034	0.7023
46	0.8	0.394837	0.7265
47	0.7954	0.415176	0.7058
48	0.762	0.406871	0.6632
49	0.7852	0.4345	0.6952
50	0.7738	0.40681	0.6631

Table 19

Test suite coverage evaluation results in percentages (Set2 Part II). We include Percentages with respect to Maximal Achievable Transition Coverage.

FSM #	State coverage	Transition coverage	PMATC
51	0.7616	0.400526	0.684901
52	0.7772	0.418788	0.691
53	0.7418	0.388171	0.679299
54	0.805	0.418439	0.7239
55	0.7834	0.392286	0.686499
56	0.7932	0.389945	0.7136
57	0.7956	0.403218	0.701599
58	0.7328	0.364674	0.671
59	0.7758	0.404768	0.696201
60	0.7766	0.396033	0.7287
61	0.7652	0.396171	0.693299
62	0.7596	0.403095	0.6772
63	0.7674	0.398343	0.6732
64	0.7868	0.409882	0.6927
65	0.7922	0.408363	0.698301
66	0.7748	0.411797	0.687701
67	0.8096	0.436348	0.728701
68	0.8042	0.431151	0.711399
69	0.7686	0.388187	0.706499
70	0.7644	0.388022	0.706199
71	0.8192	0.395769	0.720299
72	0.8152	0.406966	0.724399
73	0.81	0.425	0.714
74	0.7542	0.381768	0.691
75	0.7552	0.378737	0.7196
76	0.7622	0.417665	0.697501
77	0.7768	0.408855	0.6787
78	0.7262	0.386509	0.6532
79	0.7854	0.388989	0.7313
80	0.7648	0.423418	0.669001
81	0.7426	0.400248	0.6444
82	0.7654	0.415305	0.6811
83	0.7908	0.430307	0.7014
84	0.7984	0.435901	0.7018
85	0.7668	0.400059	0.6761
86	0.7816	0.40092	0.6976
87	0.8054	0.437143	0.7038
88	0.7792	0.406994	0.7041
89	0.7882	0.411345	0.703401
90	0.756	0.387598	0.6938
91	0.7746	0.390945	0.7037
92	0.765	0.407651	0.6767
93	0.8102	0.434573	0.7127
94	0.7446	0.415938	0.6655
95	0.7514	0.419063	0.6705
96	0.7968	0.411136	0.7236
97	0.7452	0.376667	0.6893
98	0.7638	0.405907	0.694101
99	0.7834	0.413392	0.706901
100	0.7802	0.405989	0.698301

Research Question 2. Are lower levels of Biased Mutual Information associated with higher fault coverage?

In the experiments, lower levels of BMI were correlated with higher fault coverage. We can conclude this from Tables 10 and 11, where we can observe that BMI was negatively correlated with the mutation score of the tests, with a mean correlation of -0.369134 for FSMs with an alphabet of size 5 and a mean correlation of -0.650256 for FSMs with an alphabet of size 25.

Research Question 3. Do test suites selected by BMI have higher fault coverage than those selected by test set diameter (ITSDm)?

The answer was again positive: BMI selected test suites with higher fault coverage than ITSDm. We can conclude this from Tables 12 and 13, where we can observe that BMI outperformed the ITSDm measure when selecting the best test suite from a set of 2 randomly generated test suites.

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

Fig. 4 shows that the time needed to compute BMI increased (approximately) quadratically with respect to the length of the test suite. Also, the time needed to compute the selection method was smaller than the time needed to apply a test suite as long as we need at least 0.001 s to execute each transition (Table 9).

6. Threats to validity

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

Concerning threats to internal validity, which consider uncontrolled factors that might be responsible for the obtained results, the main threat is associated with possible faults in the tools. In order to reduce the impact of this threat we tested the code with carefully constructed examples for which we could manually check the results. In addition,

Table 20

Test suite coverage evaluation results in percentages (Set3 Part I). We include Percentages with respect to Maximal Achievable Transition Coverage.

FSM #	State coverage	Transition coverage	PMATC
1	0.7778	0.415671	0.6817
2	0.7662	0.405353	0.6891
3	0.7938	0.413977	0.707901
4	0.7974	0.424458	0.7046
5	0.7744	0.399435	0.707
6	0.8006	0.415407	0.714501
7	0.788	0.405029	0.7007
8	0.7914	0.404413	0.7239
9	0.7674	0.423727	0.6822
10	0.7926	0.403295	0.7098
11	0.7738	0.393779	0.677301
12	0.8128	0.411916	0.687901
13	0.78	0.404128	0.695101
14	0.7956	0.422083	0.7091
15	0.7538	0.397337	0.6715
16	0.7974	0.393757	0.7127
17	0.8096	0.425154	0.693
18	0.7804	0.411183	0.6949
19	0.7802	0.403489	0.694001
20	0.7968	0.405114	0.713
21	0.7718	0.418235	0.711
22	0.7718	0.432911	0.684
23	0.7926	0.408864	0.7196
24	0.8026	0.424551	0.709001
25	0.7774	0.409538	0.7085
26	0.7462	0.391737	0.654201
27	0.7988	0.419461	0.700501
28	0.773	0.3925	0.7222
29	0.7804	0.399385	0.7149
30	0.7782	0.388506	0.675999
31	0.8176	0.419581	0.700701
32	0.7576	0.424494	0.6707
33	0.797	0.413684	0.707401
34	0.7876	0.411579	0.7038
35	0.799	0.399375	0.7029
36	0.801	0.405967	0.7348
37	0.8006	0.439062	0.7025
38	0.7938	0.408855	0.6787
39	0.799	0.40578	0.702
40	0.7884	0.387263	0.7358
41	0.7696	0.393559	0.6966
42	0.7598	0.390284	0.6869
43	0.7528	0.406023	0.694301
44	0.7468	0.422581	0.655
45	0.7734	0.41	0.6724
46	0.7776	0.392198	0.713799
47	0.815	0.421006	0.7115
48	0.7652	0.408963	0.6707
49	0.781	0.396497	0.7018
50	0.7974	0.42018	0.701701

Table 21

Test suite coverage evaluation results in percentages (Set3 Part II). We include Percentages with respect to Maximal Achievable Transition Coverage.

FSM #	State coverage	Transition coverage	PMATC
51	0.777	0.410424	0.6772
52	0.7694	0.403932	0.718999
53	0.7644	0.429814	0.692
54	0.7758	0.40386	0.690601
55	0.7988	0.438853	0.689
56	0.7748	0.388258	0.691099
57	0.8038	0.408324	0.7309
58	0.7874	0.395281	0.703599
59	0.782	0.436346	0.6807
60	0.7952	0.405402	0.705399
61	0.8026	0.439752	0.708
62	0.7938	0.427048	0.7089
63	0.7782	0.431503	0.6602
64	0.8098	0.435151	0.717999
65	0.7708	0.401395	0.690401
66	0.8094	0.447	0.7152
67	0.7968	0.418537	0.6864
68	0.7756	0.399535	0.687201
69	0.7998	0.42	0.714
70	0.7984	0.411573	0.732599
71	0.8216	0.424269	0.725501
72	0.8052	0.409714	0.716999
73	0.7742	0.422638	0.6889
74	0.8286	0.441212	0.727999
75	0.799	0.416667	0.7125
76	0.7854	0.382989	0.7047
77	0.7798	0.413721	0.711601
78	0.7704	0.409177	0.6956
79	0.7348	0.397831	0.6604
80	0.7762	0.398962	0.7301
81	0.7524	0.412468	0.6517
82	0.7512	0.42075	0.6732
83	0.7704	0.414479	0.6756
84	0.8016	0.412486	0.7136
85	0.7934	0.420311	0.6767
86	0.7848	0.401808	0.7112
87	0.8016	0.401932	0.7074
88	0.7922	0.40503	0.6845
89	0.7618	0.405497	0.693401
90	0.807	0.390909	0.731001
91	0.733	0.397469	0.6439
92	0.7916	0.428036	0.7191
93	0.7738	0.39676	0.7102
94	0.7826	0.412126	0.7171
95	0.8102	0.415882	0.707
96	0.807	0.452467	0.6787
97	0.79	0.425864	0.6899
98	0.794	0.395304	0.7155
99	0.8116	0.427679	0.7185
100	0.7898	0.415385	0.702

we repeated the experiments many times to reduce the impact of randomisation. Another important threat was the processor reschedule policy, which can affect the recorded times. In order to reduce the impact of this threat, we abstracted the time computation and only computed small enough time values so that the reschedule policy does not affect them. In addition, we repeated the tests and computed mean values. Another threat was that Normalised Compress Distance (NCD), used in TSDm, performs poorly with short strings. We therefore used relatively long strings; for a test suite of length 100 we have strings of $(100 \text{ input actions} + 100 \text{ output actions}) \times 2 \text{ characters per action} = 400 \text{ characters}$. We made this choice because it seemed to be a reasonable test length for the FSMs used and also because previous work noted that compression did not work for strings of length less than 128 [21]. It is possible that longer test sequences would lead to more effective compression and so NCD being a better guide; this is an issue that could be addressed by further experiments.

The main threat to external validity, which concerns conditions that allow us to generalise our findings to other situations, is the choice

of FSMs. Such a threat cannot be entirely addressed since the population of FSMs is unknown and it is not possible to sample from this (unknown) population. In order to reduce the impact of this threat we used randomly generated FSMs and a carefully constructed benchmark. A minor external threat is that if we use large alphabets for randomly generated FSMs then very few input/output pairs will be repeated. In order to address this threat we performed the experiments with different alphabet sizes, as shown in Section 5.

Finally, we considered threats to construct validity, which are related to whether we are measuring properties of interest. The aim of testing is to find faults and so it is clear that the fault detection ability of a test suite is of interest, as is the time used to obtain a solution (since there are finite resources). We used mutants to assess fault detection ability and ideally we would have also used real faults. However, we are not aware of benchmark FSMs with faulty versions; 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. Final discussion: alternative definitions

We have shown that BMI is interesting, potentially useful and that the time needed to compute it is negligible when compared to the time needed to apply extra testing. However, it is possible that some of the design decisions were not optimal and, indeed, there were alternative choices. In this section we describe some such alternatives and the results of additional experiments carried out to evaluate these. The decisions made, when designing BMI, fall into the following classes:

1. The formula used to define $bmi(t_1; t_2)$ in terms of the ‘distributions’;
2. Whether the variables used in the mutual information formula were probability distributions.

We now describe some alternatives to the choices made, along with the results of experiments that evaluated these.

7.1. The definition of $bmi(t_1; t_2)$

During the rest of this section, remember that 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 .

The definition of $bmi(t_1; t_2)$ used a transformation of the X -axis and we might have chosen a different transformation. In order to explore the impact of using a larger transformation, we considered:

$$bmi_2(\xi_{t_1}; \xi_{t_2}) = \sum_{x \in I_2} n_x \cdot \frac{\log_2(m_x + 2)}{m_x + 2}$$

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

Another option is to use the formula $\frac{n}{m}$, instead of $\frac{1}{m}$, to compute the values of $\sigma_r(x)$ (where $x \in_n t$ and $x \in_m M$). This way, we take into account also how many times the input/output pair is repeated in the test. This leads to the following formula:

$$bmi_3(\xi_{t_1}; \xi_{t_2}) = \sum_{x \in I_2} n_1 \cdot n_2 \cdot \frac{\log_2(m_x + 1)}{m_x}$$

where m_x is such that $x \in_{m_x} M$, n_1 is such that $x \in_{n_1} t_1$ and n_2 is such that $x \in_{n_2} t_2$.

Finally, we applied the previous variations with two different approaches to compute the values of $\sigma_r(x)$. Instead of using the probability explained in Section 4, we could use the number of times an input/output pair appears in the test suite:

$$\sigma_{\xi_A}(x) = \frac{1}{\#\text{test suite } I/\mathcal{O} \text{ pairs with label } x}$$

The results of experiments, using Set3 (50 states and alphabets of size 25), are given in Table 22. These show that from the seven possible combinations, five are more or less equally good, and the other two are clearly worse. Therefore, we decided to retain our approach, since it appears to keep a good balance between intuition and being faithful to the original Information Theory formulae.

7.2. Variables used in the mutual information formula

Another important choice was the decision to not use true random variables and corresponding probability distributions in the Mutual Information formula. We now describe some alternatives considered. The alternatives that we explored used combinations of the following mechanisms for generating the random variables used in the definition of BMI.

1. Whether normalisation is used;
2. Whether one takes into account the number of times a pair appears in a test;

3. How to consider the case where an input/output pair appears in both tests (i.e. how one defines the “joint probability”, corresponding to $\sigma_{\xi_{t_1, t_2}}(x_1, x_2)$, for two tests t_1, t_2 when considering the same pair $(x_1 = x_2)$).

Regarding the first point, normalisation would lead to a probability distribution (i.e. with values that sum to 1). Normalisation can be achieved by taking the sum of the values for the input/output pairs of the test and then dividing the value given for each input/output pair by this factor. This way, the sum of the probabilities of all the input/output pairs of the test is equal to 1.

Regarding the second mechanism, it would have been possible to use the *number of times* each input/output pair appears within a single test when defining the corresponding probability distribution. The resultant probabilities depend both on the test and on the FSM. The downside of this mechanism was that we lose the intuition that we previously followed, which is that the weight of each input/output pair in a test should be the probability of this pair corresponding to a particular transition of the specification (see paragraph after Example 1). Despite this, we evaluated alternatives that take into account the number of times an input/output pair appears in a test.

Finally, we considered the approach taken to define the joint probability distribution for tests t_1 and t_2 . We explored an alternative approach in which we start by giving a random variable and its probability distribution for t_1 and t_2 . Then, we compute the joint probability of both tests with the uncorrelated input/output pairs of each test (the case where the input/output pairs are different) and add all these values. This is straightforward because the joint probability of uncorrelated input/output pairs is simply the product of the probabilities of each input/output pair. As the sum of all the values of the joint probability should sum up to 1, we know the amount of probability corresponding to the correlated input/output pairs (we will call this P). Then, we defined s to be the product of the probabilities of each input/output pair modified by a factor and we define the joint probability of the correlated input/output pairs as s divided by the sum of all the s 's and multiplied by P . This gives us a joint probability for each pair of correlated input/output pairs and we could call this “a joint probability of correlated input/output pairs”. It is important to note that this joint probability is different from the product of the probabilities of each input/output pair; otherwise, in the mutual information formula we would get $\log_2(1)$ and since this is equal to 0, we would have mutual information of 0.

We tried several different combinations of these mechanisms. These alternatives are displayed in Table 23. The column “dist” corresponds to the probability distribution formula, while “joint” corresponds to the joint distribution formula for the correlated input/output pairs (for the uncorrelated input/output pairs, the joint distribution is the product of the individual distributions). In the table we assume $x_1 \in_{n_1} t_1, x_2 \in_{n_2} t_2, x_1 \in_m M, x_2 \in_m M$ and $P = 1 - S_1$.

As can be seen in Table 23, all the alternative formulations considered were outperformed by the proposed approach. This is despite some of the alternatives being notably more involved than the proposed approach. All of the alternatives achieve a mean score between 50% and 65%, with only one distribution getting more than 60%. In contrast, our proposed approach had a score of 75.0605%.

Then, with all this information, we are able to clearly state that the alternative a tester should use is the one presented in Section 4.

8. Conclusions and future work

The selection of a test suite can be a critical task because the time and resources devoted to testing are limited. In this paper we considered the problem of choosing between two alternative test suites. Solutions to this problem might be used to directly compare alternative test suites (e.g. for different sets of features). They might also be used to guide the generation of test suites in an iterative manner or to inform

Table 22
Comparing different alternative approaches.

Test	# 0.4%	# 0.5%	# 0.6%	# 0.7%	# 0.8%	Min value	Max value	% success (mean)
MI based on spec	4	33	13	0	0	0.459184	0.680851	56.9662%
MI based on test suite	0	0	3	41	6	0.666667	0.818182	75.0757%
BMI based on spec	0	0	5	36	9	0.673469	0.838384	75.3883%
BMI based on test suite	0	0	5	32	13	0.666667	0.848485	76.4054%
BMI ₂ based on spec	0	0	14	31	5	0.66	0.816327	74.2696%
BMI ₂ based on test suite	0	0	1	43	6	0.670103	0.848485	75.1613%
BMI ₃	3	22	24	1	0	0.42268	0.7	58.9412%

Table 23
Comparing different probability distributions.

#	Dist	Joint	S ₁	S ₂	S ₁	S ₂	Success
1	$\frac{1}{m \cdot s}$	$\frac{n_1}{m \cdot s_1} \cdot \frac{n_2}{m \cdot s_2} \cdot \frac{P}{S_2}$	$\sum_{\substack{x_1 \in E_1 \\ x_1 \in M}} \frac{1}{m_1}$	$\sum_{\substack{x_2 \in E_2 \\ x_2 \in M}} \frac{1}{m_2}$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 \neq x_2}} \frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2}$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 = x_2}} \frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2}$	64%
2	$\frac{n}{s}$	$\frac{\min(n_1, n_2)}{S_2}$	$\sum_{x_1 \in M} n_1$	$\sum_{x_2 \in M} n_2$	0	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 = x_2}} \min(n_1, n_2)$	55%
3	$\frac{n}{s}$	$\frac{n_1}{s_1} \cdot \frac{n_2}{s_2} \cdot \frac{1}{m_1} \cdot \frac{P}{S_2}$	$\sum_{x_1 \in M} n_1$	$\sum_{x_2 \in M} n_2$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 \neq x_2}} \frac{n_1}{s_1} \cdot \frac{n_2}{s_2}$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 = x_2}} \frac{n_1}{s_1} \cdot \frac{n_2}{s_2} \cdot \frac{1}{m_1}$	54%
4	$\frac{n}{s}$	$\frac{n_1}{s_1} \cdot \frac{n_2}{s_2} \cdot m_1 \cdot \frac{P}{S_2}$	$\sum_{x_1 \in M} n_1$	$\sum_{x_2 \in M} n_2$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 \neq x_2}} \frac{n_1}{s_1} \cdot \frac{n_2}{s_2}$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 = x_2}} \frac{n_1}{s_1} \cdot \frac{n_2}{s_2} \cdot m_1$	58%
5	$\frac{1}{m \cdot s}$	$\frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2} \cdot \frac{1}{m_1} \cdot \frac{P}{S_2}$	$\sum_{x_1 \in M} \frac{1}{m_1}$	$\sum_{x_2 \in M} \frac{1}{m_2}$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 \neq x_2}} \frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2}$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 = x_2}} \frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2} \cdot \frac{1}{m_1}$	57%
6	$\frac{1}{m \cdot s}$	$\frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2} \cdot m_1 \cdot \frac{P}{S_2}$	$\sum_{x_1 \in M} \frac{1}{m_1}$	$\sum_{x_2 \in M} \frac{1}{m_2}$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 \neq x_2}} \frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2}$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 = x_2}} \frac{1}{m_1 \cdot s_1} \cdot \frac{1}{m_2 \cdot s_2} \cdot m_1$	55%
7	$\frac{n}{m \cdot s}$	$\frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2} \cdot \frac{1}{m_1} \cdot \frac{P}{S_2}$	$\sum_{x_1 \in M} \frac{n_1}{m_1}$	$\sum_{x_2 \in M} \frac{n_2}{m_2}$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 \neq x_2}} \frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2}$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 = x_2}} \frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2} \cdot \frac{1}{m_1}$	56%
8	$\frac{n}{m \cdot s}$	$\frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2} \cdot m_1 \cdot \frac{P}{S_2}$	$\sum_{x_1 \in M} \frac{n_1}{m_1}$	$\sum_{x_2 \in M} \frac{n_2}{m_2}$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 \neq x_2}} \frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2}$	$\sum_{\substack{x_1 \in E_1 \\ x_2 \in E_2 \\ x_1 = x_2}} \frac{n_1}{m_1 \cdot s_1} \cdot \frac{n_2}{m_2 \cdot s_2} \cdot m_1$	55%

the choice of which subset of a regression test suite to use. We observed that diverse test suites have been found to be effective and proposed the use of a novel measure, BMI, based on Mutual Information to assess diversity.

Having developed BMI, and analysed a number of its properties, we reported on experiments that evaluated it. First, we randomly generated pairs of test suites and used BMI to order the test suites in each pair. We then determined how many mutants of the FSM specification were killed by each test suite. In these experiments we found that test suites with lower BMI tend to kill more mutants. This provides evidence that BMI can be used as the basis for choosing between test suites. There was also a (negative) correlation between the fault detecting ability of a test suite (i.e. the number of mutants that it killed) and the BMI of the test suite. Interestingly, we also found that BMI outperformed the previous information theoretic measure, Test Set Diameter (TSDm), when selecting a test suite from a set of two randomly generated test suites.

The positive results, when comparing BMI with TSDm, suggest that the use of diversity is improved if we introduce knowledge about the specification; previous work using diversity has concerned white-box testing and assumed that a specification is not available. The results suggest that measures of test suite diversity can be improved if one has knowledge about the *rarity* of events. This knowledge can be extracted from the specification of the developed system. Naturally, if additional information is available then it should be possible to improve on measures. As a result, we found that transition coverage was (slightly) more effective than BMI, although the computation of transition coverage took much more time than the computation of BMI. Therefore, there is potentially a trade-off between effectiveness and time taken. Importantly, however, BMI can be used whenever we have

information about input/output pair frequency and is therefore more widely applicable than transition coverage. In fact, there is potential to use BMI even if there is no specification, since it should be possible to estimate input/output pair frequency using sampling.

The results presented in this paper have some clear practical ramifications. First, testers can directly use the novel BMI measure if they have a number of test suites; they can compute the BMI of each test suite and take the one (or ones) with lower BMI. Testers can also use BMI to drive test subset selection; they can select the subset with lower BMI. In addition, if there is a limited budget for regression testing then the tester faces the problem of choosing a subset of the regression test suite; the test subset selection problem has been addressed based on white-box coverage information (see, for example, [3–5]) but, where this is not available, it is possible to instead use BMI. BMI can be used by testers when they have limited information about the specification. In particular, it can be used instead of methods based on coverage when we do not have a complete specification of the system but we have the frequency of each input/output pair.

An intuition that explains why BMI is better than a true Information Theory based measure can be the following one: BMI gives a proportional value to each input/output pair independently of the rest of the test. That is, we are giving the same weight to the same input/output pair independently of the test length. However, when using Information Theory based measures, we require a probability distribution over the input/output pairs of the test. Therefore, the weights of an input/output pair will be different in two different tests. This produces undesirable effects like the decrease of the weight of an input/output pair due to it being in a longer test than if it were in a shorter test. This can lead to situations where, for example, a test suite with a longer test with many repeated input/output pairs could be preferred to a test suite with many

shorter tests with only one repeated input/output pair between all of them.

There are several possible lines of future work. First, it would be interesting to explore the use of BMI in the task of generating new test suites from scratch. Second, we would like to perform additional experiments to compare BMI and ITSDm. Third, there is potential to apply BMI in more complex scenarios, with one such scenario being when the specification is an Extended Finite State Machine (EFSM). Note that an EFSM can be mapped to an FSM through expanding out the data, possibly after applying an abstraction. Thus, EFSM faults that lead to incorrect variable values map nicely to the type of mutation used in the experiments, in which only the final state of a transition is changed. Another interesting scenario is given when we consider distributed systems, possibly with asynchronous communications, whose specifications are represented as a variant of an FSM [65,66]. In order to confront these more complicated formalisms, we can use current work that make it possible to apply a systematic approach to the generation of mutants [67,68]. In principle, it should also be possible to apply BMI even when there is no specification, since an initial random testing phase could be used to produce estimates of input/output pair frequency. For this scenario, there is a need for experiments that explore the process of producing estimates of input/output pair frequency and also the impact of using estimates on the effectiveness of BMI. This line of work is particularly important because it should make it possible to apply BMI in a context in which we do not have access to a specification and so we cannot apply a method based on the coverage of the available test suites.

CRedit authorship contribution statement

Alfredo Ibias: Conceptualization, Software, Validation, Formal analysis, Data curation, Writing - original draft, Writing - review & editing, Visualization. **Manuel Núñez:** Conceptualization, Methodology, Writing - original draft, Writing - review & editing, Supervision, Funding acquisition. **Robert M. Hierons:** Conceptualization, Methodology, Writing - original draft, Writing - review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We would like to thank the anonymous reviewers for the careful reading of the paper and the many constructive comments, which have helped us to further strengthen the paper.

References

- [1] P. Ammann, J. Offutt, *Introduction to Software Testing*, second ed., Cambridge University Press, 2017.
- [2] G. Myers, C. Sandler, T. Badgett, *The Art of Software Testing*, third ed., John Wiley & Sons, 2011.
- [3] G. Rothermel, M.J. Harrold, Analyzing regression test selection techniques, *IEEE Trans. Softw. Eng.* 22 (8) (1996) 529–551.
- [4] Z. Li, M. Harman, R.M. Hierons, Search algorithms for regression test case prioritization, *IEEE Trans. Softw. Eng.* 33 (4) (2007) 225–237.
- [5] A. Arrieta, J.A. Agirre, G. Sagardui, Seeding strategies for multi-objective test case selection: an application on simulation-based testing, in: 22nd Annual Conf. on Genetic and Evolutionary Computation, GECCO'20, ACM, 2020, pp. 1222–1231.
- [6] B. Sarikaya, G.v. Bochmann, Synchronization and specification issues in protocol testing, *IEEE Trans. Commun.* 32 (1984) 389–395.
- [7] K. Sabnani, A. Dahbura, A protocol test generation procedure, *Comput. Netw. ISDN Syst.* 15 (1988) 285–297.
- [8] I. Pomeranz, S.M. Reddy, Test generation for multiple state-table faults in finite-state machines, *IEEE Trans. Comput.* 46 (7) (1997) 783–794.
- [9] A. Benharref, R. Dssouli, M.A. Serhani, A. En-Nouayari, R. Glietho, New approach for EFSM-based passive testing of web services, in: Joint 19th IFIP TC6/WG6.1 Int. Conf. on Testing of Software and Communicating Systems, TestCom'07, and 7th Int. Workshop on Formal Approaches To Software Testing, FATES'07, LNCS 4581, Springer, 2007, pp. 13–27.
- [10] M. Haydar, A. Petrenko, H. Sahraoui, Formal verification of web applications modeled by communicating automata, in: 24th IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'04, LNCS 3235, Springer, 2004, pp. 115–132.
- [11] W. Grieskamp, N. Kicillof, K. Stobie, V. Braberman, Model-based quality assurance of protocol documentation: tools and methodology, *Softw. Test. Verif. Reliab.* 21 (1) (2011) 55–71.
- [12] W. Huang, J. Peleska, Complete model-based equivalence class testing for nondeterministic systems, *Form. Asp. Comput.* 29 (2) (2017) 335–364.
- [13] F. Hübner, W. Huang, J. Peleska, Experimental evaluation of a novel equivalence class partition testing strategy, *Softw. Syst. Model.* 18 (1) (2019) 423–443.
- [14] 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.
- [15] 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, 4783, Springer, 2007, pp. 11–23.
- [16] C.E. Shannon, A mathematical theory of communication, *Bell Syst. Tech. J.* 27 (1948) 379–423, 623–656.
- [17] 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.
- [18] 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, *Softw. Test. Verif. Reliab.* 21 (2) (2011) 75–100.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] ITU-TSG 10/Q8 ISO/IEC JTC1/SC21/WG7, 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.
- [23] J. Tretmans, Model based testing with labelled transition systems, in: *Formal Methods and Testing*, LNCS 4949, Springer, 2008, pp. 1–38.
- [24] M. Li, P.M.B. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*, fourth ed., Springer, 2019.
- [25] R. Cilibrasi, P.M.B. Vitányi, Clustering by compression, *IEEE Trans. Inform. Theory* 51 (4) (2005) 1523–1545.
- [26] 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.
- [27] E.P. Moore, Gedanken experiments on sequential machines, in: C. Shannon, J. McCarthy (Eds.), *Automata Studies*, Princeton University Press, 1956.
- [28] F. 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.
- [29] T.S. Chow, Testing software design modeled by finite state machines, *IEEE Trans. Softw. Eng.* 4 (1978) 178–187.
- [30] M.P. Vasilevskii, Failure diagnosis of automata, *Cybernetics* 4 (1973) 653–665.
- [31] R.M. Hierons, H. Ural, Optimizing the length of checking sequences, *IEEE Trans. Comput.* 55 (5) (2006) 618–629.
- [32] F. Ipate, Bounded sequence testing from deterministic finite state machines, *Theoret. Comput. Sci.* 411 (16–18) (2010) 1770–1784.
- [33] A. Simão, A. Petrenko, N. Yevtushenko, On reducing test length for FSMs with extra states, *Softw. Test. Verif. Reliab.* 22 (6) (2012) 435–454.
- [34] R.M. Hierons, Testing from partial finite state machines without harmonised traces, *IEEE Trans. Softw. Eng.* 43 (11) (2017) 1033–1043.
- [35] R.M. Hierons, FSM quasi-equivalence testing via reduction and observing absences, *Sci. Comput. Program.* 177 (2019) 1–18.
- [36] A. Petrenko, N. Yevtushenko, Testing from partial deterministic FSM specifications, *IEEE Trans. Comput.* 54 (9) (2005) 1154–1165.
- [37] R.M. Hierons, M. Núñez, Implementation relations and probabilistic schedulers in the distributed test architecture, *J. Syst. Softw.* 132 (2017) 319–335.
- [38] R.M. Hierons, M.G. Merayo, M. Núñez, Bounded reordering in the distributed test architecture, *IEEE Trans. Reliab.* 67 (2) (2018) 522–537.

- [39] I. Hwang, A.R. Cavalli, Testing a probabilistic FSM using interval estimation, *Comput. Netw.* 54 (7) (2010) 1108–1125.
- [40] N. López, M. Núñez, I. Rodríguez, Specification, testing and implementation relations for symbolic-probabilistic systems, *Theoret. Comput. Sci.* 353 (1–3) (2006) 228–248.
- [41] R.M. Hierons, M.G. Merayo, M. Núñez, Testing from a stochastic timed system with a fault model, *J. Log. Algebr. Program.* 78 (2) (2009) 98–115.
- [42] A.V. Aho, A.T. Dabhura, D. Lee, M.Ü. Uyar, An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours, *IEEE Trans. Commun.* 39 (11) (1991) 1604–1615.
- [43] A.Y. Duale, M.Ü. Uyar, A method enabling feasible conformance test sequence generation for EFSM models, *IEEE Trans. Comput.* 53 (5) (2004) 614–627.
- [44] K. Derderian, R.M. Hierons, M. Harman, Q. Guo, Generating feasible input sequences for extended finite state machines (EFSMs) using genetic algorithms, in: 7th Genetic and Evolutionary Computation Conference, GECCO'05, ACM Press, 2005, pp. 1081–1082.
- [45] A.S. Kalaji, R.M. Hierons, S. Swift, Generating feasible transition paths for testing from an extended finite state machine (EFSM), in: 2nd Int. Conf. on Software Testing Verification and Validation, ICST'09, IEEE Computer Society, 2009, pp. 230–239.
- [46] A. Petrenko, S. Boroday, R. Groz, Confirming configurations in EFSM testing, *IEEE Trans. Softw. Eng.* 30 (1) (2004) 29–42.
- [47] A. Turlea, F. Ipate, R. Lefticaru, A test suite generation approach based on EFSMs using a multi-objective genetic algorithm, in: 19th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC'17, IEEE Computer Society, 2017, pp. 153–160.
- [48] K. Bogdanov, M. Holcombe, F. Ipate, L. Seed, S. Vanak, Testing methods for X-machines: a review, *Form. Asp. Comput.* 18 (2006) 3–30.
- [49] 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.
- [50] J.K. Blundell, M.L. Hines, J. Stach, The measurement of software design quality, *Ann. Softw. Eng.* 4 (1–4) (1997) 235–255.
- [51] 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.
- [52] D. Clark, R.M. Hierons, Squeeziness: An information theoretic measure for avoiding fault masking, *Inform. Process. Lett.* 112 (8–9) (2012) 335–340.
- [53] A. Ibbas, R.M. Hierons, M. Núñez, Using squeeziness to test component-based systems defined as finite state machines, *Inf. Softw. Technol.* 112 (2019) 132–147.
- [54] A.V. Miranskyy, M. Davison, R.M. Reesor, S.S. Murtaza, Using entropy measures for comparison of software traces, *Inform. Sci.* 203 (2012) 59–72.
- [55] K.R. Pattipati, M.G. Alexandridis, Application of heuristic search and information theory to sequential fault diagnosis, *IEEE Trans. Syst. Man Cybern.* 20 (4) (1990) 872–887.
- [56] K.R. Pattipati, S. Deb, M. Dontamsetty, A. Maitra, START: System testability analysis and research tool, *IEEE Aerosp. Electron. Syst. Mag.* 6 (1) (1991) 13–20.
- [57] 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.
- [58] S. Yoo, M. Harman, D. Clark, Fault localization prioritization: Comparing information-theoretic and coverage-based approaches, *ACM Trans. Softw. Eng. Methodol.* 22 (3) (2013) 19:1–19:29.
- [59] A. González-Sánchez, É. Piel, H.-G. Groß, A.J.C. van Gemund, Prioritizing tests for software fault localization, in: 10th Int. Conf. on Quality Software, QSIQ'10, IEEE Computer Society, 2010, pp. 42–51.
- [60] N. Alshahwan, M. Harman, Coverage and fault detection of the output-uniqueness test selection criteria, in: 24th ACM SIGSOFT Int. Symposium on Software Testing and Analysis, ISSTA'14, ACM Press, 2014, pp. 181–192.
- [61] T.M. Cover, J.A. Thomas, *Elements of Information Theory*, Wiley Interscience, 1991.
- [62] 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, QSIQ'09, IEEE Computer Society, 2009, pp. 145–154.
- [63] 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.
- [64] M.R. Garey, D.S. Johnson, *Computers and Intractability*, W. H. Freeman and Company, 1979.
- [65] M.G. Merayo, R.M. Hierons, M. Núñez, Passive testing with asynchronous communications and timestamps, *Distrib. Comput.* 31 (5) (2018) 327–342.
- [66] M.G. Merayo, R.M. Hierons, M. Núñez, A tool supported methodology to passively test asynchronous systems with multiple users, *Inf. Softw. Technol.* 104 (2018) 162–178.
- [67] P. Gómez-Abajo, E. Guerra, J. de Lara, M.G. Merayo, A tool for domain-independent model mutation, *Sci. Comput. Program.* 163 (2018) 85–92.
- [68] P. Gómez-Abajo, E. Guerra, J. de Lara, M.G. Merayo, Wodel-test: a model-based framework for language-independent mutation testing, *Softw. Syst. Model.* (2020) in press.

10.6 Coverage-Based Grammar-Guided Genetic Programming Generation of Test Suites

Authors	Alfredo Ibias, Pablo Vazquez-Gomis and Miguel Benito-Parejo
Title	Coverage-Based Grammar-Guided Genetic Programming Generation of Test Suites
Publication Type	Conference
Venue	2021 IEEE Congress on Evolutionary Computation
Year	2021
DOI/URL	https://doi.org/10.1109/CEC45853.2021.9504969
Pages	8
Authors' Contributions	Ibias and Benito-Parejo developed the theory. Ibias, Vazquez-Gomis and Benito-Parejo designed the experiments. Ibias and Vazquez-Gomis developed and executed the experiments. Ibias and Benito-Parejo wrote the manuscript. Ibias and Benito-Parejo reviewed the manuscript.

Coverage-Based Grammar-Guided Genetic Programming Generation of Test Suites

Alfredo Ibias
DTRS research group
Universidad Complutense de Madrid
28040, Madrid, Spain
aibias@ucm.es

Pablo Vazquez-Gomis
DTRS research group
Universidad Complutense de Madrid
28040, Madrid, Spain
pavazq01@ucm.es

Miguel Benito-Parejo
DTRS research group
Universidad Complutense de Madrid
28040, Madrid, Spain
mibeni01@ucm.es

Abstract—Software testing is fundamental to ensure the reliability of software. To properly test software, it is critical to generate test suites with high fault finding ability. We propose a new method to generate such test suites: a coverage-based grammar-guide genetic programming algorithm. This evolutionary computation based method allows us to generate test suites that conform with respect to a specification of the system under test using the coverage of such test suites as a guide. We considered scenarios for both black-box testing and white-box testing, depending on the different criteria we work with at each situation. Our experiments show that our proposed method outperforms other baseline methods, both in performance and execution time.

Index Terms—Genetic Programming, Coverage, Software Testing

I. INTRODUCTION

Testing software is a fundamental step on every software development process. The goal is to improve the quality of the software searching for faults in it, using as few resources as possible. However, due to its criticality, testing can cost more than 50% of the development budget [30]. Therefore, good, cheap and effective methods for testing software are fundamental for the software product cycle. One of the main techniques in the software testing field [1], [30] is to try to find faults through the execution of input sequences and comparing the outputs obtained with the expected ones. The combination of the input sequence and the expected outputs is a test suite, and the goal of the method is to use the test suites with higher fault finding ability. In order to generate such test suites, the most widely known approach is *mutation testing* [22], [32], which uses *mutants* (i.e. modified versions) of the System Under Test (SUT) (or more usually, its specification) to generate such test suites. However, one of the main issues with this method is its computational cost. More feasible approaches focus on finding good test suites with respect to a chosen

criteria, reducing the computational and resources costs but diminishing at the same time the effectiveness of the method. In this paper we present one of such approaches.

Another problem of software testing is the so called *Oracle problem* [2], [25]. This problem focuses on how to decide that a SUT is correct or not given the obtained outputs, and for extension, on how well a test suite detects a fault in the program. For this task a lot of approaches had been tried, from classical deterministic solutions [7], [11] to more evolved ones, like those based on genetic algorithms [3], [4], [31]. In this paper we use Finite State Machines (FSMs) to represent the specification of the SUT, and we used a coverage-based criterion to decide how good the test suites are detecting faults. We also use the mutation score to compare our approach to others from the literature. Mutation score is a measure from mutation testing that calculates the percentage of mutants killed by a test suite. We say that a test kills a mutant when the mutation has been discovered when executing the test.

Evolutionary computation algorithms are a well known family of algorithms and meta-heuristics that focus on the evolution of a bunch of individual solutions to obtain an approximately optimal solution. This family ranges from the Genetic Algorithms [34] based on the evolution of the genomes to the Ant Colony Optimisation algorithm [9] based on the organisation of an ant colony, passing by the Particle Swarm Optimisation algorithm [5], [23] based on the development of a flock of birds. Genetic algorithms [34] are an approximation method to find good or nearly good solutions to computationally exponential problems. They focus on generating random solutions (called *individuals*) and improving them through the mixture and mutation of the best generated ones. To decide which individual is better they use a previously chosen criteria (called *fitness function*) that should guide the *evolution*. Genetic programming [24] is an extension of genetic algorithms that manage to deal with structured types, being able to find solutions to a wider range of problems. Specifically, genetic programming used to work with tree-like structures, which can be totally free or bounded to a grammar [27]. In our work we use a genetic programming algorithm whose individuals are bounded to a grammar that will ensure that they will conform to valid test suites of the SUT.

In this paper we present a Grammar-Guided Genetic Pro-

This work has been supported by the Spanish MINECO/FEDER project FAME (RTI2018-093608-B-C31); the Region of Madrid project FORTE-CM (S2018/TCS-4314) co-funded by EIE Funds of the European Union; the Region of Madrid - Complutense University of Madrid (grant number PR65/19-22452); and the Santander - Complutense University of Madrid (grant number CT63/19-CT64/19).

gramming algorithm to generate test suites. This algorithm uses a coverage-based measure as a guide, and therefore obtains test suites that have a high coverage of the FSM that models the SUT. The goal of this algorithm is to generate test suites with a high fault detection ability, and it uses coverage-based fitness functions to that end. Specifically, we use measures based on t -way coverage, which measures how many groups of t consecutive elements of the SUT are covered by the test suite. These coverage-based fitness functions are prepared for two scenarios: a white-box scenario where we have information about the internal structure of the SUT (like states) and a black-box scenario where we do not have such information and we can only observe input and outputs. We performed experiments to compare this approach to more traditional ones and to different variations of itself. Specifically, we compared our algorithm with a grammar-guided genetic programming algorithm that uses as fitness function the mutation score of the individuals. In these experiments we obtained that our solutions generated using coverage-based fitness functions were quite effective, as they took less time to be computed, while not losing a lot of (or even having better) finding faults potential. We found that the coverage criteria based on 1-way transition coverage is the preferable choice when generating test suites if we are in a black-box scenario. For a white-box scenario, the 2-way state coverage is preferred.

In this paper, Section II introduces the basic concepts over which our algorithm is developed. Section III introduces our coverage-based grammar-guided genetic programming algorithm for generating test suites. Later, Section IV presents our experiments evaluating our algorithm. In Section V we discuss some aspects of our algorithm and our experiments. Section VI evaluates the threats to the validity of our experiments. Finally, Section VII contains the conclusions of our work and lines for future work.

II. THEORETICAL BACKGROUND

We model systems as *Finite State Machines* (FSMs). In order to define an FSM, we first introduce some 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 .

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

An FSM is a (finite) labelled transition system in which every transitions is labelled with an *input/output pair* (a pair containing an input action and an output action). We use this formalism to define specifications.

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 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 $(q, i/o, q') \in T$.

We assume that FSMs are deterministic. This simplifies our scenario so we can use genetic algorithms, without losing applicability. However, our algorithm can be used on non-deterministic FSMs with some adaptations to the specification. An FSM can be represented by a diagram in which nodes represent states and transitions are represented by arcs between the nodes. In our case, all states are final as long as they are reachable from the initial state.

We will assume the *minimal test hypothesis* [21]: the SUT can be modelled as an (unknown) object described in the same formalism as the specification (here, an FSM).

We say that a mutant $M' = (Q, q_{in}, \mathcal{I}, \mathcal{O}, T')$ of an FSM $M = (Q, q_{in}, \mathcal{I}, \mathcal{O}, T)$ is another FSM such that T and T' only differ in one transition in the form of $(q, (i, o), q') \in T$ and $(q, (i, o), q'') \in T'$ with $q' \neq q''$.

Our main goal while testing is to decide whether the behaviour of an SUT conforms to the specification of the system that we would like to build. In order to detect differences between specifications and SUTs, we need to compare their behaviours, and the main notion to define such behaviours is given by the concept of a *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 $\text{traces}(M)$ the set of traces of M . Note that $\epsilon \in \text{traces}(M)$ for every FSM M .

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

Definition 3: Let $M = (Q, q_{in}, \mathcal{I}, \mathcal{O}, T)$ 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 $t \in \text{traces}(M)$. The *length* 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 *length* of the test suite is the sum of the lengths 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 $(\lambda, \mu') \in$

$\text{traces}(M')$ 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 notion of coverage is quite simple in our context: it represents how much of the FSM a test suite traverses. We define three different coverage criteria based on the t -way coverage definition:

- t -way transition coverage: the percentage of sets of t consecutive transition labels that the test suite traverses. We define transition label as a pair input/output of the FSM (that is, a pair input/output that corresponds to the execution of a transition).
- t -way state coverage: the percentage of sets of t consecutive states that the test suite visits. We define state as a state of the FSM.
- t -way action coverage: the percentage of sets of t consecutive actions that the test suite executes. We define action as an input of the FSM.

We differentiate between these 3 types of coverage as they are the most widely used in the literature [26], [33]. Each coverage type focuses on a different element of the FSM and therefore it is mandatory to compare between the three to see which one yields better results. In order to use state coverage we need to be aware of the internal state of the FSM, therefore it can only be considered within a white-box scenario. However, action coverage and transition coverage only need observable information and can also be used in a black-box scenario. Finally, for a notion of coverage based in transitions (as defined in Def. 1), we require a white-box scenario as well, since the structure of the FSM is also needed. This last case will be studied in detail in section IV.

The t -way coverage groups the transition labels/states/actions of the FSM in sets with exactly t elements. These sets contain t consecutive elements, that is, there exists a sequence of transitions of the FSM such that the transition labels/states/actions involved in it are exactly the ones in the set. For example, for 1-way state coverage, the states of the FSM are grouped in sets with only one state, and therefore there are as many sets as states. In the case of 2-way state coverage however, the states of the FSM are grouped in sets of pairs of states. Specifically each set contains two consecutive states, that is, two states that are connected by a transition. Then, in this case there are as many sets as transitions, but not as many as combinations of two states, because if two states are not connected through a transition, then they are not consecutive.

With the t -way sets of transition labels/states/actions of the FSM, we can check the number of them that appear (without repetitions) in a test suite. The percentage of these sets that appear in the test suite will be its t -way transition/state/action coverage score. This score represents how much coverage of the SUT this test suite will provide.

Formally, we define the t -way transition/state/action coverage score as follows.

Definition 4: Given an FSM M , a grouping G (with $|G|$ elements) of its transition labels/states/actions in sets of t

consecutive elements, and a test suite that traverses s of such sets (without repetitions), the t -way transition/state/action coverage score is $\frac{s}{|G|}$.

In our work, t ranges from 1 to 3. Along this paper we will call G set to the grouping of the transition labels/states/actions in sets of t consecutive elements of an FSM.

Genetic programming is a meta-heuristic that is often used to obtain good enough solutions to complicated optimisation problems. They are non-deterministic algorithms that consider multiple possible solutions or *individuals* at a time, and combine their information in order to obtain different solutions each iteration. Since the objective is to improve the final solution, a *fitness function* evaluates each individual to prioritise those possible solutions with a better score. This method is derived from the classical genetic algorithm, with some adaptations to work with tree-like structures that we use in our work. Usually, a genetic algorithm is divided in 5 steps structured as follows:

- The *initialisation* step generates the initial *population*, acting as a seed for the whole process. Such an initialisation is usually random, in order not to bias the behaviour of the algorithm.
- The *selection* step is focused on obtaining the most suited individuals to perform the following steps, and achieving a better solution in next generations.
- The *crossover* step consists of pairing the individuals obtained in the selection step, and exchanging parts of the structure within each couple.
- The *mutation* step considers each individual after the crossover step, and with a small probability performs slight variations or mutations. This process, although might seem counter-intuitive, it tends to avoid obtaining local maximum solutions, by possibly substituting the negative-impact elements of the individual for new ones.
- The *replacement* step, finally takes the current population and its offspring, and decides which individuals amongst them conform the following generation.

The idea of genetic algorithms is to iterate the process to make the population evolve and produce a better solution. Therefore, it requires a termination criteria, which usually considers a bound on the number of iterations.

It is important to note that the loop for a genetic algorithm only considers the selection, crossover, mutation and replacement steps, as only one initialisation is required. A general flowchart of genetic programming is shown in Fig. 1.

For our genetic programming algorithm we use a grammar to guide the generation of the test suites. A grammar is a set of symbols and rules that restrict the generation capabilities of the algorithm in order to ensure the correctness of the generated individuals with respect to a chosen criteria (in our case, that they are valid test suites for the SUT).

III. COVERAGE-BASED GRAMMAR-GUIDED GENETIC PROGRAMMING ALGORITHM

In this section, we describe the specific elements of our genetic algorithm, and the structure of the steps we use, to

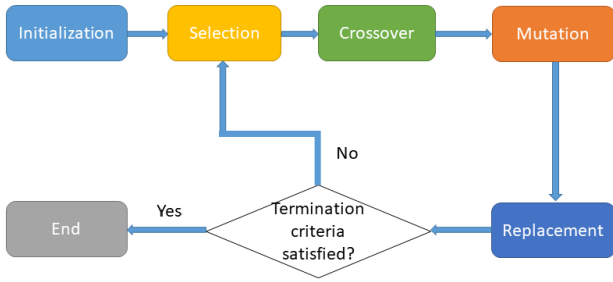


Fig. 1. GA flowchart

present a global idea of our contribution in a more detailed manner.

Our proposed algorithm is a genetic programming algorithm that works with a population of individuals of the same size. Each individual is a test suite consisting of n input/outputs pairs. However, a grammar-guided genetic programming works with tree-like structures (i.e. data structures that can be abstracted as trees) that conform to a grammar. Thus, we have to adapt our test suites to be abstractly represented as grammar-guided trees. We do such a transformation defining each node of the tree as an input/output pair (i.e. each node contains an input and an output) with a grammatical symbol, then each succession of nodes would be a succession of input/output pairs, which corresponds to a test. Finally, in order to combine the tests of the test suite into a tree, we join them with a *dummy node* that would be the root of the tree. This way, each test would be a branch of the tree. It is important to note that we work with a constrained size, which all trees share, in order not to obtain huge trees that compromise the score of the results, as a bigger size in a test suite would induce a better score. Along the rest of the paper we would use input/output pair and node interchangeably, and we would do the same for test and branch, and for test suite and tree. Finally, remark that a subtree of these trees would be a section of a test.

Since we decided to use a grammar-guided genetic programming approach, we need to generate a grammar that allows to generate test suites that conforms to the FSM. For this grammar, we define the following components [17]:

- A start non-terminal symbol S that starts the grammar.
- A non-terminal symbol TS that introduces each test of the test suite.
- A non-terminal symbol A for each state, where $A \in \mathbb{A}$ 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 $S \rightarrow TS$ to generate the initial test.
- A production rule $TS \rightarrow TS + TS$ to introduce a new test.
- A production rule $TS \rightarrow 0$ to start each test in the FSM initial state.

- A production rule $A \rightarrow 'a/b' + A$ for each transition from the left hand side state A to the right hand side state A with input/output pair (a, b) .
- A production rule $A \rightarrow 'null'$ for each state A to a terminal to represent the end of the test.

With this components, we will produce the nodes of the trees that represent test suites.

In the *initialisation* step, we generate random test suites, avoiding duplicated tests. The idea is to generate an initial population as diverse as possible, in order to have a wider spectrum to explore with the following steps.

To evaluate and compare the individuals of our population, we need a fitness function. As the main contribution of our work, we propose several coverage-based fitness functions, considering the notion of coverage defined in Def. 4. These fitness functions are built over coverage criteria to determine which test suites have a higher fault finding capability. We developed 12 fitness functions using t -way coverage criterion. In particular, we considered transition, state, action t -way coverage, and an extension of transition t -way coverage that includes the initial and final states of each transition (i.e. it considered the actual transitions). We instantiated these criteria with $t \in \{1, 2, 3\}$. We will later on compare the results that we obtain with each of these fitness functions.

In the selection step, we aim at an exploration goal, rather than an exploitation one. In that sense, we consider that our selection method should take into account the whole population, in order not to lose any genetic diversity. Therefore, our selection method simply matches pairs of individuals for the following steps.

For the grammar-guided crossover, we implemented a variation on the standard crossover, where we select a random node from each parent such that both nodes have the same grammatical symbol. Then, we exchange the selected subtrees while maintaining the grammatical correctness of the whole tree. This means that the resulting individuals still represent a valid test suite for the given SUT. However, in order to keep the length of the test suite (note that such length is the sum of the length of each test), we will have to modify the resulting trees accordingly. We have two different scenarios: first, we have to extend test suites that provide a longer subtree while receiving a shorter one; and second we have to bound test suites that receive a longer subtree than the one they provide. This is necessary to control the length of a test suite, avoiding an incremental increase of the size of the individuals. The reason to keep the length invariant is to have a reasonable comparison between the solutions, as a bigger test suite would produce inherently a better score.

We perform the extension of a subtree by randomly generating a grammatically valid continuation of the subtree. That means that we expand from the leaf of the subtree generating new nodes (i.e. input/output pairs) until reaching the adequate length. In the case that we are unable to extend such subtree up to the desired length, we generate a new random test to substitute the remaining nodes. For the bound on larger subtrees, we simply eliminate its final nodes, in order

for the test suite to match the required length. Due to this variation of the standard crossover, in most cases we add a little extra genetic information apart from the one belonging to the parents.

Our grammar-guided mutation step consists in randomly replacing tests of the test suite for newly generated ones. The idea of this mutation, is to incorporate different tests that have not been considered before for the test suite. With this procedure, we add new genetic information (previously unseen) to the individual and avoid reaching local optimum. We do not consider a mutation on the test suite as such, but on each test in the test suite. Since each test is represented by a branch of the tree, we remove such test by deleting the branch, and we include the new test by adding a new branch to the *dummy root node*. It is important to note that a test is either fully removed, or is not modified, as we do not interfere with partial branches of the tree.

Finally, the last step of our loop combines the best elements of both the parent and the offspring populations, to prepare the final individuals (either for following iterations, or the end of the process). We divided this step in several phases, starting by obtaining the average score for the offspring population. With it, we automatically consider the individuals that improve such average value to be in the final population. Next, for the remaining individuals, which have a worse score than the average, there is a probability for them to be maintained. To finish, in order to complete the size of the population, that should remain invariant, we randomly select among the best individuals from the parent generation.

The termination criteria that we consider is to perform 100 iterations. However, if during the execution we find that the last 20% of iterations do not improve the score, we terminate the process. We added such extra condition as we want to avoid an excessive amount of iterations and computing power that will not yield a better result.

IV. EXPERIMENTS

The experiments we performed aimed to compare our algorithm with another test suite generation algorithm based on mutation score. For these experiments we took as experiment subjects a set of 100 randomly generated FSMs, each one with 100 states. In them, each state has between 5 and 50 outgoing transitions, each one with a label conformed by an input/output pair. These pairs are generated from both input and output alphabets, each one with 50 elements. This set of FSM is generated with the idea of stretching the capabilities of the proposed algorithm, being big enough SUTs and trying to be as representative of real-world applications as possible.

The idea of the experiments was to compare our proposed algorithm (*coverage algorithm*) to a genetic programming algorithm whose fitness function is the mutation score (*mutation score algorithm*). *Mutation score algorithm* uses the same grammar-guided genetic programming algorithm as *coverage algorithm*, the only real change is in the fitness function, and therefore in the obtained solutions. Specifically, it generates a new set of 10 random mutants each iteration to calculate

the mutant score of the individuals of the population. In these experiments each individual of the population represents a test suite of fixed length 1000, and therefore the solutions will be test suites for the given FSM. Then, we compare the solution from both algorithms using mutation score, as it is a well established method to compare test suites [32].

The experiments were developed as follows: for each FSM we generated two test suites (one using *coverage algorithm* and other using *mutation score algorithm*). Then, we generated 100 mutants of the FSM and computed the mutation score of each test suite. We repeated this procedure for each of the 100 FSMs from the experiment subjects set, and we computed the average values for all the FSMs. Finally, we performed this whole experiment 10 times to obtain a final mean value that is less prone to the randomisation influence. We display these final averages on Table I, where:

- The *Success* columns indicate the percentage of times where *coverage algorithm* performed better than the *mutation score algorithm* (and vice-versa).
- The *Mutants Killed* columns indicate the percentage of mutants each algorithm was able to kill.
- The *Execution Time* columns indicate the time that each algorithm took to run.

The results of the experiments are displayed on Table I. There we can see that the three of t -way transition coverage, t -way state coverage and t -way action coverage beat mutation score to a extent, both in fault finding capability and in computation time. Moreover, we can observe an interesting phenomena: the mutation score of the different coverage notions is not uniform with respect to the value of t . This implies that a greater t does not imply that the resulting test suite will have a higher mutation score.

From these results arises the question of what would the situation be if we computed the t -way transition coverage using transitions instead of transition labels. We call this notion t -way extended transition coverage. It is obvious that the transitions have more information about the underlying FSM and its structure, so they could perform better, while not needing a lot of extra time as they follow a similar concept of coverage. However, the transitions can only be generated if we have an oracle (an FSM representing the SUT) or we are in a white-box testing scenario, which limits its applicability. We repeated the experiment using the transitions and obtained the results displayed on Table II. As we can observe there, the results are in the line of the ones from the other coverage types, without a huge difference in computation time. Therefore, we can conclude that the extra requirements are not worth it.

Finally, we performed a statistical hypothesis test over all the results. The null hypothesis was that the coverage-based fitness functions and the mutation score fitness function gave similar results, that is, both produced test suites with similar mutation score. We applied a one-way ANOVA test¹ where we tested whether the values of both fitness functions were,

¹Note that we could use the ANOVA test because we performed an homogeneity of variance check and it raised a positive result.

TABLE I
RESULTS OF THE COMPARISON WITH MUTATION SCORE.

Coverage Type	Success Coverage (Percentage)	Success Mutation Score (Percentage)	Mutants Killed Coverage (Percentage)	Mutants Killed Mutation Score (Percentage)	Execution Time Coverage (Seconds)	Execution Time Mutation Score (Seconds)
1 – way transition coverage	0.5644	0.4356	0.3781	0.3660	31.0099	83.9273
2 – way transition coverage	0.5585	0.4415	0.3738	0.3652	29.1904	84.5565
3 – way transition coverage	0.5131	0.4869	0.3673	0.3660	9.8079	84.2539
1 – way state coverage	0.4915	0.5085	0.3660	0.3666	6.8651	85.6319
2 – way state coverage	0.6167	0.3833	0.3827	0.3633	31.7231	86.0107
3 – way state coverage	0.5452	0.4548	0.3734	0.3636	31.2126	86.5856
1 – way action coverage	0.4995	0.5005	0.3663	0.3670	6.8224	85.9126
2 – way action coverage	0.5074	0.4926	0.3672	0.3654	31.4185	85.9683
3 – way action coverage	0.4995	0.5005	0.3684	0.3667	10.0687	86.1290

TABLE II
RESULTS OF THE COMPARISON WITH MUTATION SCORE (USING TRANSITIONS).

Coverage Type	Success Coverage (Percentage)	Success Mutation Score (Percentage)	Mutants Killed Coverage (Percentage)	Mutants Killed Mutation Score (Percentage)	Execution Time Coverage (Seconds)	Execution Time Mutation Score (Seconds)
1 – way extended transition coverage	0.6028	0.3972	0.3823	0.3663	31.2724	87.5435
2 – way extended transition coverage	0.5458	0.4542	0.3722	0.3651	29.0016	86.3313
3 – way extended transition coverage	0.4973	0.5027	0.3674	0.3656	10.1902	86.2759

on average, similar. Then, we computed the p-value for the experiments. Here, we observed an interesting situation: for 1-way and 2-way transition and extended transition coverage, and for 2-way and 3-way state coverage the obtained p-values were lower than 0.05. However, for the other measures the p-values were higher than 0.05. Therefore, we can deny the null hypothesis for the experiments with the first coverage notions with a confidence higher than 0.95, and we have to accept the null hypothesis for the experiments with the second set of coverage notions. In order to double-check our results, we performed a t-test and obtained the same p-values.

We can conclude that some of our coverage-based fitness functions are better than a mutation score function, both regarding performance and computation time. However, there is something more that we have to discuss. More precisely, we detected that the computation time in some experiments was lower by a huge margin than for others. After a careful analysis, we concluded that this behaviour was produced by the different orders of magnitude between the different G sets. For example, the number of sets of 3-way transitions, actions and extended transitions are so huge, that with a test suite of length 1000 the variation on percentage between two test suites is almost null. Alternatively, we have that the number of sets of 1-way states and actions are very small (in fact, these numbers are 100 and 50 in our experimental subjects, respectively) and therefore with test suites of length 1000

is really easy to cover all the sets, obtaining a coverage of 100%. It is important to note that these two groups correspond to the ones that confirmed the null hypothesis. This lack of improvement (either because we reached the 100%, or the improvement is negligible) triggers the second condition of the termination criteria, stopping the execution before the 100 iterations are produced, and therefore reducing the execution time. This behaviour shows that not all the coverage notions that we propose in our work are useful for the average testing practices, because some will arise pointless results.

V. DISCUSSION

During the development of our algorithm there were some critical decisions we had to make and that can arise some questions from an experienced reader. Therefore, in this section we will address such decisions. We discuss in detail our decision of using only one kind of coverage measures and our decision of using a genetic algorithm with mutation score as fitness function as a baseline algorithm for comparing with our proposed algorithm. Finally, we also address the decision of which crossover to use for our genetic algorithm.

A. Coverage Measures Choice

In traditional coverage-based literature there are two kind of coverage metrics: the first one includes in the coverage metric all the elements a test traverses, and the second one only includes the last element (or set of elements) of each

test. We decided to stick to the first kind of coverage metrics and forget about the second one because the characteristics of the first kind would help better to the evolution of the genetic algorithm. We chose the first kind of coverage metrics because it is easier to obtain different scores for two test suites with the same amount of tests, while the second kind of coverage metrics would yield the same score for those two test suites (if there are no repetitions). For example, let us say we have two test suites with only one test: the test of the first test suite has 17 input/output pairs and the test of the second test suite has 5 input/output pairs. Let us say also consider the 2-way action coverage. Then, if we use the first kind of metrics the first test suite will cover 16 sets of actions and the second test suite will cover 4 sets of actions, while if we use the second kind of metrics both test suites cover only 1 set of actions. However, it is clear that we should prefer the first test suite over the second, as suggested by the first kind of coverage measure.

B. Baseline Algorithm Choice

Mutation score is a traditional and widely known measure used in mutation testing. It is typically used as a measure to compare different test suite generation algorithms, as it has been shown to be highly correlated with the fault finding ability of a test suite [32]. That's why we use it to compare between different coverage measures and to compare with the baseline algorithm. The choice we want to discuss here is why we used it as a fitness function of our baseline algorithm.

The reasoning behind this choice is that a genetic algorithm (in this case a grammar-guided genetic programming algorithm) whose evolution is guided by mutation score as fitness function will obtain test suites that obtain a high mutation score, and therefore, will obtain the best scores later when comparing with another algorithm using mutation score. However, such algorithm will take a lot of computation time due to the high computational cost of generating and using the bunch of mutants needed to obtain a mutation score. Therefore, this kind of algorithm should be a valid baseline with which to compare our proposed algorithm.

We could have used other state-of-the-art algorithms to which compare our algorithm, like genetic algorithms using fitness functions like Test Set Diameter [10], [17], or more classical algorithms like W [11] or Wp [7] methods. However, we considered that a genetic algorithm using mutation score as fitness function will perform better as a baseline measure. Anyway, the comparison with these other methods would be matter of future work, in order to ensure the suitability of our proposed algorithm.

C. Crossover Choice

Concerning the crossover selected for our proposed grammar-guided genetic programming algorithm, our choice arises some concern due to its capability to include new (previously unseen) genetic information into the offspring. We chose this crossover because we wanted a crossover with two clear restrictions: first, the offspring had to be grammatically valid,

and second the offspring had to keep the same length than its parents. Then, the spectrum of options that we had available was limited. In fact, we could only find two crossovers that conform to those restrictions.

The first crossover we considered was simpler but also harder to produce: it consisted in finding the same grammatical symbol with the same length to the leaf in both trees. This situation happens very rarely and therefore the amount of crossovers produced was less than optimal (even when trying to produce the crossover for each pair of trees). This crossover conformed to the required restrictions because the length was maintained through the interchange of subtrees with the same number of nodes, and the grammatical correctness was ensured because both subtrees started with the same grammatical symbol. However, it obtained worse results than our selected crossover.

The second crossover was our selected crossover. It is a bit more complex but at the same time easier to occur: it consisted in finding the same grammatical symbol at both trees, exchanging the subtrees that start at such symbols, and then solving the possible problems with the length of the offspring. It is in this last step where new genetic information was generated in order to extend the shorter tree into the desired length. This crossover conforms to the required restrictions because the grammatical correctness was ensured due to both subtrees starting with the same grammatical symbol, and the length being maintained by generating or bounding the offspring. This crossover does not have problems of incapability to be produced and therefore it obtains better results than other options.

VI. THREATS TO VALIDITY

In this section we briefly discuss some of the possible threats to the validity of the results of our experiments. Concerning threats to *internal validity* (results validity), the main threat is associated with the possible faults in the developed experiments because they could lead to misleading results. In order to reduce the impact of this threat we tested our code with carefully constructed examples for which we could manually check the results. In addition, we repeated the experiments many times in order to get a mean so that the randomisation impact is reduced. Finally, there is the choice of baseline measure, that if poorly chosen can arise better results than the real ones. This concern is discussed in Section V and we consider it properly addressed.

The main threat to *external validity* (results generalisation), is the different possible systems to which we could apply our algorithm. Such a threat cannot be entirely addressed since the population of possible systems is unknown and it is not possible to sample from this (unknown) population. In order to diminish this risk, we perform our experiments over randomly generated FSMs prepared to stretch the capabilities of the algorithm.

Finally, we considered threats to *construct validity* (experiments *reality*), that is, whether our experiments reflect real-world situations or not. In our work, the main construct threat

is what would happen if we used our algorithm with much more complex methods. In order to address this concern, we have performed experiments with huge randomly generated FSMs, but there is still room for improvement and it will be matter of future work.

VII. CONCLUSIONS

Generating test suites with a high fault finding ability is fundamental for ensuring the quality of software. Moreover, performing this task in a quick and efficient manner is critical for software budgets. In this paper we have presented a grammar-guided Genetic Programming algorithm to generate such tests suites, based on coverage criteria fitness functions. We compared our proposal with another method that happens to be worst than ours. We also found that 1-way transition coverage is the preferable choice when generating test suites if we are in a black-box scenario, and 2-way state coverage if we are in a white-box scenario.

For future work, we would like to explore a wider range of coverage notions, specifically t -way coverage with $t > 3$. Evolving over this line of work, we would like to explore the significance of the relation between the size of the G set and the length of the test suites. We would like to deal with bigger numbers of mutants. Therefore, we would like to include recent work on producing and managing big sets of mutants [6], [8], [12], [13] into our framework. We would also like to explore the possibility of using another kind of evolutionary computation algorithms, instead of a genetic programming algorithm, like tree swarm optimisation [14], [15]. In another line of work, we would like to use our recent work on testing using Information Theory concepts [18]–[20] to implement genetic algorithms using the induced measures as fitness functions. Finally, we would like to extend our framework to deal with FSMs that can represent systems with distributed components [16], [28], [29].

REFERENCES

- [1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2017.
- [2] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.
- [3] M. Benito-Parejo, I. Medina-Bulo, M. G. Merayo, and M. Núñez. Using genetic algorithms to generate test suites for FSMs. In *15th Int. Work-Confer. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 741–752. Springer, 2019.
- [4] M. Benito-Parejo and M. G. Merayo. An evolutionary algorithm for selection of test cases. In *22nd IEEE Congress on Evolutionary Computation, CEC'20*, pages E–24535: 1–8. IEEE Computer Society, 2020.
- [5] C. Blum and D. Merkle, editors. *Swarm Intelligence: Introduction and Applications*. Springer, 2008.
- [6] P. C. Cañizares, A. Núñez, and M. G. Merayo. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software*, 143:187–207, 2018.
- [7] T. S. Chow. Testing software design modeled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.
- [8] P. Delgado-Pérez and I. Medina-Bulo. Search-based mutant selection for efficient test suite improvement: Evaluation and results. *Information and Software Technology*, 104:130–143, 2018.
- [9] M. Dorigo, V. Maniezzo, and A. Colnani. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics B*, 26(1):29–41, 1996.
- [10] R. Feldt, S. M. Poulding, D. Clark, and S. Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *9th IEEE Int. Conf. on Software Testing, Verification and Validation, ICST'16*, pages 223–233. IEEE Computer Society, 2016.
- [11] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite-state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.
- [12] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. A tool for domain-independent model mutation. *Science of Computer Programming*, 163:85–92, 2018.
- [13] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. Wodel-Test: a model-based framework for language-independent mutation testing. *Software and Systems Modeling (in press)*, 2021.
- [14] D. Griñán and A. Ibas. Generating tree inputs for testing using evolutionary computation techniques. In *22nd IEEE Congress on Evolutionary Computation, CEC'20*, pages E–24267: 1–8. IEEE Computer Society, 2020.
- [15] D. Griñán, A. Ibas, and M. Núñez. Grammar-based tree swarm optimization. In *2019 IEEE Int. Conf. on Systems, Man and Cybernetics, SMC'19*, pages 76–81. IEEE Press, 2019.
- [16] R. M. Hierons, M. G. Merayo, and M. Núñez. Bounded reordering in the distributed test architecture. *IEEE Transactions on Reliability*, 67(2):522–537, 2018.
- [17] A. Ibas, D. Griñán, and M. Núñez. GPTSG: a Genetic Programming Test Suite Generator using Information Theory measures. In *15th Int. Work-Confer. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 716–728. Springer, 2019.
- [18] A. Ibas, R. M. Hierons, and M. Núñez. Using Squeeziness to test component-based systems defined as Finite State Machines. *Information & Software Technology*, 112:132–147, 2019.
- [19] A. Ibas and M. Núñez. SqSelect: Automatic assessment of failed error propagation in state-based systems. *Expert Systems with Applications*, 174:114748, 2021.
- [20] A. Ibas, M. Núñez, and R. M. Hierons. Using mutual information to test from Finite State Machines: Test suite selection. *Information & Software Technology*, 132:106498, 2021.
- [21] ISO/IEC JTC1/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.
- [22] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.
- [23] J. Kennedy and R. Eberhart. Particle swarm optimization. In *3rd Int. Conf. on Neural Networks, ICNN'95*, pages 1942–1948. IEEE Computer Society, 1995.
- [24] J. R. Koza. *Genetic programming*. MIT Press, 1993.
- [25] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, 2014.
- [26] J. D. McGregor and D. A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison Wesley object technology series. Pearson / Prentice Hall, 2001.
- [27] R. I. McKay, N. X. Hoai, P. A. Whigham, Y. S., and M. O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010.
- [28] M. G. Merayo, R. M. Hierons, and M. Núñez. Passive testing with asynchronous communications and timestamps. *Distributed Computing*, 31(5):327–342, 2018.
- [29] M. G. Merayo, R. M. Hierons, and M. Núñez. A tool supported methodology to passively test asynchronous systems with multiple users. *Information & Software Technology*, 104:162–178, 2018.
- [30] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, 3rd edition, 2011.
- [31] A. Núñez, M. G. Merayo, R. M. Hierons, and M. Núñez. Using genetic algorithms to generate test sequences for complex timed systems. *Soft Computing*, 17(2):301–315, 2013.
- [32] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275 – 378. Elsevier, 2019.
- [33] S. Splaine and S. P. Jaskiel. *The web testing handbook*. STQE Pub., 2001.
- [34] M. Srinivas and L. M. Patnaik. Genetic algorithms: A survey. *IEEE Computer*, 27:17–27, 1994.

10.7 Feature Selection using Evolutionary Computation Techniques for Software Product Line Testing

Authors	Alfredo Ibias and Luis Llana
Title	Feature Selection using Evolutionary Computation Techniques for Software Product Line Testing
Publication Type	Conference
Venue	2020 IEEE Congress on Evolutionary Computation
Year	2020
DOI/URL	https://doi.org/10.1109/CEC48606.2020.9185675
Pages	8
Authors' Contributions	Ibias and Llana developed the theory. Ibias designed the experiments. Ibias developed and executed the experiments. Ibias and Llana wrote the manuscript. Ibias and Llana reviewed the manuscript.

Feature Selection using Evolutionary Computation Techniques for Software Product Line Testing

Alfredo Ibias
Universidad Complutense de Madrid
 Madrid, Spain
 aibias@ucm.es

Luis Llana
Universidad Complutense de Madrid
 Madrid, Spain
 llana@ucm.es

Abstract—Software product lines are an excellent mechanism in the development of software. Testing software product lines is an intensive process where selecting the right features where to focus it can be a critical task. Selecting the *best* combination of features from a software product line is a complex problem addressed in the literature. In this paper, we address the problem of finding the combination of features with the highest probability of being requested from a software product line with probabilities. We use Evolutionary Computation techniques to address this problem. Specifically, we use the Ant Colony Optimization algorithm to find the *best* combination of features. Our results report that our framework overcomes the limitations of the brute force algorithm.

Index Terms—Software Testing, Evolutionary Computation, Software Product Lines

I. INTRODUCTION

During the last years, software product lines (in short, SPLs) have become a widely adopted mechanism for efficient software development. They are a set of similar software-based systems produced from a set of software features that are shared between them using a common means of production. The main goal of SPLs is to increase the productivity of creating software products. They achieve this goal by selecting those software systems that are better for a specific criterion (e.g., a software system is less expensive than others; it requires less time to be executed, etc.). Currently, different approaches for representing the product line organisation can be found in the literature, such as FODA [38], RSEB [30], PLUSS [28] and SPLA [2].

The formal language SPLA was introduced in [4]. The authors present a formal language capable to express the FODA diagrams (Figure 1 shows some examples). A recent work [14] has proposed a probabilistic extension to SPLA: SPLA^P. This proposal includes a probability whenever there is a choice in the representation of the SPL. This probability allows us to know which features are requested more frequently and which feature combinations are the most popular ones. This knowledge is beneficial to make decisions about the SPL, the resources destined to each feature, and the SPL updates.

Software testing [1] is the main validation technique to assess the reliability of complex software systems. When

This work has been supported by the Spanish MINECO-FEDER (grant number FAME, RTI2018-093608-B-C31) and the Region of Madrid (grant number FORTE-CM, S2018/TCS-4314).

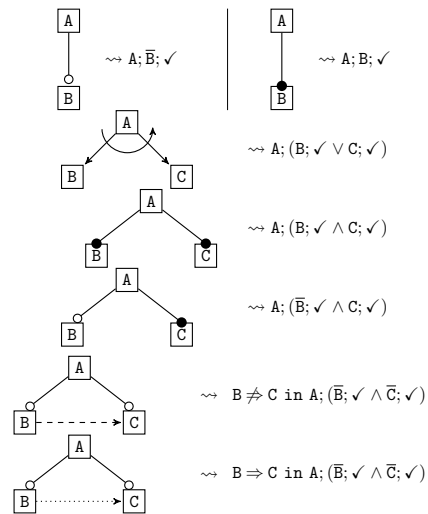


Fig. 1: Examples of translation from FODA Diagrams into SPLA.

testing SPLs [18], it is crucial to distribute the resources between the different features of the line in a smart way. We are particularly concerned about the testing resources assigned to each component of the software system. These resources are limited so, if we had information about the most requested components of the software system (what we have called features) we could assign them more testing resources. Therefore, the information about the more requested features from the SPL can be vital to distribute the resources during testing. This problem has been addressed in [14]. Besides, when producing SPLs, we want to pay attention to have a better engagement between the features that are more commonly shipped together. Therefore, more development and testing resources are ideally focused on those features and their engagement. However, it is not trivial (and sometimes not feasible) to know the probability of use of all the possible feature combinations that a product line can produce. Therefore, being able to know which feature combinations are more used can be critical when testing SPLs. In this paper, we present an approach to get those feature combinations.

Heuristic search algorithms are techniques commonly used in Mathematics and Computer Science either to optimize a

function or to find the best possible solution for a given problem. These techniques, also referred to as *metaheuristics*, can be roughly divided into three categories: *global search techniques* such as simulated annealing [40], *evolutionary techniques* such as genetic algorithms [29], and *constructive techniques* such as Ant Colony Optimization [26], [27].

In this paper, we have used an Evolutionary Computation technique to select features from a SPL. We have chosen this particular family of techniques because it is well suited for parallel searching, and it also combines the knowledge obtained by each member of the population of candidate solutions. Furthermore, by starting with a random set of candidate solutions, the algorithm can quickly obtain a candidate solution that suits our goal. More specifically, we have implemented a variation of a typical Evolutionary approach: the Ant Colony Optimization (ACO) algorithm [26], [27]. In this variation we have combined the classical ACO algorithm with a SPLA^P expressions interpreter to be able to search, in an *a priori* unknown search space, the feature combination with a higher probability that the SPLA^P expression can produce.

Then, we show the results of some experiments. In them, we can clearly see how our framework can solve the feature selection problem we have raise, beating in time (saving around 67% of time) to the standard brute force algorithm. Also, we show how our framework can obtain as good results as the ones obtained by the brute force algorithm, getting in mean feature combinations with only an 18% less probability. We also analyse some extreme cases, where the randomisation factors have had a high impact.

Finally, we would like to mention that the use of metaheuristics in testing is not new [3], [5], [21], [23], [32], [33], [36], [45]. In particular, there is some work on the application of the swarm idea to testing [31], [48]. The novelty of our approach resides in the fact that we are using this metaheuristics to the feature selection problem as a previous step before properly testing an SPL.

The rest of the paper is organised as follows. In Section II, we present some theoretical concepts that we use along with our paper. In Section III, we introduce our feature selection framework. In Section IV, we present our experiments and discuss the results. In Section V, we review some of the possible threats to the validity of our results. Finally, in Section VI, we give conclusions and outline some directions for future work.

II. PRELIMINARIES

In this section we briefly introduce the concepts that will be used along the work. First, we define the concepts of SPL, software feature, and SPLA^P process algebra.

A Software Product Line (SPL) is, as defined by The Carnegie Mellon Software Engineering Institute, “a set of software-intensive systems that share a common managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” [42]. The idea of SPLs is that they allow producing, from a set of predefined software

features, a piece of software that includes all those features. SPLA^P [14] extends these classic SPLs concepts with a notion of probability. This probability indicates the preferences of the users in front of a choice.

We have decided to represent SPLs using the SPLA^P process algebra [14]. In order to work with this algebra, we will consider a set of *features*, denoted by \mathcal{F} , and the elements A, B, C, \dots will stand for elements of \mathcal{F} . We have a special feature $\checkmark \notin \mathcal{F}$ to mark the end of a product. We consider non-degenerated probabilities in the syntax, that is, for all probability p we have $0 < p < 1$.

Definition 1: A *probabilistic software product line* is a term generated by the following grammar:

$$P ::= \checkmark \mid \text{nil} \mid A; P \mid \bar{A};_p P \mid P \vee_p Q \mid P \wedge Q \mid A \not\Rightarrow B \text{ in } P \mid A \Rightarrow B \text{ in } P \mid P \setminus A \mid P \Rightarrow A$$

where $A, B \in \mathcal{F}$, $\checkmark \notin \mathcal{F}$ y $p \in (0, 1)$. The set of all software product lines is denoted by SPLA^P.

This probabilistic process algebra has an operational semantics to guide how to interpret the expressions of the algebra. The operational semantics rules are displayed in Figure 2. A relevant property of this algebra is that each time we find a probability, the feature from the left-hand side gets the probability p , and the feature from the right-hand side gets the probability $1 - p$, but those are not the full probabilities of each feature. In fact, the probability of a single feature in a SPL is a measure of its occurrences in the set of products.

The operational semantics of an SPLA^P expression P is a tree structure. The set of products of P is computed by traversing this tree.

Definition 2: Let $P, Q \in \text{SPLA}^P$. We write $P \xrightarrow{s} Q$ if there exists a sequence of consecutive transitions

$$P = P_0 \xrightarrow{a_1}_{p_1} P_1 \xrightarrow{a_2}_{p_2} P_2 \cdots P_{n-1} \xrightarrow{a_n}_{p_n} P_n = Q$$

where $n \geq 0$, $s = a_1 a_2 \cdots a_n$ and $p = p_1 \cdot p_2 \cdots p_n$. We say that s is a trace of P .

Let $s \in \mathcal{F}^*$ be a trace of P . We define the product $[s] \subseteq \mathcal{F}$ as the set consisting of all features belonging to s .

Let $P \in \text{SPLA}^P$. We define the set of probabilistic products of P , denoted by $\text{prod}^P(P)$, as the set

$$\text{prod}^P(P) = \{(pr, p) \mid p > 0 \wedge p = \sum \{q \mid P \xrightarrow{s\checkmark}_q Q \wedge [s] = pr\}\}$$

However, computing this tree is computationally expensive, and can be infeasible in some cases. That is the reason we need to work with evolutionary computation techniques. And as this tree can be seen as a directed graph, then a convenient evolutionary technique to search this tree as a search space is the Ant Colony Optimization algorithm.

The Ant Colony Optimization algorithm is a well known algorithm in the Evolutionary Computation field. It is a distributed algorithm of search in a graph-like search space. It consists of a set of *ants*, that are the agents that explore the search space. Then, each ant look for the shortest path from the initial node to the target node, choosing their next move based on a random choice modified by the weigh of each

<p>[tick] $\frac{\checkmark \xrightarrow{a} \text{nil}}{\bar{a};_p P \xrightarrow{a} P}$</p> <p>[ofeat1] $\frac{P \xrightarrow{a} P_1}{P \vee_q Q \xrightarrow{a} P_1}$</p> <p>[cho1] $\frac{P \xrightarrow{a} P_1}{P \wedge Q \xrightarrow{a} P_1}$</p> <p>[con1] $\frac{P \xrightarrow{a} \text{nil}, Q \xrightarrow{a} \text{nil}}{P \wedge Q \xrightarrow{a} \text{nil}}$</p> <p>[con3] $\frac{P \xrightarrow{a} P_1, Q \xrightarrow{a} P_1}{P \wedge Q \xrightarrow{a} P_1}$</p> <p>[con4] $\frac{P \xrightarrow{c} P_1, C \neq A}{A \Rightarrow B \text{ in } P \xrightarrow{c} A \Rightarrow B \text{ in } P_1}$</p> <p>[req1] $\frac{P \xrightarrow{a} \text{nil}}{A \Rightarrow B \text{ in } P \xrightarrow{a} \text{nil}}$</p> <p>[req3] $\frac{A \Rightarrow B \text{ in } P \xrightarrow{a} \text{nil}}{P \xrightarrow{c} P_1, C \neq A, C \neq B}$</p> <p>[excl1] $\frac{P \xrightarrow{b} P_1}{A \not\Rightarrow B \text{ in } P \xrightarrow{b} A \not\Rightarrow B \text{ in } P_1}$</p> <p>[excl3] $\frac{P \xrightarrow{b} P_1}{A \not\Rightarrow B \text{ in } P \xrightarrow{b} P_1 \setminus A}$</p> <p>[forb1] $\frac{P \xrightarrow{a} \text{nil}}{P \Rightarrow A \xrightarrow{a} \checkmark}$</p> <p>[mand1] $\frac{P \xrightarrow{b} P_1, A \neq B}{P \Rightarrow A \xrightarrow{b} P_1 \Rightarrow A}$</p> <p>[mand3] $\frac{P \xrightarrow{a} \text{nil}}{P \Rightarrow A \xrightarrow{a} \checkmark}$</p>	<p>[feat] $\frac{A; P \xrightarrow{a} P}{\bar{a};_p P \xrightarrow{a} \text{nil}}$</p> <p>[ofeat2] $\frac{Q \xrightarrow{a} Q_1}{P \vee_p Q \xrightarrow{a} Q_1}$</p> <p>[cho2] $\frac{Q \xrightarrow{a} Q_1}{P \wedge Q \xrightarrow{a} Q_1}$</p> <p>[con2] $\frac{P \xrightarrow{a} \text{nil}, Q \xrightarrow{a} Q_1}{P \wedge Q \xrightarrow{a} Q_1}$</p> <p>[con5] $\frac{P \xrightarrow{a} P_1}{A \Rightarrow B \text{ in } P \xrightarrow{a} P_1 \Rightarrow B}$</p> <p>[req2] $\frac{P \xrightarrow{a} P_1}{A \not\Rightarrow B \text{ in } P \xrightarrow{a} P_1 \not\Rightarrow B}$</p> <p>[excl2] $\frac{P \xrightarrow{a} P_1}{A \not\Rightarrow B \text{ in } P \xrightarrow{a} P_1 \setminus B}$</p> <p>[excl4] $\frac{P \xrightarrow{a} \text{nil}}{A \not\Rightarrow B \text{ in } P \xrightarrow{a} \text{nil}}$</p> <p>[forb2] $\frac{P \setminus A \xrightarrow{a} \text{nil}}{P \Rightarrow A \xrightarrow{a} \text{nil}}$</p> <p>[mand2] $\frac{P \xrightarrow{a} P_1}{P \Rightarrow A \xrightarrow{a} P_1}$</p>
--	---

$A, B, C \in \mathcal{F}, a \in \mathcal{F} \cup \{\checkmark\}$

Fig. 2: SPLA^P operational semantics.

path and the *pheromones* released by other ants that previously performed that move.

Formally, an Ant Colony Optimization algorithm [26] needs a combinatorial optimization problem to be solved. This problem can be defined as:

Definition 3: A model $P = (\mathbf{S}, \Omega, f)$ of a combinatorial optimization problem consists of:

- a search space \mathbf{S} defined over a finite set of discrete decision variables $X_i, i = 1, \dots, n$.
- a set Ω of constraints among the variables.
- an objective function $f : \mathbf{S} \rightarrow \mathbb{R}_0^+$ to be minimised.

The generic variable X_i takes values in $D_i = v_i^1, \dots, v_i^{|D_i|}$. A feasible solution $s \in \mathbf{S}$ is a complete assignment of values to variables that satisfies all constraints in Ω . A solution $s^* \in \mathbf{S}$ is called a global optimum if and only if $f(s^*) \leq f(s) \forall s \in \mathbf{S}$.

Then, from this setup we can generate the *construction graph* $G_C(\mathbf{V}, \mathbf{E})$, where \mathbf{V} is a set of vertices and \mathbf{E} is a set of edges. This graph can be obtained from the set of solution components C in two ways: components may be represented either by vertices or by edges. Artificial ants move from vertex to vertex along the edges of the graph, incrementally building a partial solution.

Additionally, ants deposit a certain amount of pheromone on the components, that is, either on the vertices or on the edges that they traverse. The amount of $\Delta\tau$ pheromone

Set parameters;

Initialise pheromone trails;

while *termination criterion not reached* **do**

Construct Ant Solutions;

Update Pheromones;

end

Algorithm 1: Ant Colony Optimization algorithm: general scheme

deposited may depend on the quality of the solution found. Subsequent ants use the pheromone information as a guide toward promising regions of the search space.

The ACO general scheme is presented in Algorithm 1. In each iteration, each ant generates a solution. Then, the global state updates the pheromones left by the ants in their solution path. Following there is a more detailed explanation of each step:

Construct Ant Solutions: At each iteration, a set of m ants generates solutions taking elements from a finite set of available solution components $\mathbf{C} = \{c_{ij}\}, i = 1, \dots, n, j = 1, \dots, |D_i|$. The construction starts from an empty solution set $s^P = \emptyset$ and, at each step, the ant extends its partial solution adding a feasible solution element from the set $\mathbf{N}(s^P) \subseteq \mathbf{C}$, that is the set of elements of \mathbf{C} that can be added to the partial solution s^P without violating any constraint from Ω .

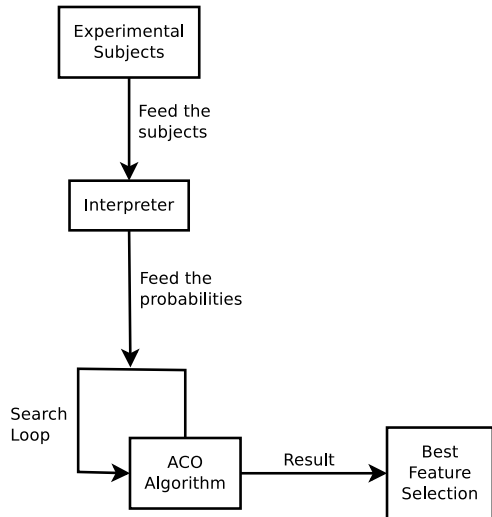


Fig. 3: Schema of the proposed feature selection framework

This process can be seen as a walk on the *construction graph* $G_C(\mathbf{V}, \mathbf{E})$. The choice of a solution component from $\mathbf{N}(s^P)$ is guided by a stochastic mechanism, which is biased by the pheromone associated with each of the elements of $\mathbf{N}(s^P)$. The rule for the stochastic choice of solution components vary across different ACO algorithms but, in all of them are inspired by the behaviour of real ants.

Update Pheromones: The pheromone update aims to increase the pheromone values associated with good or promising solutions, and to decrease those that are associated with bad ones. Usually, this is achieved by decreasing all the pheromone values through pheromone evaporation, and by increasing the pheromone levels associated with a chosen set of good solutions.

III. FEATURE SELECTION FRAMEWORK

In this section, we present our feature selection framework. Its main goal is to find a combination of features that have a high enough probability for a given SPL, that is, a SPLA^P expression. As we have already explained, we decided to rely on evolutionary computation techniques to compute these feature combinations. Specifically, we decided to use an ACO algorithm because we considered it to be the most suitable one for this problem. In future work we will address the use of other evolutionary computation techniques.

Below, we briefly describe the main components of our framework:

- A software product line, it is the system that we are working with. It is represented as a probabilistic algebra expression, specifically, as a SPLA^P expression [14].
- A SPLA^P interpreter that allows us to explore the search space generated by the SPLA^P expression without fully computing it.
- An Ant Colony Optimization algorithm. It leads the search for a feature combination with high probability.

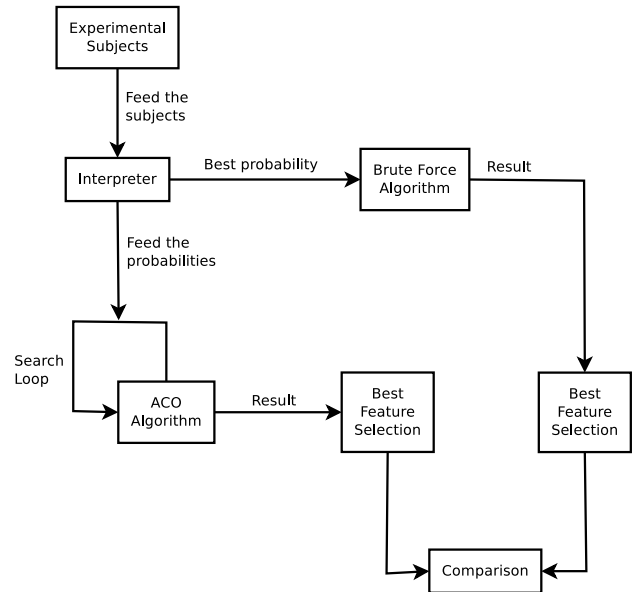


Fig. 4: Schema of the experiments flow

A graphical representation of our framework can be found in Figure 3.

In our framework, we receive a SPL, expressed as a SPLA^P expression, intending to find a set of features that fulfil the requisites of the SPL and at the same time has a high probability. This probability, coming from the SPLA^P expression, usually represents the probability of each feature to be chosen, but it does not have to be limited to that purpose [14]. However, in our scenario, we assume that the probabilities represent the likelihood of each feature to be chosen, because we are looking for the feature combination that has the higher probability to be selected and therefore the one that needs more testing focus when testing the SPL.

Then, with the SPLA^P expression, we interpret it to be able to execute an ACO algorithm over it. As our target is to find a set of features with a high probability, but without having to compute all the probabilities of all the possible combinations of features of the SPLA^P expression, we need to have an interpreter. This interpreter has to, given a feature of the SPLA^P expression, return the probability of that feature, but without computing the full SPLA^P expression tree.

For our ACO algorithm to work, we need to have a combinatorial optimization problem. Then, we need to express our problem as a combinatorial optimization one, in the following way:

- Search space \mathbf{S} : it is the full SPLA^P tree, whose decision variables are the feature to choose next.
- Set of constraints Ω : it is composed by:
 - A constraint that states that a valid path should end in a \checkmark feature.
 - A constraint that states that a valid path should fulfil the SPLA^P expression constraints.
- Objective Function f : it is the function assigning to each

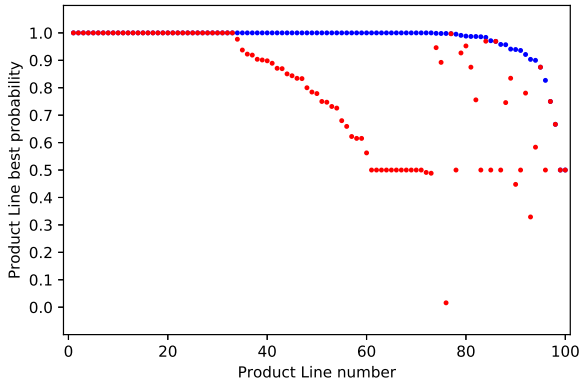


Fig. 5: Sorted obtained probabilities (blue = brute force, red = ACO)

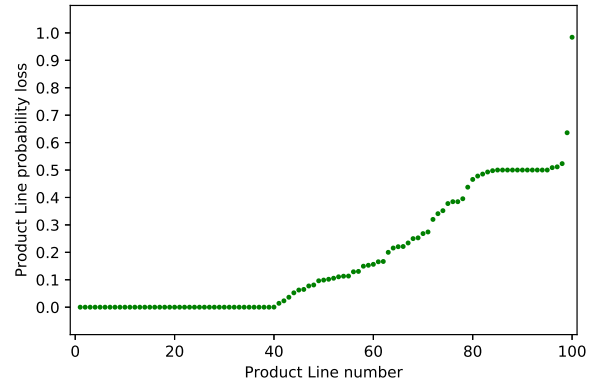


Fig. 7: Sorted probability loss

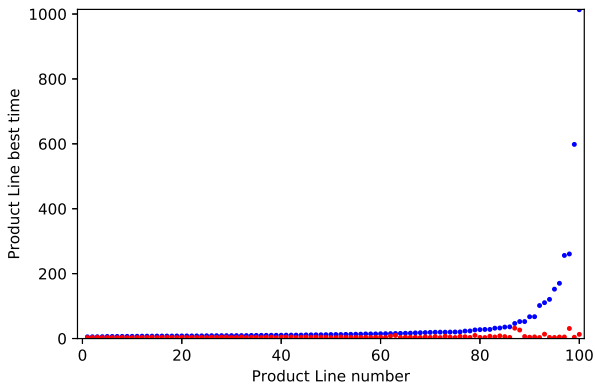


Fig. 6: Sorted obtained times (blue = brute force, red = ACO)

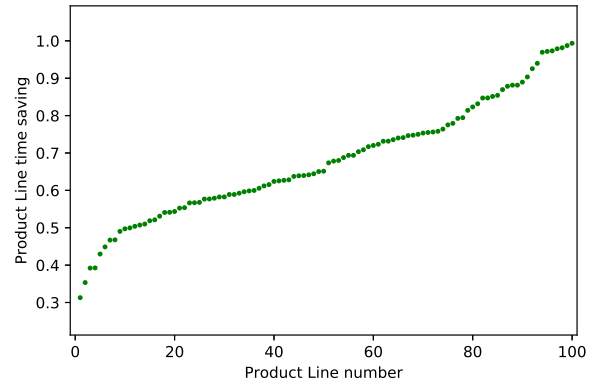


Fig. 8: Sorted time saving

set of features their probability in the $SPLA^P$ expression. In this case, we look to maximize it.

Then, with this well defined combinatorial optimization problem, the ACO algorithm follows the general scheme presented in Algorithm 1. The only modification is the fact that, when exploring the search space, the ants are generating it instead of having a memory variable storing all the information. Thus, the ACO algorithm has to work with the $SPLA^P$ interpreter in order to obtain the distances (in this case, the probabilities).

Another important fact about the ACO algorithm is that the distance between nodes is the probability of choosing the feature associated with the target node. This implies that the total distance travelled by an ant is the product of the probabilities of each step, instead of the sum of the weights as in the classical ACO algorithm.

IV. EXPERIMENTS

In this section, we present an experiment we performed intending to evaluate the suitability of our proposed framework. The schema of the experiment (graphically presented

in Figure 4) is very similar to our general framework (see Figure 3). There is, however, a slight difference. To be able to compare the performance of our ACO algorithm with respect to computing the full tree from the $SPLA^P$ expression, we decided to also compute the feature combination with higher probability by a brute force algorithm so that we can compare the performance in time versus the difference in the probabilities obtained. However, although we have computed all the probabilities from the $SPLA^P$ expression, we should not use those probabilities for the ACO algorithm, but instead, it should compute their own probabilities based on which paths the ants take.

For our experiments, we needed four elements:

- A set of $SPLA^P$ expressions as experimental subjects.
- An interpreter of $SPLA^P$ expressions.
- A brute force algorithm to find the set of features with the highest probability.
- An Ant Colony Optimization algorithm implementation¹, modified to be able to work with the $SPLA^P$ probabilistic

¹We used the code from <https://github.com/pjmattingly/ant-colony-optimization> as a starting point for our adapted ACO algorithm.

process algebra.

The *experimental subjects set* consists of 100 $SPLA^P$ expressions automatically generated using the BeTTY tool [47] and stored in an fodaA format in .xml files. The *interpreter* allows us to compute the probabilities of each feature in an ad-hoc way, so we do not fully compute the $SPLA^P$ expression tree. The *brute force algorithm* is an algorithm that asks the interpreter, for each feature of the $SPLA^P$ expression, which is its probability. It returns the feature with the highest probability, but at the cost of a longer computation. Therefore, we need to work with small $SPLA^P$ expressions to end in a reasonable time. The *ACO algorithm* is an ACO algorithm that searches the set of features with maximum probability from the $SPLA^P$ expression, using the interpreter to compute the probabilities. For our ACO algorithm, we decided to use 5 ants, and perform a maximum of 10 iterations over the main loop. The pheromone evaporation coefficient was 0.4, and the pheromone constant was 1000. Finally, the α and β values used to calculate the attractiveness of each path were 0.5 and 1.2 respectively. The ants and iterations values are so low because we are working with small $SPLA^P$ expressions (as explained before), and therefore higher values are unnecessary and a waste of resources.

With the interpreter and the ACO algorithm, we were able to implement our framework and test it against the SPLs represented as $SPLA^P$ expressions coming from the experimental subjects set. To perform interesting experiments, we took the 100 $SPLA^P$ expressions and computed their *best* selection of features, that is, the one with the highest probability. We computed those selections both with the brute force algorithm and the ACO algorithm and compared the probabilities obtained and the computation times. The code and results of the experiments can be found at <https://github.com/Colosu/FSACO>.

In Table I we can find the probabilities obtained for each $SPLA^P$ expression, as well as the computation times. In Figure 5, we can compare the probabilities obtained graphically. The same happens for times in Figure 6. In both figures, the results from the brute force algorithm are in blue and the results from the ACO algorithm are in red. They are sorted to facilitate the display of the results. Finally, in Figure 7 we can observe the sorted probability *losses* obtained, that is, the difference (in percentage) between the probabilities of the best feature combination and the one obtained by our framework. And in Figure 8 we can see the sorted time savings achieved. In mean, we have *loss* an 18.2318% of probability saving at the same time a 67.33% of time. This shows that our method is not only useful to save time. We also obtain good feature combinations: those whose probability is close to the best combination (in terms of probability).

From the results, we can make some interesting remarks. First, it is interesting to see that there are some cases where the ACO algorithm saves more than 97% of the time while losing no probability (that is, giving the feature combination with the highest probability). This is the case of trials number 8 or 34. Another interesting remark is the fact that there are around 40% of the cases where there is no *loss* of probability,

while there is always at least a 30% of time-saving. Moreover, there are only 3 cases (trials 22, 32 and 70) where the *loss* of probability is proportionally higher than the time saving, what can be a result of the randomization factors. However, also due to the randomization factors, there are cases where the time saving comes with a high probability *loss*, being the case of the trial number 32 the clearest case (although the saving in time is still high). In fact, there are only 2 cases (trials 17 and 32) where the *loss* of probability surpasses the 60%.

V. THREATS TO VALIDITY

In this section, we briefly discuss some of the possible threats to the results of our experiments validity. Concerning threats to *internal validity*, which consider uncontrolled factors that might be responsible for the obtained results, the main threat is associated with the possible faults in the developed experiment because they could lead to misleading results. To reduce the impact of this threat, we tested our code with carefully constructed examples for which we could manually check the results. Besides, we repeated the experiment with many subjects to diminish the effect of randomization factors.

The main threat to *external validity*, which concerns conditions that allow us to generalize our findings to other situations, is the different possible SPLs to which we could apply our framework. Such a threat cannot be entirely addressed since the population of possible SPLs is unknown, and it is not possible to sample from this (unknown) population. To diminish this risk, we considered different SPLs in the experiments.

Finally, we considered threats to *construct validity*, which are related to the *reality* of our experiments, that is, whether our experiments reflect real-world situations or not. In our work, the main construct threat is what would happen if we use our framework with much more complex SPLs, which is a matter of future work.

VI. CONCLUSIONS

We have proposed a new framework for feature selection in software product lines with probabilities. Our framework strongly relies on Evolutionary Computation techniques to perform feature selection. Specifically, we have used a novel variant of ACO to deal with an *a priori* unknown search space. With this new framework, we can obtain new feature combinations for a given SPL without computing all the possible feature combinations, which is a time-consuming task. Besides, to present the new framework, in this paper, we have reported some of our experiments. Their goal was to show that the *loss* in the feature combination probabilities produced by our framework pays back with the saving of time when computing those combinations.

For future work we have already identified several research directions concerning applicability, scalability, suitability and adaptability of our framework. First, we plan to adapt our framework to perform feature selection in SPLs where instead of probabilities we have costs. Since there are similarities, but also differences, between probabilities and costs we will try to

Trial Number	Brute Force Probability	ACO Probability	Probability Loss	Brute Force Time	ACO Time	Time Saving
1	0.9568	0.7458	0.2204	14.1544	3.6634	0.7411
2	0.9868	0.7558	0.2341	14.7537	4.2998	0.7085
3	1.0	1.0	0.0	7.5902	3.0380	0.5997
4	0.9578	0.5	0.4780	67.1252	4.0392	0.9398
5	1.0	0.8333	0.1666	35.2642	5.9373	0.8316
6	1.0	0.5	0.5	17.3872	3.9101	0.7751
7	1.0	0.7317	0.2682	12.4005	4.4771	0.6389
8	1.0	1.0	0.0	152.1496	3.223	0.9788
9	0.9971	0.9968	0.0003	19.8590	4.3838	0.7792
10	1.0	1.0	0.0	8.7878	3.5475	0.5963
11	0.9952	0.5	0.4976	17.9624	3.1754	0.8232
12	1.0	0.7843	0.2156	52.3167	6.2011	0.8814
13	1.0	0.9226	0.0773	15.1992	7.4489	0.5099
14	1.0	1.0	0.0	6.8014	2.9485	0.5664
15	0.9	0.5833	0.3518	7.3125	3.4301	0.5309
16	1.0	0.75	0.25	9.5622	3.5962	0.6239
17	0.9034	0.3289	0.6359	22.9624	5.5561	0.7580
18	1.0	1.0	0.0	11.3242	3.6242	0.6799
19	1.0	0.9037	0.0962	11.4791	4.2828	0.6269
20	0.9978	0.8925	0.1055	13.5067	3.6242	0.7316
21	1.0	0.8	0.1999	5.7608	3.0709	0.4669
22	1.0	0.6153	0.3846	5.5168	3.5669	0.3534
23	1.0	0.8344	0.1655	26.4979	9.5484	0.6396
24	1.0	1.0	0.0	8.0393	3.3579	0.5823
25	1.0	0.5	0.5	8.0977	3.1421	0.6119
26	1.0	0.6222	0.3777	7.4093	3.9446	0.4676
27	1.0	1.0	0.0	9.6065	3.5982	0.6254
28	1.0	1.0	0.0	12.3441	2.9144	0.7638
29	0.9910	0.9268	0.0647	260.5862	30.8953	0.8814
30	1.0	1.0	0.0	31.9025	4.7426	0.8513
31	1.0	0.9190	0.0809	52.2687	26.1559	0.4995
32	0.9978	0.0158	0.9840	36.01	3.4776	0.9034
33	0.9836	0.9696	0.0141	8.9961	3.3452	0.6281
34	1.0	1.0	0.0	120.6549	3.4089	0.9717
35	1.0	1.0	0.0	15.8293	3.2840	0.7925
36	1.0	0.8695	0.1304	7.9836	3.5726	0.5525
37	1.0	1.0	0.0	5.4251	3.2981	0.3920
38	1.0	1.0	0.0	10.4942	4.4392	0.5769
39	0.9409	0.8346	0.1129	11.1994	4.4170	0.6056
40	1.0	1.0	0.0	67.2651	5.0123	0.9254
41	1.0	0.8982	0.1017	8.8722	4.0693	0.5413
42	0.875	0.875	0.0	8.7718	3.7987	0.5669
43	1.0	1.0	0.0	6.9869	3.1898	0.5434
44	1.0	0.5	0.5	9.7102	3.1239	0.6782
45	1.0	0.4883	0.5116	6.4319	2.8708	0.5536
46	0.9983	0.9461	0.0523	110.5130	13.4314	0.8784
47	0.6666	0.6666	0.0	7.6843	3.3206	0.5678
48	1.0	1.0	0.0	101.6822	3.1101	0.9694
49	1.0	0.5625	0.4375	16.2177	4.0925	0.7476
50	1.0	1.0	0.0	8.5078	3.0497	0.6415

(a) First part.

Trial Number	Brute Force Probability	ACO Probability	Probability Loss	Brute Force Time	ACO Time	Time Saving
51	1.0	0.7258	0.2741	6.6376	3.6591	0.4487
52	1.0	0.8709	0.1290	9.8265	4.0051	0.5924
53	0.9359	0.5	0.4657	12.2820	3.0111	0.7548
54	1.0	0.9768	0.0231	32.4471	8.1153	0.7498
55	1.0	1.0	0.0	6.4097	3.2232	0.4971
56	1.0	1.0	0.0	10.5378	3.1247	0.7034
57	0.5	0.5	0.0	18.9448	3.5220	0.8140
58	1.0	1.0	0.0	27.3563	3.5678	0.8695
59	0.9882	0.9523	0.0362	256.0567	4.7459	0.9814
60	1.0	0.9375	0.0625	19.6477	7.1258	0.6373
61	1.0	0.7472	0.2527	1013.2517	12.8199	0.9873
62	1.0	0.5	0.5	14.2741	3.7750	0.7355
63	1.0	0.6590	0.3409	11.5821	4.0514	0.6502
64	1.0	0.5	0.5	10.0417	2.8096	0.7201
65	1.0	0.5	0.5	6.7181	2.8282	0.5790
66	1.0	1.0	0.0	9.9793	4.1015	0.5889
67	0.8269	0.5	0.3953	8.0312	3.2244	0.5985
68	1.0	0.5	0.5	12.9420	3.5798	0.7233
69	1.0	1.0	0.0	598.2494	3.8589	0.9935
70	1.0	0.4912	0.5087	6.3343	3.2288	0.4902
71	1.0	0.5	0.5	8.2636	3.4499	0.5825
72	1.0	1.0	0.0	27.9893	3.0902	0.8895
73	0.9714	0.5	0.4852	9.0948	4.4819	0.5071
74	1.0	0.5	0.5	23.3201	3.3990	0.8542
75	0.9212	0.7804	0.1528	8.8912	4.2803	0.5185
76	1.0	0.6153	0.3846	8.9465	3.1816	0.6443
77	1.0	0.5	0.5	14.1738	3.4967	0.7532
78	1.0	0.7787	0.2212	10.4757	4.3059	0.5889
79	1.0	1.0	0.0	7.6632	3.6695	0.5211
80	1.0	0.68	0.3199	10.5954	3.6948	0.6512
81	1.0	1.0	0.0	9.1105	2.9731	0.6736
82	1.0	1.0	0.0	7.3558	3.3774	0.5408
83	1.0	1.0	0.0	7.8904	3.0344	0.6154
84	0.9858	0.5	0.4928	170.1517	4.5620	0.9731
85	1.0	1.0	0.0	12.7440	3.4231	0.7313
86	0.75	0.75	0.0	5.2166	2.9751	0.4296
87	1.0	1.0	0.0	11.8614	3.6342	0.6936
88	1.0	1.0	0.0	14.9899	3.0819	0.7943
89	1.0	1.0	0.0	13.2409	3.4413	0.74
90	0.9393	0.4477	0.5233	10.1325	4.2891	0.5766
91	1.0	0.8505	0.1494	18.3256	5.1875	0.7169
92	1.0	0.9012	0.0987	15.5291	9.4323	0.3926
93	0.9690	0.9690	0.0	46.3790	31.8633	0.3129
94	1.0	1.0	0.0	20.1493	3.0774	0.8472
95	1.0	0.8892	0.1107	20.3595	6.3576	0.6877
96	0.5	0.5	0.0	7.6664	3.8037	0.5038
97	0.9870	0.875	0.1134	19.4409	5.9538	0.6937
98	1.0	0.8437	0.1562	16.5696	4.0402	0.7561
99	1.0	1.0	0.0	28.0511	7.1120	0.7464
100	1.0	0.5	0.5	19.6242	3.0028	0.8469

(b) Second part.

TABLE I: Results of the experiments.

incorporate into our framework recent work on formal testing of fuzzy systems [12], [13], where probabilities are *replaced* by confidences, and on testing using Information Theory concepts [37]. Second, concerning scalability, we would like to consider more complex SPLs and check whether our technique scales well. In addition, we would like to use current approaches to mutation testing [15], [20], [22] to efficiently generate and process big amount of mutants representing either non-optimal or faulty selections of features. Concerning suitability, we have two orthogonal lines of work. First, we would like to compare our ACO approach with other metaheuristics such as Bee Swarm [39] and Water Based [46] metaheuristics and Collective Intelligence [24], [25], [43], [44]. Second, we would like to consider SPLs with existing feature selections, produced by an expert, and compare the quality of the existing feature selections and the ones produced by our framework. Concerning adaptability, we would like to assess the useful-

ness of our methodology in other frameworks. In particular, we consider more complicated feature selection frameworks where we have to work with deadlock avoidance/analysis [9]–[11], [19], so that we can scale the feature selection from single systems to entire software families. A second line of work consists in applying our framework to formal models of cloud [6], [7], [16] and distributed [34], [35] systems because they are highly configurable and, therefore, will induce SPLs with many features. Finally, it is interesting the possibility of integration of our feature selection framework to existing tools like ProFeat [17], to represent product lines, PRISM [41], to analyse probabilistic systems, and MEdit4CEP-CPN [8], to represent complex events.

REFERENCES

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

- [2] C. Andrés, C. Camacho, and L. Llana. A formal framework for software product lines. *Information & Software Technology*, 55(11):1925–1947, 2013.
- [3] C. Andrés, M. G. Merayo, and M. Núñez. Multi-objective genetic algorithms: Construction and recombination of passive testing properties. In *22nd Int. Conf. on Software Engineering & Knowledge Engineering, SEKE'10*, pages 405–410. Knowledge Systems Institute, 2010.
- [4] César Andrés, Carlos Camacho, and Luis Llana. A formal framework for software product lines. *Inf. Softw. Technol.*, 55(11):1925–1947, 2013.
- [5] M. Benito-Parejo, I. Medina-Bulo, M. G. Merayo, and M. Núñez. Using genetic algorithms to generate test suites for FSMs. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 741–752. Springer, 2019.
- [6] A. Bernal, M. E. Cambronero, A. Núñez, P. C. Cañizares, and V. Valero. Improving cloud architectures using UML profiles and MZT transformation techniques. *The Journal of Supercomputing*, 75(12):8012–8058, 2019.
- [7] A. Bernal, M. E. Cambronero, V. Valero, A. Núñez, and P. C. Cañizares. A framework for modeling cloud infrastructures and user interactions. *IEEE Access*, 7:43269–43285, 2019.
- [8] J. Boubeta-Puig, G. Díaz, H. Macià, V. Valero, and G. Ortiz. MEdit4CEP-CPN: An approach for complex event processing modeling by prioritized colored Petri nets. *Information Systems*, 81:267–289, 2019.
- [9] M. Bravetti, M. Carbone, and G. Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017.
- [10] M. Bravetti, M. Carbone, and G. Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.*, 722:19–51, 2018.
- [11] M. Bravetti and G. Zavattaro. On the expressive power of process interruption and compensation. *Mathematical Structures in Computer Science*, 19(3):565–599, 2009.
- [12] I. Calvo, M. G. Merayo, and M. Núñez. A methodology to analyze heart data using fuzzy automata. *Journal of Intelligent & Fuzzy Systems*, 37(6):7389–7399, 2019.
- [13] I. Calvo, M. G. Merayo, M. Núñez, and F. Palomo-Lozano. Conformance relations for fuzzy automata. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 753–765. Springer, 2019.
- [14] C. Camacho, L. Llana, A. Núñez, and M. Bravetti. Probabilistic software product lines. *Journal of Logical and Algebraic Methods in Programming*, 107:54 – 78, 2019.
- [15] P. C. Cañizares, A. Núñez, and M. G. Merayo. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software*, 143:187–207, 2018.
- [16] P. C. Cañizares, A. Núñez, J. de Lara, and L. Llana. MT-EA4Cloud: A methodology for testing and optimising energy-aware cloud systems. *Journal of Systems and Software*, 163:110522:1–110522:25, 2020.
- [17] P. Chrszon, C. Dubsclaff, S. Klüppelholz, and C. Baier. Profeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects of Computing*, 30(1):45–75, 2018.
- [18] M. Cordy, P. Heymans, P. Schobbens, A. M. Sharifloo, C. Ghezzi, and A. Legay. Verification for reliable product lines. *CoRR*, abs/1311.1343, 2013.
- [19] F. S. de Boer, M. Bravetti, M. D. Lee, and G. Zavattaro. A petri net based modeling of active objects and futures. *Fundam. Inform.*, 159(3):197–256, 2018.
- [20] P. Delgado-Pérez and I. Medina-Bulo. Search-based mutant selection for efficient test suite improvement: Evaluation and results. *Information and Software Technology*, 104:130–143, 2018.
- [21] P. Delgado-Pérez, I. Medina-Bulo, and M. Núñez. Using evolutionary mutation testing to improve the quality of test suites. In *19th IEEE Congress on Evolutionary Computation, CEC'17*, pages 596–603. IEEE Computer Society, 2017.
- [22] P. Delgado-Pérez, Louis M. Rose, and I. Medina-Bulo. Coverage-based quality metric of mutation operators for test suite improvement. *Software Quality Journal*, 27(2):823–859, 2019.
- [23] K. Derderian, M. G. Merayo, R. M. Hierons, and M. Núñez. A case study on the use of genetic algorithms to generate test cases for temporal systems. In *11th Int. Conf. on Artificial Neural Networks, IWANN'11, LNCS 6692*, pages 396–403. Springer, 2011.
- [24] G. Díaz, H. Macià, V. Valero, J. Boubeta-Puig, and F. Cuartero. An intelligent transportation system to control air pollution and road traffic in cities integrating CEP and colored petri nets. *Neural Computing and Applications*, 32(2):405–426, 2020.
- [25] G. Díaz, H. Macià, V. Valero, J. Boubeta-Puig, and G. Ortiz. Facilitating the quantitative analysis of complex events through a computational intelligence model-driven tool. *Scientific Programming*, 2019:2604148:1–2604148:17, 2019.
- [26] M. Dorigo, M. Birattari, and T. Stützle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.
- [27] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, 2004.
- [28] M. Eriksson, J. Börstler, and K. Borg. The PLUSS approach - domain modeling with features, use cases and use case realizations. In *Int. Conf. on Software Product Lines, SPLC'05*, pages 33–44, 2005.
- [29] D. E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, 1989.
- [30] M. L. Griss, J. M. Favaro, and M. D'Alessandro. Integrating feature modeling with the RSEB. In *Int. Conf. on Software Reuse, ICSR'98*, pages 76–85, 1998.
- [31] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *22nd ACM SIGSOFT Int. Symposium on Software Testing and Analysis, ISSTA'12*, pages 78–88. ACM Press, 2012.
- [32] L. Gutiérrez-Madroñal, A. García-Domínguez, and I. Medina-Bulo. Evolutionary mutation testing for IoT with recorded and generated events. *Software - Practice & Experience*, 49(4):640–672, 2019.
- [33] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.
- [34] R. M. Hierons, M. G. Merayo, and M. Núñez. Bounded reordering in the distributed test architecture. *IEEE Transactions on Reliability*, 67(2):522–537, 2018.
- [35] R. M. Hierons and M. Núñez. Implementation relations and probabilistic schedulers in the distributed test architecture. *Journal of Systems and Software*, 132:319–335, 2017.
- [36] A. Ibiás, D. Griñán, and M. Núñez. GPTSG: a Genetic Programming Test Suite Generator using Information Theory measures. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 716–728. Springer, 2019.
- [37] A. Ibiás, R. M. Hierons, and M. Núñez. Using Squeeziness to test component-based systems defined as Finite State Machines. *Information & Software Technology*, 112:132–147, 2019.
- [38] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, 1990.
- [39] D. Karaboga and B. Akay. A survey: algorithms simulating bee swarm intelligence. *Artificial Intelligence Review*, 31(1):61, Oct 2009.
- [40] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [41] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Int. Conf. on Computer Aided Verification (CAV'11)*, volume 6806 of LNCS, pages 585–591. Springer, 2011.
- [42] J. D. McGregor. Testing a software product line. In *Testing Techniques in Software Engineering, Pernambuco Summer School on Software Engineering, PSSE'07*, pages 104–140, 2007.
- [43] V. D. Nguyen, H. B. Truong, M. G. Merayo, and N. T. Nguyen. An overview on consensus-based approaches to processing collective inconsistency and knowledge integration. *WIREs Data Mining and Knowledge Discovery*, 9(4):1311:1–1311:9, 2019.
- [44] V. D. Nguyen, H. B. Truong, M. G. Merayo, and N. T. Nguyen. Toward evaluating the level of crowd wisdom using interval estimates. *Journal of Intelligent & Fuzzy Systems*, 37(6):7279–7289, 2019.
- [45] A. Núñez, M. G. Merayo, R. M. Hierons, and M. Núñez. Using genetic algorithms to generate test sequences for complex timed systems. *Soft Computing*, 17(2):301–315, 2013.
- [46] P. Rabanal, I. Rodríguez, and F. Rubio. Applications of river formation dynamics. *Journal of Computational Science*, 22:26–35, 2017.
- [47] S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, and A. Ruiz-Cortés. Betty: Benchmarking and testing on the automated analysis of feature models. In *6th Int. Workshop on Variability Modeling of Software-Intensive Systems, VaMoS'12*, pages 63–71, 2012.
- [48] A. Windisch, S. Wappler, and J. Wegener. Applying particle swarm optimization to software testing. In *9th Genetic and Evolutionary Computation Conference, GECCO'07*, pages 1121–1128. ACM Press, 2007.

10.8 Using Ant Colony Optimisation to Select Features having Associated Costs

Authors	Alfredo Ibias, Luis Llana and Manuel Núñez
Title	Using Ant Colony Optimisation to Select Features having Associated Costs
Publication Type	Conference
Venue	33rd IFIP International Conference on Testing Software and Systems
Year	2021
DOI/URL	<i>To appear</i>
Pages	17
Authors' Contributions	Ibias, Llana and Núñez developed the theory. Ibias designed the experiments. Ibias developed and executed the experiments. Ibias and Llana wrote the manuscript. Llana and Núñez reviewed the manuscript.

Using Ant Colony Optimisation to Select Features having Associated Costs*

Alfredo Ibias¹[0000-0002-3122-4272], Luis Llana¹[0000-0003-1962-1504], and Manuel
Núñez¹[0000-0001-9808-6401]

Universidad Complutense de Madrid, Madrid 28040, Spain
{aibias, llana, manuelnu}@ucm.com

Abstract. Software Product Lines (SPLs) strongly facilitate the automation of software development processes. They combine features to create programs (called *products*) that fulfil certain properties. Testing SPLs is an intensive process where choosing the proper products to include in the testing process can be a critical task. In fact, selecting the *best* combination of features from an SPL is a complex problem that is frequently addressed in the literature. In this paper we use evolutionary algorithms to find a combination of features with low testing cost that include a target feature, to facilitate the integration testing of such feature. Specifically, we use an Ant Colony Optimisation algorithm to find one of the *cheapest* (in terms of testing) combination of features that contains a specific feature. Our results show that our framework overcomes the limitations of both brute force and random search algorithms.

Keywords: Software Product Lines · Integration Testing · Ant Colony Optimisation · Feature Selection.

1 Introduction

Software Product Lines (SPLs) define generic software products, enabling mass customisation. Generally speaking, SPLs provide a systematic and disciplined approach to developing software. SPLs encode a set of similar (software) systems that can be constructed from a specific set of features. These features can be combined according to some specific rules defining which products (that is, which combinations of features) are valid. In this paper we use FODA [22] to represent SPLs. In order to formally reason about FODA diagrams, it is important to have a formal framework to represent FODA diagrams. In previous work, we introduced SPLA [1], an algebra that can provide a precise semantics to these diagrams. The original framework was extended to manage an important aspect of features: their costs. This is captured in the process algebra SPLA-CRIS [4]. In this work, these costs will represent the cost of testing a specific feature of the product. Testing SPLs is fundamental to ensure the quality and

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

reliability of the products generated by them. When testing SPLs [26], it is crucial to distribute the testing resources between the different features of the line in a smart way. One way of distributing such resources is based on the probability of each feature being requested [18]. However, if we do not have such probabilities, we can consider the costs of testing each feature. The idea is that the products with the minimum cost will be easier to test and, therefore, will consume less resources. This situation is ideal when testing the integration of a specific feature into the SPL. For example, if we add a new feature to an existing SPL and we want to test that its integration with the other features does not produce any errors, then it is useful to have a product with lower testing cost because the integration testing process will be faster and/or cheaper.

We are going to focus on the problem of *Integration Testing of Software Components* [21]. Actually, software components can be seen as the features of an SPL. In fact, integration testing within an SPL has gained attention from researchers [7,29,33]. One important aspect of integration testing is its cost: although testing each variant of an SPL may be feasible, it is impossible to independently test all possible (maybe redundant) products [24]. In our approach, we are interested in getting the product that includes a particular feature having the smallest testing cost. Note that the order of the features may be relevant in the complexity and costs of the testing process [34].

In general, testing cost can refer to multiple concepts: from actual monetary cost of testing the integration of the feature into the product, to the necessary time to test such integration, passing through the amount of resources needed to test that integration. In our framework, we only need to know that such cost exists and that it represents the same along all individual costs of the same SPL. Therefore, along this paper we will be talking about testing cost in a broad sense and we will try to minimise it. Finally, regarding the origin of such costs, we will assume that they are provided together with the SPL. Ideally, such costs would be obtained through estimation, approximation or empirical methods and added to the SPL before using the solution presented in this paper.

It is important to clarify what we mean by computing a cheap (or expensive) product of an SPL. In our context, a cheap product is a product that has a low total testing cost compared to the cost of other products. For example, if the testing cost represents the estimated time needed to test such product, then a cheap product would be one whose aggregated time to test it is low compared with other products of the SPL (e.g. hours vs. weeks). Note that cheaper to test products will not necessarily be the ones with a smaller number of features.

Finally, we want to clarify that testing cost is not a proxy for fault detection effectiveness. We are not looking for the product that will arise more faults, but for the one that will be cheaper to test. This is so because our solution looks to fill a very specific need: we have a feature to add to an SPL and we want to cheaply test its integration into the SPL. The goal is not to find all the faults in the introduced feature, but instead ensure that it can be included into products of the SPL. This is specially useful when one has an SPL with hundreds or thousands of features and there is not enough time or resources to test all the possible combinations. Therefore, it is useful to test that the feature can be included into products and that there are no errors when used in combination with other features, what can be tested using any product. One example of such

situation appears when adding a new database to a server SPL. It is necessary to test that the added database is correctly integrated with the other features of the SPL, but the tester only needs to test the integration in one product because all the productions might have the same integration faults.

In this paper we apply Ant Colony Optimisation algorithm (ACO) [9] to select a combination of features from an SPL with the minimum testing cost that contains a given feature. This combination will be later used to test the integration of such feature into the SPL. To the best of our knowledge, this is the first attempt to develop an efficient solution to this problem if we rely on a formal approach (in our case, a process algebra). To develop this algorithm we modify, enhance and extend our recently developed framework [18] so that we select feature combinations with low testing cost from SPLs including testing costs information, and so that we have the requirement of including a given feature in the generated product.

In order to evaluate the quality of the solutions obtained by our ACO-approach, we compare our framework with a brute force algorithm (computing all the combinations of features and choosing one with the lowest cost) and a random algorithm (randomly choosing features but such that they conform a valid product). We could not compare our algorithm to other alternatives as there were no previous proposals addressing our specific scenario. Our framework takes significantly less time to compute a solution than the brute force algorithm (around a 99% saving), while obtaining total testing costs that are not much higher (around a 25% increase). It also gets solutions with lower testing cost than the ones obtained by the random algorithm (around a 15% cheaper). In order to properly compare our ACO and the random approach, we allow the random approach to run an equal amount of time as the ACO one. In conclusion, our approach represents a preferable choice to these two alternatives.

The rest of the paper is organised as follows. In Section 2 we review related work. In Section 3 we present background concepts that we use in our paper. In Section 4 we introduce our feature selection framework. In Section 5 we present our experiments and discuss the results. In Section 6 we briefly review some threats to the validity of our results. In Section 7 we discuss some considerations concerning the different choices that we took when defining our algorithm. Finally, in Section 8, we give conclusions and outline some directions for future work.

2 Related work

In this section we review previous work related to the research presented in this paper.

We have chosen FODA [22] to represent SPLs but there are other alternative approaches such as RSEB [12] and PLUSS [10]. We think that FODA represents several advantages: it is widely used and, more important, it is based on graphic models.

We are aware that we cannot compute the best, according to a given criteria, combination of features due to the combinatorial nature of the problem. In fact, we performed a small experiment to show that this is the case also in our framework. Therefore, we have to rely on an heuristic approach. Our previous work on applying heuristic approaches to testing [17, 19, 20] showed that Swarm Intelligence [36] was very suitable. Among the different approaches to implement a swarm, in this paper we have decided

to consider the Ant Colony Optimisation algorithm (ACO) [9] because it allowed us to build on top of previous work, facilitating the implementation of the approach. ACO is inspired by the behaviour of real ant colonies in nature and has been successfully used in computationally hard classical optimisation problems such as the travelling salesman problem but, to the best of our knowledge, the research presented in this paper is a novel application of ACO. Although we have used ACO, other alternative approaches in the broad field of *evolutionary algorithms* could have been selected. Evolutionary algorithms are a family of meta-heuristics that base its intelligent behaviour in the evolution of its population. Some approaches in the broad field of Artificial Intelligence consider the combination of many individuals, usually with limited intelligence, that work as a collective to either reach a goal or find a *good enough* solution to a certain problem. In particular, there are several applications of these algorithms in testing [3, 5, 30].

We have used an evolutionary computation approach to find cheap to test products but a framework supporting constraint propagation could be used. In this case, we could rely on tools like FaMa [2] and FeatureIDE [35]. However, we prefer to use the combination of a process algebra and an evolutionary computation technique because they allow us to work with a precise semantic description of each product, facilitating the task of deciding the equivalence, up to a certain criterion, of different products.

There exist evolutionary approaches for test case selection and prioritisation in SPLs [13, 25]. Despite working on testing, these solutions cannot be easily adapted to cope with our problem because we do not select/generate test cases: we select a set of features such that testing the resulting product is as cheap as possible.

Finally, more related to our work, there are evolutionary computation approaches to select features. A study [31] showed that the *Indicator-Based Evolutionary Algorithm* (IBEA) was better than other evolutionary approaches dealing with high complexity in the decision objective spaces. We cannot use this algorithm to solve our problem because IBEA strongly depends on user preferences (we do not have them). In addition, it seems like this algorithm performs better in a multi-objective optimisation problem: we think that a simpler approach, like ours, might work better in our single-objective optimisation problem but further experiments are needed to support this claim. Finally, another important difference is that they define the set of rules from the SPL as an objective of the optimisation problem because their solution can create non-valid products. In our case, we use a process algebra as the search space to ensure the correctness of the generated products. Another related study [14] proposed the SIP method, which improved previous proposals beating even the IBEA algorithm. The approach mainly focused on enhancing the search through a novel representation that hard-codified some constraints and through optimising first the constraints related to the generation of valid products. They also used their approach over real-world SPLs. However, this approach has the same concerns than the previous one: its problem is based on user preferences, it is focused on multi-objective optimisation, and, furthermore, it can produce non-valid products. All these differences make hard to adapt this kind of algorithms to our problem, as they rely on some assumptions that we do not consider and they can generate non-valid products that our approach cannot generate.

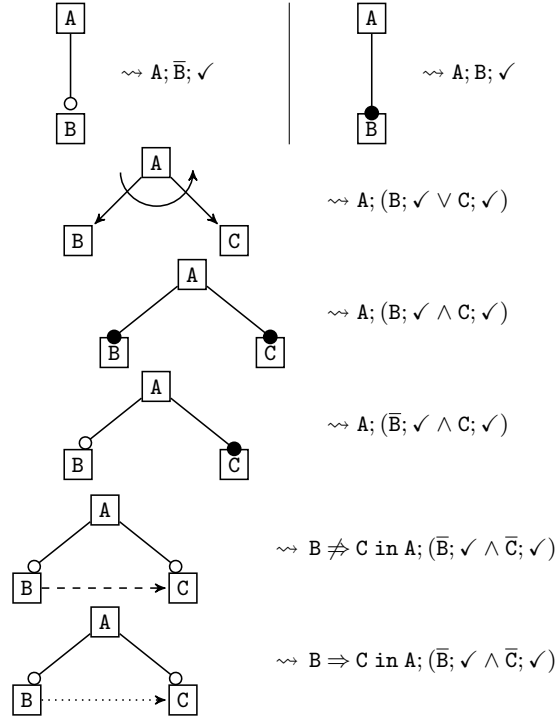


Fig. 1: Translation of FODA Diagrams into SPLA.

3 Preliminaries

In this section we present notation and introduce concepts related to the main two lines that we use in this paper: specification of Software Product Lines with costs and the Ant Colony Optimisation algorithm.

3.1 SPLA-CRIS: SPLs with costs

In this section we briefly review the formal language SPLA-CRIS. The interested reader is referred to the original work [4] for more details.

Definition 1. We will assume that we have a finite set of features \mathcal{F} and we will use A, B, C, \dots to denote single features. A Software Product Line is a term generated by the following Extended BNF-like expression:

$$P ::= \checkmark \mid \text{nil} \mid A; P \mid \bar{A}; P \mid P \vee Q \mid P \wedge Q \\ A \not\Rightarrow B \text{ in } P \mid A \Rightarrow B \text{ in } P$$

where $A, B \in \mathcal{F}$. We denote the set of terms of this algebra by SPLA.

Next we describe the operators of the algebra. The term `nil` represents an SPL with no products, while \checkmark is an SPL that has only the empty product; they are the terminal elements of the syntax. Then we have the mandatory prefix operator $A; P$ (feature A is mandatory) and the optional prefix operator $\bar{A}; P$ (A is optional). The binary operator $P \vee Q$ represents the choose-one. The binary operator $P \wedge Q$ represents the conjunction operator. These operators are associative and commutative, so they can be extended as n -ary operators. The operator $A \Rightarrow B$ in P represents the require constraint. The operator $A \not\Rightarrow B$ in P represents the exclusion constraint. Figure 1 shows the relation between these operators and FODA diagrams.

We can define an operational semantics. Given $A \in \mathcal{F} \cup \{\checkmark\}$, we will write $P \xrightarrow{A} Q$ if we can evolve from P to Q using the defined operational rules. It is important to remark that \checkmark is not a feature and, as such, it is not included in the product. This semantics is given as a set of SOS rules and the interested reader can find them, as well as detailed explanations, in our previous work [1, 4].

Single transitions can be sequentially executed to produce traces. We use ϵ to denote an empty trace and consider the usual concatenation operator $s_1 \cdot s_2$. Abusing the notation, we will write $A \in s$ if A appears in s . Traces ending with \checkmark , that we call successful, are the only ones associated with valid products. It is irrelevant the order in which the features of a trace are obtained. Given a successful trace s , $[s]$ denotes the set obtained from the elements of s .

Finally, given $P \in \text{SPLA}$, we define the products of P , denoted by $\text{prod}(P)$, as $\text{prod}(P) = \{[s] \mid s \in \text{tr}(P)\}$. \square

In order to define a cost model, we will have a cost function such that given a sequence of features (representing the part of the product that we have defined so far) and a single feature (representing the new feature that we would like to add), returns the cost of testing this new feature in the given product. This cost can represent either time and/or resources needed to perform the (integration) testing of this new feature, given the previous ones. In our framework, we assume that costs can be represented by natural numbers. Sometimes, we will not be able to compute the testing cost of integrating a new feature with the ones already chosen. For instance, if the new one is incompatible with the existing features or there are missing dependencies. Therefore, we extend the set of costs with a new symbol \perp to represent *indefiniteness*.

Definition 2. The set of costs is given by $N_\perp = N \cup \{\perp\}$. We extend arithmetic operations in the expected way: for any $x \in N_\perp$ we have $x + \perp = \perp + x = \perp$ and $x \leq \perp$.

A cost function is a function $c : \mathcal{F}^* \times \mathcal{F} \mapsto N_\perp$. \square

In order to compute the cost associated with a product we need to extend the operational semantics (see our previous work [4] for a complete definition). Intuitively, let $P \in \text{SPLA}$ be a process, c be a cost function and s be a successful trace of P . We denote by $\text{tc}(P, s)$ the cost associated with the set of features included in s according to c .

Finally, let us remind that the position of the features in the trace is not relevant to define a product although it may have an impact in its costs. Therefore, different traces

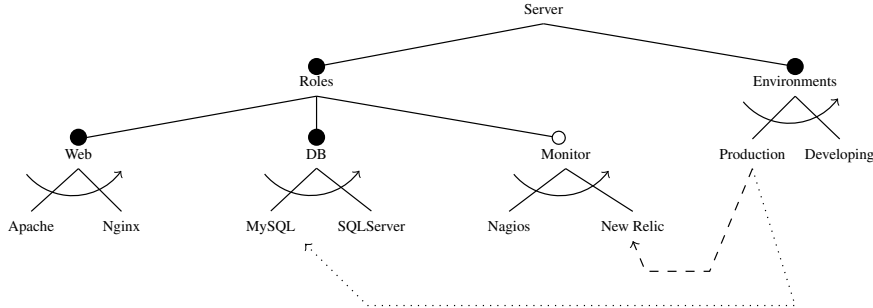


Fig. 2: FODA App Server feature diagram.

$ \begin{aligned} &Server = \\ &P \Rightarrow MS \text{ in } (\\ &P \not\Rightarrow NE \text{ in } (\\ &S; (\\ &R; (\\ &W; (AP; \checkmark \vee NG; \checkmark) \\ &\wedge \\ &D; (MS; \checkmark \vee SS; \checkmark) \\ &\wedge \\ &\bar{M}; (NA; \checkmark \vee NR; \checkmark) \\ &) \\ &\wedge \\ &E; (P; \checkmark \vee Dv; \checkmark) \\ &) \\ &) \\ &) \\ &) \end{aligned} $	<table border="1"> <thead> <tr> <th>P</th> <th>$itc(P)$</th> </tr> </thead> <tbody> <tr><td>{S, R, E, W, D, P, AP, MS}</td><td>2.10</td></tr> <tr><td>{S, R, E, W, D, P, NG, MS}</td><td>2.40</td></tr> <tr><td>{S, R, E, W, D, Dv, AP, MS}</td><td>1.10</td></tr> <tr><td>{S, R, E, W, D, Dv, NG, MS}</td><td>1.30</td></tr> <tr><td>{S, R, E, W, D, Dv, AP, SS}</td><td>1.20</td></tr> <tr><td>{S, R, E, W, D, Dv, NG, SS}</td><td>1.00</td></tr> <tr><td>{S, R, E, W, D, P, M, NA, AP, MS}</td><td>2.50</td></tr> <tr><td>{S, R, E, W, D, Dv, M, NA, AP, MS}</td><td>2.30</td></tr> <tr><td>{S, R, E, W, D, Dv, M, NE, AP, MS}</td><td>2.20</td></tr> <tr><td>{S, R, E, W, D, Dv, M, NA, AP, SS}</td><td>2.20</td></tr> <tr><td>{S, R, E, W, D, Dv, M, NE, AP, SS}</td><td>2.10</td></tr> <tr><td>{S, R, E, W, D, P, M, NA, NG, MS}</td><td>2.80</td></tr> <tr><td>{S, R, E, W, D, Dv, M, NA, NG, MS}</td><td>2, 40</td></tr> <tr><td>{S, R, E, W, D, Dv, M, NE, NG, MS}</td><td>2.30</td></tr> <tr><td>{S, R, E, W, D, Dv, M, NA, NG, SS}</td><td>2.20</td></tr> <tr><td>{S, R, E, W, D, Dv, M, NE, NG, SS}</td><td>2.10</td></tr> </tbody> </table>	P	$itc(P)$	{S, R, E, W, D, P, AP, MS}	2.10	{S, R, E, W, D, P, NG, MS}	2.40	{S, R, E, W, D, Dv, AP, MS}	1.10	{S, R, E, W, D, Dv, NG, MS}	1.30	{S, R, E, W, D, Dv, AP, SS}	1.20	{S, R, E, W, D, Dv, NG, SS}	1.00	{S, R, E, W, D, P, M, NA, AP, MS}	2.50	{S, R, E, W, D, Dv, M, NA, AP, MS}	2.30	{S, R, E, W, D, Dv, M, NE, AP, MS}	2.20	{S, R, E, W, D, Dv, M, NA, AP, SS}	2.20	{S, R, E, W, D, Dv, M, NE, AP, SS}	2.10	{S, R, E, W, D, P, M, NA, NG, MS}	2.80	{S, R, E, W, D, Dv, M, NA, NG, MS}	2, 40	{S, R, E, W, D, Dv, M, NE, NG, MS}	2.30	{S, R, E, W, D, Dv, M, NA, NG, SS}	2.20	{S, R, E, W, D, Dv, M, NE, NG, SS}	2.10	<p>Legend:</p> <ul style="list-style-type: none"> S : Server R : Roles E : Environments P : Production Dv : Developing W : Web server D : Database Server M : Monitoring service AP : Apache NG : Nginx MS : MySQL SS : SQLServer NA : Nagios NE : New Relic
P	$itc(P)$																																			
{S, R, E, W, D, P, AP, MS}	2.10																																			
{S, R, E, W, D, P, NG, MS}	2.40																																			
{S, R, E, W, D, Dv, AP, MS}	1.10																																			
{S, R, E, W, D, Dv, NG, MS}	1.30																																			
{S, R, E, W, D, Dv, AP, SS}	1.20																																			
{S, R, E, W, D, Dv, NG, SS}	1.00																																			
{S, R, E, W, D, P, M, NA, AP, MS}	2.50																																			
{S, R, E, W, D, Dv, M, NA, AP, MS}	2.30																																			
{S, R, E, W, D, Dv, M, NE, AP, MS}	2.20																																			
{S, R, E, W, D, Dv, M, NA, AP, SS}	2.20																																			
{S, R, E, W, D, Dv, M, NE, AP, SS}	2.10																																			
{S, R, E, W, D, P, M, NA, NG, MS}	2.80																																			
{S, R, E, W, D, Dv, M, NA, NG, MS}	2, 40																																			
{S, R, E, W, D, Dv, M, NE, NG, MS}	2.30																																			
{S, R, E, W, D, Dv, M, NA, NG, SS}	2.20																																			
{S, R, E, W, D, Dv, M, NE, NG, SS}	2.10																																			

Fig. 3: SPLA term.

can produce the same product but with different costs. As a consequence, we need to consider a set of costs for each product, because a product will be *equivalent* to a set of sequences.

Definition 3. Let c be a cost function. We consider the function $c_{SPLA} : SPLA \times \mathcal{P}(\mathcal{F}^*) \mapsto \mathcal{P}(\mathbf{N}_{\perp})$ defined as follows:

$$c_{SPLA}(P, p) = \{\tau c(P, s) \in \mathbf{N}_{\perp} \mid \exists s \text{ trace of } P : [s] = p\}$$

□

Example Let us illustrate the previous definitions with an example. Let us consider a *Server* consisting of a Web Server and a Database. There are two possible environments for the running server: the *production* environment and the *developing* environment. There are two possibilities for the database: MySQL or SQLServer. For the Web server we can use either Apache Web Server or Nginx. There are also two restrictions in the

case of the Production environment: First, the use of the New Relic monitor system is forbidden. Second, the use of MySQL is mandatory. Figure 2 show the FODA diagram corresponding to this description. This FODA diagram is translated to the SPLA term in Figure 3 (left) to handle the system formally. The Integration Test costs appears in the centre of Figure 3. Formally, the cost function is defined as follows: for $s \in \mathcal{F}^*$ and $A \in \mathcal{F}$, $c(s, A) = \text{itc}([sA])$ if the product $[sA]$ is listed in table and $c(s, A) = 0$ otherwise.

3.2 Ant Colony Optimisation

The Ant Colony Optimisation algorithm (ACO) [9] is a well-known algorithm in the evolutionary algorithms field. It is a distributed algorithm to explore a graph-like search space associated with a combinatorial optimisation problem. It consists of a set of *ants*, which are the agents that explore the search space. Each ant looks for the shortest path from the initial node to the target node, choosing their next move based on a random choice modified by the weigh of each path and the *pheromones* released by other ants that previously performed that move.

Definition 4. A model P of a combinatorial optimisation problem is a tuple (\mathbf{S}, Ω, f) , where \mathbf{S} is a search space defined over a finite set X_1, \dots, X_n of discrete decision variables, Ω is a set of constraints over the variables, and $f : \mathbf{S} \rightarrow R_0^+$ is the objective function to be minimised.

Each generic variable X_i takes values in $D_i = \{v_i^1, \dots, v_i^{|D_i|}\}$. A feasible solution $s \in \mathbf{S}$ is a complete assignment of values to variables such that all the constraints in Ω are satisfied. A feasible solution $s^* \in \mathbf{S}$ is called a global optimum if and only if for all $s \in \mathbf{S}$ we have $f(s^*) \leq f(s)$. \square

Once we have a model of the problem that we would like to solve, we can generate a *construction graph*. Artificial ants move from vertex to vertex along the edges of this graph, incrementally building a partial solution. During this traversal of the graph, the ants deposit a certain amount of pheromone on the edges that they traverse. The amount of pheromone deposited by each artificial ant usually depends on the *quality* of the solution reached after that specific traversal. The idea underlying ACO and the simulation of pheromone is that other ants will use the information concerning the concentration of pheromone as a hint to further explore promising regions of the search space.

The *ACO general scheme* proceeds as follows. After a preliminary step, where the main parameters and the pheromone trails are initialised, we have a main loop that iterates until we reach the termination criterion. This criterion may be based on the numbers of iterations of the loop or on the quality of the obtained solution. In each iteration of the loop, each ant generates a solution. Then, the global state updates the pheromones left by the ants in their solution path. This task consists of two main consecutive steps.

First step of the loop: Construct ant solutions. In each iteration, m ants generate solutions from a finite set of available solution components C . The construction starts from an empty solution set $s^P = \emptyset$ and, in each step, the ant extends its partial solution by adding a feasible solution element from the set of elements of C that can be added

to the partial solution s^P without violating any constraint in Ω . The choice of a solution component from this set is guided by a stochastic mechanism, which is biased by the pheromone associated with each of the elements in it. The rule for the stochastic choice of solution components varies across different ACO algorithms but they are always inspired by the behaviour of real ants. This process can be seen as a traversal of the *construction graph*.

Second step of the loop: Update pheromones. The pheromone update aims to increase the pheromone values associated with good or promising solutions and, in turn, decrease those associated with bad ones. Usually, this is achieved by decreasing all the pheromone values through *pheromone evaporation* and by increasing the pheromone levels associated with a chosen set of good solutions.

4 ACO for feature selection taking into account testing costs

Our feature selection framework finds, for a given SPL and a selected feature, a combination of features that contains said feature and such that the cost (in time and/or resources) of testing the generated product is as low as possible. We will consider that the SPL is formally defined as an SPLA-CRIS term. We use an ACO algorithm because it is the most suitable one for this problem. A comprehensive discussion about this choice can be found in Section 7. Next, we briefly describe the main components of our framework:

- An SPL represented as an SPLA-CRIS expression.
- An SPLA-CRIS interpreter that allows us to explore the search space generated by the SPLA-CRIS expression without fully computing it.
- An ACO to lead the search for a feature combination with low cost.

We combine these three components as follows. We consider an SPLA-CRIS expression and derive the structure needed to execute our ACO over it with the goal of finding a *cheap* to test product. However, we cannot compute the testing cost of all the possible combinations of features of the SPLA-CRIS expression. We will rely on an interpreter to compute the added testing cost after adding a new feature to the current selection, but without constructing the full SPLA-CRIS expression tree.

As usual, our ACO needs to have a representation of our setting as a combinatorial optimisation problem. We will define this problem as follows:

- Search space \mathcal{S} . This is the full SPLA-CRIS tree. In addition, the associated decision variables are associated to the feature that we have to choose next.
- Set of constraints Ω . We have three constraints.
 - A constraint stating that the last symbol of a valid path must be \checkmark . Remind that this is the special symbol that we use to denote successful termination, that is, the last symbol of a successful trace.
 - A constraint stating that a valid feature combination should contain the previously selected feature.
 - A constraint stating that a valid path can be generated by the definition of the SPLA-CRIS expression that we are considering.

- Objective function f . This function assigns its cost to each set of features that can be produced from the SPLA-CRIS expression. The goal of our ACO is to minimise the value of this function.

Once we have our problem redefined as a combinatorial optimisation problem, our ACO follows the general scheme presented in Section 3.2. The only adaption with respect to this general scheme is that our ants generate *on the fly* the search space while exploring it, instead of having all the information stored beforehand. Thus, our ACO has to work together with our SPLA-CRIS interpreter in order to obtain the associated costs.

It is important to note that our algorithm does not use any additional heuristic optimisation. In the literature there are some common heuristics, like removing mandatory features (i.e. computing atomic sets), that are usually used to simplify the problem at hand. In our case, as the goal is to have a lower testing cost, we cannot consider such heuristic optimisation as they would modify the obtained testing costs. For example, in the case of removing the mandatory features, that heuristic would produce testing costs that do not consider the additional testing costs that each mandatory feature would add with each added feature, costs that are not constant neither uniform between different features.

5 Experimental Results

In order to evaluate the usefulness of our ACO to find *cheap* (in terms of testing) combinations of features, according to a certain set of constraints defined by the corresponding SPL, we decided to initially compare it with a *brute force* algorithm. The brute force algorithm will effectively compute a feature combination with the lowest testing cost at the expense of a long execution time. In contrast, we will show that our framework can give feature combinations with slightly higher testing costs but having (much) shorter execution times.

We set our ACO algorithm with the following parameters:

- Number of ants: 10.
- Number of maximum iterations: 100.
- Pheromone constant: 1000.
- Pheromone evaporation coefficient: 0.4.
- α coefficient: 0.5.
- β coefficient: 1.2.

These parameters are typical parameters in the literature and they worked very well in our previous work [18]. Moreover, we did small experiments to tune the parameters and none of them show better performance than these ones.

For our experiments, we used 75 SPLA-CRIS expressions with between 10 and 85 features. These SPLA-CRIS expressions were generated using previous work with SPLA-CRIS [4], automatically generating them using the BeTTY tool [32] and storing them in an fodaA format in .xml files. The costs in these expressions are also automatically generated, and thus we consider that they represent the additional testing costs that a feature will add to the product if included in it.

Trial Number	Brute Force Cost	ACO Cost	Cost Increase	Brute Force Time	ACO Time	Time Saving
1	27	36	33.33%	1.1713	4.5798	-291.01%
2	18	27	53.33%	6.5209	8.9389	-37.08%
3	36	45	25.00%	20.5985	9.8473	52.19%
4	63	72	14.29%	4,434.4519	14.8687	99.66%
Average	36	45	25.42%	1,115.6856	9.5587	99.14%

Table 1: Comparing our approach and brute force (time is measured in seconds).

In our first experiment we evaluated these expressions through our SPLA-CRIS interpreter. Using this interpreter, we executed a brute force algorithm to compute all the possible feature combinations as well as their costs. We also executed our ACO algorithm using the SPLA-CRIS interpreter to obtain a feature combination with low cost. Due to the randomisation involved in the ACO algorithm, we executed both algorithms 15 times for each SPLA-CRIS expression and measured the mean of the results of all the computations. Unfortunately, after running during 20 hours the brute force algorithm was able to compute the solution only for four expressions (note that the longest time used by our ACO was less than 15 seconds). In Table 1 we compare the cost and computation time for these expressions.

As expected, the brute force algorithm was unable to compute, in a reasonable time, the best feature selection for most of the experiments (in fact, it was only able to compute it for the smaller expressions, the ones with less than 13 features) due to the combinatorial explosion underlying feature selection, aggravated with minimising the cost. This leaves us with only four values to compare our ACO with the brute force algorithm. In this comparison we can see that our algorithm obtains, on average, a solution that it is 25.42% more expensive than the best features combination (computed by the brute force algorithm). In contrast, it needs on average 99.14% less time to produce this solution.

Here, it is important to note that for the simplest cases, the brute force algorithm needs less time than our ACO algorithm. The reason is that the expressions are so simple that our ACO algorithm is overpowered for this task. That means that, as the expression is so small, brute force computes all the combinations quickly (because there are so few) while the ACO algorithm not only has to explore the expression, but it also needs to achieve convergence (what will take a while due to the required iterations). However, as we increase the complexity of the expressions, the brute force algorithm quickly raises its execution time a lot (due to its exponential nature), while our ACO algorithm keeps its execution time in a reasonable value.

The comparison with the brute force algorithm leaves us with so few results that we decided to perform a second experiment and compare our framework with a random algorithm. This random algorithm will give us the feature combination with lowest costs of a set of randomly generated feature combinations that represent valid products. The number of feature combinations on this set of randomly generated feature combinations

Trial Number	Random Cost	ACO Cost	Cost Saving (%)
1	52.2	52.2	0.00
2	36.0	34.8	3.33
3	50.4	50.4	0.00
4	73.8	73.8	0.00
5	63.6	63.6	0.00
6	70.2	69.0	1.71
7	63.6	61.2	3.77
8	73.2	70.8	3.28
9	67.8	67.2	0.88
10	75.0	70.8	5.60
11	63.6	63.6	0.00
12	62.4	57.0	8.65
13	91.8	87.0	5.23
14	81.6	77.4	5.15
15	70.8	66.0	6.78
16	58.2	54.0	7.22
17	83.4	79.2	5.04
18	99.6	79.8	19.88
19	78.0	76.8	1.54
20	99.0	81.0	18.18
21	80.4	79.2	1.49
22	111.0	96.6	12.97
23	70.8	67.8	4.24
24	96.0	80.4	16.25
25	76.8	69.0	10.16

Trial Number	Random Cost	ACO Cost	Cost Saving (%)
26	131.4	102.6	21.92
27	120.6	98.4	18.41
28	147.6	115.8	21.54
29	109.2	94.8	13.19
30	115.8	102.0	11.92
31	161.4	128.4	20.45
32	114.6	82.8	27.75
33	130.2	117.0	10.14
34	157.2	148.8	5.34
35	78.0	70.2	10.00
36	93.0	78.0	16.13
37	100.8	97.2	3.57
38	149.4	133.2	10.84
39	122.4	101.4	17.16
40	166.8	142.2	14.75
41	147.0	138.0	6.12
42	159.0	146.4	7.92
43	115.2	89.4	22.40
44	148.8	135.6	8.87
45	168.0	132.0	21.43
46	156.0	134.4	13.85
47	118.2	98.4	16.75
48	189.6	144.6	23.73
49	175.8	168.0	4.44
50	201.6	175.8	12.8

Trial Number	Random Cost	ACO Cost	Cost Saving (%)
51	221.4	159.6	27.91
52	136.8	120.6	11.84
53	176.4	142.8	19.05
54	151.2	123.0	18.65
55	142.2	96.0	32.49
56	208.8	176.4	15.52
57	166.8	139.8	16.19
58	145.2	105.0	27.69
59	166.8	135.0	19.06
60	175.2	135.0	22.95
61	178.2	139.2	21.89
62	185.4	167.4	9.71
63	199.2	171.0	14.16
64	254.4	171.6	32.55
65	168.6	121.2	28.11
66	173.4	146.4	15.57
67	238.8	187.8	21.36
68	210.0	191.4	8.86
69	226.2	129.0	42.97
70	201.0	133.2	33.73
71	209.4	163.8	21.78
72	309.6	196.8	36.43
73	340.8	207.0	39.26
74	285.6	150.6	47.27
75	197.4	143.4	27.36

Table 2: Results of the experiment comparing with respect to random.

will depend on how much time the algorithm is running. In our experiment, we first run the ACO algorithm and then we run the random algorithm until it overcomes the execution time the ACO algorithm needed. This way, the random algorithm always has the same (or more) time to execute as our ACO and we compare the algorithms performance, that is, the feature combination costs obtained.

We started with the same set of 75 SPLA-CRIS expressions and evaluated them using our SPLA-CRIS interpreter. For each SPLA-CRIS expression, we also used this interpreter to execute 15 times both our ACO algorithm and the random algorithm. We computed mean costs and compared them (see Table 2).

In order to present an easy visualisation of all the results, we sorted the obtained costs for the ACO approach, from lowest to highest, and produced the graphic shown in Figure 4. We also obtained the sorted percentage cost saving of the ACO algorithm with respect to the random algorithm (see Figure 5). In order to compute the cost saving of our approach with respect to the random algorithm, we proceeded as follows. For each SPLA-CRIS expression, we computed the cost using both our ACO and the random algorithm and computed the percentage difference of the ACO with respect to the random algorithm. For example, if the cost associated with the selected product by the ACO is equal to 135.0 and the cost obtained by the random algorithm, most likely for a different product but also fulfilling the constraints associated to the SPL, is 175.2, then the cost saving is equal to $100 \cdot (1 - \frac{135.0}{175.2}) \approx 22.95$.

The analysis of the results shows that our ACO algorithm always finds feature combinations with lower costs than the random algorithm (or equal cost in the worst cases). Therefore, our algorithm performs better than the random algorithm. On average, our ACO computes solutions that are 14.87% cheaper.

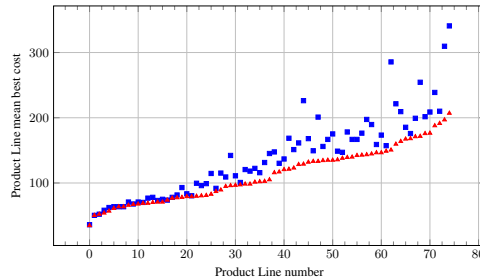


Fig. 4: Sorted obtained costs (blue = random, red = ACO).

We performed a statistical hypothesis test over the results, whose null hypothesis was that the random algorithm and our framework give similar results, that is, both obtain similar costs. We applied a one-way ANOVA test where we tested whether the results of both algorithms are similar in average. Then, we computed the p-value for the experiment, obtaining a p-value of 0.0037. This represents that there is a 00.37% of probability that the null hypothesis is fulfilled. Therefore, we can reject the null hypothesis for the experiment with a confidence higher than 99%, as its p-value is lower than 0.01. In order to double-check our results, we also performed a t-test and obtained the same p-value. Thus, the conclusion is that the performance of our ACO algorithm is better than the random algorithm.

6 Threats to Validity

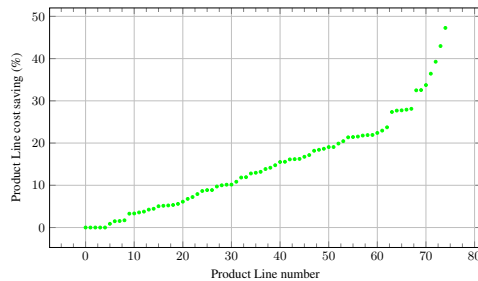
Threats to *internal validity* refer to uncontrolled factors that can affect the output of the experiments, either in favour or against our hypothesis. The main threat in this category is the possibility of having faults in the code of the experiments. We diminished this threat by carefully testing the code, even using small examples for which we knew the expected results. Additionally, in order to reduce the impact of the randomness associated with our methodology, we repeated the experiments several times.

Threats to *external validity* refer to the generality of our findings to other situations. The main threat in this category is given by the different possible SPLs to which we could apply our framework. As the population of SPLs is unknown, this threat is not fully addressable. In order to diminish this risk, we considered different SPLs in the experiments.

Finally, threats to *construct validity* refer 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 what would happen if we use our framework with real-world SPLs and/or with much more complex SPLs, which is a matter of future work.

7 Discussion about the suitability of ACO

We have shown that our ACO achieves good solutions for this task. However, it is possible that other heuristics could work better than ACO in this specific framework. Al-

Fig. 5: Sorted cost *saving*.

though this comparison should be further investigated, and it will be indeed a matter of future work, we would like to briefly justify why we decided to use an ACO algorithm.

Our main concern when developing the algorithm was that we needed to provide a much faster solution than brute force while, at the same time, being able to obtain good enough results. Therefore, we classified our problem as an *exploratory problem*. We are aware that there are many evolutionary algorithms that usually work better than a random based search. In our case, we needed a proposal able to *search* in a SPLA-CRIS expression. Fortunately, this kind of syntactical expressions can be transformed into a graph whose *final states* represent all the possible feature combinations that fulfil the expression restrictions. Since this graph can have cycles, we need to perform an extra step to unfold these cycles in order to be able to use a Genetic Programming based algorithm to search for feature combinations inside this graph. This operation would increase the complexity of the approach. In addition, since we are working with a search space based on a graph structure, an approach such as particle swarm optimisation algorithms would suffer because it needs extra adaptation phases that also will increase the complexity of the algorithm. In contrast, ACO can be easily applied to this scenario because our search space is represented as a graph where we are looking for a path from the root to a *final state*, representing a valid feature combination, with a cost as low as possible. So, in order to put into practice our approach we only needed an available interpreter [4] that transform the SPLA-CRIS expressions into appropriate graphs.

8 Conclusions and future work

Software Product Lines are a useful tool for developing software systems in an automatic way and testing them is a must. Integration testing is a process that SPLs should overcome: we test how well a new feature is integrated with the already existing features of the SPL. If we have the costs of testing each feature of the SPL, then we can select the product that contains the new feature that has a lower testing cost, so we can test its integration with the other features of the SPL in a quicker and/or cheaper way.

In this paper we have proposed a new framework for feature selection in SPLs having testing costs associated with the combination of features. This feature selection generates a product with low cost and a given feature. We have adapted ACO to deal with an *a priori* unknown search space. Therefore, our framework is able to obtain

new feature combinations for a given SPL without computing all the possible feature combinations, which is a time-consuming task. Besides, in order to assess the usefulness of the new framework, we have reported on our most representative experiments. These experiments show that our algorithm is well suited for this task and that it is preferable than other simpler algorithms. Finding sub-optimal solutions in a shorter time can be fundamental in some scenarios, as computing the optimal solution can require a huge amount of resources and time. In fact, in our own experiments we were able to compute the exact solution, by computing all the possible solutions, only for SPLs with a very small number of features. In addition, our experiments show that our algorithm is better than a random search, when giving the same time to both algorithms.

We have identified several research directions concerning applicability, scalability, suitability and adaptability of our framework. Concerning scalability, we will consider more complex SPLs and check whether our technique scales well. Although we will not be able to compare our ACO with brute force, because the latter will not compute the best solution, we want to explore the *limit* of our approach. In addition, we would like to use current mutation testing approaches [11, 28] to efficiently generate and process big amount of mutants representing either non-optimal or faulty selections of features.

With respect to suitability, we will consider two unrelated lines of work. First, although our ACO is well suited for this task, we would like to compare it with other heuristics that could work better than our proposal in this specific framework. Specifically, we would like to compare our ACO approach with other meta-heuristics based on Bee Swarm [23]. A second line of work to analyse the suitability of our framework is to consider SPLs with existing feature selections, produced by an expert, and compare their costs and the ones produced by our framework. In addition, as suggested by a reviewer, it would be interesting to take into account that products including features interacting with the new feature will be more likely to expose bugs, than products running the feature in isolation. Finally, concerning adaptability, we would like to assess the usefulness of our methodology in other frameworks. First, we would like to apply our framework to study formal models of cloud [6, 27] and distributed [15, 16] systems. We choose this type of systems because we are familiar with them and, more importantly, because they are highly configurable and, therefore, will induce SPLs with many features. Finally, we would like to evaluate whether it is possible to integrate our feature selection framework in existing tools like ProFeat [8].

Acknowledgements

We would like to thank the anonymous reviewers for the careful reading, the many constructive comments and the useful suggestions, which have helped us to further strengthen the paper.

References

1. C. Andrés, C. Camacho, and L. Llana. A formal framework for software product lines. *Information & Software Technology*, 55(11):1925–1947, 2013.

2. D. Benavides, P. Trinidad, A. Ruiz Cortés, and S. Segura. FaMa. In R. Capilla, J. Bosch, and K. C. Kang, editors, *Systems and Software Variability Management - Concepts, Tools and Experiences*, pages 163–171. Springer, 2013.
3. M. Benito-Parejo and M. G. Merayo. An evolutionary algorithm for selection of test cases. In *22nd IEEE Congress on Evolutionary Computation, CEC'20*, pages E–24535: 1–8. IEEE Computer Society, 2020.
4. C. Camacho, L. Llana, and A. Núñez. Cost-related interface for software product lines. *Journal of Logic and Algebraic Methods in Programming*, 85(1):227–244, 2016.
5. J. Campos, Y. Ge, N. Albulian, G. Fraser, M. Eler, and A. Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, 104:207–235, 2018.
6. P. C. Cañizares, A. Núñez, J. de Lara, and L. Llana. MT-EA4Cloud: A methodology for testing and optimising energy-aware cloud systems. *Journal of Systems and Software*, 163:110522:1–25, 2020.
7. I. do Carmo Machado, P. A. da Mota Silveira Neto, and E. Santana de Almeida. Towards an integration testing approach for software product lines. In *IEEE 13th Int. Conf. on Information Reuse & Integration, IRI'12*, pages 616–623. IEEE, 2012.
8. P. Chrszon, C. Dubsloff, S. Klüppelholz, and C. Baier. ProFeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects of Computing*, 30(1):45–75, 2018.
9. M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, 2004.
10. M. Eriksson, J. Borstler, and K. Borg. The PLUSS approach - domain modeling with features, use cases and use case realizations. In *9th Int. Conference on Software Product Lines, SPLC'06, LNCS 3714*, pages 33–44. Springer, 2006.
11. P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. Wodel-Test: a model-based framework for language-independent mutation testing. *Software and Systems Modeling*, 20(3):767–793, 2021.
12. M. Griss, J. Favaro, and M. D'Alessandro. Integrating feature modeling with the RSEB. In *5th Int. Conf. on Software Reuse, ICSR'98*, pages 76–85. IEEE Computer Society, 1998.
13. C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize T-Wise test configurations for software product lines. *IEEE Transactions on Software Engineering*, 40(7):650–670, 2014.
14. R. M. Hierons, M. Li, X. Liu, S. Segura, and W. Zheng. SIP: optimal product selection from feature models using many-objective evolutionary optimization. *ACM Transactions on Software Engineering and Methodology*, 25(2):17:1–17:39, 2016.
15. R. M. Hierons, M. G. Merayo, and M. Núñez. Bounded reordering in the distributed test architecture. *IEEE Transactions on Reliability*, 67(2):522–537, 2018.
16. R. M. Hierons and M. Núñez. Implementation relations and probabilistic schedulers in the distributed test architecture. *Journal of Systems and Software*, 132:319–335, 2017.
17. A. Ibias, D. Griñán, and M. Núñez. GPTSG: a Genetic Programming Test Suite Generator using Information Theory measures. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 716–728. Springer, 2019.
18. A. Ibias and L. Llana. Feature selection using evolutionary computation techniques for software product line testing. In *22nd IEEE Congress on Evolutionary Computation, CEC'20*, pages E–24502: 1–8. IEEE Computer Society, 2020.
19. A. Ibias and M. Núñez. Using a swarm to detect hard-to-kill mutants. In *2020 IEEE Int. Conf. on Systems, Man and Cybernetics, SMC'20*, pages 2190–2195. IEEE Computer Society, 2020.

20. A. Ibias, P. Vazquez-Gomis, and M. Benito-Parejo. Coverage-based grammar-guided genetic programming generation of test suites. In *23rd IEEE Congress on Evolutionary Computation, CEC'21*, pages 2411–2418. IEEE, 2021.
21. M. Jaffar-ur Rehman, F. Jabeen, A. Bertolino, and A. Polini. Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability*, 17(2):95–133, 2007.
22. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, 1990.
23. D. Karaboga and B. Akay. A survey: algorithms simulating bee swarm intelligence. *Artificial Intelligence Review*, 31(1):61–85, 2009.
24. R. Lachmann, S. Beddig, S. Lity, S. Schulze, and I. Schaefer. Risk-based integration testing of software product lines. In *11th Int. Workshop on Variability Modelling of Software-Intensive Systems, VaMoS'17*, pages 52–59. ACM Press, 2017.
25. R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, A. Egyed, and E. Alba. Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of software product lines. In *16th IEEE Congress on Evolutionary Computation, CEC'14*, pages 387–396. IEEE, 2014.
26. J. D. McGregor. Testing a software product line. In P. Borba, A. Cavalcanti, A. Sampaio, and J. Woodcock, editors, *Testing Techniques in Software Engineering: 2nd Pernambuco Summer School on Software Engineering, PSSE'07*, pages 104–140. Springer, 2010.
27. A. Núñez, P. C. Cañizares, M. Núñez, and R. M. Hierons. TEA-Cloud: A formal framework for testing cloud computing systems. *IEEE Transactions on Reliability*, 70(1):261 – 284, 2021.
28. M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275 – 378. Elsevier, 2019.
29. S. Reis, A. Metzger, and K. Pohl. Integration testing in software product line engineering: A model-based technique. In *10th Int. Conf. on Fundamental Approaches to Software Engineering, FASE'07, LNCS 4422*, pages 321–335. Springer, 2007.
30. D. S. Rodrigues, M. E. Delamaro, C. G. Corrêa, and F. L. S. Nunes. Using genetic algorithms in test data generation: A critical systematic mapping. *ACM Computing Surveys*, 51(2):article 41, 2018.
31. A. S. Sayyad, J. Ingram, T. Menzies, and H. H. Ammar. Optimum feature selection in software product lines: Let your model and values guide your search. In *1st Int. Workshop on Combining Modelling and Search-Based Software Engineering, CMSBSE'13*, pages 22–27. IEEE Computer Society, 2013.
32. S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, and A. Ruiz-Cortés. Betty: Benchmarking and testing on the automated analysis of feature models. In *6th Int. Workshop on Variability Modeling of Software-Intensive Systems, VaMoS'12*, pages 63–71, 2012.
33. J. Shi, M. B. Cohen, and M. B. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *15th Int. Conf. on Fundamental Approaches to Software Engineering, FASE'12, LNCS 7212*, pages 270–284. Springer, 2012.
34. M. Steindl and J. Mottok. Optimizing software integration by considering integration test complexity and test effort. In *10th Int. Workshop on Intelligent Solutions in Embedded Systems, WISES'12*, pages 63–68. IEEE Computer Society, 2012.
35. T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.
36. D. Wang, D. Tan, and L. Liu. Particle swarm optimization algorithm: an overview. *Soft Computing*, 22:387–408, 2018.

10.9 Using a swarm to detect hard-to-kill mutants

Authors	Alfredo Ibias and Manuel Núñez
Title	Using a swarm to detect hard-to-kill mutants
Publication Type	Conference
Venue	2020 IEEE International Conference on Systems, Man, and Cybernetics
Year	2020
DOI/URL	https://doi.org/10.1109/SMC42975.2020.9282883
Pages	6
Authors' Contributions	Ibias and Núñez developed the theory. Ibias designed the experiments. Ibias developed and executed the experiments. Ibias and Núñez wrote the manuscript. Ibias and Núñez reviewed the manuscript.

Using a swarm to detect hard-to-kill mutants

Alfredo Ibias
Universidad Complutense de Madrid
Madrid, Spain
aibias@ucm.es

Manuel Núñez
Universidad Complutense de Madrid
Madrid, Spain
manuelnu@ucm.es

Abstract—Mutation Testing is an effective testing technique that relies in the generation of mutants from the system under test. The main limitation of this technique is that the potential number of mutants is usually huge. Therefore, it is important to classify and select mutants in order to avoid repetitive, useless or excessive computations, and biased results. In this paper we focus on avoiding too many executions and/or biased results by classifying mutants into two categories: hard-to-kill and easy-to-kill mutants. We propose a new swarm intelligence algorithm to classify a set of mutants between those two classes and we show how our algorithm compares to other approaches.

Index Terms—Mutation Testing, Swarm Intelligence, Mutant Selection

I. INTRODUCTION

Software Testing [2], [31] is the most widely used technique to detect faults in software systems. Software testing includes different approaches and methodologies that target specific categories of faults. Most approaches try to increase *code coverage*, that is, try to build test suites that *traverse* all the paths of the system that are relevant with respect to a certain criterion. In this paper we focus on *mutation testing*, an approach that does not only focus on showing *where* to test, but also on helping to identify *what* should be checked for. Experimental evidence has showed that tests suites produced by mutation testing approaches were significantly better than the (high quality) manually written ones [17]. Intuitively, mutation testing considers a software system, that we would like to evaluate, and variants of this system, called *mutants*, that represent potential faults of the system. The goal is to find good test suites that *kill* all the mutants: a test case kills a mutant if the application of the test to the original system and to the mutant produces different results. If a mutant is *alive* after the application of all the test cases, then we have to analyse whether our test suite was not good enough or the mutant is equivalent to the original system.

Mutant selection is critical because it ameliorates the scalability problem associated with mutation testing: usually, we have huge amounts of potential mutants. Producing and working with a large number of mutants is impractical, as they need to be analysed, compiled, executed and killed by test cases. This is an important problem and, actually, makes difficult

Research partially supported by the Spanish MINECO/FEDER project FAME (RTI2018-093608-B-C31), the Comunidad de Madrid project FORTE-CM (S2018/TCS-4314) co-funded by EIE Funds of the European Union and the Region of Madrid - Complutense University of Madrid (grant number PR65/19-22452).

the wide applicability and large adoption of mutation testing. Classically, mutant selection tried to solve this problem through the reduction of the size of the mutants sets used in the process, defining mutant reduction strategies such as selective mutation [36], [43] and random mutant selection [1].

Previous work has focused on classifying mutants based on different characteristics. For example, it is usually assumed that *harder* to kill mutants are more useful than the easy to kill ones, but there are many categories. Hard-to-kill mutants are usually considered to be the ones killed by a small fraction of the considered test cases, but there is also work identifying hard-to-kill mutants based on the internal structure of the code of the given system under test [42]. A recent work [37] identifies different criteria to classify mutants (hard to kill, subsuming, hard to propagate and fault revealing) and show that each of them classifies different mutants as the preferable ones for mutation testing. Moreover, the authors found that there is a weak connection between these classifications and fault revelation. However, they conclude that hard-to-kill mutants are the ones more related to fault revelation.

Our goal in this paper is to classify a set of mutants between hard-to-kill and easy-to-kill mutants, with the idea that this kind of mutants gives a compromise between easy to classify and fault detection ability. We propose an approach based in the Swarm Intelligence Algorithms theory [6], [40], a family of algorithms that naturally falls into the Computational Collective Intelligence [10], [32]–[34] research area, to address this problem. In previous work other authors have proposed different approaches for mutant classification, including using machine learning methods [13], [25], [28], but we are not aware of a hard-to-kill mutant classification algorithm based on Swarm Intelligence Algorithms theory. In our work, we use a swarm of agents to apply tests to small sets of mutants with the goal of building a hard-to-kill set of mutants from the results obtained by the agents.

The paper is organised as follows. First, in Section II, we present some basic concepts needed to understand our work. Then, we explain our algorithm (Section III) and we present the experiments that we performed to assess its performance (Section IV). Finally, we present the threats to the validity (Section V) and the conclusions of our work (Section VI).

II. PRELIMINARIES

In this section we present basic information about the two main fields that we consider in this paper: mutation testing and

swarm intelligence algorithms.

A. Mutation Testing

Software Testing is a broad field of techniques with the goal of detecting faults in software systems. Essentially, testing consists in applying a set of inputs to the System Under Test (SUT), observe the generated output, and decide whether this output is consistent with the expected output. In order to perform this decision, it is necessary some kind of oracle that determines which one is the correct output. In some cases, there is no oracle available and the testers should resort to other kind of techniques to determine what constitutes a valid output [3]. One of the techniques to overcome this oracle problem is *mutation testing*. Mutation testing is not just an *academic methodology* because it has been successfully used in real software systems [38].

A *mutant* is a modified version of the SUT that includes a fault. These faults can be randomly seeded or following some guide and can be generated either automatically or manually. Then, when applying a test to a mutant, we will check whether the produced output is different to the output obtained after applying the test to the original SUT. If the outputs are different, then we say that the mutant has been killed. If the mutant is not killed then we have three different possibilities: the fault has not been executed, the fault has been executed but it has not propagated to the output, or the mutant is equivalent to the original SUT. Intuitively, mutation testing uses the SUT as a kind of oracle and generates mutants from it with the goal of having faulty versions of the SUT to assess the quality of the generated tests. Then, the quality of a test is assessed with a metric called *mutation score*, which represents the percentage of mutants that the test has killed.

Just as we can classify the tests using their mutation score, we can classify the mutants depending on how many tests kill those mutants. This idea crystallises in the concept of *hard-to-kill* mutants, that is, those mutants that are killed by a small amount of tests (as opposed to *easy-to-kill* mutants, which are those that are killed by most of the tests). In previous work, there have been different definitions of *hard-to-kill* mutants, all of them with the idea that *hard-to-kill* mutants are the hardest to find with a test. It is possible to provide a certain bound, for example, those killed by 5% or 2.5% of the tests [37]. Another possibility [42] is to mark a mutant as *hard-to-kill* if it presents a specific internal structure. In this paper we will consider as *hard-to-kill* mutants those that are the least killed, using a variable measure instead of a fixed cap.

B. Swarm Intelligence Algorithms

Swarm Intelligence Algorithms [6], [40] are a family of algorithms that base its *intelligent* behaviour in their swarm interactions. In this kind of algorithms, there is a *swarm* of agents, where each agent has little to no intelligence. Usually, these agents perform basic and repetitive tasks. The *intelligent* behaviour from the Swarm Intelligence Algorithms comes from the fact that joining all the information obtained by the individual agents allows the swarm to perform intelligent decisions. This characteristic of the Swarm Intelligence Algorithms is the

```

Set parameters;
Initialise kill matrix (all zeros);
Initialise iteration-hard-to-kill list (all mutants);
Initialise final-hard-to-kill list (empty);
while iteration-hard-to-kill list is not empty do
    Assign a mutant and a set of tests to each agent;
    Each agent applies its set of tests to its mutant;
    Each agent updates the kill matrix;
    Update iteration-hard-to-kill list;
end
Return final-hard-to-kill list;
Algorithm 1: Heuristic: general scheme

```

so called *emergence property*. This property is the basic key for the good results obtained by Swarm Intelligence Algorithms, like the widely known Particle Swarm Optimisation (PSO) algorithm [26] and its most recent variation the Tree Swarm Optimisation [20]. This last algorithm has been recently applied to the Software Testing field [19].

III. SWARM MUTANT CLASSIFICATION

We work with a scenario where we have m mutants and t tests. Our goal is to detect those mutants that are hard to kill by this set of tests, having in mind that our approach should represent a good balance between the computing time and the quality, in terms of the proportion of (un-)detected interesting mutants, of the obtained solution. We have considered a swarm heuristic that avoids the application of the full set of tests to all the mutants and that, at the same time, gives more flexibility than setting a fixed cap to decide when a mutant is *hard-to-kill* [37].

Our heuristic uses three important elements:

- *Agents* conform the swarm that performs the evaluation. We will have a agents.
- The *Kill Matrix* is the matrix where the agents store the information. It will encode which tests kill which mutants, which tests fail to kill which mutants, and which tests have not been applied to which mutants.
- The *hard-to-kill mutants lists* store the *promising* *hard-to-kill* mutants. These lists are updated after each iteration of our algorithm. We will have two *hard-to-kill* mutants lists: one for storing the considered *hard-to-kill* mutants in the current iteration (this one guides the algorithm), and one for storing the final solution.

In Algorithm 1 we present a high-level view of our heuristic. For each iteration, our heuristic has four steps:

- For each of the agents, we choose one mutant from the iteration-hard-to-kill list and choose $n \ll t$ tests¹ that will be applied to that mutant (and that have not been applied previously) and assign them to an agent. If we detect that a mutant cannot leave the iteration-hard-to-kill list with the remaining tests, we add this mutant to the final-hard-to-kill list and take another mutant.
- Each agent applies its set of n tests to its mutant.

¹Note that n is chosen by the user. For our experiments, we used $n = 5$.

- Each agent updates the kill matrix. We use the following convention:
 - 1: the mutant has been killed by that test.
 - 0: the test has not been applied to the mutant.
 - -1: the mutant has not been killed by that test.
- The iteration-hard-to-kill list is updated.

The update of the iteration-hard-to-kill list is the most critical step of the algorithm, because it is where the *emergence property* arises. In this step, the list of mutants considered to be hard-to-kill is updated. The selection is performed after each iteration as a way to guide the development of the algorithm. After each iteration we compute how many tests killed each mutant, storing the highest (max) and the lowest values (min). Then, we mark as hard-to-kill mutants the ones whose value is less than or equal to:

$$\min + \frac{\max - \min}{4}$$

Note that this bound is selected by us but other different bounds could be used. For instance, using this bound we choose all the mutants that after this iteration are in the the lowest quarter of the obtained values. Finally, we remove from the iteration-hard-to-kill list the mutants that are already in the final-hard-to-kill list. We also remove those that have already been tested with all the tests, and we add them to the final-hard-to-kill list.

An interesting property of our heuristic is that the max value does not have to be equal to the maximum number of tests that kill a mutant. In other words, even if a mutant is killed by all the tests, it is possible that the max value is lower than t (the total number of tests). This happens because the difference between max and min will be lower or equal to $\frac{4}{3} \cdot n$ and, therefore, we can have $\max < \min + \frac{4}{3} \cdot n < t$ (and that will be usually the case).

Our heuristic is able to overcome two problems when deciding which mutants are hard-to-kill and which ones are not. The first one is that, in general, it is not necessary to apply all the tests to all the mutants. This will avoid the associated costs of other algorithms based on brute force. The second one is that our heuristic is more flexible than an approach based on fixed percentages. For example, if we define hard-to-kill mutants as the mutants that are killed by at most 5% of the tests, but we have a situation where we have 100 tests and all the mutants are killed by at least 6 tests, then the set of hard-to-kill mutants will be empty. If we use our algorithm in the same situation, our set of hard-to-kill mutants will not be empty because those 6 tests will provide the min value of our range. Therefore, we will always have mutants that can be considered, under the circumstances, the *hardest-to-kill* of all the generated mutants. This implies that our heuristic will be more consistent than other algorithms like random selection.

IV. EXPERIMENTS

In this section we present the experiments that we performed to measure the usefulness of our approach. We wanted to compare our algorithm with three different alternatives: a brute force approach that consists in executing all the tests over all the mutants and afterwards determining which mutants we

consider hard-to-kill; a cap approach where we execute only the necessary tests to know if a mutant overcomes a previously fixed cap; and a random algorithm that executes randomly tests on mutants and then computes the solution. Therefore, our main goal was to answer two questions:

Research Question 1: How many test applications does our approach save when compared to a brute force approach? How different is our approach from a *cap-based* approach?

Research Question 2: What is the quality of the solution of our approach and how *good* it is compared with the solution obtained by a *cap-based* approach and a random approach?

Our heuristic strongly depends on the application of tests to mutants. However, we only use whether a given mutant was killed by a certain test; we do not use any information concerning the actual application of the test (e.g. which parts of the code of the mutant were traversed). Therefore, we only need a *kill matrix* encoding the result of the application of tests to mutants. Each position i, j of the matrix says whether the i th test kills the j th mutant. We have used the matrices provided by a recent work [13], available at <https://mutationtesting.uni.lu/farm/>. These matrices were build from a set of mutants generated from the CodeFlaws [41] and CoREBench [7] program sets, and a set of tests generated by using KLEE [9] for each program. This combination arises 1,737 matrices, with a total count of 4,778,157 mutants and 144,738 tests, what needed 8,463 CPU days of computation. Breaking down by benchmark, the CodeFlaws program set brings a total of 3,213,543 mutants and 122,261 test cases from 1,692 programs with between 1 to 322 lines of code (mean of 36 lines of code). The CoREBench program set brings a total of 1,564,614 mutants and 22,477 test cases from 45 programs with between 9,000 and 83,000 lines of code. In computation terms, the CodeFlaws benchmark needed 8,009 CPU days of computation and the CoREBench benchmark used 454 CPU days of computation to generate all their mutants.

We applied our heuristic to the kill matrix of each program, computing the number of total operations (that is, the number of tests that are applied) and computing the resulting hard-to-kill mutants. As an additional step we computed average values and some quality indicators. These last values will be useful to compare our solutions with respect to previous work [37] where hard-to-kill mutants are those killed by less than 5% of tests. Specifically, we would like to know how different are our hard-to-kill mutants sets from the ones generated using a *fixed cap* approach and how many extra operations we save. Therefore, we also implemented an algorithm to compute those mutants that are killed by 5% of the the tests or less. The algorithm is very simple: it traverses each row of the matrix until more than 5% of the tests have killed the mutant (so it is not a hard-to-kill mutant). We store the quality indicators of these sets of mutants and the number of operations (that is, the number of elements of the matrix that have been accessed) needed to compute them.

In order to have an easier visualisation of the results of our experiments, we combine all these values and plot them. In Figure 1 we have the relative number of operations for each algorithm with respect to the total number of operations obtained by all three algorithms: Brute Force, our Swarm

Mutant Classification (SMC) and the cap algorithm. We can observe that our SMC always needs less operations than the Brute Force algorithm and, depending on the program, may need less operations than the cap algorithm. On average, the Brute Force algorithm needed 70,041 operations while our SMC needed 25,255 operations and the cap algorithm needed 25,109 operations. That is, our SMC needs, on average, 61.97% less operations (that would be applications of tests if we do not have the killing matrix) while the cap algorithm saves, on average, 68.99%.

We performed a statistical hypothesis test over the results concerning operations. The null hypothesis was that the cap algorithm and our SMC algorithm give similar results, that is, both need a similar number of operations. We applied a one-way ANOVA test² where we tested whether the values of both algorithms are, on average, similar. Then, we computed the p-value for the experiment, obtaining a p-value of 0.9044. Therefore, we can confirm the null hypothesis for this experiment because its p-value is much higher than 0.05. In order to double-check our results, we performed a t-test and obtained the same p-value. Thus, the conclusion is that the needed number of operations of our SMC is equivalent to the number needed for the cap algorithm.

In Figure 2 we present the relative operations percentage saving with respect to the saving of both the cap algorithm and our algorithm. We can see how they save some operations with respect to the Brute Force algorithm and how, sometimes, our algorithm outperforms the cap algorithm.

Next, we wanted to perform a quality assessment. In order to do so, we compared our algorithm and the cap algorithm with another new algorithm: the random algorithm. This algorithm applies random tests to random mutants, filling a kill matrix, and then chose the mutants that are killed by at most a fixed number of tests. In our case, in order to compare with the cap algorithm, we decided to take the mutants killed by less than 5% of the tests. Also, in order to compare with our SMC algorithm, we decided that the number of tests that the random algorithm will apply will be equal to the number of tests applied by our algorithm.

We assessed the quality with three indicators: how many mutants that are killed by less than the 5% of tests are not included in the solution; how many mutants that are killed by more than the 25% of tests are included in the solution; and how many mutants, that are killed by less tests than the ones that kill the mutant of the solution that is killed by more tests, are included in the solution. Using these three indicators, we assess three different qualities of the hard-to-kill mutants sets, respectively: how good is the algorithm obtaining the most hard to kill mutants; how good is the algorithm avoiding mutants that cannot be considered hard-to-kill, and how good is the algorithm obtaining dense hard-to-kill mutants sets.

The results of the three algorithms are positive. For the cap algorithm, as it is pretty consistent, the values for the three indicators are equal to 0. For the random algorithm, the mean for the first indicator is 0, for the second indicator is 31.61 and

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

for the third indicator is 303.63. Finally, for our algorithm, that is more flexible than the cap algorithm and it is *less* random than the purely random algorithm, the mean for the first indicator is 24.02, the mean for the second indicator is 1.12, and the mean for the third indicator is 51.16. Analysing these results, we can conclude that the cap algorithm gives what it says: the mutants that are killed by less than the 5% of tests; that the random algorithm gives hard-to-kill mutants sets that are more hollow, what indicates that their choice criteria is less uniform; and that our algorithm gives hard-to-kill mutants sets that are a middle point between the fixed criteria and the random criteria, with more flexibility than the cap algorithm, and with a huge improvement in the choice criteria over the random algorithm.

In Figure 3 we summarise the results concerning the quality of the obtained mutants. We show the cumulative values of the first and second indicator for the two algorithms that obtained values different from 0 and their relative values with respect to the ones obtained by the other algorithm. White lines correspond to cases where all the algorithms obtained a value of 0 for both indicators.

As a recap, answering the first research question, our SMC saves 61.97% of the operations of the brute force approach and needs a similar number of operations as the cap algorithm. In fact, regarding number of operations, both algorithms are statistically equivalent. Concerning the second research question, our approach obtains hard-to-kill mutants sets of good quality. In fact, their quality is better than the quality of the solutions of a random approach, and not so far to the quality of the solutions of the cap approach. However, an advantage of our SMC over the cap approach is that it avoids some extreme cases (from both sides) that appear with it, what makes it a more reliable tool to be used.

V. THREATS TO VALIDITY

Concerning the threats to the validity of our results, most of them have been already addressed. Starting with the internal validity threats, the main concern is whether our results can be the product of internal faults in our experiments code. We addressed this concern by thoroughly testing our code with carefully constructed examples for which we could manually check the results. Another important internal validity threat is whether our results are valid in terms of time computation while using kill matrices instead of properly apply the tests to the mutants. In order to address this threat we compared the number of performed operations instead of execution times, under the assumption that the difference in execution time between tests will not be so critical as the difference in execution time between applying different number of tests. A final internal validity threat is how the randomness associated with the random algorithm affects the obtained results. In order to overcome this threat we repeated the same experiment different times and see that the mean results were similar enough to be considered representative.

Concerning threats to external validity, here arises the question of whether the kill matrices used in our work can be generalised to other families of kill matrices. Although this

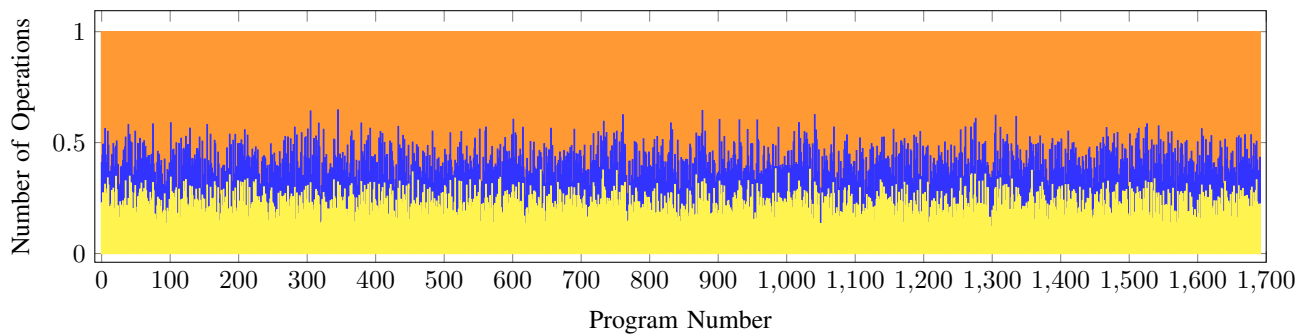


Fig. 1. Relative number of operations (orange = Brute Force, yellow = SMC, blue = cap (5%).)

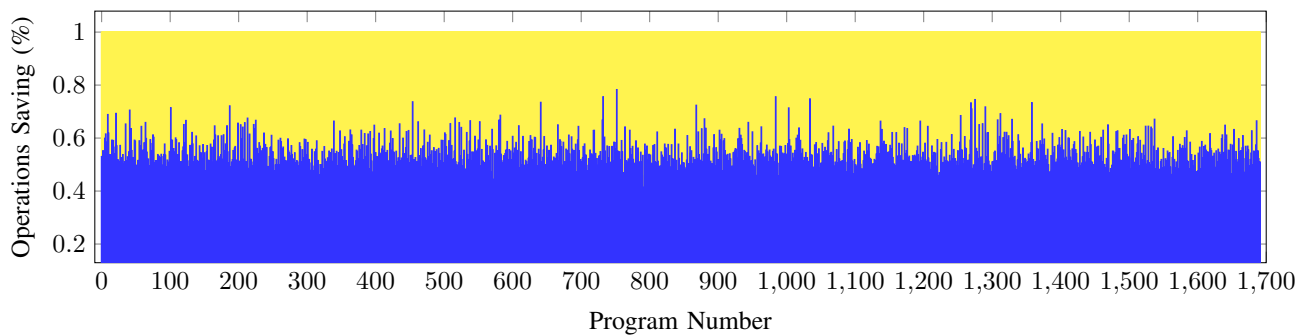


Fig. 2. Relative number of operations savings (yellow = SMC vs Brute Force, blue = cap (5%) vs Brute Force).

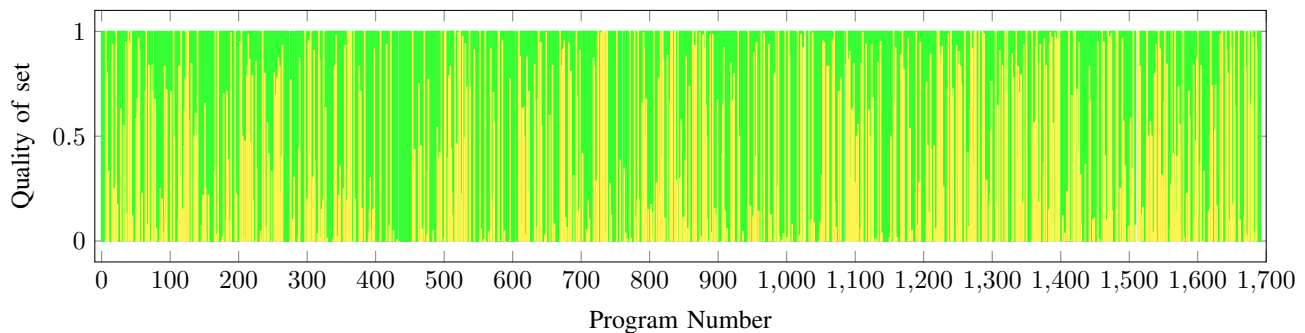


Fig. 3. Relative quality of sets of hard-to-kill mutants (yellow = SMC, green = random).

threat cannot be completely addressed, we worked with kill matrices generated from real programs and with mutants and tests generated by state-of-the-art techniques.

Finally, concerning threats to construct validity, the main concern is whether the kill matrices used in our work can be representative of real programs. Fortunately, our kill matrices were generated from real programs, so they are indeed representative of *real* kill matrices.

VI. CONCLUSIONS

Mutation testing is one of the main techniques in Software Testing. In order to perform a good and efficient mutation process, it is necessary to *filtrate* the mutants. In this work, we focused on detecting hard-to-kill mutants. As the concept of hard-to-kill mutants is too abstract, we developed a Swarm Intelligence Algorithm in order to choose a set of hard-to-kill

mutants from the set of all the mutants of a program. We performed several experiments to prove the efficiency of our algorithm when compared to other approaches to the concept of hard-to-kill mutants. We showed that our SMC is a preferable option to detect those hard-to-kill mutants.

For future work we have identified several lines. First, we would like to assess how the hard-to-kill mutants set changes when modifying the bound for choosing the hard-to-kill mutants in the main loop of our algorithm. Second, we would like to compare the efficiency of our algorithm in a weak mutation scenario, compared to the current strong mutation scenario we presented here. Third, we would like to compare our approach to other Swarm Intelligence Algorithms like Particle Swarm Optimisation [26] and its variations. Fourth, we would like to assess how related are the hard-to-kill mutants determined by

our algorithm and the set of fault revealing mutants [37]. Fifth, we would like to deal with bigger sets of mutants by including recent approaches to mutation testing [11], [15], [18] and our recent work on heuristics based on Information Theory [24]. Finally, we would like to take previous research as initial step to generalise the framework and measures to deal with asynchronous [21], [27], [29], [30], distributed [8], [16], [22], [23], IoT [14], [39] and cloud [4], [5], [12], [35] systems.

REFERENCES

- [1] A. T. Acree, A. T. Budd, R. Demillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, Georgia Institute of Technology, 1979.
- [2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2017.
- [3] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.
- [4] A. Bernal, M. E. Cambrono, A. Núñez, P. C. Cañizares, and V. Valero. Improving cloud architectures using UML profiles and M2T transformation techniques. *The Journal of Supercomputing*, 75(12):8012–8058, 2019.
- [5] A. Bernal, M. E. Cambrono, V. Valero, A. Núñez, and P. C. Cañizares. A framework for modeling cloud infrastructures and user interactions. *IEEE Access*, 7:43269–43285, 2019.
- [6] C. Blum and D. Merkle, editors. *Swarm Intelligence: Introduction and Applications*. Springer, 2008.
- [7] M. Böhme and A. Roychoudhury. CoREBench: studying complexity of regression errors. In *23rd Int. Symposium on Software Testing and Analysis, ISSA'14*, pages 105–115. ACM Press, 2014.
- [8] J. Boubeta-Puig, G. Díaz, H. Macià, V. Valero, and G. Ortiz. MEdit4CEP-CPN: An approach for complex event processing modeling by prioritized colored Petri nets. *Information Systems*, 81:267–289, 2019.
- [9] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI'08*, pages 209–224. USENIX Association, 2008.
- [10] A. Camacho, M. G. Merayo, and M. Núñez. Collective intelligence and databases in eHealth: A survey. *Journal of Intelligent & Fuzzy Systems*, 32(2):1485–1496, 2017.
- [11] P. C. Cañizares, A. Núñez, and M. G. Merayo. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software*, 143:187–207, 2018.
- [12] P. C. Cañizares, A. Núñez, J. de Lara, and L. Llana. MT-EA4Cloud: A methodology for testing and optimising energy-aware cloud systems. *Journal of Systems and Software*, 163:110522:1–110522:25, 2020.
- [13] T. T. Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen. Selecting fault revealing mutants. *Empirical Software Engineering*, 25(1):434–487, 2020.
- [14] D. Corral-Plaza, I. Medina-Bulo, G. Ortiz, and J. Boubeta-Puig. A stream processing architecture for heterogeneous data sources in the internet of things. *Computer Standards & Interfaces*, 70:103426:1–103426:13, 2020.
- [15] P. Delgado-Pérez, Louis M. Rose, and I. Medina-Bulo. Coverage-based quality metric of mutation operators for test suite improvement. *Software Quality Journal*, 27(2):823–859, 2019.
- [16] G. Díaz, H. Macià, V. Valero, J. Boubeta-Puig, and F. Cuartero. An intelligent transportation system to control air pollution and road traffic in cities integrating CEP and colored Petri nets. *Neural Computing and Applications*, 32(2):405–426, 2020.
- [17] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.
- [18] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. A tool for domain-independent model mutation. *Science of Computer Programming*, 163:85–92, 2018.
- [19] D. Griñán and A. Ibias. Generating tree inputs for testing using evolutionary computation techniques. In *22nd IEEE Congress on Evolutionary Computation, CEC'20*, pages E–24267: 1–8. IEEE Computer Society, 2020.
- [20] D. Griñán, A. Ibias, and M. Núñez. Grammar-based tree swarm optimization. In *2019 IEEE Int. Conf. on Systems, Man and Cybernetics, SMC'19*, pages 76–81. IEEE Press, 2019.
- [21] R. M. Hierons, M. G. Merayo, and M. Núñez. An extended framework for passive asynchronous testing. *Journal of Logical and Algebraic Methods in Programming*, 86(1):408–424, 2017.
- [22] R. M. Hierons, M. G. Merayo, and M. Núñez. Bounded reordering in the distributed test architecture. *IEEE Transactions on Reliability*, 67(2):522–537, 2018.
- [23] R. M. Hierons and M. Núñez. Implementation relations and probabilistic schedulers in the distributed test architecture. *Journal of Systems and Software*, 132:319–335, 2017.
- [24] A. Ibias, R. M. Hierons, and M. Núñez. Using Squeeziness to test component-based systems defined as Finite State Machines. *Information & Software Technology*, 112:132–147, 2019.
- [25] Y. Kamei and E. Shihab. Defect prediction: Accomplishments and future challenges. In *Leaders of Tomorrow Symposium: Future of Software Engineering, FOSE@SANER'16*, pages 33–45. IEEE Computer Society, 2016.
- [26] J. Kennedy and R. Eberhart. Particle swarm optimization. In *3rd Int. Conf. on Neural Networks, ICNN'95*, pages 1942–1948. IEEE Computer Society, 1995.
- [27] R. Lefticaru, R. M. Hierons, and M. Núñez. Implementation relations and testing for cyclic systems with refusals and discrete time. *Journal of Systems and Software*, 170:110738:1–110738:20, 2020.
- [28] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.
- [29] M. G. Merayo, R. M. Hierons, and M. Núñez. Passive testing with asynchronous communications and timestamps. *Distributed Computing*, 31(5):327–342, 2018.
- [30] M. G. Merayo, R. M. Hierons, and M. Núñez. A tool supported methodology to passively test asynchronous systems with multiple users. *Information & Software Technology*, 104:162–178, 2018.
- [31] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, 3rd edition, 2011.
- [32] N. T. Nguyen, D. Hwang, and E. Szczerbicki. Computational collective intelligence for enterprise information systems. *Enterprise IS*, 13(7-8):933–934, 2019.
- [33] N. T. Nguyen, E. Szczerbicki, B. Trawinski, and V. D. Nguyen. Collective intelligence in information systems. *Journal of Intelligent and Fuzzy Systems*, 37(6):7113–7115, 2019.
- [34] V. D. Nguyen and N. T. Nguyen. Intelligent collectives: Theory, applications, and research challenges. *Cybernetics and Systems*, 49(5-6):261–279, 2018.
- [35] A. Núñez, P. C. Cañizares, M. Núñez, and R. M. Hierons. TEA-Cloud: A formal framework for testing cloud computing systems. *IEEE Transactions on Reliability (in press)*, 2020.
- [36] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *15th Int. Conf. on Software Engineering, ICSE'93*, pages 100–107. IEEE Computer Society / ACM Press, 1993.
- [37] M. Papadakis, T. T. Chekam, and Y. Le Traon. Mutant quality indicators. In *13th Int. Workshop on Mutation Analysis, MUTATION'18, ICST Workshops*, pages 32–39. IEEE Computer Society, 2018.
- [38] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275 – 378. Elsevier, 2019.
- [39] J. Roldán, J. Boubeta-Puig, J. L. Martínez, and G. Ortiz. Integrating complex event processing and machine learning: An intelligent architecture for detecting iot security attacks. *Expert Systems with Applications*, 149:113251:1–113251:22, 2020.
- [40] S. Selvaraj and E. Choi. Survey of swarm intelligence algorithms. In *3rd Int. Conf. on Software Engineering and Information Management, ICSIM'20*, pages 69–73. ACM Press, 2020.
- [41] S. H. Tan, J. Yi, Y., S. Mechtaev, and A. Roychoudhury. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *39th Int. Conf. on Software Engineering, ICSE'17 Companion Volume*, pages 180–182. IEEE Computer Society, 2017.
- [42] W. Visser. What makes killing a mutant hard. In *31st IEEE/ACM Int. Conf. on Automated Software Engineering, ASE'16*, pages 39–44. ACM Press, 2016.
- [43] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.

Part VI

Bibliography

- [1] A. T. Acree, A. T. Budd, R. Demillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, Georgia Institute of Technology, 1979.
- [2] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation - GECCO'04, Genetic and Evolutionary Computation Conf.*, volume 3103 of *Lecture Notes in Computer Science*, pages 1338–1349. Springer, 2004.
- [3] F. Ahishakiye, J.-I. R. Jarabo, L. M. Kristensen, and V. Stolz. Coverage analysis of net inscriptions in coloured petri net models. In *Verification and Evaluation of Computer and Communication Systems - 14th International Conference, VECoS'20*, volume 12519 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2020.
- [4] F. Ahishakiye, J.-I. R. Jarabo, L. M. Kristensen, and V. Stolz. MC/DC test cases generation based on bdds. In *Dependable Software Engineering. Theories, Tools, and Applications - 7th International Symposium, SETTA'21*, volume 13071 of *Lecture Notes in Computer Science*, pages 178–197. Springer, 2021.
- [5] A. Aho, A. Dahbura, D. Lee, and M. Ü. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman tours. In *Protocol Specification, Testing and Verification VIII*, pages 75–86. North Holland, 1988.

-
- [6] A. V. Aho, A. T. Dahbura, D. Lee, and 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):1604–1615, 1991.
- [7] S. Ali, L. Briand, and H. Hemmati. Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems. *Software and Systems Modeling*, 11(4):633–670, 2012.
- [8] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *24th ACM SIGSOFT Int. Symposium on Software Testing and Analysis, ISSSTA'14*, pages 181–192. ACM Press, 2014.
- [9] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2017.
- [10] S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.
- [11] C. Andrés, C. Camacho, and L. Llana. A formal framework for software product lines. *Information & Software Technology*, 55(11):1925–1947, 2013.
- [12] C. Andrés, M. G. Merayo, and M. Núñez. Multi-objective genetic algorithms: Construction and recombination of passive testing properties. In *22nd Int. Conf. on Software Engineering & Knowledge Engineering, SEKE'10*, pages 405–410. Knowledge Systems Institute, 2010.
- [13] K. Androutsopoulos, D. Clark, H. Dan, R.M. Hierons, and 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*, pages 573–583. ACM Press, 2014.
- [14] R. Anido, A. R. Cavalli, L. A. Paula Lima Jr., and N. Yevtushenko. Test suite minimization for testing in context. *Software Testing, Verification and Reliability*, 13(3):141–155, 2003.
- [15] T. Apiwattanapong, R. A. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. MATRIX: Maintenance-oriented testing requirements identifier and examiner. In *1st Testing: Academia and Industry Conference - Practice And Research Techniques, TAIC PART'06*, pages 137–146. IEEE Computer Society, 2006.
- [16] T. Arbuckle. Studying software evolution using artefacts' shared information content. *Sci. Comput. Program.*, 76(12):1078–1097, 2011.

-
- [17] K. Ayari, S. Bouktif, and G. Antoniol. Automatic mutation test input data generation via ant colony. In *Genetic and Evolutionary Computation Conf., GECCO'07*, pages 1074–1081. ACM, 2007.
- [18] M. Badri, L. Badri, W. Flageol, and F. Touré. Investigating the accuracy of test code size prediction using use case metrics and machine learning algorithms: An empirical study. In *2017 Int. Conf. on Machine Learning and Soft Computing, ICMLSC'17*, pages 25–33. ACM, 2017.
- [19] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. Genes and bacteria for automatic test cases optimization in the .net environment. In *13th Int. Symp. on Software Reliability Engineering (ISSRE'02)*, pages 195–206. IEEE Computer Society, 2002.
- [20] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. Automatic test case optimization: A bacteriologic algorithm. *IEEE Softw.*, 22(2):76–82, 2005.
- [21] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Softw. Test. Verification Reliab.*, 15(2):73–96, 2005.
- [22] B. Baudry, F. Fleurey, J.M. Jézéquel, and Y. Le Traon. Automatic test cases optimization using a bacteriological adaptation model: Application to .NET components. In *17th Int. Conf. on Automated Software Engineering, ASE'02*, page 253. IEEE Computer Society, 2002.
- [23] M. H. ter Beek, A. Borälv, A. Fantechi, A. Ferrari, S. Gnesi, C. Löfving, and F. Mazzanti. Adopting formal methods in an industrial setting: The railways case. In *23rd Int. Symposium on Formal Methods, FM'19 LNCS 11800*, pages 762–772. Springer, 2019.
- [24] M. Benito-Parejo, I. Medina-Bulo, M. G. Merayo, and M. Núñez. Using genetic algorithms to generate test suites for FSMs. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 741–752. Springer, 2019.
- [25] M. Benito-Parejo and M. G. Merayo. An evolutionary algorithm for selection of test cases. In *22nd IEEE Congress on Evolutionary Computation, CEC'20*, pages E–24535: 1–8. IEEE Computer Society, 2020.
- [26] F. Bergadano and D. Gunetti. Testing by means of inductive program learning. *ACM Trans. Softw. Eng. Methodol.*, 5(2):119–145, 1996.
- [27] C. Blum and D. Merkle, editors. *Swarm Intelligence: Introduction and Applications*. Springer, 2008.

- [28] J. K. Blundell, M. L. Hines, and J. Stach. The measurement of software design quality. *Annals of Software Engineering*, 4(1–4):235–255, 1997.
- [29] Z. Bluvband, S. Porotsky, and M. Talmor. Advanced models for software reliability prediction. In *2011 Annual Reliability and Maintainability Symp.*, pages 1–5, 2011.
- [30] M Böhme. STADS: software testing as species discovery. *ACM Trans. Softw. Eng. Methodol.*, 27(2):7:1–7:52, 2018.
- [31] A. D. Bonis, L. Gasieniec, and U. Vaccaro. Generalized framework for selectors with applications in optimal group testing. In *Automata, Languages and Programming, 30th Int. Colloquium, ICALP’03*, pages 81–96, 2003.
- [32] J. Boubeta-Puig, G. Díaz, H. Macià, V. Valero, and G. Ortiz. MEdit4CEP-CPN: An approach for complex event processing modeling by prioritized colored Petri nets. *Information Systems*, 81:267–289, 2019.
- [33] C. Braunstein, A. E. Haxthausen, W.-L. Huang, F. Hübner, J. Peleska, U. Schulze, and L. V. Hong. Complete model-based equivalence class testing for the ETCS ceiling speed monitor. In *16th Int. Conf. on Formal Engineering Methods, ICFEM’14, LNCS 8829*, pages 380–395. Springer, 2014.
- [34] L. C. Briand, V. R. Basili, and C. J. Hetmanski. Providing an empirical basis for optimizing the verification and testing phases of software development. In *3rd Int. Symp. on Software Reliability Engineering, ISSRE’92*, pages 329–338, 1992.
- [35] L. C. Briand, J. Feng, and Y. Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *14th Int. Conf. on Software engineering and knowledge engineering, SEKE’02*, pages 43–50. ACM, 2002.
- [36] D. E. Brown. A method for obtaining software reliability measures during development. *IEEE Transactions on Reliability*, R-36(5):573–580, Dec 1987.
- [37] P. M. S. Bueno and M. Jino. Identification of potentially infeasible program paths by monitoring the search for test data. In *15th IEEE Int. Conf. on Automated Software Engineering, ASE’00*, pages 209–218. IEEE Computer Society, 2000.
- [38] C. Camacho, L. Llana, and A. Núñez. Cost-related interface for software product lines. *Journal of Logic and Algebraic Methods in Programming*, 85(1):227–244, 2016.

- [39] C. Camacho, L. Llana, A. Núñez, and M. Bravetti. Probabilistic software product lines. *Journal of Logic and Algebraic Methods in Programming*, 107:54–78, 2019.
- [40] J. Campos, R. Abreu, G. Fraser, and M. d’Amorim. Entropy-based test generation for improved fault localization. In *28th IEEE/ACM Int. Conf. on Automated Software Engineering, ASE’13*, pages 257–267. IEEE, 2013.
- [41] J. Campos, Y. Ge, N. Albulian, G. Fraser, M. Eler, and A. Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, 104:207–235, 2018.
- [42] P. C. Cañizares, A. Núñez, and M. G. Merayo. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software*, 143:187–207, 2018.
- [43] I. do Carmo Machado, P. A. da Mota Silveira Neto, and E. Santana de Almeida. Towards an integration testing approach for software product lines. In *IEEE 13th Int. Conf. on Information Reuse & Integration, IRI’12*, pages 616–623. IEEE, 2012.
- [44] E. G. Cartaxo, P. D. L. Machado, and F. G. de Oliveira Neto. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability*, 21(2):75–100, 2011.
- [45] A. R. Cavalli, T. Higashino, and M. Núñez. A survey on formal active and passive testing with applications to the cloud. *Annales of Telecommunications*, 70(3-4):85–93, 2015.
- [46] T. J. Cheatham, J. P. Yoo, and N. J. Wahl. Software testing: A machine learning experiment. In *1995 ACM 23rd Annual Conf. on Computer Science, CSC’95*, pages 135–141. ACM, 1995.
- [47] T. T. Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen. Selecting fault revealing mutants. *Empirical Software Engineering*, 25(1):434–487, 2020.
- [48] T. Y. Chen, J. Feng, and T. H. Tse. Metamorphic testing of programs on partial differential equations: A case study. In *26th Int. Computer Software and Applications Conf. (COMPSAC’02)*, pages 327–333. IEEE Computer Society, 2002.
- [49] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys*, 51(1):4:1–4:27, 2018.

- [50] N. Chetouane, F. Wotawa, H. Felbinger, and M. Nica. On using k-means clustering for test suite reduction. In *13th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW'20*, pages 380–385. IEEE, 2020.
- [51] W. Choi, G. C. Necula, and K. Sen. Guided GUI testing of android apps with minimal restart and approximate learning. In *2013 ACM SIGPLAN Int. Conf. on Object Oriented Programming Systems Languages & Applications, OOPSLA'13*, pages 623–640. ACM, 2013.
- [52] T. S. Chow. Testing software design modeled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.
- [53] P. Chrszon, C. Dubslaff, S. Klüppelholz, and C. Baier. ProFeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects of Computing*, 30(1):45–75, 2018.
- [54] R. Cilibrasi and P. M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.
- [55] D. Clark, R. Feldt, S. M. Poulding, and S. Yoo. Information transformation: An underpinning theory for software engineering. In *37th IEEE/ACM International Conference on Software Engineering, ICSE'15*, pages 599–602, 2015.
- [56] D. Clark and R. M. Hierons. Squeeziness: An information theoretic measure for avoiding fault masking. *Information Processing Letters*, 112(8-9):335–340, 2012.
- [57] D. Clark, R. M. Hierons, and K. Patel. Normalised Squeeziness and Failed Error Propagation. *Information Processing Letters*, 149:6–9, 2019.
- [58] M. Cordy, P. Heymans, P. Schobbens, A. M. Sharifloo, C. Ghezzi, and A. Legay. Verification for reliable product lines. *CoRR*, abs/1311.1343, 2013.
- [59] J. Couchet, D. Manrique, J. Rios, and A. Rodríguez-Patón. Crossover and mutation operators for grammar-guided genetic programming. *Soft Computing*, 11(10):943–955, 2007.
- [60] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley Interscience, 1991.
- [61] Y. Dai, M. Xie, Q. Long, and S. Ng. Uncertainty analysis in software reliability modeling by bayesian analysis with maximum-entropy principle. *IEEE Transactions on Software Engineering*, 33(11):781–795, 2007.

- [62] M. Dave and R. Agrawal. Software testing and information theory. In *Smart Trends in Information Technology and Computer Communications*, pages 323–330, Singapore, 2016. Springer Singapore.
- [63] P. Delgado-Pérez, I. Medina-Bulo, and J. J. Domínguez-Jiménez. Mutation testing. In *Encyclopedia of Information Science and Technology*, pages 7212–7221. IGI Global, 3rd edition, 2014.
- [64] P. Delgado-Pérez, I. Medina-Bulo, and M. Núñez. Using evolutionary mutation testing to improve the quality of test suites. In *19th IEEE Congress on Evolutionary Computation, CEC'17*, pages 596–603. IEEE Computer Society, 2017.
- [65] P. Delgado-Pérez, L. M. Rose, and I. Medina-Bulo. Coverage-based quality metric of mutation operators for test suite improvement. *Software Quality Journal*, 27(2):823–859, 2019.
- [66] Z. Demirezen. *An Information Theory Based Representation Of Software Systems And Design*. PhD thesis, University of Alabama – Birmingham, 2012.
- [67] K. Derderian, M. G. Merayo, R. M. Hierons, and M. Núñez. Aiding test case generation in temporally constrained state based systems using genetic algorithms. In *10th Int. Conf. on Artificial Neural Networks, IWANN'09, LNCS 5517*, pages 327–334. Springer, 2009.
- [68] K. Derderian, M. G. Merayo, R. M. Hierons, and M. Núñez. A case study on the use of genetic algorithms to generate test cases for temporal systems. In *11th Int. Conf. on Artificial Neural Networks, IWANN'11, LNCS 6692*, pages 396–403. Springer, 2011.
- [69] J. Derrick and E. Boiten. Testing refinements of state-based formal specifications. *Software Testing, Verification and Reliability*, 9(1):27–50, 1999.
- [70] I. Dinca, A. Stefanescu, F. Ipate, R. Lefticaru, and C. Tudose. Test data generation for event-b models using genetic algorithms. In *Software Engineering and Computer Systems - Second International Conference, ICSECS'11*, volume 181 of *Communications in Computer and Information Science*, pages 76–90. Springer, 2011.
- [71] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.
- [72] M. Dorigo, V. Maniezzo, and A. Coloni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics B*, 26(1):29–41, 1996.

- [73] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, 2004.
- [74] V. H. S. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. C. Dias, and M. d. P. Guimarães. Machine learning applied to software testing: A systematic mapping study. *IEEE Trans. Reliab.*, 68(3):1189–1212, 2019.
- [75] D. G. e Silva, M. Jino, and B. T. de Abreu. Machine learning methods and asymmetric cost function to estimate execution effort of software testing. In *3rd Int. Conf. on Software Testing, Verification and Validation, ICST'10*, pages 275–284. IEEE Computer Society, 2010.
- [76] O. Ekundayo and S. Viriri. Facial expression recognition: A review of trends and techniques. *IEEE Access*, 9:136944–136973, 2021.
- [77] K. El-Fakih, A. Petrenko, and N. Yevtushenko. FSM test translation through context. In *18th Int. Conf. on Testing Communicating Systems, TestCom'06, LNCS 3964*, pages 245–258. Springer, 2006.
- [78] M. C. F. P. Emer and S. R. Vergilio. Gptest: A testing tool based on genetic programming. In *GECCO'02: Genetic and Evolutionary Computation Conf.*, pages 1343–1350. Morgan Kaufmann, 2002.
- [79] M. Eriksson, J. Borstler, and K. Borg. The PLUSS approach - domain modeling with features, use cases and use case realizations. In *9th Int. Conference on Software Product Lines, SPLC'06, LNCS 3714*, pages 33–44. Springer, 2006.
- [80] R. Feldt, S. M. Poulding, D. Clark, and S. Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *9th IEEE Int. Conf. on Software Testing, Verification and Validation, ICST'16*, pages 223–233. IEEE Computer Society, 2016.
- [81] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *1st IEEE Int. Conf. on Software Testing Verification and Validation Workshops*, pages 178–186. IEEE Computer Society, 2008.
- [82] J. Ferrer, F. Chicano, and E. Alba. Estimating software testing complexity. *Information & Software Technology*, 55(12):2125–2139, 2013.
- [83] G. Fraser and F. Wotawa. Using model-checkers for mutation-based test-case generation, coverage analysis and specification analysis. In *International Conference on Software Engineering Advances (ICSEA'06)*, page 16. IEEE Computer Society, 2006.
- [84] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

- [85] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *Int. Symp. on Software Testing and Analysis, ISSA '02*, pages 134–143. ACM, 2002.
- [86] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite-state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.
- [87] P. García-Sánchez, A. Tonda, A. M. Mora, G. Squillero, and J. J. Merelo. Automated playtesting in collectible card games using evolutionary algorithms: A case study in hearthstone. *Knowledge-Based Systems*, 153:133 – 146, 2018.
- [88] M.-C. Gaudel. Testing can be formal, too! In *6th Int. Joint Conf. CAAP/FASE, Theory and Practice of Software Development, TAPSOFT'95, LNCS 915*, pages 82–96. Springer, 1995.
- [89] D. E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, 1989.
- [90] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. A tool for domain-independent model mutation. *Science of Computer Programming*, 163:85–92, 2018.
- [91] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. Wodel-Test: a model-based framework for language-independent mutation testing. *Software and Systems Modeling (in press)*, 2021.
- [92] A. González-Sánchez, É. Piel, R. Abreu, H.-G. Groß, and A. J. C. van Gemund. Prioritizing tests for software fault diagnosis. *Softw. Pract. Exp.*, 41(10):1105–1129, 2011.
- [93] A. González-Sánchez, É. Piel, H.-G. Groß, and A. J. C. van Gemund. Prioritizing tests for software fault localization. In *10th Int. Conf. on Quality Software, QSIC'10*, pages 42–51. IEEE Computer Society, 2010.
- [94] K. Goseva-Popstojanova and S. Kamavaram. Assessing uncertainty in reliability of component-based software systems. In *14th Int. Symp. on Software Reliability Engineering (ISSRE'03)*, pages 307–320, 2003.
- [95] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ACM SIGSOFT Symposium on Software Testing and Analysis, ISSA '02*, pages 112–122. ACM Press, 2002.
- [96] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011.

- [97] D. Griñán and A. Ibias. Generating tree inputs for testing using evolutionary computation techniques. In *22nd IEEE Congress on Evolutionary Computation, CEC'20*, pages E-24267: 1–8. IEEE Computer Society, 2020.
- [98] D. Griñán, A. Ibias, and M. Núñez. Grammar-based tree swarm optimization. In *2019 IEEE Int. Conf. on Systems, Man and Cybernetics, SMC'19*, pages 76–81. IEEE Press, 2019.
- [99] M. L. Griss, J. M. Favaro, and M. D'Alessandro. Integrating feature modeling with the RSEB. In *Int. Conf. on Software Reuse, ICSR'98*, pages 76–85, 1998.
- [100] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *22nd ACM SIGSOFT Int. Symposium on Software Testing and Analysis, ISSTA'12*, pages 78–88. ACM Press, 2012.
- [101] Q. Guo, R. M. Hierons, M. Harman, and K. Derderian. Computing Unique Input/Output sequences using genetic algorithms. In *3rd Int. Workshop on Formal Approaches to Testing of Software, FATES'03, LNCS 2931*, pages 164–177. Springer, 2003.
- [102] Q. Guo, R. M. Hierons, M. Harman, and K. Derderian. Improving test quality using robust unique input/output circuit sequences (UIOCs). *Information & Software Technology*, 48(8):696–707, 2006.
- [103] A. Gupta, N. Gupta, and R. Kumar-Garg. Implementing weighted entropy-distance based approach for the selection of software reliability growth models. *Int. J. Comput. Appl. Technol.*, 57(3):255–266, 2018.
- [104] L. Gutiérrez-Madroñal, A. García-Domínguez, and I. Medina-Bulo. Evolutionary mutation testing for IoT with recorded and generated events. *Software - Practice & Experience*, 49(4):640–672, 2019.
- [105] V. Le Hanh, K. Akif, Y. Le Traon, and J.-M. Jézéquel. Selecting an efficient OO integration testing strategy: An experimental comparison of actual strategies. In *ECOOP'01 - Object-Oriented Programming, 15th European Conf.*, volume 2072 of *Lecture Notes in Computer Science*, pages 381–401. Springer, 2001.
- [106] M. Harman, S. A. Mansouri, and Y. Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, Department of Computer Science, King's College London, London, UK, April 2009.
- [107] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

- [108] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology*, 22(1):6:1–6:42, 2013.
- [109] H. Hemmati, Z. Fang, and M. V. Mantyla. Prioritizing manual test cases in traditional and rapid release environments. In *8th IEEE Int. Conf. on Software Testing, Verification and Validation, ICST'15*, pages 1–10. IEEE Computer Society, 2015.
- [110] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. Comparing white-box and black-box test prioritization. In *38th Int. Conf. on Software Engineering, ICSE'16*, pages 523–534. ACM Press, 2016.
- [111] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize T-Wise test configurations for software product lines. *IEEE Transactions on Software Engineering*, 40(7):650–670, 2014.
- [112] F. C. Hennie. Fault-detecting experiments for sequential circuits. In *5th Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110. IEEE Computer Society, 1964.
- [113] R. Hewett. Mining software defect data to support software testing management. *Appl. Intell.*, 34(2):245–257, 2011.
- [114] R. M. Hierons. Reaching and distinguishing states of distributed systems. *SIAM Journal on Computing*, 39(8):3480–3500, 2010.
- [115] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luetzgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2):9:1–9:76, 2009.
- [116] R. M. Hierons, M. G. Merayo, and M. Núñez. Mutation testing. In Phillip A. Laplante, editor, *Encyclopedia of Software Engineering*, pages 594–602. Taylor & Francis, 2010.
- [117] R. M. Hierons, M. G. Merayo, and M. Núñez. Controllability through nondeterminism in distributed testing. In *28th IFIP WG 6.1 Int. Conf. on Testing Software and Systems, ICTSS'16, LNCS 9976*, pages 89–105. Springer, 2016.
- [118] R. M. Hierons, M. G. Merayo, and M. Núñez. An extended framework for passive asynchronous testing. *Journal of Logical and Algebraic Methods in Programming*, 86(1):408–424, 2017.

-
- [119] R. M. Hierons, M. G. Merayo, and M. Núñez. Bounded reordering in the distributed test architecture. *IEEE Transactions on Reliability*, 67(2):522–537, 2018.
- [120] R. M. Hierons and M. Núñez. Implementation relations and probabilistic schedulers in the distributed test architecture. *Journal of Systems and Software*, 132:319–335, 2017.
- [121] R. M. Hierons and H. Ural. Optimizing the length of checking sequences. *IEEE Transactions on Computers*, 55(5):618–629, 2006.
- [122] R. M. Hierons and H. Ural. Checking sequences for distributed test architectures. *Distributed Computing*, 21(3):223–238, 2008.
- [123] R. M. Hierons and H. Ural. The effect of the distributed test architecture on the power of testing. *The Computer Journal*, 51(4):497–510, 2008.
- [124] J. Huo and A. Petrenko. Covering transitions of concurrent systems through queues. In *16th Int. Symposium on Software Reliability Engineering, ISSRE’05*, pages 335–345. IEEE Computer Society, 2005.
- [125] I. Hwang and A. R. Cavalli. Testing a probabilistic FSM using interval estimation. *Computer Networks*, 54(7):1108–1125, 2010.
- [126] I. Hwang, A. R. Cavalli, M. Lallali, and D. Verchère. Applying formal methods to PCEP: an industrial case study from modeling to test generation. *Software Testing, Verification and Reliability*, 22(5):343–361, 2012.
- [127] A. Ibias. Using mutual information to test from Finite State Machines: Test suite generation. *Submitted for publication, -:-*, 2022.
- [128] A. Ibias, D. Griñán, and M. Núñez. GPTSG: a Genetic Programming Test Suite Generator using Information Theory measures. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN’19, LNCS 11506*, pages 716–728. Springer, 2019.
- [129] A. Ibias, R. M. Hierons, and M. Núñez. Using Squeeziness to test component-based systems defined as Finite State Machines. *Information & Software Technology*, 112:132–147, 2019.
- [130] A. Ibias and L. Llana. Feature selection using evolutionary computation techniques for software product line testing. In *22nd IEEE Congress on Evolutionary Computation, CEC’20*, pages E–24502: 1–8. IEEE Computer Society, 2020.

- [131] A. Ibias, L. Llana, and M. Núñez. Using ant colony optimisation to select features having associated costs. In *33rd IFIP Int. Conf. on Testing Software and Systems, ICTSS'21*, pages –. IEEE, 2021.
- [132] A. Ibias and M. Núñez. Estimating fault masking using Squeeziness based on Rényi's entropy. In *35th ACM Symposium on Applied Computing, SAC'20*, pages 1936–1943. ACM Press, 2020.
- [133] A. Ibias and M. Núñez. Using a swarm to detect hard-to-kill mutants. In *2020 IEEE Int. Conf. on Systems, Man and Cybernetics, SMC'20*, pages 2190–2195. IEEE Press, 2020.
- [134] A. Ibias and M. Núñez. **SqSelect**: Automatic assessment of failed error propagation in state-based systems. *Expert Systems with Applications*, 174:114748, 2021.
- [135] A. Ibias and M. Núñez. Squeeziness for Non-Deterministic Systems. *Submitted for publication, -:-*, 2022.
- [136] A. Ibias, M. Núñez, and R. M. Hierons. Using mutual information to test from Finite State Machines: Test suite selection. *Information & Software Technology*, 132:106498, 2021.
- [137] A. Ibias, P. Vazquez-Gomis, and M. Benito-Parejo. Coverage-based grammar-guided genetic programming generation of test suites. In *IEEE Congress on Evolutionary Computation, CEC'21*, pages 2411–2418. IEEE, 2021.
- [138] F. Ingrand and M. Ghallab. Deliberation for autonomous robots: A survey. *Artif. Intell.*, 247:10–44, 2017.
- [139] F. Ipate. Bounded sequence testing from deterministic finite state machines. *Theoretical Computer Science*, 411(16-18):1770–1784, 2010.
- [140] ISO/IEC JTC1/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.
- [141] M. Jaffar-ur Rehman, F. Jabeen, A. Bertolino, and A. Polini. Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability*, 17(2):95–133, 2007.
- [142] G. Jahangirova, D. Clark, M. Harman, and P. Tonella. An empirical study on failed error propagation in java programs with real faults. *CoRR*, abs/2011.10787, 2020.
- [143] A. K. Jain, J. Mao, and K. M. Mohiuddin. Artificial neural networks: A tutorial. *IEEE Computer*, 29(3):31–44, 1996.

- [144] K. Jalbert and J. S. Bradbury. Predicting mutation score using source code and test suite metrics. In *1st Int. Workshop on Realizing AI Synergies in Software Engineering, RAISE'12*, pages 42–46. IEEE, 2012.
- [145] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *8th IEEE Int. Working Conf. on Source Code Analysis and Manipulation (SCAM'08)*, pages 249–258. IEEE Computer Society, 2008.
- [146] Y. Jia and M. Harman. Higher order mutation testing. *Inf. Softw. Technol.*, 51(10):1379–1393, 2009.
- [147] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.
- [148] Y. Kamei and E. Shihab. Defect prediction: Accomplishments and future challenges. In *Leaders of Tomorrow Symposium: Future of Software Engineering, FOSE@SANER'16*, pages 33–45. IEEE Computer Society, 2016.
- [149] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, 1990.
- [150] J. Kennedy and R. Eberhart. Particle swarm optimization. In *3rd Int. Conf. on Neural Networks, ICNN'95*, pages 1942–1948. IEEE Computer Society, 1995.
- [151] F. Khomh, B. Chan, Y. Zou, and A. E. Hassan. An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox. In *18th Working Conf. on Reverse Engineering, WCRE'11*, pages 261–270. IEEE Computer Society, 2011.
- [152] T. M. Khoshgoftaar and T. G. Woodcock. Software reliability model selection: a cast study. In *2nd Int. Symp. on Software Reliability Engineering, ISSRE'91*, pages 183–191, 1991.
- [153] B. Kim, K.-T. Kim, and E. Y. Kim. Reconstruction of degraded images using genetic algorithm for archive film restoration. In *Int. Conf. on Image Processing, ICIP'09*, pages 2765–2768. IEEE, 2009.
- [154] K. N. King and A. J. Offutt. A fortran language system for mutation-based software testing. *Softw. Pract. Exp.*, 21(7):685–718, 1991.
- [155] Z. Kohavi. *Switching and Finite State Automata Theory*. McGraw-Hill, 1978.

- [156] J. R. Koza. *Genetic programming*. MIT Press, 1993.
- [157] M. Kumar, A. Sharma, and R. Kumar. Fuzzy entropy-based framework for multi-faceted test case classification and selection: an empirical study. *IET Softw.*, 8(3):103–112, 2014.
- [158] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *23rd Int. Conf. on Computer Aided Verification, CAV’11, LNCS 6806*, pages 585–591. Springer, 2011.
- [159] R. Lachmann, S. Beddig, S. Lity, S. Schulze, and I. Schaefer. Risk-based integration testing of software product lines. In *11th Int. Workshop on Variability Modelling of Software-Intensive Systems, VaMoS’17*, pages 52–59. ACM Press, 2017.
- [160] J. W. Laski, W. Szermer, and P. Luczycki. Error masking in computer programs. *Software Testing, Verification and Reliability*, 5(2):81–105, 1995.
- [161] G. I. Latiu, O. A. Cret, and L. Vacariu. Automatic test data generation for software path testing using evolutionary algorithms. In *2012 3rd Int. Conf. on Emerging Intelligent Data and Web Technologies*, pages 1–8. IEEE Computer Society, 2012.
- [162] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines: A survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
- [163] M.-C. Lee and T. Chang. Software measurement and software metrics in software quality. *Int. Journal of software engineering and its application*, 7:15–33, 01 2013.
- [164] R. Lefticaru and F. Ipate. Automatic state-based test generation using genetic algorithms. In *9th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC’07*, pages 188–195. IEEE Computer Society, 2007.
- [165] R. Lefticaru and F. Ipate. An improved test generation approach from extended finite state machines using genetic algorithms. In *Software Engineering and Formal Methods - 10th International Conference, SEFM’12*, volume 7504 of *Lecture Notes in Computer Science*, pages 293–307. Springer, 2012.
- [166] M. Li and P. M. B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, 4th edition, 2019.

- [167] G. Lin and J. Wang. Software operational profile reduction and classification testing based on entropy theory. In *2009 Int. Conf. on Artificial Intelligence and Computational Intelligence*, volume 1, pages 73–77, 2009.
- [168] J.-C. Lin and P.-L. Yeh. Automatic test data generation for path testing using gas. *Inf. Sci.*, 131(1-4):47–64, 2001.
- [169] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, A. Egyed, and E. Alba. Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of software product lines. In *16th IEEE Congress on Evolutionary Computation, CEC'14*, pages 387–396. IEEE, 2014.
- [170] Y. Lou, J. Chen, L. Zhang, and D. Hao. Chapter one - A survey on regression test-case prioritization. *Adv. Comput.*, 113:1–46, 2019.
- [171] J. M. Luna, J. R. Romero, and S. Ventura. Design and behavior study of a grammar-guided genetic programming algorithm for mining association rules. *Knowledge and Information Systems*, 32(1):53–76, 2012.
- [172] T. Mantere and J. T. Alander. Evolutionary software engineering, a review. *Appl. Soft Comput.*, 5(3):315–331, 2005.
- [173] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Automatic testing of gui-based applications. *Softw. Test. Verification Reliab.*, 24(5):341–366, 2014.
- [174] R. Marinescu, C. Secleanu, H. Le Guen, and P. Pettersson. *A Research Overview of Tool-Supported Model-based Testing of Requirements-based Designs*, volume 98 of *Advances in Computers*, chapter 3, pages 89–140. Elsevier, 2015.
- [175] W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In *2nd Int. Workshop on Defects in Large Software Systems, DEFECTS'09*, pages 1–5. ACM Press, 2009.
- [176] P. May, K. Mander, and J. Timmis. Software vaccination: An artificial immune system approach to mutation testing. In *Artificial Immune Systems, Second Int. Conf., ICARIS'03*, volume 2787 of *Lecture Notes in Computer Science*, pages 81–92. Springer, 2003.
- [177] P. May, J. Timmis, and K. Mander. Immune and evolutionary approaches to software mutation testing. In *6th Int. Conf. on Artificial Immune Systems, ICARIS'07, LNCS 4628*, pages 336–347. Springer, 2007.

- [178] J. D. McGregor. Testing a software product line. In *Testing Techniques in Software Engineering, Pernambuco Summer School on Software Engineering, PSSE'07*, pages 104–140, 2007.
- [179] J. D. McGregor and D. A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison Wesley object technology series. Pearson / Prentice Hall, 2001.
- [180] R. I. McKay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010.
- [181] G.H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34:1045–1079, 1955.
- [182] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.
- [183] M. G. Merayo, R. M. Hierons, and M. Núñez. Passive testing with asynchronous communications and timestamps. *Distributed Computing*, 31(5):327–342, 2018.
- [184] M. G. Merayo, R. M. Hierons, and M. Núñez. A tool supported methodology to passively test asynchronous systems with multiple users. *Information & Software Technology*, 104:162–178, 2018.
- [185] C. C. Michael, G. E. McGraw Jr., M. Schatz, and C. C. Walton. Genetic algorithms for dynamic test data generation. In *1997 Int. Conf. on Automated Software Engineering, ASE'97*, pages 307–308. IEEE Computer Society, 1997.
- [186] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.
- [187] T. Minohara and Y. Tohma. Parameter estimation of hyper-geometric distribution software reliability growth model by genetic algorithms. In *6th Int. Symp. on Software Reliability Engineering, ISSRE'95*, pages 324–329. IEEE Computer Society, 1995.
- [188] A. V. Miranskyy, M. Davison, R. M. Reesor, and S. S. Murtaza. Using entropy measures for comparison of software traces. *Information Sciences*, 203:59–72, 2012.
- [189] D. B. Mishra, R. Mishra, A. A. Acharya, and K. N. Das. Test data generation for mutation testing using genetic algorithm. In *Soft Computing for Problem Solving - SocProS'17*, volume 817 of *Advances in Intelligent Systems and Computing*, pages 857–867. Springer, 2017.

- [190] M. Mitchell. *An introduction to genetic algorithms*. MIT Press, 1998.
- [191] S. N. Mohanty. Models and measurements for quality assessment of software. *ACM Comput. Surv.*, 11(3):251–275, 1979.
- [192] E. P. Moore. Gedanken experiments on sequential machines. In C. Shannon and J. McCarthy, editors, *Automata Studies*. Princeton University Press, 1956.
- [193] A. Muhamed, L. Li, X. Shi, S. Yaddanapudi, W. Chi, D. Jackson, R. Suresh, Z. C. Lipton, and A. J. Smola. Symbolic music generation with transformer-gans. In *35th AAAI Conf. on Artificial Intelligence, AAAI'21*, pages 408–417. AAAI Press, 2021.
- [194] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, 3rd edition, 2011.
- [195] S. Naito and M. Tsunoyama. Fault detection for sequential machines. In *IEEE Fault Tolerant Computer Systems*, pages 238–243. IEEE Computer Society, 1981.
- [196] N. F. M. Nasir, N. Ibrahim, M. M. Deris, and M. Z. Saringat. Test case and requirement selection using rough set theory and conditional entropy. In *Computational Intelligence in Information Systems*, pages 61–71, Cham, 2019. Springer Int. Publishing.
- [197] A. Núñez, M. G. Merayo, R. M. Hierons, and M. Núñez. Using genetic algorithms to generate test sequences for complex timed systems. *Soft Computing*, 17(2):301–315, 2013.
- [198] A. J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *«UML»'99: The Unified Modeling Language - Beyond the Standard, Second Int. Conf.*, volume 1723 of *Lecture Notes in Computer Science*, pages 416–429. Springer, 1999.
- [199] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *15th Int. Conf. on Software Engineering, ICSE'93*, pages 100–107. IEEE Computer Society / ACM Press, 1993.
- [200] H. Okamura, Y. Etani, and T. Dohi. A multi-factor software reliability model based on logistic regression. In *IEEE 21st Int. Symp. on Software Reliability Engineering, ISSRE'10*, pages 31–40. IEEE Computer Society, 2010.
- [201] H. Okamura, Y. Etani, and T. Dohi. Quantifying the effectiveness of testing efforts on software fault detection with a logit software reliability growth model. In *2011 Joint Conf of 21st Int'l Workshop on Software Measurement and the 6th Int'l Conf. on Software Process and*

- Product Measurement, IWSM/Mensura'11*, pages 62–68. IEEE Computer Society, 2011.
- [202] H. Okamura, Y. Watanabe, and T. Dohi. Estimating mixed software reliability models based on the EM algorithm. In *2002 Int. Symp. on Empirical Software Engineering (ISESE'02)*, pages 69–78. IEEE Computer Society, 2002.
- [203] A. Oliveira, R. Freitas, A. Jorge, V. Amorim, N. Moniz, A. C. R. Paiva, and P. J. Azevedo. Sequence mining for automatic generation of software tests from GUI event traces. In *Intelligent Data Engineering and Automated Learning - IDEAL'20 - 21st Int. Conf.*, volume 12490 of *Lecture Notes in Computer Science*, pages 516–523. Springer, 2020.
- [204] M. Papadakis, T. T. Chekam, and Y. Le Traon. Mutant quality indicators. In *13th Int. Workshop on Mutation Analysis, MUTATION'18, ICST Workshops*, pages 32–39. IEEE Computer Society, 2018.
- [205] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman. Mutation testing advances: An analysis and survey. In Atif M. Memon, editor, *Advances in Computers*, volume 112, pages 275 – 378. Elsevier, 2019.
- [206] R. P. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Softw. Test. Verification Reliab.*, 9(4):263–282, 1999.
- [207] K. R. Pattipati and M. G. Alexandridis. Application of heuristic search and information theory to sequential fault diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(4):872–887, 1990.
- [208] K. R. Pattipati, S. Deb, M. Dontamsetty, and A. Maitra. START: System testability analysis and research tool. In *IEEE Conference on Systems Readiness Technology, 'Advancing Mission Accomplishment'*, pages 395–402, 1990.
- [209] J. Peleska. Model-based avionic systems testing for the airbus family. In *23rd IEEE European Test Symposium, ETS'18*, pages 1–10. IEEE Computer Society, 2018.
- [210] M. Pelikan, M. Hauschild, and F. G. Lobo. Estimation of distribution algorithms. In *Springer Handbook of Computational Intelligence*, Springer Handbooks, pages 899–928. Springer, 2015.
- [211] A. Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In *4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067*, pages 196–205. Springer, 2001.

- [212] A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in EFSM testing. *IEEE Transactions on Software Engineering*, 30(1):29–42, 2004.
- [213] A. Petrenko and N. Yevtushenko. Testing from partial deterministic FSM specifications. *IEEE Transactions on Computers*, 54(9):1154–1165, 2005.
- [214] A. Petrenko, N. Yevtushenko, G. von Bochmann, and R. Dssouli. Testing in context: Framework and test derivation. *Computer Communications*, 19:1236–1249, 1996.
- [215] J. H. Poore and C. J. Trammell. Application of statistical science to testing and evaluating software intensive systems. In *Science and Engineering for Software Development: A Recognition of Harlin D. Mills Legacy*, pages 40–57, 1999.
- [216] J. H. Poore, G. H. Walton, and J. A. Whittaker. A constraint-based approach to the representation of software usage models. *Inf. Softw. Technol.*, 42(12):825–833, 2000.
- [217] S. M. Poulding and R. Feldt. Generating controllably invalid and atypical inputs for robustness testing. In *2017 IEEE Int. Conf. on Software Testing, Verification and Validation Workshops, ICST'17*, pages 81–84, 2017.
- [218] S. Reis, A. Metzger, and K. Pohl. Integration testing in software product line engineering: A model-based technique. In *10th Int. Conf. on Fundamental Approaches to Software Engineering, FASE'07*, pages 321–335. Springer, 2007.
- [219] A. Rényi. On measures of entropy and information. In *4th Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*, pages 547–561. University of California Press, 1961.
- [220] D. S. Rodrigues, M. E. Delamaro, C. G. Corrêa, and F. L. S. Nunes. Using genetic algorithms in test data generation: A critical systematic mapping. *ACM Comput. Surv.*, 51(2), 2018.
- [221] A. Rosenfeld, O. Kardashov, and O. Zang. Automation of android applications functional testing using machine learning activities classification. In *5th Int. Conf. on Mobile Software Engineering and Systems, MOBILESoft@ICSE'18*, pages 122–132. ACM, 2018.
- [222] R. H. Rosero, O. S. Gómez, and G. D. R. Rafael. 15 years of software regression testing techniques - A survey. *Int. J. Softw. Eng. Knowl. Eng.*, 26(5):675–690, 2016.

- [223] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.
- [224] R. Sagarna, A. Arcuri, and X. Yao. Estimation of distribution algorithms for testing object oriented software. In *9th IEEE Congress on Evolutionary Computation, CEC'07*, pages 438–444. IEEE Computer Society, 2007.
- [225] R. Sagarna and J. A. Lozano. Variable search space for software testing. In *Int. Conf. on Neural Networks and Signal Processing, 2003*, volume 1, pages 575–578 Vol.1, Dec 2003.
- [226] R. Sagarna and J. A. Lozano. Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms. *Eur. J. Oper. Res.*, 169(2):392–412, 2006.
- [227] A. Samarah, A. Habibi, S. Tahar, and N. N. Kharm. Automated coverage directed test generation using a cell-based genetic algorithm. In *11th Annual IEEE Int. High-Level Design Validation and Test Workshop*, pages 19–26. IEEE Computer Society, 2006.
- [228] J. Sant, A. L. Souter, and L. G. Greenwald. An exploration of statistical models for automated test case generation. *ACM SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.
- [229] R. A. Santelices and M. J. Harrold. Applying aggressive propagation-based strategies for testing changes. In *4th Int. Conf. on Software Testing, Verification and Validation, ICST'11*, pages 11–20. IEEE Computer Society Press, 2011.
- [230] S. Sarkar, G. M. Rama, and A. C. Kak. Api-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Trans. Software Eng.*, 33(1):14–32, 2007.
- [231] K. Sayre. *Improved Techniques For Software Testing Based On Markov Chain Usage Models*. PhD thesis, The University of Tennessee, Knoxville, 1999.
- [232] S. Segura, G. Fraser, A. B. Sánchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016.
- [233] S. Selvaraj and E. Choi. Survey of swarm intelligence algorithms. In *3rd Int. Conf. on Software Engineering and Information Management, ICSIM'20*, pages 69–73. ACM Press, 2020.
- [234] E. Serna M., E. Acevedo M., and A. Serna A. Integration of properties of virtual reality, artificial neural networks, and artificial intelligence

- in the automation of software tests: A review. *Journal of Software: Evolution and Process*, 31(7):2159, 2019.
- [235] M. Shafique and Y. Labiche. A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer*, 17(1):59–76, 2015.
- [236] L. Shan and H. Zhu. Testing software modelling tools using data mutation. In *2006 Int. Workshop on Automation of Software Test, AST'06*, pages 43–49. ACM, 2006.
- [237] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, 1948.
- [238] J. Shi, M. B. Cohen, and M. B. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *15th Int. Conf. on Fundamental Approaches to Software Engineering, FASE'12*, pages 270–284. Springer, 2012.
- [239] Q. Shi, Z. Chen, C. Fang, Y. Feng, and B. Xu. Measuring the diversity of a test set with distance entropy. *IEEE Trans. Reliab.*, 65(1):19–27, 2016.
- [240] A. Simão, A. Petrenko, and N. Yevtushenko. On reducing test length for FSMs with extra states. *Software Testing, Verification and Reliability*, 22(6):435–454, 2012.
- [241] J. Smith and T. C. Fogarty. Evolving software test data - ga's learn self expression. In *Evolutionary Computing, AISB Workshop*, volume 1143 of *Lecture Notes in Computer Science*, pages 137–146. Springer, 1996.
- [242] Y. Song and Y. Gong. Web service composition on iot reliability test based on cross entropy. *Comput. Intell.*, 36(4):1650–1662, 2020.
- [243] S. Splaine and S. P. Jaskiel. *The web testing handbook*. STQE Pub., 2001.
- [244] M. Srinivas and L. M. Patnaik. Genetic algorithms: A survey. *IEEE Computer*, 27:17–27, 1994.
- [245] M. Steindl and J. Mottok. Optimizing software integration by considering integration test complexity and test effort. In *10th Int. Workshop on Intelligent Solutions in Embedded Systems, WISES'12*, pages 63–68. IEEE Computer Society, 2012.
- [246] J. Strug and B. Strug. Machine learning approach in mutation testing. In *Testing Software and Systems - 24th IFIP WG 6.1 Int. Conf.*,

- ICTSS'12*, volume 7641 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2012.
- [247] A. Turlea, F. Ipate, and R. Lefticaru. A hybrid test generation approach based on extended finite state machines. In *18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC'16*, pages 173–180. IEEE, 2016.
- [248] A. Turlea, F. Ipate, and R. Lefticaru. A test suite generation approach based on EFSMs using a multi-objective genetic algorithm. In *19th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC'17*, pages 153–160. IEEE Computer Society, 2017.
- [249] A. Turlea, F. Ipate, and R. Lefticaru. Generating complex paths for testing from an EFSM. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS'18*, pages 242–249. IEEE, 2018.
- [250] S. Udeshi, X. Jiang, and S. Chattopadhyay. Callisto: Entropy-based test generation and data quality assessment for machine learning systems. In *13th IEEE Int. Conf. on Software Testing, Validation and Verification, ICST'20*, pages 448–453. IEEE, 2020.
- [251] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.
- [252] M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 4:653–665, 1973.
- [253] W. Visser. What makes killing a mutant hard. In *31st IEEE/ACM Int. Conf. on Automated Software Engineering, ASE'16*, pages 39–44. ACM Press, 2016.
- [254] D. Wang, D. Tan, and L. Liu. Particle swarm optimization algorithm: an overview. *Soft Computing*, 22:387–408, 2018.
- [255] X. Wang, S.-C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *31st Int. Conf. on Software Engineering, ICSE'09*, pages 45–55. IEEE Computer Society, 2009.
- [256] Y. Wang, M. Ü. Uyar, S. S. Batth, and M. A. Fecko. Fault masking by multiple timing faults in timed EFSM models. *Computer Networks*, 53(5):596–612, 2009.
- [257] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

- [258] J. A. Whittaker and M. G. Thomason. A markov chain model for statistical software testing. *IEEE Trans. Software Eng.*, 20(10):812–824, 1994.
- [259] J.A. Whittaker and J.H. Poore. Markov analysis of software specifications. *ACM Transactions on Software Engineering and Methodology*, 2(1):93–106, 1993.
- [260] A. Windisch, S. Wappler, and J. Wegener. Applying particle swarm optimization to software testing. In *9th Genetic and Evolutionary Computation Conference, GECCO'07*, pages 1121–1128. ACM Press, 2007.
- [261] J. K. Wolf. Born again group testing: Multiaccess communications. *IEEE Trans. Information Theory*, 31(2):185–191, 1985.
- [262] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.
- [263] M. R. Woodward and Z. A. Al-Khanjari. Testability, fault size and the domain-to-range ratio: An eternal triangle. In *12th Int. Symposium on Software Testing and Analysis, ISSTA'00*, pages 168–172. ACM Press, 2000.
- [264] L. Yang. *Entropy and software systems: towards an information-theoretic foundation of software testing*. PhD thesis, Washington State University, 2011.
- [265] L. Yang, Z. Dang, and T. R. Fischer. Information gain of black-box testing. *Formal Asp. Comput.*, 23(4):513–539, 2011.
- [266] L. Yang, Z. Dang, T. R. Fischer, M. S. Kim, and L. Tan. Entropy and software systems: towards an information-theoretic foundation of software testing. In *Workshop on Future of Software Engineering Research, FoSER'10*, pages 427–432, 2010.
- [267] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.
- [268] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transactions on Software Engineering and Methodology*, 22(3):19: 1–29, 2013.
- [269] D. Zhang, C. Nie, and B. Xu. A markov decision approach to optimize testing profile in software testing. In *9th Int. Conf. for Young Computer Scientists, ICYCS'08*, pages 1205–1210. IEEE Computer Society, 2008.

-
- [270] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang. Predictive mutation testing. In *25th Int. Symp. on Software Testing and Analysis, ISSA '16*, pages 342–353. ACM, 2016.
- [271] R. Zhao, W. Wang, Y. Song, and Z. Li. Diversity-oriented test suite generation for EFSM model. *IEEE Transactions on Reliability*, 69(2):611–631, 2020.
- [272] K. Zhou, X. Wang, G. Hou, J. Wang, and S. Ai. Software reliability test based on markov usage model. *JSW*, 7(9):2061–2068, 2012.
- [273] X. Zhu, B. Zhou, L. Hou, J. Chen, and L. Chen. An experience-based approach for test execution effort estimation. In *9th Int. Conf. for Young Computer Scientists, ICYCS'08*, pages 1193–1198. IEEE Computer Society, 2008.
- [274] F. Zhuo, B. Lowther, P. W. Oman, and J. R. Hagemester. Constructing and testing software maintainability assessment models. In *1st Int. Software Metrics Symp., METRICS'93*, pages 61–70, 1993.
- [275] R. Zuo. Information theory, information view, and software testing. In *7th Int. Conf. on Information Technology: New Generations, ITNG'10*, pages 998–1003, 2010.

