Random Testing: Theoretical Results and Practical Implications

Andrea Arcuri, *Member*, *IEEE*, Muhammad Zohaib Iqbal, *Member*, *IEEE*, and Lionel Briand, *Fellow*, *IEEE*

Abstract—A substantial amount of work has shed light on whether random testing is actually a useful testing technique. Despite its simplicity, several successful real-world applications have been reported in the literature. Although it is not going to solve all possible testing problems, random testing appears to be an essential tool in the hands of software testers. In this paper, we review and analyze the debate about random testing. Its benefits and drawbacks are discussed. Novel results addressing general questions about random testing are also presented, such as how long does random testing need, on average, to achieve testing targets (e.g., coverage), how does it scale, and how likely is it to yield similar results if we rerun it on the same testing problem (predictability). Due to its simplicity that makes the mathematical analysis of random testing tractable, we provide precise and rigorous answers to these questions. Results show that there are practical situations in which random testing is a viable option. Our theorems are backed up by simulations and we show how they can be applied to most types of software and testing criteria. In light of these results, we then assess the validity of empirical analyzes reported in the literature and derive guidelines for both practitioners and scientists.

Index Terms—Coupon collector, random testing, theory, Schur function, predictability, partition testing, adaptive random testing.

1 Introduction

Formally[1] proving that nontrivial software is correct (i.e., satisfying the intended behavior) is in general either impossible or too difficult to be done in practice. Therefore, testing is the most common software verification technique. Usually, testing software consists of the execution of some inputs and then checking whether the obtained outputs are the expected ones. Even on simple programs, there can be an extremely large number of possible inputs. For example, a routine that takes a 32-bit variable as input would have $2^{32} \approx 4$ billion possible different inputs. Exhaustive testing is hence usually infeasible. In this paper, we analyze the technique called *random testing* (RT) [24], which is one of the most used automated testing techniques in practice.

The general purpose of random testing is to generate as many test cases as possible in such a way that they help uncover as many faults or hit as many coverage targets as possible. For example, random test cases can be sampled until a desired amount of feasible branches in the system under test (SUT) are executed (i.e., branch coverage). Because test cases trigger failures and do not directly uncover faults, from a mathematical standpoint we cannot consider faults as targets. Rather, our targets will be to make specific oracles fail, thus triggering visible failures. Oracles can be assertions of various types or direct comparisons with expected outputs. We use the term "failure type" to denote a specific oracle failure. From a mathematical point of view, these two problems (i.e., triggering specific failure types and satisfying coverage criteria) are equivalent, as we will show. In particular, they are instances of the *Coupon Collector*'s problem [35], which is a well-known urn problem in probability analysis. Surprisingly, to the best of our knowledge, we have not found in the literature any work connecting random testing with the Coupon Collector's problem.

In contrast to random testing, in *partition testing* the input space is divided in subdomains. Test cases are chosen with the constraint that at least one test case is chosen from each subdomain. The division can be based on functional criteria defined using the requirements specification or based on code coverage criteria (e.g., coverage of all the control flow paths). The goal is to have "similar" test cases that are grouped together into equivalence classes, and then choose test cases from these classes. It is commonly *believed* that using this type of information *always* leads to "better" testing [36]. However, this has been proven (both theoretically and empirically) to be wrong in many conditions (e.g., see [16], [49]). In this paper, we review the empirical and theoretical results on the comparisons between random and partition testing. We also discuss how the design of a study may affect the results and conclusions.

Let S be the set of all possible test cases for a given SUT. Let T be a set of n *targets* $\{T_1, \ldots, T_n\}$, where a target T_i can be, depending on the selected test strategy, a type of failure or a particular code partition we want to cover. Each target T_i is executed by a subset of test cases with cardinality t_i. Assuming uniform sampling, it follows that a random test case covers the target T_i with probability $p_i = t_i/|S|$. If these targets are disjoint, then to cover all of these targets we will need at least n test cases.

1. A conference version of this paper was published in the 2010 ACM International Symposium on Software Testing and Analysis (ISSTA) [8].

• *The authors are with the Certus Software V&V Center, Simula Research Laboratory, PO Box 134, Lysaker 1325, Norway. E-mail: {arcuri, zohaib, briand}@simula.no.*

Manuscript received 30 Nov. 2010; revised 27 Oct. 2011; accepted 13 Nov. 2011; published online 7 Dec. 2011.
Recommended for acceptance by A. Orso and P. Tonella.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2010-11-0355.
Digital Object Identifier no. 10.1109/TSE.2011.121.

On average, how many random test cases do we need to sample to cover all feasible targets? In this paper, we formally prove nontrivial lower bounds (i.e., $\gg n$) to this value that are valid for *any* SUT. If we do not have any a priori knowledge about the SUT, then we have formally proven that those lower bounds are tight, i.e., they *cannot* be improved upon.

Scalability is an important problem that needs to be considered, as testing in the large is a critical issue in the software industry. Testing techniques that work well on small-medium size software could end up not scaling up to larger software. Large empirical studies on real-world software are not so common and this is in part due to the huge computational time that is required to carry them out. We formally study the scalability of random testing compared to testing techniques that specifically generate test cases aiming at each target independently, e.g., Genetic Algorithms that try to cover control flow branches one at a time [28]. We show that, under certain conditions, random testing can scale up much better than such alternatives and thus becomes more cost-effective on large systems. We explain the reasons for this behavior and provide practical suggestions on how to improve current testing techniques.

Some authors empirically studied whether random testing is *predictable* [13]. The motivation is to assess the extent to which random testing results (e.g., the testing targets that are covered) can be affected by chance. If we run random testing twice, are we going to obtain similar results, that is, cover similar targets? In this paper, we answer this question by providing the exact probability function that describes this stochastic process. This formal result is valid for *any* SUT and complements existing empirical studies (as in [13]) in the sense that it helps to explain their results.

The main contributions of this paper are:

- We survey and analyze the empirical and analytical results regarding the comparison of random and partition testing.
- We provide nontrivial, tight lower bounds valid for *any* SUT for the average number of test cases that must be sampled by two random testing strategies to cover predefined targets.
- We show that random testing can better scale up compared to a certain category of partition testing techniques that may be faster on smaller size software, but are unlikely to fare as well on large systems.
- We provide the exact formula that describes the predictability of two random testing runs, which can be used to compare testing techniques in the general case. We then derive a nontrivial upper bound valid for the lack of predictability for *any* SUT. The validity of existing empirical studies is assessed in light of these theoretical results. Random testing is shown to be more predictable than what was reported by previous empirical analyzes in the literature.
- From a more general standpoint, this paper proposes an initial, formal framework to help interpret empirical studies about random testing and help explain their possible discrepancies in results. Combining our survey and framework we also derive practical guidelines on using and experimenting with random testing.

The paper is organized as follows: Section 2 provides general background information regarding mathematical concepts/results that will be used throughout the paper. Section 3 describes and defines the properties of random testing. Analytical and empirical comparisons of random testing with partition testing are critically analyzed in Section 4. Section 5 presents novel theoretical results on random testing. The practical implications of our analyses and of the literature review are discussed in Section 6. Finally, Section 7 concludes the paper.

2 Background

2.1 Asymptotic Notations

In the analysis of algorithms, it is often important to study their *scalability* based on the *size* of the addressed problem [14]. How is the performance of an algorithm going to be affected by larger problem instances? A typical example is sorting algorithms, such as Merge Sort, whose runtime increases for longer arrays. In this case, the length l of the arrays defines the size of the problem.

Asymptotic analyses are useful to study the performance of algorithms for increasing size l without having to deal with all their low-level details. Asymptotic bounds are expressed only as functions of l, but they are valid only for large values of l. In particular, given the functions $f(l)$ (e.g., the performance of an algorithm, which can be an arbitrarily complex function) and a "simpler" function $g(l)$, in the literature the following notations are used:

- **Upper Bound**: $f(l) = O(g(l))$ if and only if there exists a positive real number c and a real number l_0 such that $|f(l)| \leq c|g(l)| \; \forall l > l_0$.
- **Lower Bound**: $f(l) = \Omega(g(l))$ if and only if there exists a positive real number c and a real number l_0 such that $|f(l)| \geq c|g(l)| \; \forall l > l_0$.
- **Tight Bound**: $f(l) = \Theta(g(l))$ if and only if $f(l) = \Omega(g(l))$ and $f(l) = O(g(l))$.
- **Dominated**: $f(l) = o(g(l))$ if and only if $\lim_{l \to \infty} f(l)/g(l) = 0$.

For example, given $f(l) = 5l^2 + 2l + 42$, we have $f(l) = \Theta(l^2)$, but at the same time it is also true that $f(l) = O(2^l)$ and $f(l) = \Omega(1)$. In the case of Merge Sort, we have $\Theta(l \log l)$. Notice that, often, the O notation is misused and misinterpreted in the literature. For example, it is not uncommon that an algorithm $\mathcal{A}$ with complexity $O(l)$ is wrongly claimed to be better than another algorithm $\mathcal{B}$ with complexity, for example, $O(l^2)$. The O notation is just an upper bound, which does not imply that it should be close to the lowest/tightest upper bound. In the above example, it could be that the real complexities of $\mathcal{A}$ and $\mathcal{B}$ are $\Theta(\log(l))$ and $\Theta(1)$, respectively, and so $\mathcal{B}$ would turn out to be better.

2.2 Expectation of a Random Variable

To study randomized algorithms, it is useful to represent their performance as *random variables* [35], and then study their properties, such as their expected value (i.e., arithmetic average). A random variable X can assume different values from a domain D, each one with a specific probability p_i (see [17] for more details). In this paper, we

will deal only with discrete variables. For example, if X represents the resulting value of a thrown dice, we would have $D = \{1, 2, 3, 4, 5, 6\}$ and $p_i = 1/6$ for each of the six outcomes (assuming a fair dice that is not biased). The expectation $E[X]$ of a random variable X is its mean value, which is formally defined as

$$E[X] = \sum_{x \in D} xP(X = x),$$

where $P(X = x)$ is the probability of X assuming the value x. In the case of a thrown dice, we have $E[X] = \sum_{i=1}^{6} i \times 1/6 = 3.5$.

For many problems, we do not know the distribution of X. To get information on X, we can run k experiments and collect the resulting outputs X_j. For example, if we throw a dice three times (i.e., $k = 3$), and we obtain the values (3,1,5), then $X_1 = 3$, $X_2 = 1$, and $X_3 = 5$. We can *estimate* $E[X]$ with the following formula:

$$\mu' = \frac{1}{k} \sum_{j=1}^{k} X_j.$$

Notice that, although $\lim_{k \to \infty} \mu' = E[X]$, μ' is just an estimation of the real average $E[X]$. In this paper, most of the theoretical results we present pertain to $E[X]$ and not μ'. If we want to run experiments to confirm the theory, then a large k might be required to obtain $\mu' \approx E[X]$. For example, when, in this paper, we prove an upper bound b for $E[X]$ (i.e., $E[X] \leq b$), then we may obtain $\mu' > b$ for low values of k.

2.3 Geometric Distribution

The geometric distribution is very typical in probability analysis and many phenomena can be described with it [17]. Given p the probability of an event e in a trial, we might want to know how many trials x we need on average before obtaining e. For example, if we are throwing a die and our event e is the outcome 6, we might want to know how many dice x we need to throw before obtaining a 6. This is a typical example of a geometric distribution, which assumes independence of the trials. Properties of this distribution are

$$P(X \leq x) = 1 - (1 - p)^x \ \forall x \geq 0, \tag{1}$$

$$P(X = x) = (1 - p)^{x-1} p \ \forall x \geq 1,$$

$$E[X] = 1/p.$$

Notice that $P(X = x) = 0$ for nonpositive values of x.

2.4 Coupon Collector's Problem

The Coupon Collector is a typical problem in the probability field [17], [35]. It is an urn problem in which there are n coupons/balls with different colors. The balls are randomly extracted with replacement. In other words, every time we pick up a ball, we reinsert it in the urn. The goal is to pick up each ball at least once. The probability of drawing a ball of a specific color is $1/n$.

A more general case is when there are n colors and there are more balls than n. In this case, using the notation that we will use later on in a testing context, the probability of drawing a ball with a specific color c is $p_c = t_c/|S|$, where t_c is the number of balls with color c and $|S|$ is the total number of balls in the urn. The goal would be to have a sequence of drawings in which each color appears at least once.

Let X be the random variable representing how many drawings we need to collect all the n coupons; its properties are

$$E[X] = \sum_{i=1}^{n} (-1)^{i+1} \sum_{J;|J|=i} \frac{1}{P_J}, \tag{2}$$

$$P(X \leq x) = 1 - \sum_{i=1}^{n} (-1)^{i+1} \sum_{J;|J|=i} (1 - P_J)^x,$$

where J is, in turn, every subset of $\{1, \ldots, n\}$ and $P_J = \sum_{j \in J} p_j$.

There has been a lot of work to understand the dynamics of the Coupon Collector's problem. For example, Rosén [40] studied the probability distribution of the *bonus sum* of the balls. We can add a secondary value to each ball representing its "bonus," and we might want to know the sum of these bonuses after l balls have been collected. In [40], it is proven that the sum of these bonuses follows a normal distribution. Another example is [42], in which quick-to-compute, nontrivial lower and upper bounds for $E[X]$ have been proven. Those bounds are important because computing the exact value of $E[X]$ requires an exponential number in n of arithmetic operations. So, even if we have all the values for the probabilities p_i, calculating $E[X]$ in (2) might not be possible because it is too time consuming.

A central concept that is used throughout the paper is the notion of harmonic number [35]. It is denoted H_n and is defined as $H_n = \sum_{i=1}^{n} 1/i = \log n + \Theta(1)$. When all the probabilities p_c are equal, we have $E[X] = nH_n$ (see [35]).

Several problems can be considered as instances of the Coupon Collector's problem. For example, how many dice do we need to throw on average before observing each of the six values at least once? By considering the dice outputs as coupons, the answer is $6H_6 = 14.7$.

2.5 Random Generators

In computer programs, for some types of algorithms it is necessary to use some *random* values. For example, search-based algorithms have a random component, and they are widely used in software engineering [25]. How do we obtain such random values? Using true random sources such as atmospheric noise[2] can be, in general, too expensive for most applications and large scientific studies. A more practical solution is to use deterministic mathematical formulas to generate sequences of values that look random. In these cases, we would actually be considering *pseudorandom* generators.

A typical example to generate a sequence of values v is the use of linear congruential formulas [31], such as the following:

$$v_{i+1} = (a \times v_i + c) \bmod m,$$

for some constant a, c, and m. Given a starting point v_0, this formula generates a sequence of period m (i.e., this

2. http://www.random.org/, accessed October 2010.

sequence will be repeated again every m values). Usually, the starting point v_0 (the so-called *seed*) is chosen with some "randomness" as, for example, the current time expressed in milliseconds given by the CPU clock. However, whatever seed is used, there would still be a sequence of m values that repeats itself. The seed only determines from where to start in this deterministic sequence.

A linear congruential formula is what is at the moment used in programming languages such as Java (see, for example, `java.util.Random` in Java 1.6). In particular, in Java we have $m = 2^{48}$. Although this number seems sufficient for most applications, it is unlikely to be enough for large empirical analyses. For example, consider the fact that we need to sample pairs of integer numbers to test a function that takes as input pairs of integers. Given 2^{32} values for an integer, the possible number of pairs would be 2^{64}. The random generator in the Java API would not able to sample all these possible pairs because it has a period of 2^{48}. It could only sample from a tiny fraction of the input domain, which size is $2^{48}/2^{64} = 2^{-16}$ of the input domain. This is a problem that is independent of how many random numbers are generated: The needed numbers (e.g., to cover some particular branches in the code) could all be in the large subdomain from which it is impossible to sample. Furthermore, there are also other problems such that the numbers are not sampled with uniform probabilities [33].

For these reasons, to conduct empirical studies in software engineering it would be better to use more sophisticated pseudorandom generators, such as MersenneTwister [33]. This pseudorandom generator is the default choice in statistical tools such as R [38] and search algorithm libraries such as ECJ.[3]

3 Random Testing

To test software, test cases from the input domain need to be chosen, run, and then the results need to be checked against an oracle. Testing all possible inputs is not feasible as the size of the input domain is typically arbitrarily large. Hence, only a (tiny) fraction of the possible test cases can be evaluated.

Arguably, the simplest technique to select test cases is random testing [24]. Test cases are chosen at random from the input domain based on some distributions. For verification purposes, often a uniform distribution is used to avoid biases (i.e., each test case is as likely to be sampled as any other). On the other hand, for validation purposes (reliability estimation) a distribution based on usage profiles is usually preferred [43].

When the input domain consists of numerical inputs, it is easy to uniformly choose random test cases from it. But it is not always clear how to do that when more complex types of test cases are used. One solution would be to consider their binary representation and then choose each bit with uniform probability, where k is the length in bits of a test case. However, there are three main problems:

- Elements in the binary space could not be mappable to the test case domain. For example, an enumeration of six elements that is represented with $k = 3$ bits would have $2^3 - 6 = 2$ values that are invalid. If the ratio of these invalid elements over the input domain size 2^k is low, this would not be a particular problem. Random binary sequences would be sampled, and the ones that are not valid would be sampled again. But if this ratio is high (i.e., most of the binary strings are not valid test cases), then it would be necessary to define sampling algorithms that are specific to the input domain. Designing specific algorithms that sample test cases with uniform probability can be trivial for enumerations, but it can be challenging for complex data structures (e.g., trees and graphs).
- The binary length k of the test cases can be variable. For example, in testing object-oriented software, the test cases are usually sequences of method calls [45], [18]. Theoretically, there would be no upper bound on such lengths. But constraints are common because running and analyzing too long sequences is impractical. Choosing these constraints is problem dependent [3], and a proper choice with little a priori knowledge can be difficult.
- Even when bounds on the length are used (e.g., maximum length K for binary strings), using a pure uniform distribution is unwise. There would be a total of $\sum_{i=1}^{K} 2^i = 2^{K+1} - 2$ possible test cases. Nearly half of them (2^K) would have length K, then nearly a quarter of them (2^{K-1}) would have length $K - 1$, and so on. Hence, random sequences would have very high probability of having a length that is close to the upper bound K. For example, if we choose $K = 100$, a random sequence would have a length that is at least 97 with probability $\simeq 0.9$. Sampling a sequence with length less than 10 would be extremely unlikely. To cope with this problem, the length could be first chosen with uniform probability, and then among the sequences with that length a test case can be chosen uniformly.

Even in the case of numerical inputs, it is often not wise to use a uniform probability for selection of test cases. For example, consider a function that takes as input a single 32-bit integer x. A block inside a branch such as "if(x==0)" would have an extremely low probability to be covered. Therefore, it is a very common practice in the literature to restrict the range for integer input variables, using, for example, $[-100, 100]$. However, such a restricted range could make it impossible to cover some of the testing targets, e.g., think, for example, of the case "if(x==128)." Unfortunately, we are not aware of any large empirical analysis on real-world software regarding this problem, so the impact of using constrained ranges for random testing is unclear. It could be that in most of the cases a constrained range is sufficient, or it might be exactly the opposite. As rules of thumb based on strong empirical evidence do not appear in the literature, we can only suggest using *both* unconstrained random testing as well as constraints that are common in the literature, such as $[-100, 100]$. For example, when one needs to sample an integer at random, with probability $1/2$ it could be sampled from $[-100, 100]$, and from outside that range otherwise.

3. http://cs.gmu.edu/~eclab/projects/ecj/, accessed October 2010.

Although choosing random inputs seems trivial, there are problems that need to be dealt with. They can be addressed in practice, but the result is that uniform random testing in a pure mathematical sense is either infeasible or not recommended. In general, one would want to use a distribution as close as possible to the uniform one. But, when this is not possible or when there is domain knowledge that can be exploited to introduce bias in selecting test cases that are likely to be more effective, then using different types of sampling distribution is a reasonable choice.

As described in the introduction, we have a set of testing targets $T = \{T_1, \ldots, T_n\}$. Their order is not relevant. Testing targets depend on the selected coverage criterion (e.g., paths for path coverage). The goal is to find a set of test cases that is able to cover all the targets. The probability that a *random* test case covers T_i is p_i. These probabilities are independent and do not change during the testing process. In other words, the test cases sampled so far have no effect on the sampling distribution of new test cases (i.e., the sampling is memoryless). If the sampling is done with uniform probability, then $p_i = t_i/|S|$, where t_i is the number of test cases covering T_i and $|S|$ is the total number of test cases. In this paper, the results we present are expressed as functions of p_i. The probability distribution by which test cases are randomly sampled is not important (e.g., whether uniform distribution is used or not) because each target will still have some probability p_i of being covered with a test case sampled at random regardless of the sampling distribution. We are not dealing with the problem of how to calculate the exact p_i given a particular sampling distribution. Notice that the failure triggering ability of a test case is directly dependent on its length [1], as well as its ability of covering testing targets [3]. A testing technique can hence be tailored to sample longer test sequences rather than shorter ones. However, as we just discussed, the theoretical results presented in this paper are independent of the sampling distribution.

To simplify the mathematical notation and make the paper more readable, unless otherwise stated we assume that these targets T_i are feasible, i.e., all the probabilities are strictly positive: $p_i > 0$. We will discuss and study the consequences when this property does not hold later in the paper. In particular, all the theorems in Section 5 are discussed in the context of infeasible targets. Furthermore, we do not put constraints on the input domain S, i.e., it can be either finite or infinite. If S is infinite, $p_i > 0$, and uniform sampling is used, then it follows that there will be an infinite number of test cases covering T_i. In practice, due to testing budgets, S is never infinite. For example, if the SUT takes as input a list object, then that list can be arbitrarily long. But it would not be feasible to use a list of trillions of nodes as input data. Given $p_i > 0$, the results we present in this paper are valid for both the finite and infinite cases, unless otherwise stated.

An *assumption* we make in this paper is that the targets in T are disjoint. In other words, a test case can only cover one of those T_i. This is not a particularly serious limitation: When a failure is triggered, the execution of the test case normally halts and in this case the targets (failure types) are disjoint. Coverage criteria such as path coverage yields true partitions. This is not usually the case for branch and statement coverage. At any rate, any nonpartitioning testing strategy can be transformed into one that produces true partitions (as described in [23]), but it is not necessarily easy to analyze these cases.

Regarding the probabilities p_i for coverage criteria where test cases necessarily cover at least one target, we would expect $\sum p_i = 1$ since the targets are disjoint. This is not the case when the targets are failure types since a test case may not trigger any failure. In that case, we would have $\sum p_i \ll 1$. However, this is not going to change anything in the following analytical results. We will simply assume $\sum p_i = \alpha \leq 1$. When all the probabilities p_i are equal, we have $p_i = \alpha/n = p^{\mu}$.

The following definition gives a high level pseudocode for random testing. For simplicity, we omit the details of how the found test cases are given as output to the user.

Definition 1 (RT).

while $T \neq \emptyset$
sample a new random test case $s \in S$
remove from T *the target covered by* s.

To ease the notation, to represent the random variables describing the number of test cases random testing samples before terminating, given a vector $\mathcal{P}$ of probabilities p_i for each target T_i, we use $RT(\mathcal{P})$. The values of the random variable $RT(\mathcal{P})$ depend on the SUT and the testing strategy since they determine $\mathcal{P} = \{p_1, \ldots, p_n\}$. We use $\mathcal{P}^{\mu}$ to represent the special case when all the p_i are equal.

The strategy RT is an instance of the Coupon Collector's problem (see Section 2.4). In fact, the colors are simply the testing targets we want to cover (e.g., branches if we are dealing with branch coverage). The balls are the test cases. The total number of test cases is $|S|$. The probability of covering a target c will hence depend on the number t_c of test cases that cover that target: $t_c/|S|$. If a test case does not cover any target (e.g., when the targets are failure types, then it is likely that many test cases do not trigger any failure), this can be considered as a ball with no color which is simply discarded when drawn.

Practically all the results regarding the Coupon Collector's problem can be directly applied to analyze random testing (even in the case of triggering failures where $\alpha < 1$). For example, Rosén [40] studied the probability distribution of the *bonus sum* of the balls. We can add a secondary value to each ball representing its "bonus," and we might want to know the sum of these bonuses after l balls have been collected. What is the connection with random testing? For example, we can assign a value to each oracle based on its severity (see, for example, [46], [21]) and thus evaluate the cost-effectiveness of testing. The cost-effectiveness of testing can be measured as the sum of the severity values associated with the failure types triggered by running l random test cases. The study of the distribution properties of this sum is already provided in [40] (e.g., it follows a normal distribution). It would be important to investigate if other works related to the Coupon Collector's problem would be useful to study the properties of random testing. This paper is a first

attempt to do so. Notice that if there is only one target to cover, then random testing would be simply described with a geometric distribution (see Section 2.3).

4 Comparisons with Partition Testing

In this section, we critically analyze the empirical and theoretical results regarding the comparison of random testing with partition testing. This survey, combined with the results of the next section, will enable us to draw practical conclusions regarding the usage and experimenting of random testing. An important objective of software testing activities is to find at least one test case that is able to trigger a failure (if any fault exists in the SUT). In this context, we have only *one* testing target T_i, which represents whether a failure is revealed or not.

4.1 Partition Testing

Partition testing refers to a family of testing techniques in which the input domain D is divided in $k \geq 2$ subdomains D_i. Test cases are chosen from these domains with the constraint of having at least one test case from each of them, hence $l \geq k$ (where l is the number of test cases that are selected and executed). For example, if a program takes as input one integer, the input domain of integers can be divided between negative and nonnegative values. Two test cases with values -4 and 7 could be chosen from these subdomains. When $k = 1$, partition testing is equivalent to random testing.

In the literature, there are many kinds of partition techniques [36]. These can be based on functional properties defined based on the specification and requirements. For example, each functionality of the software can be considered as a different subdomain to test. Partitions can also be based on structural criteria, as, for example, statement coverage, branch coverage, and path coverage. The test cases would be partitioned based on which code structures they execute. Other types of partitions exist as well in which, for example, the test cases are divided based on which "mutants" they kill in mutation testing [15].

The term partition testing is not mathematically correct because many of these dividing strategies in the literature do not produce disjoint partitions. For example, in statement coverage in general it would not be possible to have for each statement a test case that executes only that statement. A more appropriate term is to call such testing subdomain testing. Similar to statement coverage, branch coverage and mutation testing do not produce disjoint subdomains, whereas in path testing we would have true partitions. To be consistent with the literature, in this paper we use the term partition testing rather than subdomain testing. Notice that any subdomain division can be transformed into a disjoint partition. If two domains overlap, then their intersection can be separated and considered as a third partition.

In partition testing, the input space is divided into equivalence classes. The idea is to group together similar test cases. There should be a relation between these classes and their *failure rate*. The partitions are designed on assumptions on where and how faults can appear. Ideally, we would like to have all the failing test cases grouped in the same classes. Choosing one test case from each group would hence guarantee getting at least one test case that shows a fault (if it exists). In the literature, a partition with all failing test cases or all passing test cases is called either *homogeneous* [23] or *revealing* [50].

The failure rate θ of the SUT is the probability that a test case, randomly drawn according to some distributions (e.g., an operational profile), fails. If we assume a uniform usage of the SUT, then θ is the ratio of failing test cases (m) over the total number of possible inputs (d), i.e., $\theta = m/d$. In the case of a partition D_i, θ_i is the ratio of failing test cases in that partition (m_i) over its cardinality (d_i), i.e., $\theta_i = m_i/d_i$. A partition D_i would be homogeneous when either $\theta_i = 0$ or $\theta_i = 1$. Given l test cases, the sampling rate is $\sigma = l/d$ and, for each partition D_i, we would have $\sigma_i = l_i/d_i$.

It is quite intuitive to understand why partition testing could be an effective testing strategy. For example, in branch coverage, if there is a fault in a particular branch, then we could expect that the some/many test cases executing that branch would show a failure. The test cases that do not execute that branch (and hence belong to another partition) would all give correct outputs. If only a few test cases execute that faulty branch, there would be only a small probability that a test case chosen at random would execute it. There are a number of issues with practical applications of partition testing. Following, we discuss the most important of them.

Once a partition strategy is chosen, sampling test cases for each partition D_i is, in general, a difficult task. For example, in branch coverage there could be predicates that are difficult to satisfy. Several techniques have been designed to address this issue, for example, based on symbolic execution tools [19] and search algorithms [34]. Not only can they be computationally expensive, but they also cannot guarantee to find feasible solutions in reasonable time (constraint satisfaction is a NP-complete problem). Furthermore, some partitions could be empty, i.e., $D_i = \emptyset$. This could happen, for example, if some branches are infeasible. Proving that a branch/path is infeasible is an undecidable problem. All effort spent in sampling test cases from a $D_i = \emptyset$ would be wasted. Unfortunately, not being able to sample test cases for a partition D_i does not mean that it is empty. If no test case for D_i has been obtained, testers need to decide whether or not to spend more effort on it.

Choosing and running test cases can be expensive, but it is not the only cost. These tasks can be automated, so their cost can be reduced. However, the outputs of the execution of the test cases need to be checked for correctness. To do that, an *oracle* is required. If it is done manually (i.e., by software testers that manually check the results for each test case), then the cost of the oracle can become prohibitive for large l. In these cases, the cost of choosing and running test cases could become relatively negligible. However, oracles can be automated, for example, when a formal specification or a model of the SUT is provided. Relying on whether, for any test case, the SUT crashes (e.g., throws unexpected exceptions and segmentation faults) could also be considered an oracle. This latter option is likely, however, to miss many types of fault.

At this point, it is important to clarify the fact that random testing can be used as a technique to generate test

cases for partition testing (e.g., for path coverage). If we need to generate a test case for a partition D_i, we can sample test cases at random until we find a test case belonging to that partition. During this process, depending on the specifics of the partition strategy, we might have to generate and run several test cases to verify whether they belong to partition D_i, e.g., whether a test case covers a particular path. So, in this case what would be the difference between random testing and partition testing? Since test cases are generated at random and run in both cases, are these test techniques equivalent? There are two distinct situations to consider. In the first case, the oracle is automated and its cost is negligible compared to the cost of running the test cases. We are then in a situation where partition testing (based on random selection) and random testing are equivalent because the information about partitions is not used as oracles will systematically be evaluated. In the second case, where the cost of the oracle is significant, then partition testing with random selection is different from random testing. Assume that, given a fixed testing budget, we can only generate l test cases given the cost of oracle evaluations, e.g., manual evaluation. With random testing, we can generate and run l test cases that then will be evaluated with an oracle. For partition testing, we can use random testing to generate and run $z \gg l$ test cases until each partition is covered at least once. Then, we can use an oracle to evaluate only l of them with the constraint that each partition is covered by at least one of these l test cases. In this scenario, partition testing has a greater computational cost for test case generation (execution to determine whether a test case belongs to a partition) than random testing, but the cost of the oracle is the same.

4.2 Metrics for Comparisons

Comparing two (or more) testing strategies is not trivial. There is not (yet) an absolute and precise method to assess whether a technique is better than another. Intuitively, if a technique is able to find all the faults in less time than another technique (or this latter finds only a subset), then the former would be considered to be better. Unfortunately, in a comparison it is usually rare that a technique both subsumes the other (i.e., it finds at least the same faults) and is faster.

Random testing uses l test cases. In partition testing, there would be l_i test cases chosen for each partition D_i. To perform fairer comparisons, usually $l = \sum_{i=1}^{k} l_i$ so that both techniques use the same number of test cases.

A fault in the software can generate one or more failures. A particular failure can be generated by different faults. Since software testing triggers failures, it is more practical to consider metrics based on detected failures rather than faults. In the literature, the following metrics have been used to compare random testing against partition testing when the same number l of test cases is used:

- **E-measure**: expected number of triggered failures.
- **P-measure**: probability of finding at least one failing test case.
- **F-measure**: expected number of test cases required to trigger at least one failure.

However, comparisons based on the above metrics ignore the cost of selecting the test cases. Although it can be automated, it could still be expensive for partition testing, whereas for random testing it would be negligible. Estimating that cost is, in general, difficult and accounting for it would significantly complicate theoretical analyses. However, if it turns out that partition testing is worse or only slightly better than random testing with respect to one or more of the above metrics, that would confirm the importance of random testing because it would be more cost effective. The other way around would not necessarily mean that partition testing is better, unless the difference in the metric scores is sufficiently large. However, if the oracle is not automated, then the cost of sampling test cases from the partitions would very likely be negligible.

These metrics also ignore the distinction among the different types of faults. For example, some faults can be more important than others. Different techniques can find different sets of faults. Again, including this type of information in a metric is, in general, difficult and would significantly increase the complexity of the comparisons. For example, Ciupa et al. [13] studied what type of faults random testing can find. They compared random testing against user reports and manual testing. They found out that the nature of the faults found by these techniques is different, and none of them subsumes the others. Therefore, just counting the number of faults without accounting for their characteristics could be considered too coarse a metric.

Chen et al. [10] studied empirically and analytically the characteristics of these three measures applied to random testing. On one hand, they claimed that the P-measure and E-measure have normal sampling distributions. Mean, median, and mode are hence expected to be similar, though at the same time variance is high. Hence, large sample sizes are required to obtain reliable estimates. On the other hand, the F-measure follows a geometric distribution. This has the effect that its median value is substantially smaller than its mean value. The claim that the E-measure has "a normal sampling distribution" (in [10, page 599]) is, however, not correct. It is easy to see that the E-measure follows a *binomial* distribution $B(l, \theta)$ [17], where l is the number of test cases and θ is the probability of triggering a failure. The empirical results in [10] follows from the normal approximation of the binomial distribution [17]. In fact, a binomial variable $B(l, \theta)$ is the sum of l Bernoulli variables with parameter θ. According to the *central limit theorem*, the *sum* of l random variables converges to the normal distribution. But how many test cases l do we need to obtain a good approximation of the normal distribution? In the case of the binomial distribution, rules of thumb exist [39]:

$$l\theta > 5, \tag{3}$$

$$l(1-\theta) > 5. \tag{4}$$

Now, in the case of random testing, we would typically have low values for θ, thus satisfying (4) most of the time in real-world scenarios. What about the condition in (3)? It is important to note that $l\theta$ is actually the expected value of a binomial variable $B(l, \theta)$. That means that, to have the E-measure be approximated by a normal distribution, we need enough test cases l such that the average value of found failures is more than five.

In empirical analyses in which randomized algorithms are employed, it is a necessity to account for their randomness. For example, conclusions based on a single run of a randomized algorithm are unreliable due to the fact that often randomized algorithms have high variance and running them twice may have a high probability of producing different results. For example, the variance of a geometric distribution with parameter p would be equal to $(1-p)/p^2$ [17], [35]. It is hence essential to base scientific conclusions regarding randomized algorithms on a large enough number of runs (e.g., 100), and then use rigorous statistical methods to assess and compare their performance. Unfortunately, software engineering research rarely reports such rigorous procedures [6]. Unreliable results would have the negative effect of preventing effective technology transfer and would limit the impact of research on practice.

4.3 First Empirical Comparisons

Initially, random testing was seen as a poor testing technique that should have no space in testing practices. A random technique could not be better than a technique that exploits knowledge of the problem, as, for example, partition testing. At least this is what was thought until Duran and Ntafos empirically showed this belief was unfounded [16] by providing empirical examples in which random testing can be an effective testing strategy.

The probability P_r that random testing finds at least one failure with l test cases (i.e., P-measure) is

$$P_r = 1 - (1-\theta)^l,$$

where $(1-\theta)$ is the probability that a test case does not trigger any failure and $(1-\theta)^l$ is the probability that no test case out of l triggers any failure at all. This is the same formula of the geometric distribution presented in (1), where $p = \theta$ and $x = l$.

In the case of partition testing, the probability P_p that partition testing triggers at least one failure is

$$P_p = 1 - \prod_{i=1}^{k} (1-\theta_i)^{l_i}.$$

In their study, Duran and Ntafos analyzed the conditions for which $P_p \geq P_r$. To simplify their analysis, they made the assumption that the D_i are actual disjoint partitions, and that there is at most one failing test case in each partition. Furthermore, random selection is done with *replacement*. In other words, they assume the same test case can be chosen more than once. In general, this is not a problem, because real software testing problems have large input domains compared to the number l of used test cases. Replicated test cases therefore occur with low probability. Checking every time whether a random test case has already been sampled could have a high cost compared to the problem of having just a few replicated test cases.

Assuming that a random test case would belong to any partition D_i with same probability is too simplistic. They hence considered the probabilities p_i that a random test belongs to the partition D_i, therefore

$$P_r = 1 - (1-\theta)^l = 1 - \Big(1 - \sum_{i=1}^{k} p_i\theta_i\Big)^l,$$

$$\theta = \frac{1}{k}\sum_{i=1}^{k} \theta_i.$$

One of the experiments they carried out considered a 25 partition scheme, with $l = k = 25$ and $l_i = 1$. Different values of θ_i were considered. These were chosen at random: 2 percent of times it was $\theta_i \geq 0.98$ and, in the other cases, $\theta_i < 0.049$. The values for p_i were chosen with uniform distribution. They found that in 14 out of 50 trials $P_r \geq P_p$, i.e., random was not worse a third of the time. In other cases, there was not much difference between the two testing techniques. Since these experiments were carried out on synthetic data, they also studied random testing applied to a series of real programs. They found that random testing was able to find many (but not all) of the faults in those programs.

The results of Duran and Ntafos were considered counterintuitive. Therefore, Hamlet and Taylor [23] carried out a larger set of similar experiments. Similarly to the results in [16], they found that partition testing was better, but not by much.

Hamlet and Taylor [23] were critical of the model used by Duran and Ntafos [16] because they used an average failure rate of $\theta = 0.04$, which can be considered extremely high and unrealistic for released programs. Furthermore, a uniform distribution for p_i was considered unrealistic as well. They carried out another set of experiments in which they correlated the failure rate θ_i of the partitions with the probabilities p_i that a random test falls in that partition D_i. In the empirical analysis, they considered two types of partitions, one with low p_i (i.e., "hidden partitions") and the other with high p_i (i.e., "exposed partitions"). When the hidden partitions had high failure rate θ_i, then partition testing was superior. On the other hand, when, in the hidden partitions, the failure rates were smaller than the average θ, then random testing performed better.

4.4 Analytical Analyses

After the counterintuitive empirical results in [16], [23], Weyuker and Jeng [49] analyzed this problem in an analytical way. They used the same types of assumptions, with the additional constraint that, for each partition, $1 \leq l_i \leq d_i$. This is not particularly unrealistic because, often, $l_i = 1$ in real-world scenarios.

They analyzed the cases in which P_p is either minimized or maximized. On one hand, it is maximized when there is a partition D_i that contains only failing test cases: $\theta_i = 1$ and hence $P_p = 1$. On the other hand, P_p is minimized when $l_1 = \cdots = l_k$, $\sum_{i=1}^{k-1} d_i = l - 1$ and $d_k = d - l + 1$, where all m failing test cases belong to the partition D_k. In this particular case we would have $P_p = m/d_k = m/(d - l + 1)$.

They found that if $d_1 = \cdots = d_k$ and $l_1 = \cdots l_k$ (i.e., the partitions have all the same size, and we choose the same number of test cases from each partition), then necessarily $P_p \geq P_r$. If it also holds that $m_1 = \cdots = m_k$, then $P_p = P_r$. This is a general result which states that partition testing cannot perform worse than random testing if the partitions are of same size and we pick up the same number of test

cases from each of them. However, this result does not hold if random testing is not based on a uniform distribution, i.e., when there is at least one partition for which $p_i \neq d_i/d$. In that case, it can be worse, equal, or better. The result $P_p = P_r$ is then generalized even for the cases in which the partitions have different sizes. This happens for $\theta_1 = \cdots = \theta_k$, from which $\theta = \theta_i$ follows.

However, this latter result is based on the assumption that failure rates have precise known values. Before doing any testing, the actual failure rates are not known, and our decision on which testing strategies to use (i.e., random or partition testing) can only be based on our *expectation* $\bar{\theta}$ of the failure rate. To get more realistic results, Gutjhar [22] studied the case when the failure rates for the partitions are modeled as independent random variables $\bar{\theta}_i$. The P-measure is hence an estimation, i.e., $\bar{P}_r$ for random testing and $\bar{P}_p$ for partition testing. Under the assumption that we choose only one test case from each subdomain (i.e., $l_i = 1$), if the expected value of the failure rates is equal in each subdomain (i.e., $\bar{\theta}_1 = \cdots = \bar{\theta}_k$), then Gutjhar [22] proved the following bound:

$$\frac{\bar{\theta}}{1-(1-\bar{\theta})^k}\bar{P}_p \leq \bar{P}_r \leq \bar{P}_p.$$

If $\bar{\theta}$ is small compared to $1/k$, then the above lower bound is close to $1/k$, hence

$$\frac{1}{k}\bar{P}_p \leq \bar{P}_r \leq \bar{P}_p.$$

The result of Weyuker and Jeng [49] is a special case in which the variance of the random variables is zero. Gutjhar's result [22] tells us that partition testing can be up to k times better than random testing (where k is the number of subdomains). This upper bound would be reached (or got close to) when the partitions are homogeneous or there are many small subdomains and a few large ones. This latter case is actually very common in practice. For example, in branch coverage, equality predicates would generate very unbalanced partitions. Therefore, Gutjhar [22] concludes that the earlier studies [16], [23] did not show a clear superiority of partition testing because the partitions had comparable sizes d_i.

Based on the analytical results in [49], Chen and Yu [11] generalized the condition for which $P_p \geq P_r$. Under the same assumptions but different sizes d_i for the partitions and different l_i, they proved that it is sufficient to have $\sigma_1 = \cdots = \sigma_k$ to guarantee $P_p \geq P_r$. In other words, if we choose the number l_i of test cases from each partition proportionally to its cardinality d_i, then we cannot perform worse than random testing.

Boland et al. [9] used the mathematical technique called majorization and Schur functions to further generalize these results. Let θ^m be the average value of the fault rates θ_i, i.e., $\theta^m = \sum_{i=1}^{k} \theta_i/k$. In [49], the condition to get $P_p \geq P_r$ is that all the failure rates θ_i are equal. Under the same assumptions, Boland et al. [9] generalized this result by proving that it is just sufficient that $\theta^m \geq \theta$ to guarantee $P_p \geq P_r$. Note that, if all the fault rates θ_i are equal, then $\theta^m = \theta_i = \theta$. In a similar way, they generalize the results of Gutjhar [22] regarding failure rates modeled as independent random variables. Under the same assumptions, $\bar{\theta^m} \geq \bar{\theta}$ is sufficient to guarantee $\bar{P}_p \geq \bar{P}_r$.

All the results described so far are based on the P-measure (see Section 4.2). This is a coarse measure because it does not take into account how many faults a testing technique is able to find. Chen and Yu [12] extended their analyses to the cases in which the expected number of failures detected (E-measure) is used to compare random testing against partition testing. They still considered uniform random selection with replacement, but they also considered the cases in which the subdomains are not disjoint.

The expected number E_r of failures detected for random testing is

$$E_r = \frac{ml}{d} = l\theta,$$

whereas for partition testing it is

$$E_p = \sum_{i=1}^{k} \frac{m_i l_i}{d_i} = \sum_{i=1}^{k} l_i \theta_i.$$

When the subdomains are not actual partitions (i.e., they are not disjoint), to be consistent with the notation in [12] we use P_s for the P-measure and E_s for the E-measure.

When for all partitions we have $d_i \geq l_i \geq 1$, if $l > m$, then E_p is maximized when all partitions with failing test cases have $m_i = l_i = d_i$. In this case, we obtain $E_p = m$. On the other hand, if $l \leq m < d$, then E_p is maximized when for all partitions but one we have $m_i = l_i = d_i$, and only one test case is taken ($l_j = 1$) from the only partition D_j with nonfailing inputs ($m_j < d_j$). In this case:

$$E_p = l - \frac{d-m}{d-l+1}.$$

Similarly with P_p, E_p is maximized when partitions are homogeneous. The conditions for which E_p is minimized are the same as for P_p [12].

When the partitions are not disjoint and $m > 0$, E_s is maximized when one partition is the entire input space and only one test case is chosen from it. All the other partitions should consist of only failing inputs. In this case, $E_s = l - (d-m)/d$. On the other hand, for $m < d$, the worst value for E_s is obtained when one partition is the entire input space and only one test case is chosen from it, but all the other partitions should consist of only nonfailing inputs. In this case, $E_s = m/d$.

When comparing partition testing against random testing, the expected number of failures triggered is equal ($E_p = E_r$) when a proportional strategy is used: $\sigma_1 = \cdots = \sigma_k$. The two measures are also the same when the failure rates of the partitions are all the same: $\theta_1 = \cdots = \theta_k$. The condition for which $E_p \geq E_r$ is when either $(\theta_i - \theta_j)(\sigma_i - \sigma_j) \geq 0$ for all pairs of partitions or $(\theta_i - \theta)(\sigma_i - \sigma) \geq 0$ for all partitions. Similarly, we have $E_p \leq E_r$ when either $(\theta_i - \theta)(\sigma_i - \sigma_j) \leq 0$ or $(\theta_i - \theta)(\sigma_i - \sigma) \leq 0$. In particular:

$$E_p - E_r = \sum_{i=1}^{k} d_i(\theta_i - \theta)(\sigma_i - \sigma). \tag{5}$$

These results formally prove and quantify the intuition that partition testing performs better when the sampling is

more concentrated in the partitions with greater failure rates. However, more interestingly, it performs worse than random testing otherwise.

Regarding nondisjoint subdomains, Chen and Yu [12] proved that it is sufficient to have $\sigma_1 = \cdots = \sigma_k$ to guarantee $E_s \geq E_r$. In other words, for the E-measure, partition testing with proportional sampling cannot be worse than random testing, regardless of whether the subdomains are disjoint or not.

The P-measure is quite different from the E-measure, but there are correlations between the two. In [12], it was proven:

- For any given program, $P_r \leq E_r = l(1 - (1 - P_r)^{1/l})$.
- For any given program and partition testing strategy, $P_s \leq E_s \leq l(1 - (1 - P_s)^{1/l})$.
- If $E_s \geq E_r$, then $P_s \geq P_r$. Equality holds only if for all partitions we have $\theta_i = \theta$.
- If $P_s < P_r$, then $E_s < E_r$. However, the converse is not necessarily true.

If the objective is to show that some partition testing technique works better than random testing, then the E-measure is more useful than the P-measure. If one partition testing technique has better value for the E-measure than random testing, then it will also necessarily be the case for the P-measure.

In conclusion, the results above show that there are potential situations in which random testing is proven to be more cost-effective. Whether such situations are frequently encountered in practice remains to be investigated.

4.5 Real-World Results

To assess whether a testing technique is actually effective, empirical analyses are necessary on real-world software containing real faults. In fact, theoretical analyses are usually based on a series of assumptions to make them tractable. These assumptions might not be valid in real software. For example, all the theoretical comparisons in Section 4 between partition and random testing ignore, out of necessity to make the mathematical formulas tractable, the additional computational cost of partition testing. Furthermore, simulations on synthetic data have similar limitations. Important features (from the point of view of testing) could be missing or wrongly examined due to an incomplete understanding of the problem. For example, this is what happened in Adaptive Random Testing (see [4]), where the high cost of distance calculations was mostly ignored in the empirical investigations based only on simulations. Another example is Combinatorial Integration Testing, where if one considers the practical aspects of the computational cost of generating test suites and the size of real-world industrial configuration spaces, then there are several practical situations in which random testing is a viable and recommendable alternative [5]. For these reasons, in this section we report some applications of random testing in real-world contexts, as they show that there exist some real-world scenarios in which random testing is useful. As for any testing technique, the practitioners need to know its properties to make an informed decision on when to apply it because no testing technique is always going to be the best option in all contexts. However, notice that often random testing is used with some slight variations. Depending on the magnitude of the variations, the theoretical results discussed in this paper might not directly apply to these variants.

Empirical analyses on real data can also be difficult for several reasons. Obtaining the data from industry is not always possible, and open source projects may not be representative or the quality of the data in their repository may be questionable. Using and developing testing tools that work on real-world software can be challenging. Running industrial case studies can be very time consuming and, as a result, very few of them are reported in the literature. Regardless of the size of the case study, conclusions, as for any empirical study, can be hard to generalize. Therefore, empirical studies on real-world software, though of the utmost importance, are only one among several ways we need to use to investigate testing techniques and develop a reliable body of knowledge.

Groce et al. [20] analyzed the use of random testing to test software used in space missions. In particular, they tested a complex implementation of a flash file system. They reported the results of testing done for 25 weeks. Different types of faults were found and corrected during the first 20 weeks. At the date of the paper, the longest continuous run of successful tests consisted of more than 3.5 billion random operations. In this context, random testing was very useful because an oracle could be automatically derived. Billions of operations on the SUT were automatically checked. Other file systems such as Solaris and EXT3 in Linux were used as reference implementations. Groce et al. [20] tried to use more sophisticated techniques than random testing, as, for example, model checking. But these simply did not scale to the actual software (written in C) that had to be released.

Random testing can generate long sequences of operations/inputs that reveal failures. But these could be so long that no software tester would analyze them and locate where the faults are. A test case that is too difficult to understand is of little interest. Fortunately, these sequences have many operations that do not affect fault detection. A postprocessing can be employed to remove these useless operations. For example, we can remove one operation at a time as long as the failure is still manifested by the reduced sequence. Other more sophisticated techniques can be used to minimize sequences, as it was done in [20]. In their experiments, it was possible, on average, to reduce by 92.5 percent the length of the revealing test sequences.

Pacheco et al. [37] applied random testing on 14 widely used object-oriented libraries, for a total of roughly 780,000 lines of code. They start from an empty test sequence, and then they add function calls one at a time. The system is not completely tested at random because simple heuristics are used to narrow down the domain from which the function calls are chosen and to prevent the inclusion of useless function calls. Contract assertions in the SUT were used as oracles. Pacheco et al. [37] found that random testing scaled to this large case study, and it had better fault detection and coverage than the formal techniques they compared it to.

In [7], [30], we introduced a methodology to automatically test embedded systems based on environment models. The main motivation was to support black-box testing by

independent testing teams. Environment models modeled with a combination of UML, MARTE (UML extension for embedded, real-time software), and OCL (constraint language which is part of UML) were used to automatically generate environment simulators that interact with the SUT. Each test case was a specific simulation setting. The simulations were very complex because each test case involved the execution of hundreds of thousands of lines of code for several seconds, many threads running in parallel, communications through TCP sockets, and write/read accesses to the file system. Error states were part of the environment models in order to generate automated oracles. Random testing was successfully used to find simulation settings for which these error states were reached, and therefore revealed failures in the SUT. In other words, by using random testing, new critical faults were automatically found in the production code of an industrial real-time, embedded system.

In [41], Sharma et al. analyzed the application of random testing on a common benchmark used in the literature of test data generation for object-oriented software, i.e., "container classes." They compared random testing with the most advanced systematic testing technique for that kind of software. The results of that empirical analysis show that random testing fared as good as that systematic technique.

5 Novel Theoretical Results on Random Testing

The literature review we presented in this paper shows that there has been a lot of work aimed at understanding when random testing could be a better option than partition testing. However, there are several important properties of random testing that have received only a little attention in the literature. In this section, we present novel theoretical results regarding some properties of random testing that have not been formally studied before. Some of these results were first presented in [8], and they are on four main properties: general *lower bounds* on the execution time (Section 5.1), *comparisons* of algorithms (Section 5.2), *scalability* (Section 5.3), and finally, *predictability* (Section 5.4) of random testing.

5.1 Effectiveness of Random Testing

One way to define the *effectiveness* of random testing is as the expected number of test cases that are required to cover all the feasible targets in T. Because random testing is a stochastic process, this value will be a random variable.

Because random testing is an instance of the Coupon Collector's problem (see Section 3), it follows that its expected value is

$$E[RT(\mathcal{P})] = \sum_{i=1}^{n} (-1)^{i+1} \sum_{J;|J|=i} \frac{1}{P_J}, \tag{6}$$

where J is a subset of $\{1, \ldots, n\}$ and $P_J = \sum_{j \in J} p_j$.

This equation precisely describes the effectiveness of random testing when we know the probabilities p_i. But what about if we need to test an SUT on which we cannot make any assumption on the values p_i? A trivial lower bound to the expectation E is n because we need at least n different test cases to cover n different disjoint targets. The following theorem provides the answer to this research question by providing a lower bound that is higher than the trivial n and that is tight (no higher, lower bound exists that is valid for all SUTs).

Theorem 1. *For any SUT and n testing targets T_i, the expected value of $RT(\mathcal{P})$ to cover all the targets T_i is lower bounded by nH_n/α: $E[RT(\mathcal{P})] \geq nH_n/\alpha$. When all the probabilities are equal ($\mathcal{P}^\mu$), then $E[RT(\mathcal{P}^\mu)] = nH_n/\alpha$.*

Proof. The proof comes from the literature of the Coupon Collector's problem. In fact, $E[RT(\mathcal{P})]$ is a Schur-convex function which has minimal value when all the probabilities are equal, i.e., for $\mathcal{P} = \mathcal{P}^\mu$ (see, for example, [29], [42]). When $\alpha = 1$, i.e., $p^\mu = 1/n$, then the expected value for the Coupon Collector's problem is simply nH_n [35]. Using (6), it follows that for the general case ($\alpha \leq 1$) we have $E[RT(\mathcal{P}^\mu)] = nH_n/\alpha$.

Another easier way to look at it would be to follow the proof in [35]. X_i being the random variable representing the number of trials in the ith epoch in which we have collected i different coupons so far, we have

$$RT(\mathcal{P}^\mu) = \sum_{i=0}^{n-1} X_i.$$

The probability of drawing one of the remaining $n - i$ coupons is

$$q_i = \alpha \frac{n-i}{n}.$$

Because X_i is geometrically distributed, therefore we have

$$\begin{aligned} E[RT(\mathcal{P}^\mu)] &= E\left[\sum_{i=0}^{n-1} X_i\right] \\ &= \sum_{i=0}^{n-1} E[X_i] \\ &= \sum_{i=0}^{n-1} \frac{n}{\alpha(n-i)} \\ &= \frac{n}{\alpha} \sum_{i=0}^{n} \frac{1}{i} \\ &= \frac{nH_n}{\alpha}. \end{aligned}$$

□

Theorem 1 states that there are testing problems that can be solved by random testing in $\Theta(n \log n)$, on average, when $\mathcal{P} = \mathcal{P}^\mu$. However, the expected number of sampled test cases in real-world testing problems would be in general much higher than these lower bounds as $\mathcal{P}$ might significantly depart from $\mathcal{P}^\mu$. Notice that Theorem 1 is also valid in the presence of infeasible targets.

Because it will be useful for the discussion and proofs in the next sections (e.g., Theorem 3 in Section 5.3), let us define another type of random testing in which each target is naively sought in a specific order ($RT0$). The following definition gives its high-level pseudocode.

Definition 2 (RT0).

while $T \neq \emptyset$
  choose and remove T_i *from* T
  while T_i *is not covered*
    sample a new random test case $s \in S$.

A lower bound for the effectiveness of this random testing variant is given by the following theorem.

Theorem 2. *For any SUT and* n *testing targets* T_i*, the expected value of* $RT0(\mathcal{P})$ *to cover all the targets* T_i *is lower bounded by* n^2/α*:* $E[RT0(\mathcal{P})] \geq n^2/\alpha$*, where* α *is the sum of probabilities in* $\mathcal{P}$*. When all the probabilities are equal* ($\mathcal{P}^\mu$)*, then* $E[RT0(\mathcal{P}^\mu)] = n^2/\alpha$.

Proof. Covering a particular target T_i follows a geometric distribution with parameter p_i (see [10], [17]). Therefore, for the strategy $RT0$ we obtain the following expected number of test cases:

$$E[RT0(\mathcal{P})] = \sum_{i=1}^{n} \frac{1}{p_i}. \tag{7}$$

When all the probabilities p_i are equal, from (7) it follows that the expected time of $RT0(\mathcal{P}^\mu)$ is

$$E[RT0(\mathcal{P}^\mu)] = \sum_{i=1}^{n} \frac{1}{\alpha/n} = \frac{n}{\alpha} \sum_{i=1}^{n} 1 = \frac{n^2}{\alpha}.$$

We hence just need to prove that $E[RT0(\mathcal{P}^\mu)]$ is the lowest possible value for $E[RT0(\mathcal{P})]$. This could be done by proving that $E[RT0(\mathcal{P})]$ is a Schur-convex function (see [9] for a description and application in software testing). For the sake of clarity and readability, we prefer to prove it in a (longer) way that we believe is easier to understand.

Let us define any generic $\mathcal{P}^g \neq \mathcal{P}^\mu$ in terms of $\mathcal{P}^\mu$. In particular, we would have $p_i = p^\mu + z_i$, where $-1 \leq z_i \leq 1$. It easily follows $\sum_{i=1}^{n} z_i = 0$ and $\exists i : z_i \neq 0$. Let us divide the n variables z_i in three separated sets, where $a_i \in A$ represents the positive values (i.e, $z_i > 0$), $b_i \in B$ the negative ones, and finally, $c_i \in C$ represents $z_i = 0$. It follows $\sum_{a_i \in A} a_i = \sum_{b_i \in B} -b_i$. We can hence write the expected time of $RT0(\mathcal{P}^g)$ as

$$\begin{aligned} E[RT0(\mathcal{P}^g)] &= \sum_{i=0}^{n} \frac{1}{p^\mu + z_i} \\ &= \sum_{a_i \in A} \frac{1}{p^\mu + a_i} + \sum_{b_i \in B} \frac{1}{p^\mu + b_i} + \sum_{c_i \in C} \frac{1}{p^\mu}. \end{aligned}$$

In a similar way, we can write

$$\begin{aligned} E[RT0(\mathcal{P}^\mu)] &= \sum_{i=0}^{n} \frac{1}{p^\mu} \\ &= \sum_{a_i \in A} \frac{1}{p^\mu} + \sum_{b_i \in B} \frac{1}{p^\mu} + \sum_{c_i \in C} \frac{1}{p^\mu}. \end{aligned}$$

To prove $E[RT0(\mathcal{P}^g)] > E[RT0(\mathcal{P}^\mu)]$, hence we just need to prove

$$\sum_{a_i \in A} \frac{1}{p^\mu + a_i} + \sum_{b_i \in B} \frac{1}{p^\mu + b_i} > \sum_{a_i \in A} \frac{1}{p^\mu} + \sum_{b_i \in B} \frac{1}{p^\mu},$$

which can be rewritten as

$$\sum_{b_i \in B} \frac{1}{p^\mu + b_i} - \sum_{b_i \in B} \frac{1}{p^\mu} > \sum_{a_i \in A} \frac{1}{p^\mu} - \sum_{a_i \in A} \frac{1}{p^\mu + a_i}.$$

This relation is verified by

$$\begin{aligned} \sum_{b_i \in B} \frac{1}{p^\mu + b_i} - \sum_{b_i \in B} \frac{1}{p^\mu} &= \sum_{b_i \in B} \frac{-b_i}{p^\mu (p^\mu + b_i)} \\ &> \sum_{b_i \in B} \frac{-b_i}{(p^\mu)^2} \\ &= \frac{1}{(p^\mu)^2} \sum_{b_i \in B} -b_i \\ &= \frac{1}{(p^\mu)^2} \sum_{a_i \in A} a_i \\ &> \sum_{a_i \in A} \frac{a_i}{p^\mu (p^\mu + a_i)} \\ &= \sum_{a_i \in A} \frac{1}{p^\mu} - \sum_{a_i \in A} \frac{1}{p^\mu + a_i}. \end{aligned}$$

□

5.2 Comparison of Random Strategies

When a new testing problem is addressed, or when a new testing algorithm is designed on a known testing problem, often it is common practice to compare the effectiveness of any new algorithm against random testing. In fact, random testing is often used as a baseline for algorithm comparisons as it usually represents the easiest and cheapest (e.g., in terms of implementation time) option practitioners would take if no automated testing tool is available.

The naive technique $RT0$ that we described in Section 5.1 is clearly worse than RT. Addressing a set of testing targets in a specific order cannot be better than allowing any order in which they can be solved. So, algorithm comparisons should always be against RT and not $RT0$. However, in the literature there are several cases in which empirical studies use $RT0$ rather than RT (e.g., [52], [28], [26], [47]).

But how much worse is $RT0$? We analyzed this research question in details in [8]. That analysis shows that there are some cases in which the difference between the two random testing strategies are negligible, but there are cases in which RT can be up to n/H_n times faster than $RT0$ (this happens for $\mathcal{P}^\mu$). For example, if $n = 100$, then RT could be up to 19.27 times faster. Answering this research question is important because it shows that certain published results may be due to the use of $RT0$, though the novel proposed techniques may still fare better than RT.

5.3 Scalability

The most important challenge in the software industry lies in the testing of large (sub)systems and components. It is therefore important for test techniques to be scalable. Assuming an automated oracle (or if the testing targets are related to code/specification coverage), one motivation for random testing is its ability to generate large numbers of test cases and therefore exercise large systems with extremely large test suites. One important question in the

context of this paper is to determine how scalable random testing is compared to alternative testing techniques. Some testing techniques may work well on smaller systems but do not necessarily scale up well to larger software. This has been proven, for example, for some local search techniques applied to structural coverage [2]. In this context, at a high level, scalability captures the relationship between the number of targets and the expected number of test cases required to cover all of these targets.

Let us assume a testing technique called V that is applied on each of the n targets. The specifics of the technique is not important (e.g., Symbolic Execution or Genetic Algorithms), as long as it focuses on one target at a time. For each target T_i, we are assuming that V is faster than random testing by a constant factor $c_i \leq c_M$, where c_M is a constant. If the expected number of test cases of random testing on a target T_i is $1/p_i$, then on the same target the expected time of V is $1/(c_i p_i)$. Without any a priori knowledge of the SUT, this is a reasonable assumption in which the cost of a testing technique V is directly proportional to how many test cases cover each target.

Which testing technique scales better? Random testing or V? To answer this research question, we first need to formally define under which scalability model we want to do such an analysis. Assuming n feasible targets, we consider the case of $z = kn$ feasible targets, where $k \geq 1$ is the scalability factor. We want to study the expected performance of V and random testing to cover all the z targets based on the scalability factor k when we know their performance for $k = 1$. When we consider a higher number of targets because they are disjoint, then we need to guarantee that each associated probability p_i^k is still a valid value. In particular, given $\sum_{i=1}^{n} p_i^{k=1} = \alpha$, then in our scalability model we assume α to be constant and only consider the case $\sum_{i=1}^{nk} p_i^{k>1} = \alpha$.

When we consider larger software (i.e., $k > 1$), we need to define how the different probabilities $p_i^{k>1}$ should be considered. In our scalability model, we use only one restriction in addition to $\sum_{i=1}^{nk} p_i^{k>1} = \alpha$. Given $p_d^{k=1}$, the probability of the most difficult target for $k = 1$ (i.e., the lowest value), then, for all values of k, the most difficult target for the case $k > 1$ should have value $p_d^{k>1} \leq p_d^{k=1}/g(k)$ for some function $g(k)$, e.g., $g(k) = k$. In other words, the most difficult target at scalability factor k should be at most $g(k)$ times more difficult than when $k = 1$. In this scalability model, it is just important to consider the most difficult target, the properties of the other targets being negligible. Under these assumptions, we can prove the following theorem:

Theorem 3. *Under the scalability model described in Section 5.3, if $g(k) = o(k^2/\log(k))$, then the expected runtime (based on the scalability factor k) $E[RT^k]$ is dominated by the expected runtime $E[V^k]$: $E[RT^k] = o(E[V^k])$. In other words, random testing scales better than V for larger k.*

Proof. To prove this theorem, we need to prove that, when $g(k) = o(k^2/\log(k))$, then

$$\lim_{k\to\infty} \frac{E[RT^k]}{E[V^k]} = 0.$$

We will first prove an upper bound for $E[RT^k]$ and then a lower bound for $E[V^k]$. With these bounds, we can prove an upper bound for $E[RT^k]/E[V^k]$ that converges to 0 for k that tends to infinite, which would prove this theorem.

Using Theorem 1 and the theory of the Coupon's Collector problem, an upper bound for $E[RT^k]$ can be constructed by considering all the probabilities p_i in their worst case, i.e., $p_i = p_d/g(k)$:

$$\begin{aligned} E[RT^k] &\leq \frac{1}{\sum_{i=1}^{z} \frac{p_d}{g(k)}} z H_z \\ &= \frac{g(k)}{z p_d} z(\log(z) + \Theta(1)) \\ &= \frac{g(k)}{p_d}(\log(k) + \log(n) + \Theta(1)) \\ &= \Theta(g(k) \times \log(k)). \end{aligned}$$

Using the same procedure (and Theorem 2), we can construct a lower bound for $E[V^k]$:

$$\begin{aligned} E[V^k] &= \sum_{i=1}^{z} \frac{E[RT0_i]}{c_i} \\ &\geq \frac{1}{c_M} \sum_{i=1}^{z} E[RT0_i] \\ &\geq \frac{1}{c_M} \sum_{i=1}^{z} \frac{z}{\alpha} \\ &= \frac{z^2}{c_M \alpha} \\ &= \frac{n^2}{c_M \alpha} k^2 \\ &= \Theta(k^2). \end{aligned}$$

Finally, if $g(k) = o(k^2/\log(k))$, then

$$\lim_{k\to\infty} \frac{E[RT^k]}{E[V^k]} \leq \lim_{k\to\infty} \frac{\Theta(g(k) \times \log(k))}{\Theta(k^2)} = 0.$$

□

Notice that $g(k) = k$ and $g(k) = k\log(k)$ satisfy $g(k) = o(k^2/\log(k))$, whereas $g(k) = k^2$ does not (for the definition of o, please recall Section 2.1). Whether such a requirement is realistic in practice should be the focus of empirical studies. Furthermore, it is important to note that $g(k) = o(k^2/\log(k))$ is just a sufficient condition, but it might not be necessary. In other words, that condition might be stricter than necessary.

How can we explain the better scalability of random testing compared to V? The reason is that random testing intrinsically works as a *parallel search*, in which all the targets are considered at the same time, whereas V works as a *sequential search*. The computational power used to cover a specific target is not employed to help covering other remaining targets. However, for a low number of targets, random testing can be slow because it samples test cases at random, and some targets can be particularly difficult. But with a larger number of targets z, its parallel nature eventually pays off, yielding better results than any V.

The above results provide support to the fact that testing techniques should not be of the V type, otherwise even a "naive" technique like random testing could scale better. To

improve upon testing techniques of the V type, we need to transform them in a form of *parallel search*, and some examples are already present in the literature. For example, in the case of *search algorithms* [34], multi-objective fitness functions that consider more targets at the same time can be designed (see [27], [18]). In a similar way, proper *seeding strategies* (e.g., initialization of the population in Genetic Algorithms) can help search techniques (see, for example, [51]). In the context of software testing, once some of the targets are covered, this implies reusing the test cases from previous attempts to initialize the population used by certain search algorithms to cover the remaining targets (some basic examples can be found in [48]).

Notice that in this section we have assumed we have only feasible targets. However, it is easy to see that, in case of infeasible targets, random testing would be further advantaged compared to V.

5.4 Predictability of Two Runs

When using random testing, how much do we rely on chance? In other words, how predictable are the results of random testing? One way to rephrase this is: How similar are the results of two random testing runs likely to be? More precisely, running random testing for l test cases we would cover some of the targets in T. If we run random testing again, how likely are we to cover the same targets? Or will they likely be completely different? This research question was empirically addressed by Ciupa et al. [13]. They ran experiments on a set of common library programs written in Eiffel. Given different assertions in the SUT, the goal was to find test cases for which different failure types occur. Their conclusion was that random testing appears "rather unpredictable in terms of the actual detected faults."

In this section, we provide the mathematical formula that describes the stochastic process of the predictability of random testing. We formally prove that Ciupa's et al. statement [13] only describe a particular scenario that may not generalize to most situations.

Let us first define the difference $\mathcal{D}$ among two runs of random testing. We can represent the output of random testing as a binary vector v of length n. The binary value $v[i]$ represents whether the target T_i has been covered ($v[i] = 1$) or not ($v[i] = 0$). The difference between two runs that produce v_1 and v_2 is hence defined by

$$\mathcal{D}(v_1, v_2) = \sum_{i=1}^{n} |v_1[i] - v_2[i]|.$$

Obviously, it holds that $0 \leq \mathcal{D}(v_1, v_2) \leq n$. Because the vectors v are randomly generated by random testing, we can consider $\mathcal{D}$ as random variable with the vector of probabilities p_i as parameter. In other words, given a particular SUT with specific probabilities p_i for the testing targets in T, we want to study the distribution of the random variable $\mathcal{D}$. In particular, we are interested in its expected value $E[\mathcal{D}]$.

This random variable $\mathcal{D}$ is dependent on the number l of sampled test cases. Obvious properties are

$$E[\mathcal{D}_{l=0}] = 0,$$
$$E[\mathcal{D}_{l\to\infty}] = 0.$$

In fact, if no test case is sampled (i.e., $l = 0$), then v_1 and v_2 necessarily contain only 0 values. If we run random testing for an infinite amount of time (i.e., $l \to \infty$), then all targets would be covered, so no difference between v_1 and v_2. But what is the expected value of $\mathcal{D}$ for a generic number of test cases l and SUT? This research question is answered by the following theorem:

Theorem 4. *The expected value of the random variable $\mathcal{D}$ is equal to*

$$E[\mathcal{D}] = \sum_{i=1}^{n} 2(1-p_i)^l(1-(1-p_i)^l).$$

Proof. Let us consider the expected value of $\mathcal{D}_i = |v_1[i] - v_2[i]|$, which is

$$\begin{aligned} E[\mathcal{D}_i] &= 0P(\mathcal{D}_i = 0) + 1P(\mathcal{D}_i = 1) \\ &= P(\mathcal{D}_i = 1) \\ &= P(v_1[i] = 0 \ \wedge \ v_2[i] = 1) \\ &\quad + P(v_1[i] = 1 \ \wedge \ v_2[i] = 0) \\ &= 2P(v_1[i] = 0 \ \wedge \ v_2[i] = 1) \\ &= 2P(v_1[i] = 0)P(v_2[i] = 1) \\ &= 2(1-p_i)^l(1-(1-p_i)^l). \end{aligned}$$

We can hence conclude the proof with

$$E[\mathcal{D}] = \sum_{i=1}^{n} E[\mathcal{D}_i] = \sum_{i=1}^{n} 2(1-p_i)^l(1-(1-p_i)^l).$$

□

To support the theoretical results in this section, we carried out a series of simulations with different vectors $\mathcal{P}$. Each vector is defined by the length n of targets. For reasons of computational time and readability, we consider only three different vectors, $\mathcal{P}_0$, $\mathcal{P}_1$, and $\mathcal{P}_2$. In all cases, for simplicity we chose $\alpha = 1$ and $n = 32$. Notice that our theoretical results are valid for any value of α (the sum of the probabilities p_i) and n (the number of targets).

The first vector is simply $\mathcal{P}_0 = \mathcal{P}^{\mu}$. In the second vector, we consider half of the probabilities (i.e., $n/2$) to be equal to $1/(10n)$. In other words, $p_1 = p_2 = \cdots = p_{n/2} = 1/(10n)$. The remaining probabilities are chosen such that their total sum for $\mathcal{P}_1$ is equal to 1. Finally, for $\mathcal{P}_2$ we consider a geometric allocation. In particular, $p_1 = 1/2$ and $p_{i+1} = (1/2)p_i$, where, however, $p_n = 1 - \sum_{i=1}^{n-1} p_i$ (so all probabilities sum up to $\alpha = 1$).

The vector $\mathcal{P}_0$ is important because it represents, as we will show, a "boundary condition" point. On the other hand, $\mathcal{P}_1$ represents a situation in which there are two groups: easy and difficult targets. All the probabilities in the same group are equal. Finally, in $\mathcal{P}_2$ we have several types of targets, increasing from very trivial (i.e., $p_1 = 1/2$) to very difficult ones (e.g., $p_{n-1} = 1/2^{n-1}$). All these probabilities are different from each other.

In our simulations, we need to define a stopping criterion for random testing for various sample sizes of test suites. Sample sizes will cover the following range: $l = 2^i$ where $i \in \{0, \ldots, 10\}$.

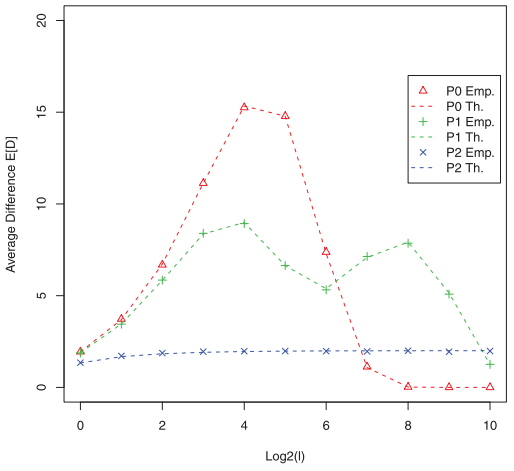

Fig. 1. Predictability of random testing.

In Fig. 1, we plot the expected value of $E[\mathcal{D}]$ for the previously described vectors and the average number of test cases obtained by 1,000 repeated experiments with different random seeds (for a total of $3 \times 11 \times 1{,}000 = 33{,}000$ experiments). The first observation is that the simulation confirms Theorem 4. Second, the results seem to suggest that the further away $\mathcal{P}$ is from the uniform distribution, the more predictable it is. However, this statement is only valid for a certain range of the number of sampled test cases. Eventually, when this number becomes large enough, as expected, all targets are covered and random testing is fully predictable. Since in practice we expect $\mathcal{P}$ to look more like $\mathcal{P}_2$ than $\mathcal{P}_0$, random testing should be predictable in most cases. For example, typically, certain faults or blocks are much harder to uncover or reach than others.

Theorem 4 gives the formula that describes how to calculate $E[\mathcal{D}]$. In Fig. 1, we can see that $E[\mathcal{D}]$ can assume a wide range of values. Because $0 \leq \mathcal{D} \leq n$, we could expect the same for its expected value $E[\mathcal{D}]$, i.e., $0 \leq E[\mathcal{D}] \leq n$. In other words, in the infinite number of possible SUTs, we could expect to have at least one SUT for each possible value $E[\mathcal{D}]$ can assume in $[0, n]$. Counterintuitively, the following theorem proves this conjecture wrong and provides an upper bound:

Theorem 5. *For any SUT, any n testing targets T_i and any value l for the limit of sampled test cases, the expected value of the random variable $\mathcal{D}$ is upper bounded by $n/2$, i.e., $E[\mathcal{D}] \leq n/2$.*

Proof. Given the function $f(x) = x(1-x)$, we start from proving that $f(x)$ is maximized for $x = 1/2$. This will turn out to be useful when we analyze the formula for $E[\mathcal{D}]$ given in Theorem 4. For $x = 1/2$, we have $f(1/2) = 1/4$. By contradiction, let us assume that there exists a value y for which $f(y) > 1/4$. Then, it follows $f(y) = y(1-y) = y - y^2 > 1/4 = (1/2)^2$ and, hence, $y^2 - y + (1/2)^2 = (y - 1/2)^2 < 0$, which is not possible.

From Theorem 4, we can hence write $E[\mathcal{D}] = 2\sum_{i=1}^{n} f(x_i)$, where $x_i = (1-p_i)^l$. Therefore, $E[\mathcal{D}]$ is maximized when we have for all the x_i the equality $(1-p_i)^l = 1/2$. In this case, $E[\mathcal{D}] = 2\sum_{i=1}^{n} 1/4 = n/2$. □

Now, we are in a position to answer the research question *"is random testing predictable?"* Theorem 5 states that, on average, the difference in covered targets can be at most $n/2$, so the results of random testing can never be completely unpredictable.

Theorem 4 gives us the tool to explain the conditions for which random testing is more predictable. Simulations in Fig. 1 show that for $\mathcal{P}_0$, we have a first phase in which the predictability is high (i.e., low value of $E[\mathcal{D}]$). Then, with the increase of l the predictability decreases ($E[\mathcal{D}]$ increases) down to a level that is close to the theoretical bound $n/2$ (in our case 16), thus confirming the validity of the upper bound provided by Theorem 5.

But we cannot simply state that $E[\mathcal{D}]$ always increases to a maximum level and then decreases. In fact, this does not happen for the case $\mathcal{P}_1$. The variable $E[\mathcal{D}]$ can fluctuate as the number of test cases l increases. Of particular interest is the case of $\mathcal{P}_2$, in which the predictability is quite high since $E[\mathcal{D}]$ has a low value that is mostly independent of the values of l.

We can use our formal results above to explain the empirical results in [13]. The different failure types in that case study have similar probabilities. The number of sampled test cases was large enough to be very likely to cover some of those targets, but not large enough to ensure that most of them would be covered. If the same empirical analysis was carried out again with less or more sampled test cases, our theorems and simulations suggest that very different conclusions would be obtained.

In [13], random testing was run 30 times with different seeds. Only execution time for a run was reported (i.e., 90 minutes). The number l of test cases was not provided. To analyze their specific claims, for simplicity we will hence make the reasonable assumption that each test case run has average duration ν minutes with negligible low variance ≈ 0. We can hence assume that for each run of 90 minutes we have $l = 90/\nu$ test cases.

In the empirical analysis in [13] it was reported that roughly 25 percent of failure types were found by only one or two of the 30 runs, while 19 percent of failure types were found by all the 30 runs. What would happen if we would run those experiments again with different numbers of test cases l? Let us consider the case in which we assume that 25 percent of targets are found by only one run out of 30. Each of those $n/4$ targets would have a probability p_i of being covered by a random test case. Covering a specific target with l test cases would have the following probability:

$$P(cover) = 1 - (1-p_i)^l.$$

Because the 30 runs are independent, the probability that only one run covers that target is

$$\begin{aligned} P(single) &= \binom{30}{1} \times P(cover) \times (1 - P(cover))^{29} \\ &= 30(1-(1-p_i)^l)(1-p_i)^{29l}. \end{aligned}$$

Based on these equations, we would like to give an estimation of p_i in that case study [13]. We do not seek to get an exact value, we just want to obtain a *reasonable* value to study the effect of what would happen if we ran more test cases (i.e., if we increase l). This would be essential to verify whether the claims in [13] can be generalized or not.

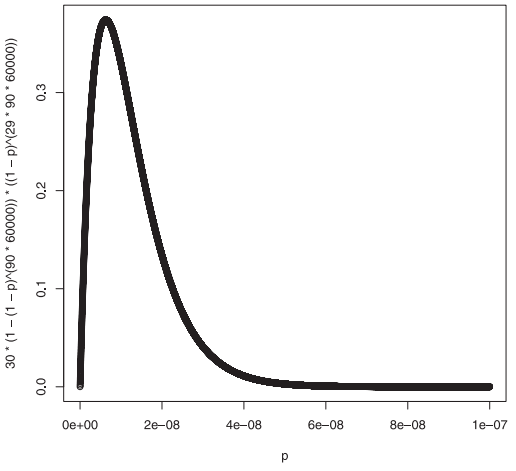

Fig. 2. Probability that only one out of 30 runs covers a specific single target T_i for different values of p_i.

To study the impact of increasing l on the empirical results in [13], we need approximate estimates of the values of l and p_i for that study so as to compute $P(single)$. We first estimate l by assuming a reasonable time to run a test case ($\nu = 1/60{,}000$, i.e., a test case takes one millisecond to run), therefore setting l equal to $60{,}000 \times 90 = 54 \times 10^5$. Fig. 2 shows $P(single)$ for a range of p_i values for that estimated value of l. The most likely p_i value for the study in [13] is the one that maximizes the chances of observing what was reported. Fig. 2 shows that $P(single)$ has the highest value when $p_i \approx 10^{-8}$. For that value of p_i, the expected number of test cases to cover one of the above difficult targets would be $1/p_i = 10^8$, which would take roughly 1,666 minutes (roughly 27 hours), which is slightly less than 20 times the 90 minutes reported in the study. This much smaller execution time therefore explains why many targets were only found by one or two runs. If random testing was run for one day (which is a feasible amount of time) instead of an arbitrary 90 minutes, the conclusions of the study would have been very likely the opposite of what was reported: Random testing is highly predictable.

Another arguable claim in [13] is that in their case study random testing "*is almost certain to detect a fault for any class within 30s*" [13]. In 30 seconds we would have $0.5/\nu = 30{,}000$ test cases. Mathematically, that claim can be translated into $P(cover) = 1 - (1 - \sum_{i=1}^{n} p_i)^{30{,}000} > 0.99$. Since the failure rate of the software is then $\theta = \sum_{i=1}^{n} p_i$, this translates into $\theta > 0.0002$, which can be considered as a very high failure rate since it would mean a failure every 5,000 random test cases. For most industrial software, in particular safety and business critical software, in general we would not expect such a high failure rate [32]. Even when we consider small and trivial software, in which faults are introduced with a mutation testing tool, the failure rates are still much lower [4]. Therefore, it is clear that the authors' claim is based on very particular circumstances [13] which are unlikely to be representative of most industrial software and would therefore not be applicable in general.

Notice that these discussions are based on the assumption that $\nu = 1/60{,}000$, which may not be fully accurate but is unlikely to be a strong departure from the real value. The testing algorithm in [13] is not a canonical random testing (see Section 3), despite the title of the paper, though it can be considered fairly similar. The differences are, however, small enough that we can reasonably apply our results to the work in [13]. If we had a higher value for ν, not only would our critique still be valid, it would even be stronger because the failure rate would be even higher. Unfortunately, the value of ν is not provided in [13]. To make the results of an empirical analysis involving random testing easier to interpret and analyze, it is advisable to always report the number of test cases that are generated and run.

To summarize, based on the above formal results, we can conclude that random testing cannot be completely unpredictable (Theorem 5). There are conditions where it can be very predictable (e.g., $\mathcal{P}_2$) as well as quite unpredictable (e.g., $\mathcal{P}_0$). In the latter case, predictability can be achieved by increasing (when possible) the number of sampled test cases. However, how many more test cases are needed cannot be answered in general without knowing the values of p_i. In fact, if we add more test cases, we can end up in a case such as $\mathcal{P}_1$, in which for $\log_2(l) = 4$ we have that, with more test cases, $E[\mathcal{D}]$ first decreases and then increases again. Furthermore, in some situations there might be stringent testing budgets, thus making it impossible to increase the number of test cases.

Notice that the results of this section directly extend to the cases in which there are infeasible targets. In fact, because infeasible targets will never be covered, the random variable $\mathcal{D}$ is not affected by them (because it is based only on the difference of covered targets). Regarding Theorem 5, in general we would not know how many n feasible targets and m infeasible targets there are. However, it is trivial to see that the bound $n/2$ would be lower than (or equal to) half of the total number of targets (i.e., $(n+m)/2$) for *any* number m of infeasible targets.

5.5 Threats to Validity

Threats to *internal validity* for this work come from the fact that mathematical analyses can be subject to errors. To reduce the probability of this being the case, we have also carried out simulations and have shown their results to confirm our theorems. But simulations cannot be used to prove that a proof is error free. Furthermore, in this paper we have investigated some of the published work in the literature, in particular [13], [26], [28], [47], [52]. Although we have carefully read these papers several times, we cannot exclude that we have misinterpreted some of their results.

Threats to the *external validity* of formal analyses depend on the *assumptions* that are made. Depending on the validity of these assumptions, the application of the theoretical results to real-world scenarios could be compromised. In this paper, the main assumption we make is that the testing targets T_i are disjoint. We have shown that this is not a problem when the targets are different failure types and for coverage strategies such as path coverage. In case of testing strategies such as branch coverage, the results could be still valid in some relevant cases. In fact, when the testing problem consists of covering a subset of difficult branches after a first phase of random testing is carried out, then these branches could be mostly disjoint targets.

6 Practical Implications

The results discussed in Section 4 identify the conditions for which random testing fares better or worse than partition testing. For example, (5) tells us that the sampling and failure rates play a crucial role. But do these conditions in which random testing can be better actually appear in practice? Section 4.5 discussed some successful applications of random testing, which provides some evidence to the reality of those theoretical results. However, identifying and proving that there are conditions for which random testing can be a better option, and empirically showing that there are some cases in which it is indeed better, does not mean that those conditions appear *often* in practice. To make an informed decision, a practitioner needs guidelines about where and when a particular testing strategy should be used, as it is unlikely that any strategy is going to be the best option for all testing problems. Random testing is also a strategy that is commonly employed by researchers as a baseline to assess the benefits of other strategies. The results surveyed and presented in this paper are therefore useful to understand and interpret existing and new empirical studies. This section focuses on providing guidelines to both practitioners and scientists based on the survey and new theoretical results presented above.

Our theoretical analysis includes a number of new theorems. We formally proved that random testing always needs, on average, at least nH_n/α iterations to find test data to cover all the feasible targets, and that boundary conditions happen when all the testing targets have identical probability (Theorem 1). Furthermore, random testing can scale better than some types of partition testing techniques (Theorem 3). Finally, we formally proved the predictability distribution of random testing (Theorem 4) where the further away from the uniform distribution the testing targets are, the more predictable random testing is. Increasing the number of sampled test cases can either increase or decrease predictability. When there is a great deal of variance in the target probabilities, then random testing becomes highly predictable regardless of the number of test cases. In any case, the expected number of different targets covered by two runs is never more than half the total number of targets (Theorem 5). There is an interesting aspect when we look at both Theorem 1 and Theorem 4: The closer the probability distribution of the targets is to the uniform distribution, the fewer test cases to sample but the less predictable the test results are (i.e., better performance is linked to worse predictability).

The first question a practitioner needs to ask himself is whether there is any automated oracle. In this context, an automated oracle is a mechanism that can automatically verify whether a testing target has been covered or not. One of the advantages of random testing is the possibility of generating a large number of test cases at a relatively low cost. However, if one needs to manually check if the targets are covered, only a small number of test cases can be run and evaluated, and random testing is unlikely to be the right strategy. This is a typical example in unit testing with white-box test techniques. For example, it is a common practice in industry to manually write test cases for each unit (e.g., functions and classes) with the aim of maximizing statement or branch coverage, and then manually writing assert statements to check whether the behavior of the unit is as expected. In this case, generating test data randomly is unlikely to be a good strategy as only a few test cases can be run and analyzed. So, it would be better to focus on writing a small number of test cases targeting all statements or branches.

As previously discussed (see Section 4.1), random testing can be used to generate test data to maximize code coverage in the context of white-box testing. For this particular type of target, we always have an "automated oracle": When we run a test case, we can always track which parts of the code have been executed. As a result, we can run random testing for as long as needed, and then give to the user only a minimized subset of sampled test cases that maximize coverage. However, code coverage often results in very unbalanced partitions. For example, only a tiny fraction of the input domain typically covers some of the code targets, where common examples are code branches whose predicates are based on numerical comparisons. Both (5) and Theorem 1 suggest using random testing in this context is a bad idea. The former tells us that partition testing is likely to work best with widely different sampling rates or failure rates for targets. The latter tells us random testing is most effective when all targets have equal probabilities of being reached. None of these conditions are usually met when performing white-box testing. Furthermore, for this testing problem there are research tools that have been empirically shown to give much better results than random testing (e.g., Pex [44] for C# software). However, there are always exceptions, as, for example, specific types of software for which random testing gives good results even for code coverage (e.g., for container classes [41]).

In black-box testing, when automated oracles are available, for example, derived from behavior models of the SUT, then random testing can in some cases become a viable option. For example, state invariants from state models can be used to instrument programs and obtain test oracles. In practice, black-box testing is commonly applied for testing large systems or even systems of systems. In many cases, such testing is performed by independent test teams who have knowledge of the application domain and requirements, but have had no involvement in the system design or do not even have access to the source code. Furthermore, because no single individual typically has a good understanding of all the system's functionalities, many stakeholders must typically get involved in system test specifications and writing them becomes a time-consuming challenge. On the other hand, during the same time random testing could be left running 24/7 without any manual intervention, where, for example, millions of test cases could be automatically run and automatically evaluated. As a further advantage, random testing is not affected by infeasible targets. Equation (5) tells us that random testing will work best when differences in failure rates of partitions remain small. This is what one would expect if, in the context of black-box testing, the test engineers in charge of test specifications have an insufficient understanding of the system functionalities and assumptions.

Once the test specifications are completed, test engineers need to automatically generate test cases from each of these

partitions. But if a particular partition is defined by a complex logical constraint (predicate) and one wants to automatically solve it, then a constraint solver needs to satisfy two conditions to be applicable: 1) It must be efficient or otherwise, in the same amount of time, random testing could fare better by generating and executing many test cases, 2) it must be able to generate alternative test data satisfying the same constraint as partitions are usually not homogenous and we need to sample more than one test case from each partition. The latter condition is only fulfilled if the constraint solver is not deterministic or biased in generating solutions. An alternative option would be to keep track of all the test cases belonging to that partition generated so far, and increase the constraint by adding that the new solutions have to be different from the generated so far. As this approach would increase the complexity of the constraint linearly in the number of sampled test cases, it might have negative effects on the performance of the constraint solver. Unfortunately, the research literature only features a limited number of empirical analyses of black-box, system level testing of real-world industrial systems. Therefore, it is still too premature to draw definitive conclusions.

To conclude, as the availability of automated tools for black-box system testing is currently limited, and considering the high cost and expertise requirements for manually writing system-level test cases or generating test data using constraint solvers or other analytical tools, random testing might turn out, despite its limitations, to be a viable first option to try out (particularly when limited automated testing tool support), and use as a complement to other techniques. As we have shown in this paper, there are situations in which random testing can be indeed effective, scalable, and predictable.

There are also several guidelines we can derive from the survey and theorems that are targeted at scientists studying RT or comparing it with more sophisticated test strategies. Since (5) tells us the failure and sampling rates of partitions have a direct effect on the difference in effectiveness between partition testing and random testing, these two factors must be accounted for when running case studies and comparing possibly inconsistent results across studies. Theorem 1 and Theorem 4 indicate that the probability distribution of targets strongly affects the effectiveness and predictability of RT, respectively. The more uniform the probability distribution, the more effective and the less predictable RT gets. Variation of effectiveness across studies can therefore be in part explained by target probability distributions. In situations where RT is less predictable, it is also particularly advised to execute a large number of test cases to better support the statistical analysis. Finally, it is important to keep in mind that the choice of the specific random testing and partition testing algorithms can have a significant impact (Theorem 2 and Theorem 3). As a result we recommend using RT instead of $RT0$ and making sure as well that the partition testing algorithm is designed as much as possible as a parallel search algorithm (Section 5.3), thus making it more scalable. Theorem 3 also suggests that when comparing random and partition algorithms, scalability must be studied as one technique may scale better than an other and thus lead to different results for larger test problems.

7 Conclusion

In this paper, we have surveyed and analyzed the properties and applications of random testing that were reported in the literature. Furthermore, we have provided novel formal results regarding the effectiveness, scalability, and predictability of random testing. Random testing is often used as a comparison baseline to assess other testing techniques and, therefore, studying its properties is important for driving and interpreting empirical research. This will eventually lead to a clearer understanding of the strengths and limitations of random testing. Since it has been increasingly seen as a plausible alternative technique for testing in the large, the potential impact of such research on practice is significant.

We have proven nontrivial, tight lower bounds for the expected number of test cases sampled by random testing to cover predefined targets. We have shown which type of random testing should be used as a baseline of comparison to evaluate other testing techniques in a fair manner. We discussed the scalability of random testing, which, under certain conditions, can fare better than a large class of partition testing techniques. In addition, we provided practical advice on how to improve some of these testing techniques (e.g., always keep track of the so-called collateral coverage, use seeding strategies). Finally, we formally proved the mathematical formula that describes the predictability of random testing and we proved a general lower bound for it. These results are also used to assess the conclusions of some of the empirical analyses reported in the literature and discuss their validity. Together with our survey results, we rely on these formal results to provide guidelines on the usage and experimenting of random testing.

The analyses in this paper further support the use of random testing in a real-world software [16], [23], [49]. Though successful applications of random testing and its variants appear in the literature (e.g., [20], [37]), overall the reported results have not been consistent. As a result, the jury is still out on whether and when random testing is a good option and more, better designed and reported studies are needed. This paper provides a theoretical basis that may help better interpret empirical results in the future and explain differences among studies. Our theoretical results also show that there are some conditions in which random testing is viable testing option that should be seriously considered.

For future work, it would be important to extend the results of this paper for the cases in which the probabilities of covering testing targets are modeled as random variables. A similar exercise was carried out by Gutjhar [22] when he analytically compared random testing with partition testing. Another important research question to investigate is how to extend these results for the cases when the testing targets are not disjoint, as, for example, in branch coverage.

Acknowledgments

The work described in this paper was supported by the Norwegian Research Council. This paper was produced as part of the ITEA-2 project called VERDE.

REFERENCES

[1] J.H. Andrews, A. Groce, M. Weston, and R.G. Xu, "Random Test Run Length and Effectiveness," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 19-28, 2008.

[2] A. Arcuri, "Theoretical Analysis of Local Search in Software Testing," *Proc. Symp. Stochastic Algorithms, Foundations and Applications*, pp. 156-168, 2009.

[3] A. Arcuri, "A Theoretical and Empirical Analysis of the Role of Test Sequence Length in Software Testing for Structural Coverage," *IEEE Trans. Software Eng.*, http://doi.ieeecomputersociety.org/10.1109/TSE.2011.44, 2011.

[4] A. Arcuri and L. Briand, "Adaptive Random Testing: An Illusion of Effectiveness?" *Proc. ACM Int'l Symp. Software Testing and Analysis*, pp. 265-275, 2011.

[5] A. Arcuri and L. Briand, "Formal Analysis of The Probability of Interaction Fault Detection using Random Testing," *IEEE Trans. Software Eng.*, doi:10.1109/TSE.2011.85, 2011.

[6] A. Arcuri and L. Briand, "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," *Proc. ACM/IEEE Int'l Conf. Software Eng.*, pp. 1-10, 2011.

[7] A. Arcuri, M.Z. Iqbal, and L. Briand, "Black-Box System Testing of Real-Time Embedded Systems Using Random and Search-Based Testing," *Proc. IFIP Int'l Conf. Testing Software and Systems*, pp. 95-110, 2010.

[8] A. Arcuri, M.Z. Iqbal, and L. Briand, "Formal Analysis of the Effectiveness and Predictability of Random Testing," *Proc. ACM Int'l Symp. Software Testing and Analysis*, pp. 219-229, 2010.

[9] P.J. Boland, H. Singh, and B. Cukic, "Comparing Partition and Random Testing via Majorization and Schur Functions," *IEEE Trans. Software Eng.*, vol. 29, no. 1, pp. 88-94, Jan. 2003.

[10] T.Y. Chen, F.C. Kuo, and R. Merkel, "On the Statistical Properties of Testing Effectiveness Measures," *J. Systems and Software*, vol. 79, no. 5, pp. 591-601, 2006.

[11] T.Y. Chen and Y.T. Yu, "On the Relationship between Partition and Random Testing," *IEEE Trans. Software Eng.*, vol. 20, no. 12, pp. 877-980, Dec. 1994.

[12] T.Y. Chen and Y.T. Yu, "On the Expected Number of Failures Detected by Subdomain Testing and Random Testing," *IEEE Trans. Software Eng.*, vol. 22, no. 2, pp. 109-119, Feb. 1996.

[13] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer, "On the Number and Nature of Faults Found by Random Testing," *Software Testing, Verification and Reliability*, vol. 21, pp. 3-28, 2009.

[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, second ed. MIT Press and McGraw-Hill, 2001.

[15] R.A. DeMillo, R.J. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34-41, Apr. 1978.

[16] J.W. Duran and S.C. Ntafos, "An Evaluation of Random Testing," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 438-444, July 1984.

[17] W. Feller, *An Introduction to Probability Theory and Its Applications*, third ed., vol. 1. Wiley, 1968.

[18] G. Fraser and A. Arcuri, "Evolutionary Generation of Whole Test Suites," *Proc. Int'l Conf. Quality Software*, pp. 31-40, 2011.

[19] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed Automated Random Testing," *Proc. ACM Conf. Programming Language Design and Implementation*, pp. 213-223, 2005.

[20] A. Groce, G. Holzmann, and R. Joshi, "Randomized Differential Testing as a Prelude to Formal Verification," *Proc. ACM/IEEE Int'l Conf. Software Eng.*, pp. 621-631, 2007.

[21] W.J. Gutjahr, "Optimal Test Distributions for Software Failure Cost Estimation," *IEEE Trans. Software Eng.*, vol. 21, no. 3, pp. 219-228, Mar. 1995.

[22] W.J. Gutjahr, "Partition Testing vs. Random Testing: The Influence of Uncertainty," *IEEE Trans. Software Eng.*, vol. 25, no. 5, pp. 661-674, Sept./Oct. 1999.

[23] D. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence," *IEEE Trans. Software Eng.*, vol. 16, no. 12, pp. 1402-1411, Dec. 1990.

[24] R. Hamlet, "Random Testing," *Encyclopedia of Software Eng.*, pp. 970-978, Wiley, 1994.

[25] M. Harman, "The Current State and Future of Search Based Software Engineering," *Proc. Future of Software Eng.*, pp. 342-357, 2007.

[26] M. Harman, F. Islam, T. Xie, and S. Wappler, "Automated Test Data Generation for Aspect-Oriented Programs," *Proc. Eighth ACM Int'l Conf. Aspect-Oriented Software Development*, pp. 185-196, 2009.

[27] M. Harman, S.G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem," *Proc. Int'l Workshop Search-Based Software Testing*, 2010.

[28] M. Harman and P. McMinn, "A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search," *IEEE Trans. Software Eng.*, vol. 36, no. 2, pp. 226-247, Mar./Apr. 2010.

[29] L. Holst, "Extreme Value Distributions for Random Coupon Collector and Birthday Problems," *Extremes*, vol. 4, no. 2, pp. 129-145, 2001.

[30] M.Z. Iqbal, A. Arcuri, and L. Briand, "Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies," *Proc. ACM/IEEE Int'l Conf. Model Driven Eng. Languages and Systems*, pp. 286-300, 2010.

[31] D. Knuth, *The Art of Computer Programming*, vol. 3. Addison Wesley 1973.

[32] M. Lyu, *Handbook of Software Reliability Engineering*. McGraw-Hill, Inc., 1996.

[33] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Trans. Modeling and Computer Simulation*, vol. 8, no. 1, pp. 23-30, 1998.

[34] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105-156, 2004.

[35] M. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge Univ. Press, 1995.

[36] G. Myers, *The Art of Software Testing*. Wiley, 1979.

[37] C. Pacheco, S.K. Lahiri, M.D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," *Proc. ACM/IEEE Int'l Conf. Software Eng.*, pp. 75-84, 2007.

[38] R Development Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2008.

[39] J.A. Rice, *Mathematical Statistics and Data Analysis*, second ed. Duxbury Press, 1994.

[40] B. Rosén, "Asymptotic Normality in a Coupon Collector's Problem," *Probability Theory and Related Fields*, vol. 13, no. 3, pp. 256-279, 1969.

[41] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov, "Testing Container Classes: Random or Systematic?" *Proc. Fundamental Approaches to Software Eng.*, 2011.

[42] S. Shioda, "Some Upper and Lower Bounds on the Coupon Collector Problem," *J. Computational and Applied Math.*, vol. 200, no. 1, pp. 154-167, 2007.

[43] P. Thévenod-Fosse and H. Waeselynck, "An Investigation of Statistical Software Testing," *Software Testing, Verification and Reliability*, vol. 1, no. 2, pp. 15-25, 1991.

[44] N. Tillmann and N.J. de Halleux, "Pex—White Box Test Generation for .NET," *Proc. Int'l Conf. Tests And Proofs*, pp. 134-253, 2008.

[45] P. Tonella, "Evolutionary Testing of Classes," *Proc. ACM Int'l Symp. Software Testing and Analysis*, pp. 119-128, 2004.

[46] M.Z. Tsoukalas, J.W. Duran, and S.C. Ntafos, "On Some Reliability Estimation Problems in Random and Partition Testing," *IEEE Trans. Software Eng.*, vol. 19, no. 7, pp. 687-697, July 1993.

[47] T. Vos, A. Baars, F. Lindlar, P. Kruse, A. Windisch, and J. Wegener, "Industrial Scaled Automated Structural Testing with the Evolutionary Testing Tool," *Proc. IEEE Int'l Conf. Software Testing, Verification and Validation*, pp. 175-184, 2010.

[48] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary Test Environment for Automatic Structural Testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841-854, 2001.

[49] E.J. Weyuker and B. Jeng, "Analyzing Partition Testing Strategies," *IEEE Trans. Software Eng.*, vol. 17, no. 7, pp. 703-711, July 1991.

[50] E.J. Weyuker and T.J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," *IEEE Trans. Software Eng.*, vol. 6, no. 3, pp. 236-246, May 1980.

[51] D. White, A. Arcuri, and J. Clark, "Evolutionary Improvement of Programs," *IEEE Trans. Evolutionary Computation*, vol. 15, no. 4, pp. 515-538, Aug. 2011.

[52] Y. Zhan and J.A. Clark, "A Search-Based Framework for Automatic Testing of Matlab/Simulink Models," *J. Systems and Software,* vol. 81, no. 2, pp. 262-285, 2008.

Andrea Arcuri received the BSc and the MSc degrees in computer science from the University of Pisa, Italy, in 2004 and 2006, respectively, and the PhD degree in computer science from the University of Birmingham, England, in 2009. Since then, he has been a research scientist at Simula Research Laboratory, Norway. His research interests include search-based software testing and analyses of randomized algorithms. He is a member of the IEEE.

Muhammad Zohaib Iqbal received the MS degree in systems and software engineering from Mohammad Ali Jinnah University, Islamabad, Pakistan. He is currently working toward the PhD degree in the Simula Research Laboratory and the Department of Informatics, University of Oslo, Norway. Before that, he worked as a lecturer in the Department of Computer Science, International Islamic University, Islamabad, Pakistan. He has also been on the faculty of the Department of Computer Science as a lecturer at Mohammad Ali Jinnah University, Pakistan. His research interests include model driven development of software systems, especially with UML and its related technologies, search-based testing, and model-based testing of software systems. He is a member of the IEEE.

Lionel Briand is a professor of software engineering at the Simula Research Laboratory and the University of Oslo, leading the project on software verification and validation. Before that, he was on the faculty of the Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, where he was a full professor and held the Canada Research Chair in Software Quality Engineering. He has also been the Software Quality Engineering Department head at the Fraunhofer Institute for Experimental Software Engineering, Germany, and worked as a research scientist for the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland. He has been on the program, steering, or organization committees of many international IEEE and ACM conferences. He is the editor-in-chief of the *Empirical Software Engineering* (Springer) and is a member of the editorial boards of *Systems and Software Modeling* (Springer) and *Software Testing, Verification, and Reliability* (Wiley). He was on the board of the *IEEE Transactions on Software Engineering* from 2000 to 2004. His research interests include: model-driven development, testing and quality assurance, and empirical software engineering. He is a fellow of the IEEE.