Adaptive Test Case Allocation, Selection and Generation using Coverage Spectrum and Operational Profile

Antonia Bertolino, Breno Miranda, Roberto Pietrantuono, *Senior Member, IEEE,* and Stefano Russo, *Senior Member, IEEE*

Abstract—We present an adaptive software testing strategy for test case allocation, selection and generation, based on the combined use of *operational profile* and *coverage spectrum*, aimed at achieving high delivered reliability of the program under test. *Operational profile-based testing* is a black-box technique considered well suited when reliability is a major concern, as it selects the test cases having the largest impact on failure probability in operation. *Coverage spectrum* is a characterization of a program's behavior in terms of the code entities (e.g., branches, statements, functions) that are covered as the program executes. The proposed strategy - named *covrel+* - complements operational profile information with white-box coverage measures, so as to adaptively select/generate the most effective test cases for improving reliability as testing proceeds. We assess *covrel+* through experiments with subjects commonly used in software testing research, comparing results with traditional operational testing. The results show that exploiting operational and coverage data in an integrated adaptive way allows generally to outperform operational testing at achieving a given reliability target, or at detecting faults under the same testing budget, and that *covrel+* has greater ability than operational testing in detecting hard-to-detect faults.

Index Terms—Software testing, Reliability, Operational testing, Random testing, Sampling

1 Introduction

This article presents a novel software testing strategy that integrates black- and white-box testing techniques for improving the delivered reliability of a software product. The strategy, named *covrel+*, builds on two branches of the software testing literature: operational-profile based testing and code coverage-based techniques for test case selection and generation.

Operational-profile based testing – or simply *operational testing*, OT - is a black-box technique well suited when reliability is a major concern, as it selects the test cases having the largest impact on failure probability in operation. To this aim, OT selects test cases based on an operational profile estimate [1]. An *operational profile* is a quantitative characterization of how a system will be used [2]. Various approaches have been proposed to define it [3]; a commonly used one is to decompose the input domain D of a program P under test into m partitions D_i ($i = 1..m$), and to describe the operational profile as a probability distribution over them. Precisely, the profile is described as a set of values, p_i, denoting the probability that an input is selected from partition D_i and such that $\sum_{i=1}^{m} p_i = 1$ [4].

OT is a pillar of *software reliability engineering* practices [5]. We assume – in line with [6] - that the reliability R of P is defined as: $R = 1 - \sum_{t \in F} p_t$, where F is the set of inputs leading to failure (or *failure points*), and p_t is the expected probability of occurrence in operation of input t. The operational testing strategy we consider here is the so-called *partition-based OT* [7]: it first selects a partition D_i randomly according to the generated profile (i.e., with probability p_i); then it generates a test case for D_i by selecting an input within D_i according to a uniform distribution, as, e.g., in [4] and [7].

Coverage-based testing refers to many different white-box testing techniques that use a signature of a program's behavior (*spectrum*) obtained by tracking the coverage of entities (e.g., statements, branches, functions) in execution [8]. Most coverage-based testing techniques consider *hit spectra*, which only list which program entities have been executed (at least once). *Count spectra* provide richer information, by measuring how many times each entity is executed.

Program count spectra have found several applications beyond their original use in program optimization [9], and are used extensively in software analysis and testing [10]. Reps *et al.* [11] proposed to use differences between path spectra for identifying changes in program behavior. Following this idea, code profiling information has been used to analyze the executions of different versions of programs, e.g., in regression testing [12], or to compare traces of failed and successful runs in fault diagnosis [13] [14].

In *covrel+* we also use count spectra, yet differently from previous approaches, not to differentiate between program behaviors, but to identify less exercised program entities. We exploit such information to complement operational testing, with the ultimate goal of improving the delivered reliability [6]. In particular, we aim at counteracting the saturation effect ([5], Chapter 13), by which the continued application

- *A. Bertolino is with ISTI - CNR, Pisa, Italy. E-mail: antonia.bertolino@isti.cnr.it.*
- *B. Miranda is with Federal University of Pernambuco, Brazil. E-mail: bafm@cin.ufpe.br.*
- *R. Pietrantuono and S. Russo are with Università degli Studi di Napoli Federico II, Napoli, Italy. E-mail: {roberto.pietrantuono, stefano.russo}@unina.it.*

Manuscript received Feb 28, 2019.

of a testing technique eventually loses efficacy. To this aim, we use the notion of *operational coverage* introduced in [15], which clusters entities into different "importance" groups according to their count spectra, and assigns them different weights used in operational testing.

Covrel+ supports **test cases allocation, selection and generation** for reliability improvement. Given P and D, the problem of *allocation* is to divide an available budget $Budget$ of test cases among a given set of m partitions D_i of D. To this aim, we need to identify the (optimal) number of test cases $|T_i|$ to allocate to each partition D_i (such that $\sum_{i=1}^{m} |T_i| \leq Budget$) in order to test P with respect to a given testing goal (here, for improving reliability). The problem of *selection* is to identify the most effective subset of test cases TS within an available test suite T, with respect to a testing objective. Finally, the problem of *generation* is the construction of a test suite TG. The idea of combining operational testing with count spectra for improving reliability was first presented in [16], with reference to the problem of test cases selection. Here, we extend this own work by providing a more detailed definition and analysis of the test selection strategy, and by presenting its generalization to also address the problem of test cases generation.

The paper is structured as follows. Section 2 describes related work. Section 3 presents the proposed *covrel+* strategy. Section 4 describes the experiments. Section 5 discusses the results and their statistical significance, as well as the threats to validity. Section 6 contains concluding remarks.

2 Related Work

An overview of the broad research fields of test case allocation for reliability improvement and coverage-based test selection and generation is beyond reach within one paper. We refer the reader to the highly referenced and still relevant Lyu's handbook [5] for details about OT, and to references [17], [18] and [19] for test allocation, selection and generation, respectively. Here we focus on related work that from different perspectives explores the relationship between reliability and code coverage.

Many empirical studies, e.g., [20] [21] among the most recent ones, assess the effectiveness of coverage-based testing. However only a few of them consider test effectiveness in terms of delivered reliability. Among the earliest studies, Del Frate and coauthors [22] found a correlation between increase (decrease) in reliability and increase (decrease) in at least one code coverage measure. In a later work Frankl and Deng [23] performed a case study comparing various approaches and showed that as coverage increases, the probability to achieve high reliability targets increases as well. They show also that the probability to reach very high reliability values would require extremely large test sets, and it is doubtful whether the improvement is worth the cost. This is what motivates our work: *covrel+* explores the usage of coverage *in combination* with traditional operational testing for reliability improvement.

Several authors have proposed to integrate test coverage information into models used to evaluate reliability as faults are found and removed (a.k.a. Software Reliability Growth Models or SRGMs). A recent short compendium of such coverage-integrated SRGMs is given by Alrmuny [24]. Although the basic motivation is the same, i.e., that reliability improvement can be impacted by observed coverage measures, the usage that we make of such measures is different. In SRGMs coverage information is used to better tune reliability estimation, as in [25]; in *covrel+* we use coverage information for driving test selection/generation, building on the concept that coverage measures can provide guidance in identifying what parts of a program should be exercised when augmenting a test suite [26] [27].

A similar concept inspires the so-called "accelerated testing method" SRAT in [28] that proposed to reduce a reliability test suite by weighting and clustering the test cases according to their achieved coverage. In comparison with *covrel+*, the SRAT approach only considered the coverage hit spectrum, and used this information to reduce the number of test cases to be executed, considering test cases that cover already exercised entities as redundant. In contrast, we exploit coverage information to select or generate test cases that would most likely cover rarely exercised entities.

A novel aspect of *covrel+* is the use of the coverage *count spectrum* (instead of the most commonly used hit spectrum). Count spectra are typically used for fault localization (namely, spectrum-based fault localization or SBFL). In [15], we first introduced the notion of operational coverage based on count spectra, and in [29] we showed that it can be used as both an adequacy and a selection criterion for operational profile based testing. To the best of our knowledge, we are the first to propose here the usage of count spectra in combination with operational profile to guide both test selection and generation for reliability-oriented testing.

Several approaches have been proposed for automated test case generation driven by coverage requirements. Some exploit Dynamic Symbolic Execution [30] or Search-based techniques [31] to derive a set of test cases that would cover all targeted program entities. In principle such approaches could be integrated within our adaptive allocation technique, however they do not target specifically reliability improvement, but aim at detecting a high number of faults.

As we use coverage information to complement OT, the closest approaches to *covrel+* are those that use variants of random test generation. In particular, Randoop [32] is a popular tool for automated feedback-directed test generation: it uses the results of previous executions (e.g., exceptions or other errors) to randomly and effectively generate further bug-finding test cases. For test generation, we also use feedback from previous executions, but with a different aim that is catching the test inputs that contribute mostly to reliability improvement. In adaptive random testing (ART) [33], test cases are generated randomly, evenly spread across the input domain: in this sense, ART also embeds a notion of similarity among test cases, as we do in *covrel+* for test case generation (see Section 3). Differently from us, in ART the similarity is measured as the distance in the input domain and not based on coverage. Moreover, ART does not consider the operational profile and does not explicitly aim at increasing reliability: it has been mostly evaluated in terms of test cases needed to detect the first failure.

A further research thread is the statistical structural testing approach to test generation, initiated by the seminal work of Thévenod-Fosse and Waeselynck [34]. It aims at determining an input distribution such that random

sampling test cases from it maximizes the probability to cover all targeted code elements. Various techniques have been then proposed to implement such idea. For instance, Petit and Gotlieb [35] formulated the goal of covering all paths in a program as a stochastic constraint problem, thus providing an algorithmic solution naturally preventing infeasible paths; Poulding and Clark [36] used automated search, showing that it could provide a general method for finding a suitable input distribution. Similarly to *covrel+*, statistical testing assigns probabilities to test inputs based on the code elements they cover. However, in statistical testing coverage is used as a means for increasing the fault finding effectiveness. The goal of *covrel+* is to improve reliability, hence it assigns input probabilities based on the operational profile: intuitively, coverage information is used to distinguish among test inputs yielding a same usage probability.

Based on our review of current literature, we believe that *covrel+* combines several means (including operation profile, adaptivity, count spectrum, similarity measure) into a completely novel and powerful approach.

3 THE *Covrel+* STRATEGY

3.1 Assumptions

Covrel+ shares the following assumptions with OT:

a) The input domain D can be decomposed into m partitions $D_1, \ldots, D_m$. These are obtained according to some partitioning criterion (e.g., functional or structural), usually depending on the information available to test designers, and on testing objectives.
b) The operational profile can be described as a probability distribution p_i over the partitions (with $\sum_{i=1}^{m} p_i = 1$). Inputs within a partition have the same probability of runtime occurrence.
c) A test case leads to failure or success, and we are able to determine what is the case (perfect oracle).
d) Test case runs are independent, i.e., the execution of a test is not constrained by that of previous tests. This assumption merely affects the way test cases are defined, since when test cases are sequences of tasks that are not independent, they can be grouped to form a single test case, so that after the test the system goes back to the initial state [4].
e) The output of a test case is independent of the history of testing: a failing test case is always such, independently of the previously run test cases.

Assumptions $a)$ and $b)$ are for the applicability of *covrel+*, since it works on partitions of the input space and requires an operational profile defined on them. Assumptions $c)$, $d)$ and $e)$ are not necessary for *covrel+* applicability, but their violation is expected to negatively impact its performance. An imperfect oracle (assumption c)), as well as the dependence of test case runs and/or of their output (assumptions $d)$ and $e)$), can make the output of a test in repeated executions unpredictable. In such situations, the allocation of test cases to partitions would be affected by tests marked as failing but not due to actual faults in the code (or, conversely, by tests marked as correct in some executions and failing in others). Hence, violations of assumptions from $c)$ to $e)$ could make *covrel+* behave differently from one execution to another.

Clearly, *covrel+* assumes additionally that coverage information of test cases is available or it can be obtained.

3.2 Strategy overview

The objective of *covrel+* is to improve delivered reliability efficiently by means of a focused test case derivation strategy: we use here the generic term *derivation* to mean either selection or generation. In fact *covrel+* applies the same underlying idea to solve two problems in software testing, namely test case selection and test case generation. In both cases the strategy proceeds iteratively and adaptively: the shared idea is to gain knowledge as testing proceeds so as to dynamically adapt to *i)* which regions of the input space, and then *ii)* which test cases within each region, are expected to contribute more to reliability improvement at the next iteration. *Covrel+* searches for those test cases with potentially the highest contribution to reliability, by combining their ability of revealing high-occurrence failures with the ability of finding still undetected faults. To do so it uses *learning* and *adaptation* to dynamically characterize the input domain partitions and derive test cases within them.

The *covrel+* strategy is sketched in Figure 1. It exploits iteratively the results of test executions per partition (namely, number of exposed failures and current coverage), in order to drive test allocation and selection/generation at the subsequent iteration. More precisely, each iteration foresees two main phases: *(i)* the derivation of test cases within each partition: if a test suite is available, the most effective test cases are chosen (**selection**); otherwise new test cases have to be generated, which we do by applying a similarity-driven random approach (**generation**); *(ii)* the **allocation** of test cases to partitions (i.e., how many test cases should be within each partition at the next iteration).

Selection and generation are both informed by the *cumulative count spectrum*, which counts, at each iteration, how many times the program entities have been exercised. Test case generation also uses *partition count spectrum*, which characterizes how many times the program entities are exercised by the test cases associated to partitions.

Program entities are ordered according to their count in the spectrum, and then classified into *importance groups*.

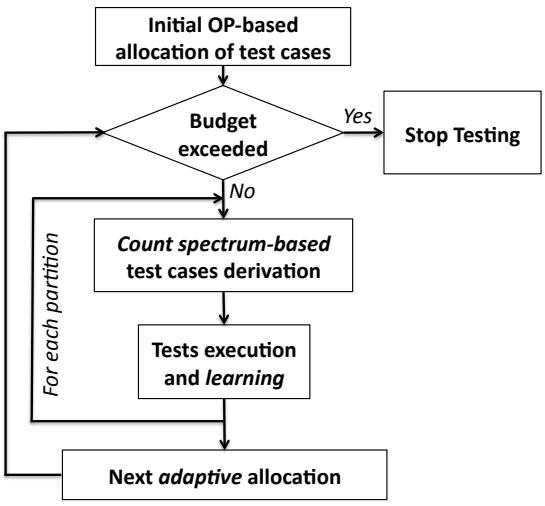


Fig. 1. The *covrel+* strategy

This is because the last phase allocates test cases adaptively through an algorithm based on the *Importance Sampling* (IS) method [37] (used in previous own work [38]). *Covrel+* then uses group membership differently for selection and generation. This is explained in the following subsections.

3.3 Allocation of test cases to partitions

Covrel+ allocates test cases adaptively depending on where more tests are actually needed. This is accomplished by means of the Importance Sampling method, which is an inference method to approximate the *true* unknown distribution of a variable of interest. Here, the distribution of interest is the number of test cases for each partition that would maximize the delivered reliability. The algorithm represents the beliefs (hypotheses) about this distribution by means of sets of "samples". Each sample is associated with a probability that the belief is true: these probabilities are updated iteratively by examining some new samples of the hypotheses, and a larger number of samples are drawn from hypotheses with a larger probability. The goal is to converge, in few iterations, to the "true" best distribution of test cases.

An *update rule* establishes how the probability of each hypothesis is modified based on new collected samples. The number of test cases per partition is determined accordingly. The IS-based algorithm exploits information about the failing tests observed in previous iterations; in addition, for test case generation the algorithm exploits the similarity between the partition and the set of least covered entities. Test case selection does not exploit similarity because there is a priori knowledge of which test cases cover which entities, and that information, combined with the cumulative count spectrum, suffices to select the best cases based on their contribution for covering the rarely exercised entities. (Clearly, the knowledge about how test cases cover the program is not available in generation beforehand.)

In test case selection, to smoothly direct testing towards the partitions with a high expected (un)reliability contribution in the next iteration, the IS-based algorithm uses the observed failure rates φ_i, defined as number of failing tests over number of executed tests by the operational profile values p_i. We denote with $\theta_i = p_i\varphi_i$ the weighted failure rate (normalized so that $\sum_{i=1}^{m} \theta_i = 1$).

In test case generation, the IS-based algorithm uses:

- The *similarity score* (denoted with σ_i), defined as the *Jaccard similarity* between the set of least covered entities from the cumulative count spectrum and the partition count spectrum.[1] We use the similarity coefficients between the cumulative and partition spectra as predictors of which partition(s) would have higher probability of generating test cases that would exercise the least covered entities. The rationale is that if partition D_i has a high Jaccard similarity with the set of "low" entities from the cumulative count spectrum, then D_i covers many "low" entities. Hence, with more test cases for D_i, we may increase the coverage of the so far rarely exercised entities.
- The similarity and failure rate *balance factor*, whose value at current iteration is β_i. It determines the weight (in [0,1]) to be assigned to failure rate (weight $= \beta_i$) and to similarities (weight $= 1 - \beta_i$) in assessing the importance of that partition. The expectation is that the failure rate will decrease over iterations; hence, as testing time goes on, test cases are more and more generated using the similarity score. The value β'_i to be used in the next iteration is computed as $\beta'_i = \beta_i \cdot \phi$, where ϕ is a *discount factor* that progressively reduces the impact of the failure rate information on determining the importance of that partition, increasing the impact of similarity.

1. The Jaccard similarity coefficient of two sets A and B is defined as $\text{JS}(A, B) = |A \cap B| \,/\, |A \cup B|$.

More formally, let us denote with π the probability vector at current iteration, whose *i-th* element π_i represents the likelihood that testing from partition i contributes to improve reliability. The probability vector π' to be used at next iteration for test allocation is computed according the following *update rule*:

$$\pi'_i = \gamma\pi_i + (1-\gamma)[\theta_i\beta_i + \sigma_i(1-\beta_i)]. \tag{1}$$

This assignment tends to explore the input domain by progressively moving tests to partitions where unreliability contribution was still small; this allows detecting hard-to-detect faults after easier ones. The β_i factor regulates the impact of failure rate and similarity in determining π'_i. The smoothness of the adaptation is determined by the parameter $\gamma \in [0, 1]$, regulating how the algorithm considers past iterations' results with respect to current ones. The π_i values are then normalized so as to sum up to 1. Starting from π'_i, $|T'_i|$ tests are allocated to partition D_i at the next iteration by a simple procedure to assign, proportionally, more tests to domains with higher π_i values [38], so as: $|T'_i| \approx |T'|\pi'_i$, where $|T'|$ is the number of test cases to be derived in the next iteration, until the testing budget is exhausted. Hence, at each iteration a vector AL is obtained: $AL = \{|T'_1|, |T'_2|, \ldots |T'_m|\}$. The best $|T'|$ is computed by an *adaptive* implementation of Importance Sampling [39]. Based on desired error and confidence, this IS variant progressively reduces the number of required samples as more information becomes available, using the formula:

$$|T'| = \frac{1}{2\xi}\chi^2_{\rho-1,1-\delta} \approx \frac{\rho-1}{2\xi}\{1 - \frac{2}{9(\rho-1)} + \sqrt{\frac{2}{9(\rho-1)}}z_{1-\delta}\}^3 \tag{2}$$

where:

ξ	error we want to tolerate between the sampling-based estimate and the true distribution;
$1-\delta$	desired confidence in this approximation;
ρ	number of partitions from which at least one test case has been drawn in the previous iteration;
χ^2	chi-square distribution with $\rho - 1$ degrees of freedom evaluated with significance level δ;
$z_{1-\delta}$	normal distribution evaluated with significance level δ.

The algorithm is triggered by an initial static allocation of a small number of tests to start up the algorithm [40]. Several strategies can be chosen, depending on the initial knowledge about failure likelihood of partitions (e.g., via expert judgment about partition criticality). Assuming no such initial knowledge, tests can be allocated following the traditional OT, i.e., proportionally to the expected usage of

the partition – giving more tests to partitions whose inputs are expected to be more exercised; this is the choice in our experiments. In general, the bigger the initial number of test cases, the better the initial learning can be, but the later the adaptation will start. In the experiments, we opted for a number of initial test cases equal to the number of partitions.

Summarizing, the output of the allocation step is the computation of the best number of test cases $|T'|$ to run and their distribution to partitions, AL. Next, we detail how tests are selected or generated within partitions.

3.4 Selection of test cases within a partition

The selection step cares about picking the test cases for partitions among those not yet executed (without-replacement selection). Algorithm 1 sketches *covrel+* procedure for allocation plus selection. TS denotes the list of test cases selected from the suite T. TS_i is the sublist selected from partition D_i and $|TS_i| = |T'_i|$ as suggested by the allocation step.

In the first iteration the selection within each partition D_i is random as test cases are equally ranked (line 4). Within the loop on partitions (line 7-11), the highest ranked test cases for D_i are selected to compose the current TS_i subset.

The learning phase takes place during execution of tests for partitions (procedure *RunTests*, called at line 9), updating the failure rate based on observed results (line 18) and the cumulative count spectrum (CCS) derived while tests are executed (line 19). After tests execution and learning, *covrel+* evaluates the remaining test cases and re-ranks them according to *how* they cover the program entities using the count spectrum and coverage information (line 12). Then, the next allocation vector AL is computed using the updated failure rate (FR, line 13), suggesting how many tests should be selected in the next step: $AL = \{|TS_1|, |TS_2|, \ldots |TS_m|\}$.

To increase the chances to find "difficult" failure points, the aim is to select test cases that cover entities so far *rarely* exercised. The test case rank is computed accordingly, by assigning weights to the *importance groups*. For the three groups *high*, *medium* and *low* used in the experiments in Section 4, the chosen criterion is to assign weights so that the *high* and the *medium* groups are one order of magnitude less important than the *medium* and the *low* group, respectively. When multiple test cases achieve the same rank, one of them is randomly selected.

Algorithm 1: *covrel+* test case selection

Input : T, the test suite from which test cases can be selected;
CT, coverage information of the existing test cases;
OP, operational profile;
$Budget$, maximum number of test cases;

Output: TS, the list of test cases selected

```
1  TS ← EmptyList()
2  FR ← FailureRate(∅)                          ▷ FR is the vector of φ_i
3  CCS ← CumulativeCountSpectrum(∅)
4  AL ← IS-basedAllocation(OP)                  ▷ First alloc. based on OP
5  RTC ← T
6  while |TS| < Budget do
7      foreach (D_i) do
8          TS_i ← Select(RTC, AL_i)             ▷ select test cases for D_i
9          RunTests(TS_i)                       ▷ execute and learn
10         TS ← TS ∪ TS_i                       ▷ add selected to the output list
11         T ← T − TS_i                         ▷ remove (if select w/o replacement)
12     RTC ← RankTestCases(T, CCS, CT)          ▷ re-ranking
13     AL ← IS-basedAllocation(OP, FR)          ▷ next allocation
14 return TS
15 procedure RunTests(TS_i)
16     foreach (tc ∈ TS_i) do
17         result, trace ← RunTestCase(tc)
18         FR ← UpdateFailureRates(FR, result)
19         CCS ← UpdateCumulativeCountSpectrum(CCS,
             trace)
```

Selection walk-through: The following example shows how *covrel+* selects test cases. Assume that iteration n received the allocation $AL = \{P_1 : 0; P_2 : 4; P_3 : 0\}$, and resulting failure rates are $FR = \{P_1 : 0.1; P_2 : 0.75; P_3 : 0.45\}$. At iteration $n{+}1$ the IS-based procedure (Algorithm 1, line 4) defines the allocation $AL = \{P_1 : 1; P_2 : 9; P_3 : 6\}$, i.e., 1 test case to partition P_1, 9 test cases to partition P_2, and 6 to P_3. Then, based on the cumulative count spectrum computed at iteration n, entities are assigned to different importance groups: precisely, e_2, e_5, and e_8 are considered to belong to the *low* importance group, e_4, e_7, and e_9 to the *medium* one, and e_1, e_3, and e_6 to the *high* one. To select 1 test case for partition P_1, the algorithm (line 8) considers the ranks of the remaining test cases from the test suite T that belong to partition P_1. Assume these test cases are TC_1, TC_2, and TC_3, and their respective entities coverage is as displayed in Table 1. To calculate the rank of a test case (TCR) the algorithm computes the weighted sum of covered entities, assigning the weights 1, 10, and 100 to entities in the *high*, *medium*, and *low* group, respectively:

$$TCR = (100 \cdot n_l) + (10 \cdot n_m) + (1 \cdot n_h), \qquad (3)$$

where n_l, n_m and n_h are the number of covered entities belonging to groups *low*, *medium* and *high*, respectively. Thus TC_2, which covers 3 entities from the low importance group, 1 from the medium, and 1 from the high, is given the rank $311 (= 3{*}100{+}1{*}10{+}1{*}1)$; as this is the highest rank, TC_2 is the test case selected to be run for partition P_1. After TC_2 is run (line 17), the failure rates vector FR and the cumulative count spectrum are updated accordingly (lines 18 and 19). These steps are then repeated for partitions P_2 and P_3, concluding iteration $n + 1$.

TABLE 1
Computation of ranks for test cases in partition P_1 at iteration $n + 1$.

Entity	CCS	Importance Group	TC_1	TC_2	TC_3
e_1	206	high	x	x	x
e_2	6	low	x	x	
e_3	221	high	x		x
e_4	109	medium	x		
e_5	22	low		x	x
e_6	209	high	x		
e_7	114	medium		x	
e_8	4	low		x	
e_9	178	medium	x		
		Test Case Rank:	123	311	102

3.5 Generation of test cases within a partition

The procedure followed in test case allocation and generation is sketched in Algorithm 2. TG denotes the list of generated test cases. TG_i is the sublist generated from partition D_i and $|TG_i| = |T'_i|$ as suggested by the allocation step.

Algorithm 2: *covrel+* test case generation

```
Input : OP, the operational profile;
        Budget, maximum number of test cases;
Output: TG, the list of test cases generated
1  TG ← EmptyList()
2  FR ← FailureRate(∅)                    ▷ FR is the vector of φ_i
3  CCS ← CumulativeCountSpectrum(∅)
4  PCS ← PartitionCountSpectrum(∅)
5  SIM ← Similarity(PCS, CCS)             ▷ SIM is the vector of σ_i
6  AL ← IS-basedAllocation(OP)            ▷ First alloc. based on OP
7  while |TG| < Budget do
8      foreach (D_i) do
9          TG_i ← GenerateTests(AL_i)     ▷ generate cases for D_i
10         RunTests(TG_i)                 ▷ execute and learn
11         TG ← TG ∪ TG_i                 ▷ add generated to the output list
12     SIM ← UpdateSimilarity(PCS, CCS)
13     AL ← IS-basedAllocation(OP, FR, SIM)
14 return TG
15 procedure RunTests(TG_i)
16     foreach (tc ∈ TG_i) do
17         result, trace ← RunTestCase(tc)
18         FR ← UpdateFailureRates(FR, result)
19         CCS ← UpdateCumulativeCountSpectrum(CCS,
             trace)
20         PCS ← UpdatePartitionCountSpectrum(PCS,
             trace)
```

Initially, *covrel+* allocates test cases to partitions according to OT (line 6). The generation of test cases for D_i (line 9), is done by interfacing *covrel+* with a test generation engine (the one we used is explained in Section 4.3.4). The amount of test cases to be generated is defined by the IS-based procedure in the allocation step (line 13). Notice that *covrel+* is not meant as an original test case generation strategy and, in principle, any existing test case generation tool or strategy can be used (e.g., Randoop [32], Dynamic Symbolic Execution [41], Evosuite [31]), as long as it can be guided towards generating test cases from a specified partition. The budget to partitions for the next iteration is computed by the adaptive allocation procedure (line 13) by combining the OT information with data about *i)* the failure rates in previous iterations, and *ii)* the *similarity score*.

The computation of the similarities (line 12) consists of:

- computation of a *cumulative count spectrum* of code coverage for all tests executed in previous iterations;
- identification of the set of least covered entities ("low" entities) from the cumulative count spectrum;
- characterization of each partition with a *partition count spectrum* (PCS), which collects the spectra of test cases already executed for the partition;
- measurement of the similarities between the spectrum of "low" entities and the count spectrum of D_i.

Generation walk-through: Let us assume that iteration n received allocation $AL = \{P_1 : 4, P_2 : 6, P_3 : 5\}$ and, at the end of it, the failure rates were $FR = \{P_1 : 0, P_2 : 0, P_3 : 1\}$. Table 2 displays the cumulative count spectrum as well as the partition count spectra for partitions P_1, P_2, and P_3 after iteration n. According to the CCS, three entities are considered to belong to the *low* group: e_2, e_5, and e_8. Based on the PCS for partition P_1, we can tell that the set of test cases that belong to that partition have covered all of the three *low* entities, which yields a similarity of 100% when compared with the set of *low* entities from the CCS. The PCS for partition P_2 did not cover any *low* entity and it is assigned a 0% similarity with the CCS. Finally, the PCS for partition P_3 has covered one out of three *low* entities, yielding a similarity of 33% with the CCS. The similarity vector (Algorithm 2, line 12) is thus defined as $SIM = \{P_1 : 1.0, P_2 : 0, P_3 : 0.33\}$; it is used by the IS-based procedure (line 13) to define the allocation AL for the iteration $n + 1$. Let us now assume that the allocation for iteration $n + 1$ is defined as $AL = \{P_1 : 26, P_2 : 2, P_3 : 18\}$. When the `GenerateTests` procedure is called (line 9), it interfaces with the test generation engine to provide exactly the number of test cases allocated for each partition, i.e., 26 test cases for partition P_1, 2 for P_2, and 18 for P_3.

TABLE 2
Cumulative count spectrum and partition count spectra (PCS) for partitions P_1, P_2, and P_3 after iteration n.

Entity	Cumulative Count Spectrum	Importance Group	PCS (P_1)	PCS (P_2)	PCS (P_3)
e_1	206	high	21	157	28
e_2	6	low	6	-	-
e_3	221	high	18	118	85
e_4	109	medium	14	28	67
e_5	22	low	22	-	-
e_6	209	high	43	85	81
e_7	114	medium	28	72	14
e_8	4	low	1	-	3
e_9	178	medium	85	5	88
		Similarity:	100%	0%	33%

3.6 Overhead analysis

We analyze the factors determining the overhead (in terms of CPU time) of the *covrel+* strategy. Both variants of *covrel+* use an instrumented version of the program to collect coverage information, which entails an execution time overhead like any coverage-based technique. The latter is known to vary between 10% and 30%, depending on the instrumentation tool, the language, the coverage criterion and the subject under test [42], [43].

Besides this, the overhead factors specific to our approach include: *i)* the importance sampling time, which is computed both for test selection and test generation; *ii)* the time for similarity computation (only for the test generation). Both functions take arrays of data whose size is exclusively dependent on the number of partitions and they have no other dependency on the specific subject under test. Therefore, in the following we assess the time needed for both algorithms under a varying number of partitions.

3.6.1 Importance Sampling overhead

We ran the IS algorithm on a number of partitions ranging from a minimum of 2 to a maximum of 100. Each run is repeated 100 times, and basic statistics are collected (mean, median, standard deviation, semi-interquartile range). Figures 2 and 3 plot the results; we can see that: *i)* the overall average time for one execution of the IS algorithm is less than a millisecond (8.60E-5 seconds); *ii)* the time does not

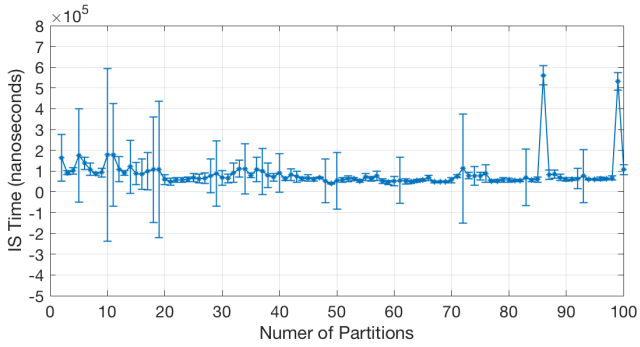


Fig. 2. Importance Sampling time: average and standard deviation

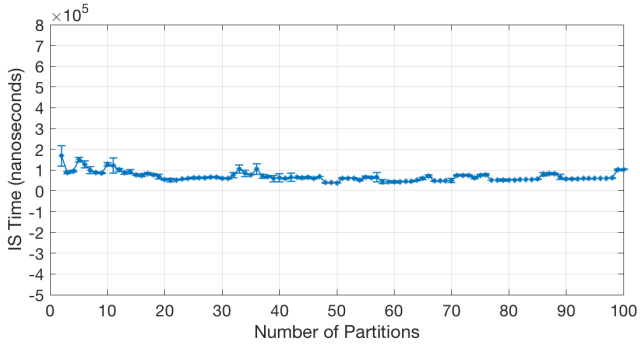


Fig. 3. Importance Sampling time: median and SIQR

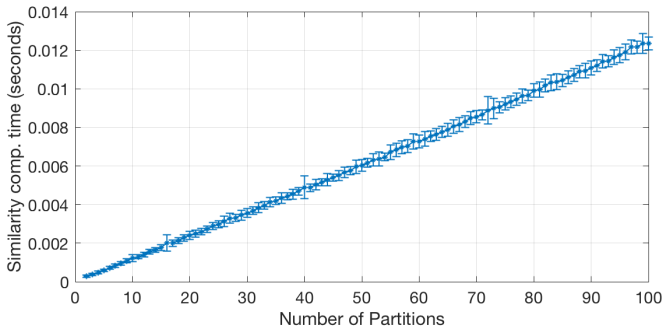


Fig. 4. Similarity computation time: average and standard deviation

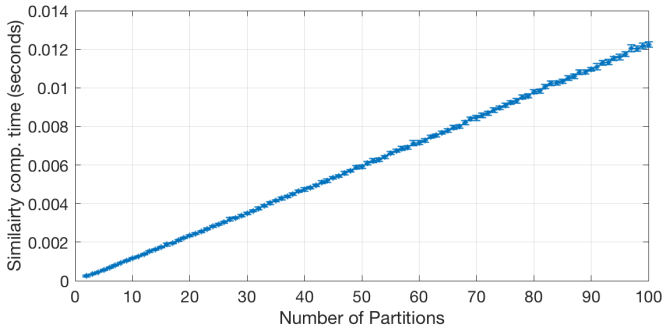


Fig. 5. Similarity computation time: median and SIQR

increase with the number of partitions; *iii)* the standard deviation in most cases is small too (in the average, it is 4.78E-5 seconds, hence the coefficient of variation is about 55%); *iv)* the median and semi interquartile range (SIQR), which are more stable indicators with respect to outliers, confirm the trends, giving smaller values than the average/standard deviation pair (the mean of medians is 6.91E-5 seconds and the mean of SIQR is 5.65E-6 seconds).

The IS algorithm is invoked at each iteration in a run. The algorithm is expected to converge in few iterations, the exact number depending on the desired error between the approximated and true distribution and on the confidence in the approximation (namely, on ξ and δ in Equation 2). For instance, in a setting with error ξ=0.1, confidence δ=0.05 and 5 partitions, we had between 4 and 5 iterations.

3.6.2 Similarity computation overhead

Like the IS case, we ran the similarity computation algorithm on a number of partitions ranging from 2 to 100, each repeated 100 times. Figures 4 and 5 plot the results. We can see that: *i)* the similarity computation time varies linearly with the number of partitions, *ii)* the time varies from 2.78E-4 seconds with 2 partitions to 1.24E-2 seconds with 100 partitions; *iii)* the standard deviation in most cases is relatively small (2.74E-4 seconds in the average) and slightly increasing with the number of partitions (from 5.72E-5 to 7.07E-4 seconds), but always keeping a small coefficient of variation, between 2.5% and 21.24%; *iv)* the median and SIQR confirm the trend, giving smaller values than the average/standard deviation pair (the medians going from 2.51E-4 to 1.22E-2 seconds and the mean of SIQR being 8.16E-5 seconds). Similarity is also computed at each iteration.

3.6.3 Trade-off analysis

The temporal overhead in a testing session caused by the IS algorithm and the similarity computation time are worth to be incurred depending on the gain yielded by *covrel+*.

Let us denote the average test case execution time for a single test as t, the IS time as t_{IS}, and the similarity computation time as t_{SC}. To compare *covrel+* against a generic coverage-based technique (using the same instrumentation technique/tool) in a testing session, we use the following Equation to express the points in which the additional overhead of *covrel+* and its gain with respect to the competing technique are equal:

$$(t_{IS} + t_{SC}) \cdot K = Q \cdot t \quad (4)$$

where K is the number of iterations, Q is the number of test cases saved by applying *covrel+*, and $Q \cdot t$ the total time saving. The first term is the total additional overhead; the second term is the gain. Under this Equation, applying *covrel+* is better if the gain is bigger than the overhead.

For instance, fixing the test case execution time t, *covrel+* is advantageous if the number of saved test cases is at least: $Q > \frac{(t_{IS}+t_{SC})\cdot K}{t}$. Similarly, if we solve it with respect to t, we have the minimum average test case execution time that makes *covrel+* advantageous: $t > \frac{(t_{IS}+t_{SC})\cdot K}{Q}$.

If we compare our approach with a technique not based on coverage information, then we need to include the instrumentation overhead too. For instance, let us assume the instrumentation entails an overhead equal to 10%. The previous equation becomes: $(t_{IS} + t_{SC}) \cdot K + 0.1 \cdot N \cdot t = Q \cdot t$, with N being the number of executed test cases. In this case, *covrel+* is better than the alternative technique if: $Q > \frac{(t_{IS}+t_{SC})\cdot K}{t} + 0.1 \cdot N$ (or if $t > \frac{(t_{IS}+t_{SC})\cdot K}{Q-0.1\cdot N}$).

4 EVALUATION

4.1 Objective

We assess the performance of *covrel+* against OT considering both the *test selection* and *test generation* settings. We choose OT as a baseline because it is a well-known and intuitive strategy, which like *covrel+* aims at improving reliability. However, it has to be considered that as *covrel+* exploits additional information, it also exhibits higher costs (discussed in subsections 3.6.1 and 3.6.2). Hence, the comparison of *covrel+* performance against OT should be interpreted also in the light of the above presented trade-off analysis (subsection 3.6.3).

The following research questions are targeted:

- **RQ1**. *Is covrel+ more efficient than OT (in terms of number of tests required) at achieving a target reliability value?*
- **RQ2**. *Does covrel+ detect more faults than OT under the same testing budget?*

It is known that as testing proceeds, the number of faults detected per effort unit progressively decreases: testing uncovers typically many faults initially, and the detection of residual faults becomes harder and harder. Hence to characterize the ability of our strategy to detect hard-to-detect faults, we formulate the third research question:

- **RQ3**. *How does covrel+ perform compared to OT as testing proceeds?*

4.2 Subjects

Past studies on testing techniques based on operational profile have used for experimentation three types of relatively small subjects: *i)* programs from business-/mission-critical domains, such as telecommunication, space and nuclear systems (used in [1], [5] and [44]), respectively); *ii)* example programs with faults and test suites already available, such as the `space` program by ESA (used in [22], [45], [7]), and the similar `orecolo` and `autopilot` programs (used in [25]); *iii)* subjects available from the SIR repository[2], such as gzip and GCC (used in [4]). More recently, operational testing has been applied also to Web applications, with failure data inferred in various ways (e.g., from log data) [46], [47], [48].

Experimenting with *covrel+* requires the availability of source code, faults and operational profile. Subjects including all such features (e.g., industrial systems) are not freely available and artificially constructing them (e.g., on large open source applications) in a representative way would be hard. For these reasons, in the present study we opted for repeatability and verifiability of controlled experiments on subjects widely used in software engineering research, as in case *iii)* above. We considered the following programs from SIR: grep, gzip, flex, and sed. Grep is a command-line utility for searching lines matching a given regular expression in the provided file(s); gzip is an application used for file compression and decompression; flex is used for generating scanners that are able to recognize lexical patterns in text; sed is a stream editor that performs text transformations on an input stream. The first three of these programs are available in five versions containing seeded faults, whereas sed contains seven versions. All of them have test suites derived through the category-partition method [49], available as *tsl* (test specification language) files. These are used for evaluating *test selection*, where test cases are selected from the test suites.

TABLE 3
Study subjects selected from the SIR repository

Subject	LoC	SIR test suite	Seeded faults	Detectable faults
gzip v1	4,594	214	16	7
gzip v2	5,083	214	7	3
gzip v4	5,233	214	12	3
gzip v5	5,745	214	14	5
flex v1	9,558	567	19	16
flex v2	10,274	670	20	13
flex v3	10,296	670	17	9
flex v4	11,447	670	16	11
grep v1	9,463	809	18	5
grep v2	9,987	809	8	4
grep v3	10,124	809	18	8
grep v4	10,143	809	12	3
sed v2	9,867	360	5	5
sed v3	7,146	360	6	6
sed v5	13,398	370	4	4
sed v6	13,413	370	6	6
Total:	145,771	8,129	198	108

After a preliminary analysis we excluded from the study the versions whose test suite could not identify any of the seeded faults available for them. This happened for grep v5, gzip v3, and sed v1. We also excluded version 5 of flex due to its anomalous characteristics: 3 faults could be revealed by more than 80% of the test cases, and 2 others could be revealed by ≈99% of tests; the vast majority of the test cases available in the test suite would be able to reveal 100% of the seeded faults – so all faults would be detected after few tests. After these exclusions, 18 versions remained. During the execution of the *test generation* study, no faults could be revealed for sed v4 and v7 by all the test cases generated. Hence we do not report results for these two versions.

The final remaining 16 study subjects and the related details are listed in Table 3. Column "LoC" shows the number of lines of code of each subject.[3] The column "Detectable faults" contains the number of faults, from the set of seeded faults, that could be detected by the SIR test suite.

Both OT and *covrel+* distribute test cases to different partitions. The partitioning criterion for these subjects is based on functionalities. We inspected the *tsl* (test specification language) file made available with the subject (which specifies the function units and their input space features in terms of parameters, categories and choices, according to category-partition testing terminology) along with the user manual, in order to infer the main functionalities. Each functionality is associated with a partition; we ended up with 4 partitions for grep, 5 for gzip, 3 for sed, and 6 for flex. On these subjects, a *fault matrix* – i.e., a mapping of which faults can be revealed by which test cases of the test suite - is built for evaluating the approach in the *test selection* setting. In particular, for each subject and version, we run the available test suite and, with the support of SIR tools,

2. SIR is the *Software-artifact Infrastructure Repository*, widely used for controlled experimentation about testing and program analysis, available at: http://sir.unl.edu/portal/index.html.

3. Collected using the CLOC utility (http://cloc.sourceforge.net).

extracted the fault matrix. This does not apply for the *test generation* case, since the test suite from SIR is not used.

4.3 Experimental methodology

4.3.1 Compared techniques

We compare three variants of *covrel+* against the partition-based OT strategy described in the Introduction. The variants of *covrel+* are based on three different coverage criteria, namely: *branch*, *function* and *statement* coverage.

4.3.2 Metrics

We evaluate the techniques using the following metrics.

The efficiency in achieving a desired reliability level (RQ1) is measured in terms of *number of executed test cases to detect a given amount of faults*. In particular:

- For *test selection*, we count the *number of test cases required to reveal all the faults detectable by that test suite* (see column *Detectable faults* in Table 3). We denote with $N_{covrel+}$ and N_{OT} the number of test cases required by *covrel+* and the OT baseline, respectively.
- For *test generation*, we count the *number of test cases required to reveal a same amount of faults by the generated test cases*. Since the full set of seeded faults (column *Seeded faults* in Table 3) might be detected in unacceptable (and unpredictable) long time, the amount of faults to be detected is determined as the number of faults detected by a fixed number of first 1,000 test cases. Namely, 1,000 test cases are first executed for each technique: then, the technique that detects more faults stops its execution, while the other technique keeps executing until it detects the same amount of faults. The cost incurred by the latter technique (i.e., the looser) is the *additional number of test cases required to detect the same amount of faults*. The cost of the former one (i.e, the winner) is zero.

To answer RQ2 and RQ3, we count the *number F of faults detected by the same number T of executed test cases*. In particular:

- For *test selection*, we adopt the following criterion to fix the number of test cases: given the number of test cases to detect all detectable faults by *covrel+* and by OT ($N_{covrel+}$ and N_{OT}), we take the minimum T_{Total} between the two, and consider percentages of them (e.g., 10%, 20%, ..., 100%). Thus, $T = x\% \cdot T_{Total}$, with $x \in [1, 100]$. This allows computing the failure intensity as testing proceeds (for RQ3) and when it stops, i.e., when x= 100% (for RQ2).
- For *test generation*, we adopt the following criterion to fix the number of test cases: *covrel+* is executed first, and testing ends when either all the seeded faults have been detected or a maximum number of test cases (set to T_{max}=1,000) has been executed. OT testing is then run with the same amount of tests T_{Total} as *covrel+*. Similarly to the previous case, $T = x\% \cdot T_{Total}$, with $x \in [1, 100]$, and when x= 100%, the final failure rate is obtained.

4.3.3 Experiments

Three controlled experiments have been designed to address the research questions. The first experiment evaluates *covrel+* against OT for the *test selection* problem. In this case, testing ends when all faults detectable by the available test suite have been detected (these are generally not all the seeded faults). This experiment suffices for test selection, since the number of test cases required to compute the metrics for answering RQ2 and RQ3 (T_{Total}) is derived from $N_{covrel+}$ and N_{OT}, which are the metrics to answer RQ1.

The second and third experiments evaluate *covrel+* against OT for the *test generation* problem. In this case, one experiment does not suffice to address all research questions, as we do not know if and when the undetectable faults (namely, Seeded minus Detectable faults) will be detected by the generated test cases. Hence:

- One experiment is executed by fixing a number of faults to detect (between 0 and the number of seeded faults – for each subject, shown in Table 3) and running tests until that number of faults is detected. This will allow answering RQ1;
- One more experiment is executed by fixing the number of test cases and looking at how many faults are detected. This will allow answering RQ2 and RQ3.

In each experiment, both *covrel+* and OT are run on all the subjects. Specifically, the three variants of *covrel+* (based on *statement*, *function* and *line* coverage) are executed for each subject. For any subject/variant pair, the execution is repeated 50 times for test selection and 30 times for test generation (because of the greater cost required by the two test generation experiments compared to the test selection experiment) with a new random profile generated for each of such repetitions. After each repetition of *covrel+*, operational testing is run using exactly the same profile used by *covrel+* so as to have a fair comparison. We call *testing session* the experimental run of a single repetition. Thus, the total number of *test scenarios* is: 3 experiments x 2 techniques x 3 variants x 16 subjects = 288 (96 for the test selection experiment, 192 for the two test generation experiments). The total number of testing sessions is 96 scenarios x 50 repetitions plus 192 scenarios x 30 repetitions, namely: 4,800 + 5,760 = 10,560 testing sessions.

4.3.4 Experimental procedure and parameter values

The automated procedure followed in a testing session consists of these steps:

1) Generate the operational profile;
2) Repeat:
   a) Select/generate the next test case for the technique under evaluation (*covrel+* or OT);
   b) Execute the test case and observe if it exposes a failure or not;
   c) If a failure occurs, remove the fault(s);[4]

   until T tests are executed (T chosen depending on the research question, as explained in Section 4.3.2);
3) Compute the metrics presented above.

4. Note that for each failure the tester could remove more faults; we choose to remove all the faults, hence the repetition of the test case does no longer lead to failure.

As for allocation, the values for the parameters of the Importance Sampling method (Equations 1 and 2) are set as follows: learning factor $\gamma = 0.5$; confidence $(1 - \delta) = 0.99$; desired maximum error $\xi = 0.1$; discount factor for similarity and failure rate balance $\phi = 0.5$.

For selection, based on the criterion defined in Section 3.4, the weights of the three importance groups are set as follows: $W_{high} = 10^{-1}$, $W_{medium} = 10^{0}$, and $W_{low} = 10^{1}$.

For generating test cases, we developed a script to enable *covrel+* with the ability of requesting an arbitrary number of test cases for any given partition (the script generates input values for our study subjects rather than test code). First we identified all the possible input parameters and environment flags that could influence the behavior of our study subjects (e.g., for gzip, the `-S` and `--suffix` flags can be used to define the suffix to be used for compressed files; the `--fast` and `--best` flags can be used to regulate the speed of compression; and so on). We then investigated which input data, if any, was required by the subject (e.g., gzip can receive a file to be compressed, decompressed, or tested) and created the necessary support files (for gzip we created directories containing multiple files to be compressed; multiple compressed files to be tested or decompressed; etc). Finally, once we identify which flags and parameters belong to which partition, our script can generate random test cases by choosing the necessary input variables and combining them with arbitrary input data to be used by the subject (e.g., if gzip has a partition dedicated to the decompression of files, all test cases generated for such partition should always contain either the `-d` or the `--decompress` flag).

We assess the statistical significance of results by means of appropriate tests. To determine if there is a significant difference between techniques and to separate out the effect of subjects we use the *Friedman test*. This is a non-parametric test for repeated (hence dependent) data measures; it does not assume normality of observations, homoscedasticity of variances, independence of data among compared samples, and it works well under balanced designs as ours.

To measure the "effect size" we assess the magnitude of the difference adopting the *Vargha and Delaney test* [50], as suggested in [51], using the $\hat{A}_{12}(x, y)$ statistic. The latter represents the probability that the metric's value for technique x is greater than for technique y – namely, the probability that a randomly selected observation from one sample is bigger than one randomly selected from the other sample.

5 RESULTS

We now report and discuss the results of the experiments. With the aim of supporting the independent verification and replication, we make available the artifacts produced as part of this work.[5] The replication package includes the implementation of the algorithms, input data, raw data used for the statistical analyses, and additional results.

5.1 RQ1: Testing efficiency

5.1.1 Test Selection

Figure 6 shows, for each scenario, how many times *covrel+* required less test cases than OT to detect all the faults detectable by the provided test suite (last column in Table 3) – i.e., *covrel+* "wins" the comparison; how many times OT required less test cases (OT wins), and how many times they required the same amount of tests (ties). In the majority of cases (38 out of 48 scenarios), *covrel+* wins more often than OT. However, this information is not enough, because OT could win less often than *covrel+* but with a great number of saved test cases at each win – meaning that it could be advantageous in some specific scenarios. Therefore, we also report in Table 4 the average, over the repetitions in each scenario, of the number of test cases required by a technique to detect all detectable faults.

The results show that *covrel+* requires generally less test cases than OT (the *covrel+* average values are lower than OT ones in 40 out of 48 cases). Looking at columns' means and medians, this is true for all the three variants of *covrel+*, with *branch*, *function* and *statement* coverage. The differences are remarkable: OT required, in the average, 76 more test cases than *covrel+* for the *branch* criterion case, 45 more test cases for *function* and 76 more test cases for *statement*. Hence, *branch* and *statement* coverage allow saving more tests.

TABLE 4
Test Selection: Average number of executed test cases to detect all detectable faults

Subject	*Branch*		*Function*		*Statement*	
	covrel+	OT	*covrel+*	OT	*covrel+*	OT
gzip v1	118.40	143.20	137.70	177.50	123.74	144.12
gzip v2	28.22	13.42	10.66	25.48	28.64	14.46
gzip v4	25.62	153.38	25.18	153.92	24.70	150.88
gzip v5	66.16	129.50	62.42	127.14	60.20	134.76
flex v1	15.86	25.14	17.56	28.16	18.12	24.32
flex v2	354.32	501.84	275.78	487.12	314.30	474.28
flex v3	611.38	399.26	542.34	489.10	621.82	437.26
flex v4	227.62	314.28	95.88	351.16	256.52	303.72
grep v1	129.72	602.16	303.46	626.68	112.78	575.54
grep v2	321.34	195.32	622.46	169.80	357.56	190.58
grep v3	156.36	179.32	72.0	159.6	96.74	171.08
grep v4	99.70	509.72	610.68	461.86	107.70	483.54
sed v2	113.50	185.98	84.18	223.76	72.06	154.42
sed v3	43.90	104.20	61.62	97.40	42.20	114.14
sed v5	10.80	23.26	10.58	20.50	10.20	22.02
sed v6	47.32	112.72	70.20	127.12	60.10	132.18
Mean	**148.14**	**224.54**	**187.67**	**232.89**	**144.21**	**220.46**
Median	106.6	166.35	78.09	164.7	84.4	152.65

Figure 7 shows the results by subject, averaged over the three criteria, highlighting the percent relative differences between *covrel+* and OT (labeled with the absolute values). *Covrel+* is more costly than OT in 3 out of 16 cases. These are grep v2, flex v3 and gzip v2. In all the other cases (13 out of 16), *covrel+* is better. In 10 of these 13 cases, the percentage of test cases required by *covrel+* over the sum of test cases (*covrel+* and OP) ranges from 30% to 40%.

It is worth investigating the three cases where OT turned out to be more effective than *covrel+*. Inspecting results, we conjecture that the combination of the following causes make OT to behave better than *covrel+* for those subjects: *i)* the tuning of the IS algorithm parameters of *covrel+* (namely, desired error and confidence), which determine the speed at which the "focus" (in terms of number of allocated test cases) is shifted towards more failure-prone regions, coupled with *ii)* a highly skewed distribution of faults in those subject, with a concordant operational profile. The three

5. http://labsedc.isti.cnr.it/covrel/.

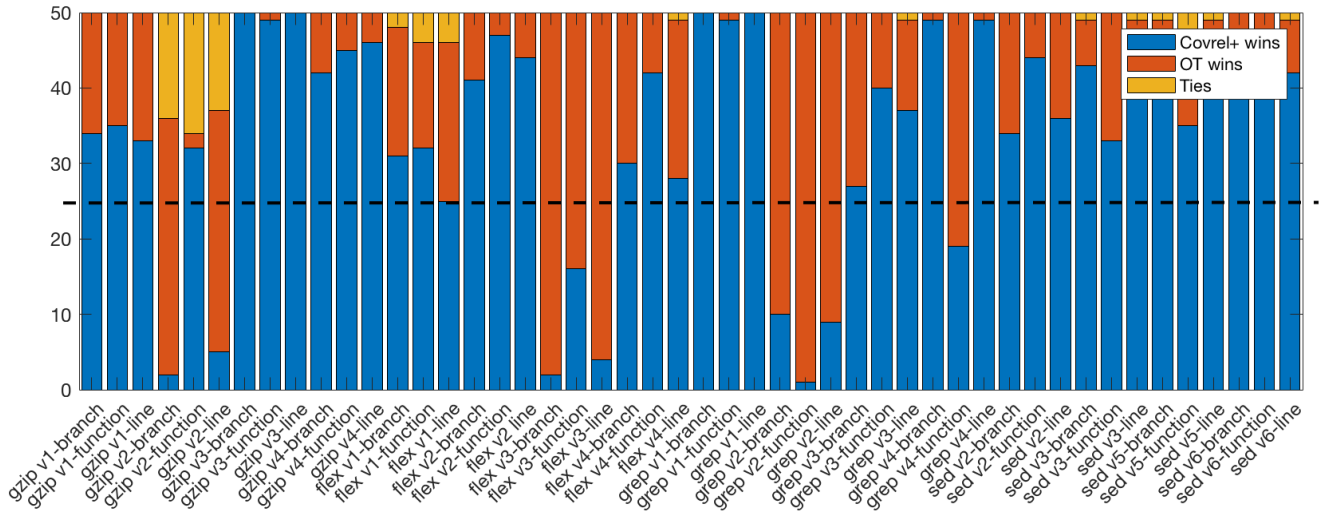


Fig. 6. Test Selection: number of wins of *covrel+* and *OT*

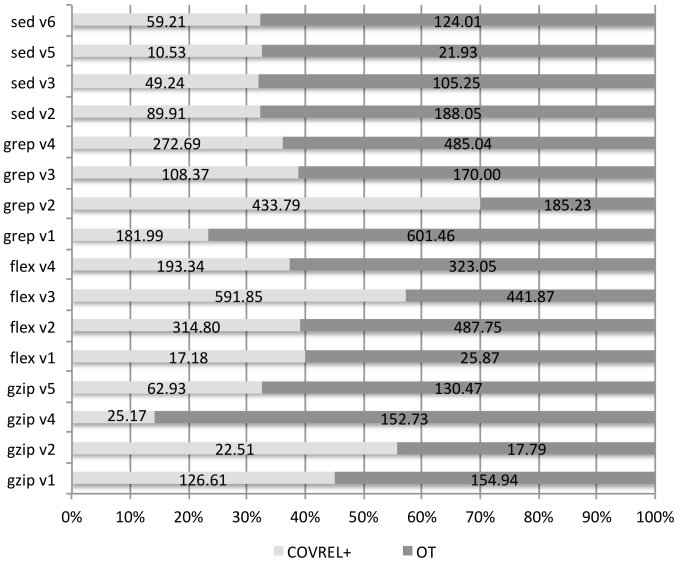


Fig. 7. Test Selection: Additional test cases by subject

subjects have all faults concentrated in one or few partitions and, at the same time, the profile has high selection probability for that partition; this is the best scenario for OT. In such cases, the IS procedure (which tries to give chances also to other partitions so as to gauge their failure rate) allocates some more tests before identifying the critical partition(s), depending on the tuning of its parameters. These should be tailored for the subject under test.

Table 5 reports the results of the Friedman test (the *p-value*) and the effect size (the $\hat{A}_{1,2}$ statistic). The difference between *covrel+* and OT is statistically significant by a large extent; the effect size shows that the probability for a randomly selected value from the *covrel+* sample to be greater than a value from OT sample is 0.2324 – in this case, the lower the better for *covrel+*.

5.1.2 Test Generation

Similarly to test selection, Figure 8 shows, for each scenario, how many times *covrel+* required less test cases than OT

TABLE 5
Test Selection: Hypothesis test. Number of test cases required to detect all detectable faults

Pairwise Comparison		
	covrel+	OT
Mean	160.01	225.96
Median	90.03	157.01
p-value	**3.01 E-93**	-
Effect size	**0.2324**	-

to detect the same amount of faults, how many times OT required less test cases, and how many they required the same amount of tests. Even if still outperformed by *covrel+*, OT improves compared to the test selection experiment, as in 17 of 48 scenarios it wins more often than *covrel+*.

Table 6 reports the additional number of test cases (with respect to 1,000, see Section 4.3.2) spent by a technique to detect the same amount of faults as the other technique. Average values are lower for *covrel+* than for OT in 36 out of 48 cases. From mean and median values, it is evident that *covrel+* requires generally less test cases than OT for a given quality objective (amount of faults to detect). This is true for all the three variants of *covrel+*, and the differences are remarkable: OT required, on average, 236 more test cases than *covrel+* with the *branch* coverage criterion, 123 more with *function* and 145 more with *statement*.

Figure 9 shows the results by subject, averaged over the three criteria, highlighting the relative difference (in percentage) between *covrel+* and OT (labeled with the absolute values). *Covrel+* is less costly than OT in 12 out of 16 cases. In 2 cases (*gzip v1* and *v5*) OT requires, on average, many test cases less; in the remaining 2 cases (*gzip v2* and sed v6) OT is better by a small margin. The worst cases for *covrel+* are explained by the same reasons we discussed for the test selection experiment. While some subjects are the same as in the test selection experiment, other are different (e.g., sed instead of flex): this is due to the difference in the test suite, being test cases generated from scratch in this experiment.

Table 7 reports the results of the Friedman test and the effect size for test generation, which are similar to those of Table 5 for the problem of test selection.

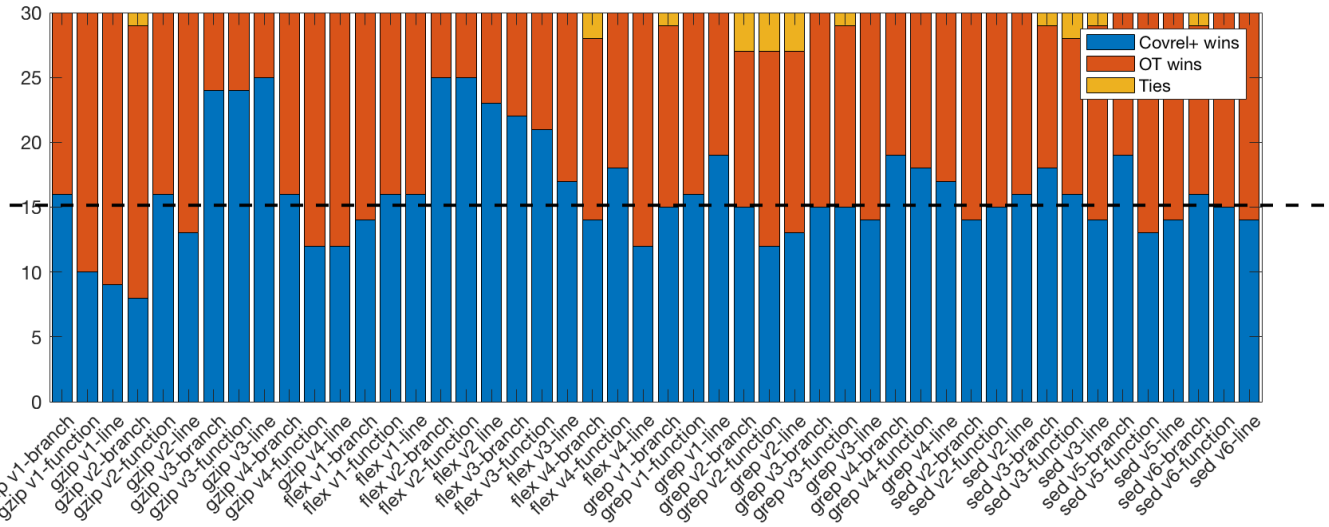


Fig. 8. Test Generation: number of wins of *covrel+* and OT

TABLE 6
Test Generation: Additional average number of executed test cases to detect a same amount of faults

Subject	*Branch*		*Function*		*Statement*	
	covrel+	OT	*covrel+*	OT	*covrel+*	OT
gzip v1	973.83	911.43	1217.27	441.87	976.40	198.43
gzip v2	8.00	3.33	5.60	7.53	4.43	5.50
gzip v4	37.40	904.1	40.57	739.23	27.40	1416.20
gzip v5	55.50	38.29	81.47	56.17	84.27	56.70
flex v1	22.90	35.57	19.13	34.17	28.30	32.77
flex v2	50.40	769.23	27.47	1047.30	32.13	533.20
flex v3	34.90	400.47	33.40	565.66	65.97	110.23
flex v4	9.27	12.30	8.63	21.73	17.47	11.30
grep v1	52.50	222.60	41.50	243.60	39.72	215.63
grep v2	2.29	2.43	2.23	1.83	2.23	2.87
grep v3	20.63	32.43	19.89	22.63	17.77	19.20
grep v4	23.67	40.70	29.83	46.10	18.80	42.80
sed v2	398.60	1158.67	488.70	1129.30	371.40	1122.59
sed v3	6.77	12.90	9.07	6.70	8.03	8.23
sed v5	723.17	1654.70	1372.30	993.37	790.93	1039.37
sed v6	13.53	14.67	8.27	16.40	24.73	11.43
Mean	**152.08**	**388.36**	**212.85**	**335.85**	**156.875**	**301.65**
Median	29.28	39.5	28.65	51.13	27.85	49.75

TABLE 7
Test Generation: Hypothesis test. Number of additional required test cases to detect the same amount of faults

Pairwise Comparison		
	covrel+	OT
Mean	173.93	341.95
Median	27.88	44.45
p-value	**6.81 E-14**	-
Effect size	**0.2652**	-

5.1.3 Answering RQ1

Based on the results achieved in our studies, we can positively answer RQ1 for both test selection and generation: *Covrel+ is significantly more efficient than OT as in the greatest majority of cases it required a lower number of (selected or generated) test cases to find the same number of faults. In particular covrel+ outperformed OT in 13 subjects for test selection, and in 12 subjects for test generation out of the 16 considered ones.*

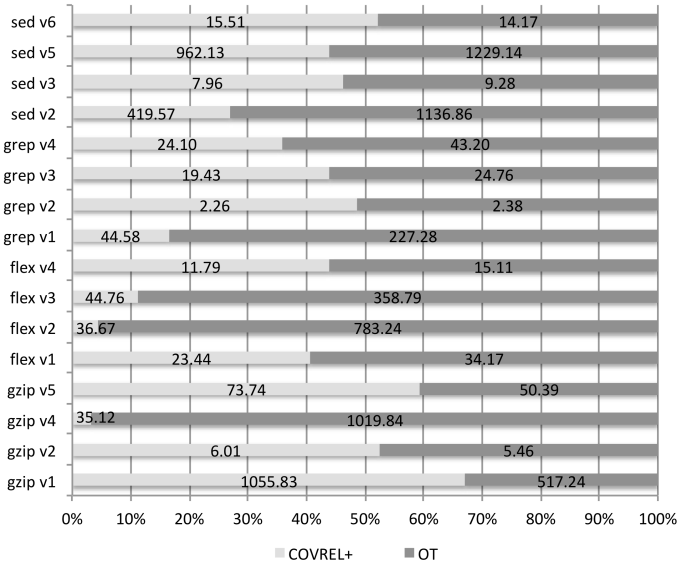


Fig. 9. Test Generation: Additional test cases by subject

5.2 RQ2: Fault detection

5.2.1 Test Selection

Table 8 reports the number of faults detected by an amount of test cases determined as $T = min(N_{covrel+}, N_{OT})$, obtained by the first experiment (the same experiment as RQ1 for test selection). The mean and median values confirm that *covrel+* detects more faults than OT with a same number of test cases. This is true for all the three variants of *covrel+*. Unlike RQ1, in which the *branch* criterion has the highest gain, in this case the highest gain over OT is attained by the *statement* criterion, followed by the *branch* criterion.

The results by subject in Figure 10 show that *covrel+* is more effective than OT at detecting faults in 12 out of 16 cases. In one case (flex v1), the performance are the same; in the remaining 3 cases (grep v2, flex v3, gzip v2, the same as for RQ1) *covrel+* detected less faults in the average.

Table 9 reports the overall average and the results of the Friedman test and the effect size. Note that in this case (RQ2) the greater the effect size value, the better for *covrel+*.

TABLE 8
Test Selection: Average number of faults detected with a same amount of test cases ($T = min(N_{covrel+}, N_{OT})$)

Subject	*Branch*		*Function*		*Statement*	
	covrel+	OT	*covrel+*	OT	*covrel+*	OT
gzip v1	6.52	6.06	6.20	6.46	6.46	6.34
gzip v2	0.82	0.94	0.54	0.74	0.68	0.86
gzip v4	2.34	1.40	2.60	1.56	2.42	1.48
gzip v5	4.70	4.24	4.62	3.96	4.78	4.10
flex v1	3.12	3.18	3.24	3.18	3.28	3.28
flex v2	7.44	6.74	7.58	6.18	7.36	6.44
flex v3	5.50	6.60	5.62	6.26	5.52	6.60
flex v4	3.88	3.74	3.78	3.86	3.84	3.82
grep v1	5.32	2.44	5.22	3.30	5.18	2.50
grep v2	2.96	3.34	2.96	3.34	3.16	3.30
grep v3	4.62	4.64	5.06	4.72	4.82	4.16
grep v4	3.98	2.68	3.66	3.72	3.98	2.68
sed v2	4.10	3.76	4.24	3.28	4.20	3.78
sed v3	5.44	4.70	4.98	4.90	5.06	4.90
sed v5	1.84	1.54	2.06	1.80	1.96	1.64
sed v6	3.62	3.06	3.78	3.50	3.94	2.80
Mean	**4.14**	**3.69**	**4.13**	**3.79**	**4.16**	**3.66**
Median	4.04	3.54	4.01	3.61	4.09	3.54

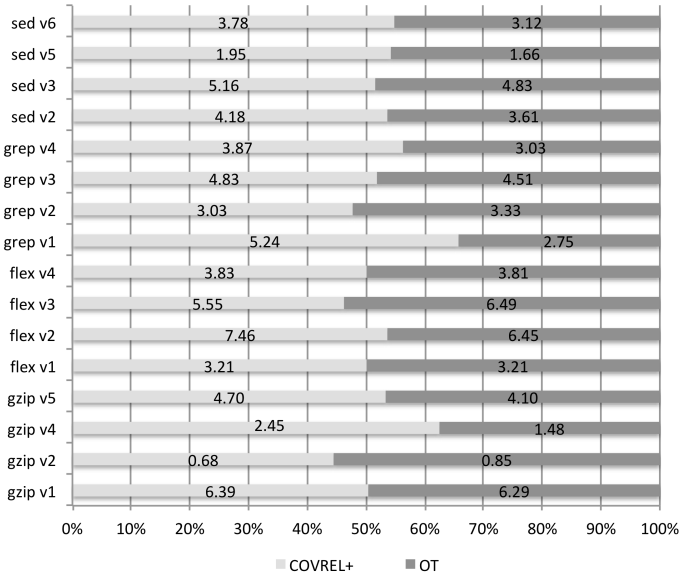


Fig. 10. Test Selection: Percentage and number of faults by subject

TABLE 9
Test Selection: Hypothesis test. Number of faults detected by the same amount of tests

Pairwise Comparison		
	covrel+	OT
Mean	4.15	3.72
Median	4.03	3.47
p-value	**1.44 E-41**	-
Effect size	**0.6018**	-

5.2.2 Test Generation

Table 10 reports the results in terms of number of detected faults by the same amount of test cases (T=1,000), obtained by the third experiment (which is the second experiment for test generation evaluation). Also in this case, mean and median values of faults detected by *covrel+* are bigger than OT under a fixed budget. This is true for all the three variants of *covrel+* – with *branch, function* and *statement* coverage, with again a better performance of the *statement* configuration followed by *branch* and *function*.

TABLE 10
Test Generation: Average number of faults detected with the same amount of test cases (T=1,000)

Subject	*Branch*		*Function*		*Statement*	
	covrel+	OT	*covrel+*	OT	*covrel+*	OT
gzip v1	5.97	5.13	6.00	4.83	6.00	5.00
gzip v2	3.00	1.83	3.00	1.77	3.00	1.73
gzip v4	5.00	4.70	5.00	4.59	5.00	4.67
gzip v5	5.00	3.53	5.00	3.17	5.00	3.33
flex v1	16.00	11.70	15.90	12.13	16.00	11.20
flex v2	14.97	13.93	15.00	13.90	14.97	13.67
flex v3	9.00	7.23	8.93	7.27	8.97	6.40
flex v4	10.97	6.63	11.00	5.90	10.90	7.13
grep v1	3.00	2.37	3.00	2.33	3.00	2.29
grep v2	1.00	1.00	1.00	0.90	1.00	0.80
grep v3	5.00	1.70	4.97	1.43	5.00	1.37
grep v4	2.00	0.97	2.00	1.12	2.00	0.73
sed v2	5.00	4.43	5.00	4.17	5.00	4.23
sed v3	2.00	0.50	2.00	0.50	2.00	0.67
sed v5	4.00	3.13	3.90	3.00	3.93	3.07
sed v6	6.00	3.23	5.97	3.70	6.00	3.87
Mean	**6.12**	**4.50**	**6.10**	**4.42**	**6.11**	**4.39**
Median	5.00	3.38	5.00	3.44	5.00	3.60

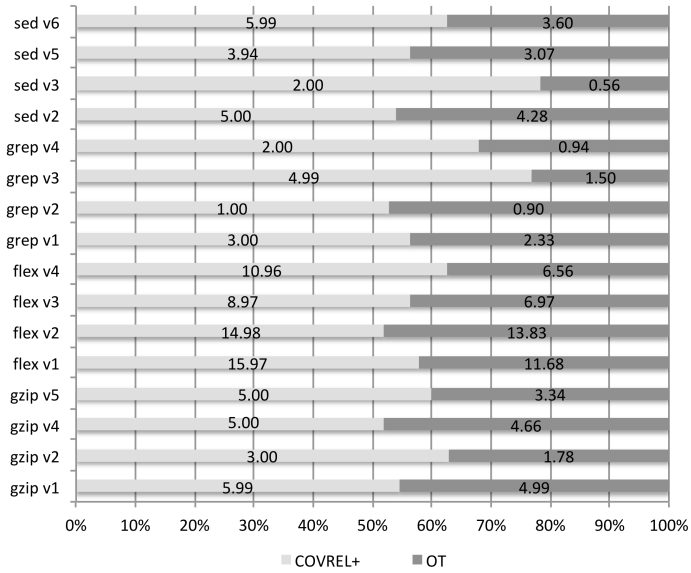


Fig. 11. Test Generation: Percentage and number of faults by subject

TABLE 11
Test Generation: Hypothesis test. Number of faults detected by the same amount of tests

Pairwise Comparison		
	covrel+	OT
Mean	6.11	4.43
Median	5.00	3.43
p-value	$<$ **1.0 E-324**	-
Effect size	**0.6478**	-

The results by subject in Figure 11 show that *covrel+* is more effective than OT at detecting faults in all the cases. This is statistically confirmed by hypothesis testing: results of the Friedman test and the effect size are in Table 11.

5.2.3 Answering RQ2

We can positively answer RQ2 for both test selection and generation:

Covrel+ is significantly more effective than OT as by executing a same number of (selected or generated) test cases, in the greatest

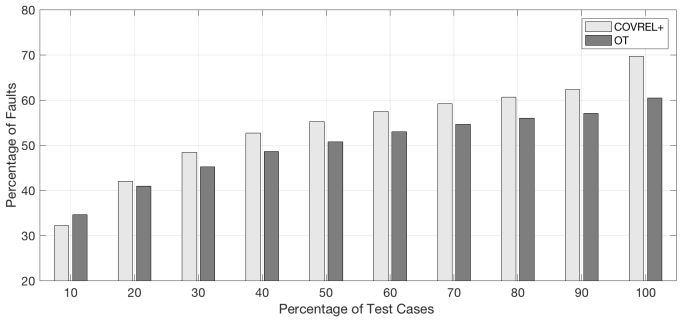


Fig. 12. Test Selection: Faults vs test cases (percentages)

majority of cases it detected a higher number of faults. In particular covrel+ outperformed OT in 12 subjects for test selection, and in all subjects for test generation out of the 16 considered ones.

5.3 RQ3: Fault-detection evolution

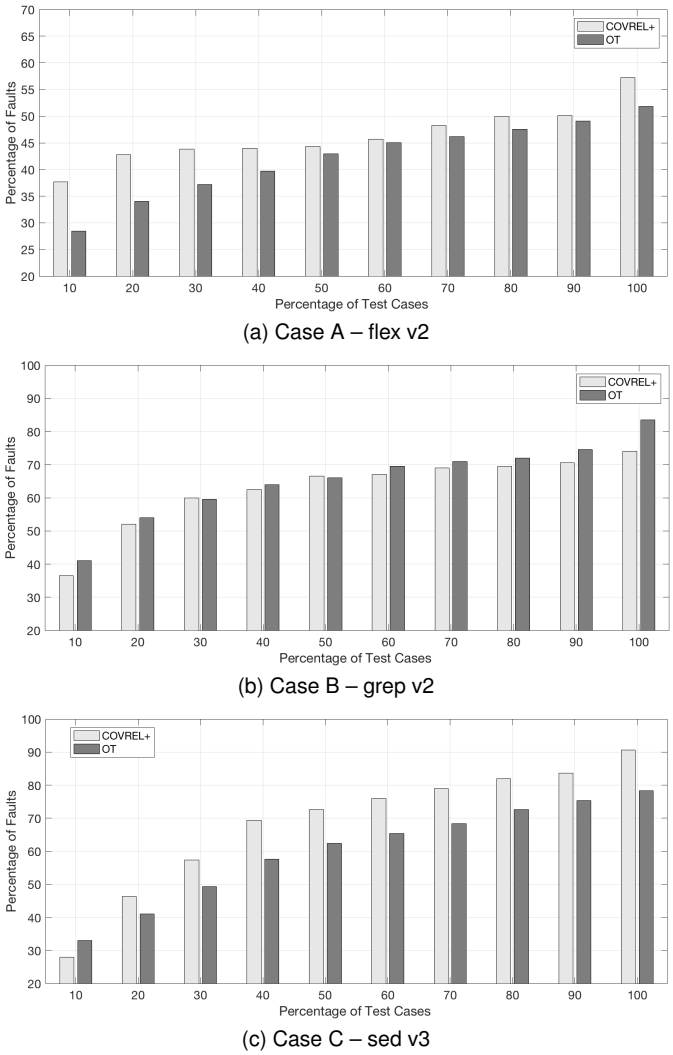
5.3.1 Test Selection

Figure 12 shows the percentage of faults detected over those detectable by the test suite after the execution of a given percentage of test cases with respect to T. Similarly to RQ2, T is taken, at each repetition of a given test scenario, as $T = min(N_{covrel+}, N_{OT})$, namely as the minimum between the number of test cases to detect all detectable faults by *covrel+* and by *OT*, so as to have the same amount of tests for both approaches. The Figure represents the evolution of the fault detection ability by the two techniques in the test selection experiment.[6] On average, *covrel+* performs better after about 20% of test cases – the total number of test cases being the minimum between $N_{covrel+}$ and N_{OT}. The difference between the two approaches increases with the number of executed test cases, getting to about 70% vs. 60% of detected faults over the detectable ones.

We emphasize the following three cases, which relate to the general trend in different ways:

- **Case A** (Figure 13a, flex v2). This case highlights that *covrel+* can be better since the beginning, not only in later stages. The behavior is observed more often in the flex cases (8 out of 12 pairs <*subjects, coverage criterion*>) test scenarios (8 out of 12 scenarios), while for the other subjects it is more rare – overall it happens in 15 out of 48 test scenarios. A possible explanation is that most faults in the flex subject escape traditional operational testing, namely they are more easily detectable if coverage information is exploited. The common trend is OT being better at the beginning and being overcome later.
- **Case B** (Figure 13b, grep v2). This is an example of OT being better than *covrel+*. It, in general, happened for 9 out of 48 cases (pairs <*subjects, coverage criterion*>), in 3 subjects – see Figure 10. In such cases, *covrel+* "wastes" test cases looking for scarcely covered entities, but the plausible explanation for the better performance of OT is that most faults are in code snippets highly covered by most test cases – namely there are not hard-to-detect faults.
- **Case C** (Figure 13c, sed v3). This example highlights the gain of *covrel+* when almost all faults are detected, i.e., in later stages. The marginal gain is higher, confirming that *covrel+* performs better when few residual faults are left in the code.

6. Note that, since T refers, in a given repetition, to either *covrel+* or to OT (depending on who wins the comparison), we have that when T refers to *covrel+*, the number of detected faults by *covrel+* is 100% and detected faults by OT are less than 100%; when T refers to *OT*, the opposite happens. Figure 12 reports the average over repetitions, thus the percentage of detected faults is never 100%, because none of the two approaches wins all the comparisons for a given scenario.



(a) Case A – flex v2

(b) Case B – grep v2

(c) Case C – sed v3

Fig. 13. Test Selection: Faults vs test cases (percentages) for subjects: (a) flex v2, (b) grep v2, (c) sed v3

5.3.2 Test Generation

Figure 14 shows the percentage of faults detected over all the seeded faults after the execution of a given percentage of test cases. It represents the evolution of the detection ability by the two techniques. The pattern is similar to the one observed for selection, confirming the better performance of *covrel+* in later stages of testing where few residual hard-to-detect faults remain in the code. However, in this case *covrel+* overcomes OT after about the 70% of test cases – the total number of test cases being 1,000 in this experiment. These test cases make the approaches to reveal about 50% of seeded faults; from the trend in Figure 14 and the results of RQ1 (where more than 1,000 test cases are generally executed), it is clear that the fault detection ability would still

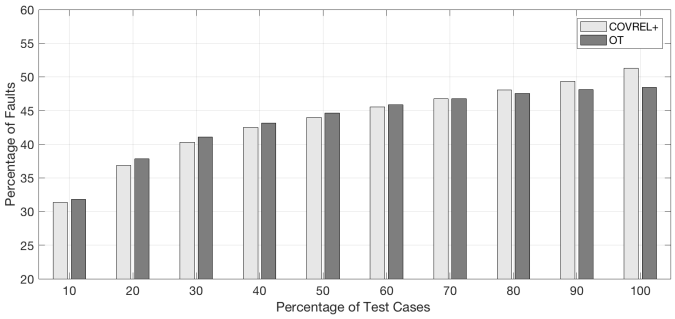


Fig. 14. Test Generation: Faults vs test cases (percentages)

increase in favor of *covrel+* with more test cases, basically because of its capability to detect harder faults. This is a feature observed also in our previous study [16], where the performance of *covrel* on subsets of hard-to-detect faults was studied for the test selection problem.

Similarly to the test selection experiments, the following cases are highlighted:

- **Case A** (Figure 15a, sed v6). The example shows a case where *covrel+* is better since the beginning. It happens only in two other cases with flex (like in the test selection case) but always by a very small margin. In particular, in the flex cases, results of the two approaches are comparable and the differences are always small since the beginning, because of the characteristics of seeded faults which make the effort of looking for hard to reach code less gainful.
- **Case B** (Figure 15b, flex v1). It is the case of OT being comparable or even slightly better than *covrel+*. This happens for 2 out of 48 cases (<*subjects, coverage criterion*> pair) for the test generation case and by a very small margin. The average of results over the three coverage criteria per subject in Figure 11 shows that OT never overcomes *covrel+*.
- **Case C** (Figure 15c, grep v1). This case confirms the relevant gain that *covrel+* can achieve in the last 20% of test cases also in the test generation case, which is almost 15% in the plotted case.

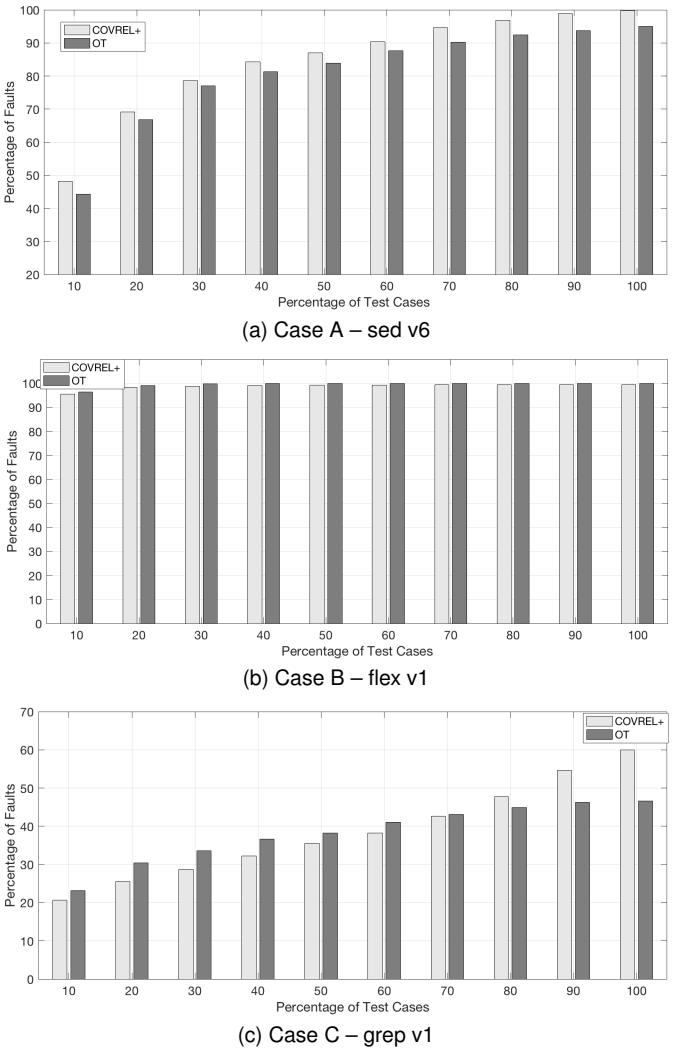


Fig. 15. Test Generation: Faults vs test cases (percentages) for subjects: (a) sed v6, (b) flex v1, (c) grep v1

5.3.3 Answering RQ3

Concerning RQ3, the results generally confirm our expectation that *covrel+* improvement in effectiveness over OT tends to increase for more mature stages of testing.

As testing proceeds and faults are removed, the improvement in the fault detection effectiveness of covrel+ over OT tends to increase. In particular using covrel+ for test selection the marginal difference becomes about 10% when executing the last 10% of the selected test cases. Using covrel+ for test generation, OT performs better until 70% of the test suite, and covrel+ performs better afterwards with increasing gain.

5.4 Threats to validity

Beyond our efforts in the accurate design and execution of experiments, results might still suffer from validity threats.

Threats to **internal validity** concern aspects of the study settings that could bias the observed results. Both the selection and generation of tests by *covrel+* and by the baseline OT technique include some random steps, and thus without proper controlling the experiment settings, an observed difference in the outcomes might be due to random variability and not to actual differences in efficiency or effectiveness. To prevent this, for each configuration we repeated the experiment 50 and 30 times for selection and generation, respectively, and in Section 5 we reported the average outcomes.

For the considered subjects, we manually identified the domain partitions used to derive the operational profiles that in our experiments are used as proxy for true profiles. If our partitioning is not appropriate, the observed results might not be related to the actual reliability. Using averaged data over multiple repetitions corresponding to as many profiles helps to mitigate this risk, but further experiments with true profiles may be needed to fully neutralize it.

Concerning test case selection, another threat to internal validity might derive from the usage of the test suites available in the SIR repository, which: *i)* do not achieve full coverage, and *ii)* do not detect all faults in the repository. We might have observed outcomes that are impacted by such characteristics of the test suites, rather than by the "treatment" under study. To mitigate this risk, we could have manipulated the test suite, e.g., by adding more test cases. However, the SIR repository data represent a "golden

standard" for benchmarking purposes and are used in several similar studies. Thus, to make the study more objective and repeatable, we preferred to undergo this potential threat and use each subject as it is provided with all of its artifacts.

As described in Section 4.3.4, to apply *covrel+* we had to assign values to some parameters that certainly affect its results. Although we chose the assigned values after some tuning, we did not perform a rigorous study and other values could have produced different results for *covrel+*. Parameters of the IS algorithm, together with characteristics of the faults distribution of subjects under test, are also conjectured as possible cause of some results in favor of OT. However, in order to make definitive conclusions on such specific worst cases for *covrel+*, further empirical analyses are needed. Indeed, with more accurate calibration we could have improved our approach without affecting OT results.

Threats to **construct validity** concern confounding aspects by which what we observed is not truly due to the supposed cause. To compare the efficiency of techniques (RQ1), we had to establish a target reliability value. In test selection we considered it as finding all detectable faults. In test generation we set it as either again all detectable faults or otherwise as the number of faults detected after 1,000 test cases by the most efficient technique. Such targets would not be meaningful in real practice, because obviously we do not know a priori how many faults exist. Other criteria to compare the "treatment" (*covrel+*) against the baseline might produce different outcomes. A true confirmation could only be achieved through comparing actual reliability improvements either in production or under a real operational profile: in the context of the present study we could not perform either, but the experimentation of *covrel+* within an industrial context is planned as future work.

Finally, threats to **external validity** concern aspects of the study that may impact the generalizability of results. These may suffer from low representativeness of the used subjects and faults. As for the latter, we considered seeded faults, and subjects with real faults might yield different results; control for this threat can be achieved only by conducting additional studies using subjects with real faults. As for subjects' representativeness, the study covered in total 16 versions of four C programs. Even though they belong to four subjects only, it is well known that for those four subjects from SIR differences among versions are quite significant, so that they could be considered as different programs. However, generalizability demands for additional studies with a range of diversified subjects. In particular, the subjects we employed do not properly match neither the size nor the type of applications for which OT is typically performed, e.g., large-scale enterprise or web systems. Even though we made our best effort to reproduce an operational-profile based testing environment, it remains to be established how well *covrel+* would scale on such larger applications. As mentioned in Section 4.2, experimenting with *covrel+* requires the availability of source code, faults and operational profile. While large subjects are available as open source software, the main obstacle for us to evaluate *covrel+* on them has been the unavailability of faults and profiles. Although a realistic profile could be devised for a large system with a huge manual effort, nevertheless its actual representativeness would be a threat to external validity. In the present study, we opted for the repeatability and verifiability of controlled experiments on subjects widely used in software engineering research, leaving experiments with large programs to future empirical studies that we hope to conduct with industrial partners.

6 Conclusions

When targeting product reliability, software testing is often based on the operational profile. This is because operational testing exploits information on typical product usage so as to expose failures (and then remove the causing bugs) that will occur in operation with higher probability, thus contributing more to the delivered (un)reliability. However, operational testing pays by its nature little attention to failures with low probability of occurrence, thus as testing proceeds, the reliability tends to achieve some stable level that becomes difficult to improve further.

To overcome this limitation, we have presented *covrel+*, a hybrid testing strategy that integrates white-box coverage measures based on count spectra into black-box operational testing. *Covrel+* can be used in two problems in software testing, namely the selection of test cases (if a test suite is available) or the generation of test cases (if not). It works iteratively by allocating a test budget (number of test cases) to operational profile-based partitions of the input domain of the program under test, and then selecting or generating test cases within partitions exploiting count spectra. This way, as more and more failures are exposed and the causing faults removed, the strategy progressively biases the remaining budget toward exposing failures with lower occurrence probability, ultimately improving reliability.

We have evaluated *covrel+* through experiments with subjects taken from a repository widely used for benchmarking testing techniques. The results show that *covrel+* generally outperforms operational testing at achieving a given reliability target, or at achieving a higher reliability level under the same testing budget, thanks to its greater ability to detect hard-to-detect faults as testing proceeds.

The application of the proposed strategy requires practitioners to find a trade-off between the benefits over operational testing and the additional cost for collecting coverage information. *Covrel+* is meant to be employed when the use of count spectra (and the associated overhead) may overcompensate the cost of executing a high number of tests, inherent in traditional operational testing to improve reliability beyond the saturation point.

In future we plan to assess *covrel+* feasibility within a real world industrial process. This may well reveal possible obstacles to its adoption or also some hidden costs that we might have overlooked. We hope that the potential gains we observed on the assessed benchmarks will be confirmed and that practitioners expert guidance will help to further improve *covrel+* usefulness and performance.

Acknowledgments

This work has been supported by the PRIN 2015 project "GAUSS" funded by MIUR. B. Miranda wishes to thank the postdoctoral fellowship jointly sponsored by CAPES and FACEPE (APQ-0826-1.03/16; BCT-0204-1.03/17). We thank the editor and the anonymous reviewers for comments which greatly helped to improve the paper.

REFERENCES

[1] J. Musa, "Software reliability-engineered testing," *Computer*, vol. 29, no. 11, pp. 61–68, 1996.

[2] ——, "Operational profiles in software-reliability engineering," *IEEE Softw.*, vol. 10, no. 2, pp. 14–32, 1993.

[3] C. Smidts, C. Mutha, M. Rodríguez, and M. J. Gerber, "Software testing with an operational profile: OP definition," *ACM Computing Surveys*, vol. 46, no. 3, p. 39, 2014.

[4] J. Lv, B.-B. Yin, and K.-Y. Cai, "On the asymptotic behavior of adaptive testing strategy for software reliability assessment," *IEEE Trans. Softw. Eng.*, vol. 40, no. 4, pp. 396–412, 2014.

[5] M. Lyu, Ed., *Handbook of Software Reliability Engineering*. Hightstown, NJ, USA: McGraw-Hill, 1996.

[6] P. Frankl, R. Hamlet, B. Littlewood, and L. Strigini, "Evaluating testing methods by delivered reliability," *IEEE Trans. Softw. Eng.*, vol. 24, no. 8, pp. 586–601, 1998.

[7] K.-Y. Cai, C.-H. Jiang, H. Hu, and C.-G. Bai, "An experimental study of adaptive testing for software reliability assessment," *J. Syst. Softw.*, vol. 81, no. 8, pp. 1406–1429, 2008.

[8] M. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation of program spectra," *ACM SIGPLAN Notices*, vol. 33, no. 7, pp. 83–90, 1998.

[9] T. Ball, P. Mataga, and M. Sagiv, "Edge profiling versus path profiling: The showdown," in *Proc. 25th ACM Symp. on Principles of Programming Languages*. ACM, 1998, pp. 134–148.

[10] T. Ball and J. R. Larus, "Using paths to measure, explain, and enhance program behavior," *Computer*, vol. 33, no. 7, pp. 57–65, 2000.

[11] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," *ACM SIGSOFT Softw. Eng. Notes*, vol. 22, no. 6, pp. 432–449, 1997.

[12] T. Xie and D. Notkin, "Checking inside the black box: regression testing by comparing value spectra," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 869–883, 2005.

[13] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. Van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, 2009.

[14] W. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, 2016.

[15] B. Miranda and A. Bertolino, "Does code coverage provide a good stopping rule for operational profile based testing?" in *Proc. 11th Int. Workshop Automation of Software Test*. ACM, 2016, pp. 22–28.

[16] A. Bertolino, B. Miranda, R. Pietrantuono, and S. Russo, "Adaptive coverage and operational profile-based testing for reliability improvement," in *Proc. 39th Int. Conf. Softw. Eng.* IEEE, 2017, pp. 541–551.

[17] L. Fiondella and S. Gokhale, "Optimal allocation of testing effort considering software architecture," *IEEE Trans. Rel.*, vol. 61, no. 2, pp. 580–589, 2012.

[18] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Testing, Verification Rel.*, vol. 22, no. 2, pp. 67–120, 2012.

[19] S. Anand et al.; A. Bertolino, J. Li, and H. Zhu (Orchestrators and Editors), "An orchestrated survey on automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, 2013.

[20] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proc. 36th Int. Conf. Softw. Eng.* ACM, 2014, pp. 435–445.

[21] P. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *Proc. 22nd Int. Conf. Software Analysis, Evolution and Reengineering*. IEEE, 2015, pp. 560–564.

[22] F. Del Frate, P. Garg, A. Mathur, and A. Pasquini, "On the correlation between code coverage and software reliability," in *Proc. 6th Int. Symp. Softw. Rel. Eng.* IEEE, 1995, pp. 124–132.

[23] P. Frankl and Y. Deng, "Comparison of delivered reliability of branch, data flow and operational testing: A case study," *ACM SIGSOFT Softw. Eng. Notes*, vol. 25, no. 5, pp. 124–134, 2000.

[24] D. Alrmuny, "A comparative study of test coverage-based software reliability growth models," in *Proc. 11th Int. Conf. on Information Technology: New Generations*. IEEE, 2014, pp. 255–259.

[25] M.-H. Chen, M. Lyu, and W. Wong, "Effect of code coverage on software reliability measurement," *IEEE Trans. Rel.*, vol. 50, no. 2, pp. 165–170, 2001.

[26] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," in *Proc. 14th Int. Symp. Softw. Rel. Eng.* IEEE, 2003, pp. 442–453.

[27] B. Miranda and A. Bertolino, "Scope-aided test prioritization, selection and minimization for software reuse," *J. Syst. Softw.*, vol. 131, pp. 528–549, 2017.

[28] S. Wang, Y. Wu, M. Lu, and H. Li, "Software reliability accelerated testing method based on test coverage," in *Proc. Annual Reliability and Maintainability Symp.*, 2011, pp. 1–7.

[29] B. Miranda and A. Bertolino, "An assessment of operational coverage as both an adequacy and a selection criterion for operational profile based testing," *Software Quality Journal*, 2017.

[30] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Systems Design and Implementation*. USENIX Association, 2008, pp. 209–224.

[31] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proc. 19th ACM SIGSOFT Symp. and 13th European Conf. on Foundations of Softw. Eng.* ACM, 2011, pp. 416–419.

[32] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. 29th Int. Conf. Softw. Eng.* IEEE, 2007, pp. 75–84.

[33] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *J. Syst. Softw.*, vol. 83, no. 1, pp. 60–66, 2010.

[34] P. Thévenod-Fosse and H. Waeselynck, "An investigation of statistical software testing," *Softw. Testing, Verification Rel.*, vol. 1, no. 2, pp. 5–26, 1991.

[35] M. Petit and A. Gotlieb, "Uniform selection of feasible paths as a stochastic constraint problem," in *Proc. 7th Int. Conf. on Quality Software*. IEEE, 2007, pp. 280–285.

[36] S. Poulding and J. A. Clark, "Efficient software verification: Statistical testing using automated search," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 763–777, 2010.

[37] C. Bishop, *Pattern Recognition and Machine Learning*, ser. Information Science and Statistics. New York, NY, USA: Springer, 2006.

[38] D. Cotroneo, R. Pietrantuono, and S. Russo, "RELAI testing: a technique to assess and improve software reliability," *IEEE Trans. Softw. Eng.*, vol. 42, no. 5, pp. 452–475, 2016.

[39] D. Fox, "Adapting the sample size in particle filters through KLD-sampling," *Int. J. Robotics Res.*, vol. 22, no. 12, pp. 985–1003, 2003.

[40] M. Sridharan and A. Namin, "Prioritizing mutation operators based on importance sampling," in *Proc. 21st Int. Symp. Softw. Rel. Eng.* IEEE, 2010, pp. 378–387.

[41] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving search-based test suite generation with dynamic symbolic execution," in *Proc. 24th Int. Symp. Softw. Rel. Eng.* IEEE, 2013, pp. 360–369.

[42] M. L. Soffa, K. R. Walcott, and J. Mars, "Exploiting hardware advances for software testing and debugging (NIER track)," in *Proc. 33rd Int. Conf. Softw. Eng.* ACM, 2011, pp. 888–891.

[43] K. Herzig, "Testing and Continuous Integration at Scale: Limits, Costs, and Expectations," in *Proc. 11th Int. Workshop on Search-Based Software Testing*. ACM, 2018, pp. 38–38.

[44] L. Strigini and B. Littlewood, "Guidelines for statistical testing," ESA/ESTEC, Tech. Rep. PASCON/WO6-CCN2/TN12, 1997.

[45] A. Pasquini, A. Crespo, and P. Matrella, "Sensitivity of reliability-growth models to operational profile errors vs. testing accuracy," *IEEE Trans. Rel.*, vol. 45, no. 4, pp. 531–540, 1996.

[46] C. Kallepalli and J. Tian, "Measuring and modeling usage and reliability for statistical Web testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 11, pp. 1023–1036, 2001.

[47] P. Tonella and F. Ricca, "Statistical Testing of Web Applications," *J. Softw. Maint. Evol.*, vol. 16, no. 1-2, pp. 103–127, 2004.

[48] J. Hao and E. Mendes, "Usage-based Statistical Testing of Web Applications," in *Proc. 6th Int. Conf. on Web Engineering*. ACM, 2006, pp. 17–24.

[49] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Commun. ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[50] A. Vargha and H. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *J. Educational Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[51] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Testing, Verification Rel.*, vol. 24, no. 3, pp. 219–250, 2014.

Antonia Bertolino is Research Director of the National Research Council at ISTI, Pisa, Italy. Her research covers software engineering and dependability, with particular interest in software testing methods and tools. She serves as an area editor for the Elsevier *Journal of Systems and Software*, and as an associate editor of *ACM Trans. on Software Engineering and Methodology*, Springer *Empirical Software Engineering Journal*, and Wiley *Journal of Software: Evolution and Process*. She served as the General Chair of the ACM/IEEE Conference ICSE2015 in Florence, Italy.

Breno Miranda is postdoctoral researcher at the Federal University of Pernambuco, Brazil, where he earned the Master degree in Computer Science in 2011. He received the PhD in Computer Science from University of Pisa in 2016. Currently, he collaborates with the Software Engineering and Dependable Computing laboratory at the Italian National Research Council (CNR) in Pisa. His research interests include software testing, recommender systems, and mining software repositories.

Roberto Pietrantuono is Assistant Professor at Federico II University of Naples, Italy, where he got his PhD degree in Computer and Automation Engineering in 2009. His research interests are in the area of software reliability engineering, particularly in the software verification of critical systems, software testing, and software reliability analysis. He is Senior Member of the IEEE.

Stefano Russo is Professor of Computer Engineering at Federico II University of Naples, Italy, where he teaches Software Engineering and Distributed Systems, and leads the Dependable Systems and Software Engineering Research Team (DESSERT). He co-authored over 160 papers in the areas of software testing, software aging, middleware technologies, mobile computing. He is Associate Editor of *IEEE Trans. on Services Computing*, and Senior Member of the IEEE.