Software Qual J
DOI 10.1007/s11219-017-9388-0

An assessment of operational coverage as both an adequacy and a selection criterion for operational profile based testing

Breno Miranda[1,2] · **Antonia Bertolino**[2]

Abstract While the relation between code coverage measures and fault detection is actively studied, only few works have investigated the correlation between measures of coverage and of reliability. In this work, we introduce a novel approach to measuring code coverage, called the operational coverage, that takes into account how much the program's entities are exercised so to reflect the profile of usage into the measure of coverage. Operational coverage is proposed as (i) an adequacy criterion, i.e., to assess the thoroughness of a black box test suite derived from the operational profile, and as (ii) a selection criterion, i.e., to select test cases for operational profile-based testing. Our empirical evaluation showed that operational coverage is better correlated than traditional coverage with the probability that the next test case derived according to the user's profile will not fail. This result suggests that our approach could provide a good stopping rule for operational profile-based testing. With respect to test case selection, our investigations revealed that operational coverage outperformed the traditional one in terms of test suite size and fault detection capability when we look at the average results.

Keywords Coverage testing · Operational coverage · Operational profile-based testing · Program Spectra · Relative coverage

✉ Breno Miranda
bafm@cin.ufpe.br; breno.miranda@isti.cnr.it

Antonia Bertolino
antonia.bertolino@isti.cnr.it

[1] Federal University of Pernambuco, Recife, PE, 50740-540, Brazil

[2] ISTI - CNR, Via Moruzzi 1, 56124 Pisa, Italy

Published online: 22 September 2017

1 Introduction

Software testing can be based on many criteria (Ammann and Offutt 2016; Bertolino 2007; Zhu et al. 1997). Among them, *operational profile-based* testing and*coverage-based* testing provide two quite diverse approaches.

The former is a black-box testing approach: the test cases are selected from the input domain, trying to reproduce how the software will be used in practice. The aim is to rapidly detect those failures that would occur most frequently in operation (Musa 1993). Indeed, operational profile-based testing is grounded on the notion that not all faults have the same importance: depending on how it will be exercised by the users, a program can show quite different levels of reliability (Lyu et al. 1996).

On the other hand, coverage-based testing is white-box: a program is tested until all, or a pre-defined percentage of, targeted code entities (e.g., statements or branches) have been executed at least once. Starting from the 70's, code coverage criteria have been actively studied, and many techniques and tools have been proposed (Zhu et al. 1997). Coverage-based testing is appealing because it provides in automated way a quantitative feedback from a testing session, i.e., the ratio between the entities covered and their total number. Such coverage measure, used as a supplement to other non-coverage-based testing methods, can be an effective tool (Staats et al. 2012), for example to decide whether a test suite derived using another black-box method is adequate, as well as to pinpoint portions of code that have not yet been tested.

What remains controversial is the relation between test coverage and testing effectiveness, and has been the subject of countless analytical and empirical studies (see, e.g., Basili and Selby 1987; Wong et al. 1994, among the earliest ones). After three decades, the debate on the topic does not seem to decline, and current testing research still seeks an answer to questions such as "Is Branch Coverage a Good Measure of Testing Effectiveness?" (Wei et al. 2012). A recent large scale experiment (Inozemtseva and Holmes 2014) concludes that high coverage measures achieved by a test suite do not necessarily indicate that the latter also yields high effectiveness. Thus, generating test cases for coverage as a target may be risky, as warned from many sides (e.g., Marick 1999; Staats et al. 2012).

On the other side, we observe that almost all studies assessing the effectiveness of coverage testing have used as a measure of effectiveness the faults (or mutations) detected without distinguishing their probability of failure in use (e.g., Inozemtseva and Holmes 2014; Kochhar et al. 2015; Staats et al. 2012; Wei et al. 2012; Wong et al. 1994, just to cite a few). Thus, we still know little about how (and if) coverage testing is related to *delivered reliability* (Frankl et al. 1998). An exception is the early study by Del Frate and coauthors (Del Frate et al. 1995), who observed that the relation between coverage and reliability varied widely with subject's size and complexity.

In addition, all studies so far (concerned with either faults or mutants detected, or delivered reliability) considered traditional coverage measures, which require that *all* entities are covered *at least once*. In other terms, all entities are considered as having same relevance for the purpose of completing coverage. This assumption seems contradictory with the very idea of reliability according to which user's functions should be exercised more or less frequently accordingly to user profiles, and in doing this, code entities as well will consequently be exercised with different frequencies.

In our research, we have been investigating for some time novel coverage measures that customize coverage to user's relevance (Miranda 2014, 2016; Miranda and Bertolino 2015, 2016b). In particular, in previous work, we distinguished between *relevant* entities and *not*

relevant ones depending on the usage domain. Our intuition is that we could find better correlation between coverage and reliability if we took into account how much the program's entities are exercised, i.e., we distinguish between entities that are very often exercised from those that are scarcely exercised, to reflect the profile of usage into the measure of coverage.

Profiling of code entities according to different usage profiles is referred to as *program spectra* (Harrold et al. 1998). The idea of using program spectra in software engineering tasks is not new: program spectra have been used, among others, for regression testing (Xie and Notkin 2005) and fault localization (Wong et al. 2016). To the best of our knowledge, however, our research is the first attempt to tune coverage measures based on program spectra, for the purpose of reflecting the importance of program entities.

More precisely, in a recent paper (Miranda and Bertolino 2016a), we introduced the first of its kind coverage criterion for operational profile-based testing using program spectra and called the *Operational Coverage*. We first assessed the performance of the proposed approach as an adequacy criterion, i.e., to support the decision on when to stop testing.

In this paper, we extend that work and assess also the performance of the proposed operational coverage for the task of selecting test cases (*selection criterion*) for operational profile -based testing.

The preliminary results we obtained seem to sustain our intuition that spectra-based coverage may be taken as both a stopping rule *and* a selection criterion for operational profile -based testing, better than traditional coverage. Such conclusion can be most usefully applied when the developer can leverage field data collected from profiling usage of the instrumented software (like the scenario described in Orso et al. 2003). Our proposed coverage measure may improve scalability of automated testing based on field data, because it provides an evaluation of test thoroughness that is customized to the usage profile, and thus testing resources can be better calibrated.

In summary, the contributions of the paper (extending preliminary definition and results in Miranda and Bertolino 2016a) include:

- a method to measuring code coverage that exploits program count spectra
- the design of a study of using operational coverage as an *adequacy criterion* for operational profile-based testing
- the design of a study of using operational coverage as a *selection criterion* for operational profile-based testing

The rest of the paper is structured as follows: in the next section, we introduce operational coverage. Then, in Section 3, we present the settings of our empirical evaluations, including the Research Questions and the study subjects. The results of adopting operational coverage are reported and illustrated separately for adequacy and selection in Sections 4 and 5, respectively. We discuss threats to the validity of our study in Section 6. Related work (Section 7) and Conclusions (Section 8) complete the paper.

2 Operational profile-based coverage

An operational profile provides a quantitative characterization of how a system is used (Musa 1993). Software testing based on the operational profile ensures that testing resources are focused on the most frequently used operations and, thus, maximizes the reliability level that is achievable within the available testing time (Musa 1993). So, it can be a good testing strategy when safety is not an issue.

Our proposed approach is meant to provide practical adequacy and selection criteria for operational profile-based testing. As we start from the assumption that the developer adopted an operational profile-based testing strategy, we also can assume that either an operational profile is derived by domain experts during the specification stage, or that, as schematized in Bullet 1 of Fig. 1, this profile is obtained from real world usage, e.g., by monitoring field data by means of an infrastructure such as Gamma (Orso et al. 2002).

If this developer selects a test suite from the operational profile, how can they decide whether the test suite is adequate and testing can be stopped, or otherwise more test cases should be derived? In the latter case, which additional test cases should be selected (Bullet 7 of Fig. 1)? This is where our approach can help. It foresees two main steps:

1. Classify the entities according to their usage frequency with respect to the operational profile under testing
2. Calculate operational coverage based on the importance of the entities covered

We rate the importance of entities based on their frequency of usage, i.e., we make use of program spectra (Harrold et al. 1998). A program spectrum characterizes a program's behavior by recording the set of entities that are exercised as the program executes. In this work, we investigated coverage of three types of entities and correspondingly adopted three types of spectra:

– **Branch-count spectrum (BCS):** for each conditional branch in a given program P, the spectrum indicates the number of times that branch was executed.
– **Statement-count spectrum (SCS):** for each statement in a given program P, the spectrum indicates the number of times that statement was executed.
– **Function-count spectrum (FCS):** for each function in a given program P, the spectrum indicates the number of times that function was executed.

Based on the spectra, we classify the entities (step 1 of our approach) into different importance groups. In this work, we used three groups: *high*, *medium*, and *low*, but other different groupings could be decided. To cluster entities into group, again, different methods could be applied. In our investigations, we opted for ordering the list of entities according to their frequency and assigning the first 1/3 entities to the high frequency group; the second 1/3 entities to the medium frequency group; and the last 1/3 entities to the low frequency group. Surely, the importance of a given entity could be assigned in many different ways and the effect of choosing one approach or another should be investigated in future work.

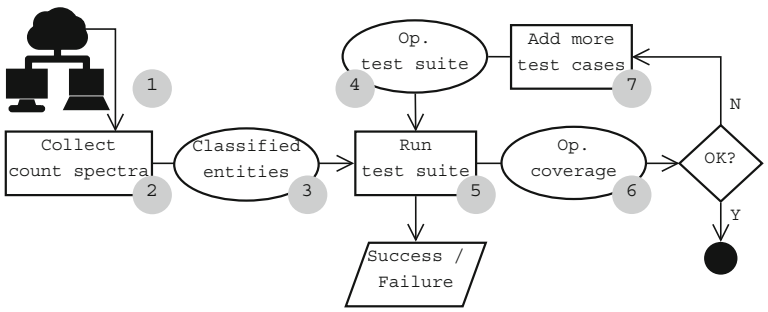

Fig. 1 Overview of the approach

We calculate operational coverage (step 2) by computing the weighted arithmetic mean of the rate of covered entities according to (1) below. Again, we observe that there may exist many other different ways in which we could calculate operational coverage and (1) is just one of the many possibilities.

$$\textbf{Operational Coverage} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i} \cdot 100(\%) \tag{1}$$

where:

n = number of importance groups
x_i = the rate of covered entities from group i
w_i = the weight assigned to group i

When the weights are normalized such that they sum up to 1, Equation 1 can be reduced to:

$$\sum_{i=1}^{n} w_i x_i \cdot 100(\%)$$

In this work, we assigned the weights for the importance groups (the w_i of (1)) in such a way that the *medium* group is three times more important than the *low*, and the *high* group, on its turn, is three times more important than the *medium* one.

Example To provide some intuition on how the operational coverage is calculated, we provide a dummy example.

Let us consider the set of entities in Table 1 that are classified into high, medium, or low, based on the count spectrum, and the set of test cases displayed in Table 2.

Let us assume that the test cases are executed in ascending order, i.e., from TC_1 to TC_4. After TC_1 is performed, two out of three *low* entities and two out of three *medium* entities are covered. No *high* entity is exercised by TC_1. Assuming that the weights assigned for the low, medium, and high groups are 1, 3, and 9, respectively, we have that:

$$Operational\ coverage = \frac{1 \times \frac{2}{3} + 3 \times \frac{2}{3} + 9 \times 0}{1 + 3 + 9} \approx 20.5\%$$

After TC_2 is performed, two additional entities, both *high*, are covered and the operational coverage will be 66.7%. When TC_3 is performed, 100% operational coverage is

Table 1 Example of entities classified into different importance groups based on their frequency of execution

Entity	Count spectrum	Importance group
e_1	28	Low
e_2	43	Medium
e_3	10	Low
e_4	59	Medium
e_5	107	High
e_6	114	High
e_7	95	High
e_8	85	Medium
e_9	14	Low

Table 2 Test cases and entities (from Table 1) they cover classified into different importance groups

TC #	Entities covered	Importance group
1	e_1	Low
	e_2	Medium
	e_3	Low
	e_4	Medium
2	e_3	Low
	e_5	High
	e_6	High
3	e_7	High
	e_8	Medium
	e_9	Low
4	e_1	Low
	e_5	High
	e_9	Low

achieved. By comparison, had the coverage been measured in the traditional way, the coverage achieved after TC_1, TC_2, TC_3, and TC_4, would had been 44.4%, 66.7%, 100%, and 100%, respectively.

3 Exploratory study

We conducted an exploratory study to assess the usefulness of adopting the proposed operational coverage criterion for test adequacy and selection. In this section, we discuss the settings of the study. More precisely, we focus on the following research questions:

- **RQ1**: *Does operational coverage provide a good stopping rule (adequacy criterion) for operational profile based testing?*
- **RQ2**: *Does operational coverage provide a good test case selection criterion for operational profile based testing?*

3.1 Study subjects

In order to carry out our exploratory study and to investigate our research questions in a realistic setting, we looked for subjects in the Software-artifact Infrastructure Repository (SIR) (Do et al. 2005). SIR contains a set of real, non-trivial programs that have been extensively used in previous research. For selecting our subjects, the only prerequisite was that they should contain faults (either real or seeded ones) and a test suite associated with them.

We considered three subjects — grep, gzip, and sed — as these are frequently used in academia for software testing research. grep is a command-line utility that searches for lines matching a given regular expression in the provided file(s); gzip is a software application used for file compression and decompression; and sed is a stream editor that performs basic text transformations on an input stream. grep and gzip are available from SIR with 6

Table 3 Details about the study subjects considered in our investigations

Subject	Version	LoC	# Seeded faults	# Failing test cases	# Faults revealed
grep	v3	10124	18	1664	5
gzip	v4	5233	12	1638	5
sed	v2	9867	5	3195	4
	Total:	25224	35	6497	14

sequential versions (1 baseline version and 5 variant versions with seeded faults) whereas sed contains 8 sequential versions (1 baseline version and 7 variant versions with seeded faults). Because each version contains a different number of seeded faults, for our study we selected, from each subject, the version that contained the highest number of faults that could be revealed by our test pool (detailed next). We then proceeded with version 3 of grep, version 4 of gzip, and version 2 of sed. Because we adopted only one version of each subject, throughout the rest of this paper, we refer to them by the subjects' names only without reporting the version.

Table 3 provides some additional details about the study subjects. Column "LoC" shows the lines of code[1] of each subject. The meaning of the last two columns is explained later on.

3.2 Operational profile

Operational profiles can be defined in many different ways. Here, following Musa (1993), we express it as the list of operations that are expected to be invoked by users along with their associated occurrence probabilities. Ideally, it is developed during system specification with the participation of the system experts (e.g.: system engineers, designers, etc) and domain experts (e.g.: analysts, customers, etc). Because such an ideal operational profile was not available for the subject systems we investigated, we ourselves defined the operational profile for each subject. We accomplished this task by getting acquainted with the system (after carefully reading the user manual for the version of the subject being investigated) and by following Musa's stepwise approach (Musa 1993). For experimental purposes, we stopped the process of creating the operational profile just before assigning the occurrence probabilities for each operation identified, as these are taken as an independent variable of our study.

Table 4 displays an excerpt of the list of operations we identified for grep. A graphical version containing the full list of operations identified (grouped according to different usage paths) is available at http://bit.ly/op_grep.

3.3 Study settings

Besides the study subjects obtained from SIR and the subjects' operational profiles developed by ourselves, for each of the three investigated subjects, we also created a few additional artifacts required for our study. Some of them were derived once and for all:

– **Test pool.** For each subject investigated, we created a test pool containing 10k test cases uniformly distributed among the operations in the subject's operational profile.

[1] Collected using CLOC (http://cloc.sourceforge.net/).

Table 4 List of operations identified for grep

ID	Description
Op_{001}	Look for a single pattern in a single input file
Op_{002}	Look for multiple patterns, obtained from a file, in a single input file
...	...
Op_{191}	Interpret the pattern as a list of fixed strings and match any of them
Op_{192}	Interpret the pattern as an extended regular expression and print the number of matching lines

- **Fault matrix.** For each subject considered, we run the set of 10k test cases over the baseline version first (the one without any faults enabled), and then over the faulty versions of that subject. To get a precise mapping of which test cases would reveal which faults, we compile one faulty version for each seeded fault available. For grep, for example, we have one baseline version and 18 faulty versions, which accounts for 190k test cases run to generate the fault matrix. The second last column and the last column of Table 3 refer to the results of running the set of 10k test cases over the studied subjects: they display the number of failing test cases and of seeded faults that could be revealed, respectively.
- **Operation matrix.** Each test case in the test pool is created for a specific operation in the operational profile and this mapping (test case, operation) is stored in the operation matrix.

Other artifacts, on the other hand, were created on a "per observation" basis. For each subject, we made 500 observations, each one related to a different operational profile. More precisely, for each observation, we derived the following artifacts:

- **Customized operational profiles.** For deriving a customized operational profile, we randomly select an arbitrary number of operations and randomly assign their respective occurrence probabilities in a way that the sum of the individual probabilities is equal to 1.
- **Importance groups.** For each customized operational profile, we classified the program entities into different importance groups (*high*, *medium*, or *low*) based on their frequency of usage. In order to do so, we exercise the subject with randomly generated test cases that are derived according to the operations and their respective occurrence probabilities defined in the customized operational profile. Observe that this set of randomly generated test cases is completely separated from the test pool previously introduced. We considered three different count spectra, each one related to a different coverage criterion: the branch-count, the statement-count, and the function-count spectrum.

We also used the gcov[2] and lcov[3] utilities for collecting accurate coverage metrics and our own code to automate the majority of the steps followed during this study.

[2] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[3] http://ltp.sourceforge.net/coverage/lcov.php

4 Adequacy study (RQ1)

In this section we report the results of our investigations with respect to the adoption of operational coverage as a stopping rule (RQ1). We start by describing the specific tasks and procedures associated with this exploratory study. Then, we present a summary of the study results along with an answer to our research question.

4.1 Tasks and procedures

For each subject (and for each one of the 500 customized profiles created per subject) the following tasks are performed:

1. We carry out operational profile-based testing by selecting the next test case to be run (from the 10k set) according to the occurrence probabilities defined in the customized operational profile.
2. After each test case is run, we calculate:
   (a) the traditional coverage achieved
   (b) the operational coverage achieved (calculated according to (1))
   (c) the probability of failure for the next test case, θ

The coverage metrics (items 2a and 2b) are calculated for the three adequacy criteria considered in this study and we stop testing if none of them increases after a sequence of 10 test cases.

4.1.1 Calculating the probability of failure for the next test case

We illustrate the way we calculate the probability of failure (item 2c) through a simple example pointing to the artifacts described in Section 3.3:

- *Customized operational profile:* we assume a very simple profile including only four operations: Op_1 with occurrence probability $Pr = 0.6$, Op_2 with $Pr = 0.3$, Op_3 with $Pr = 0.1$, and Op_4 with $Pr = 0.0$.
- *Test pool:* by construction, the test pool contains the same amount of test cases for each operation in the operational profile. For this example, we assume the test pool contains 2500 test cases per operation.
- *Fault matrix:* we assume only one fault, Fault 1, and that from the Fault matrix we can see it is revealed by 100 test cases in the 10k pool.
- *Operation matrix:* from the operation matrix, we can match operations to fault-revealing test cases (in the fault matrix). We assume we get: 50 failing test cases for Op_1, 30 for Op_2, 0 for Op_3, and 20 for Op_4.

Then, when no test case has been run, the probability that the next test will fail is:

$$\theta_{F_1} = \frac{(50 \times 0.6) + (30 \times 0.3) + (0 \times 0.1) + (20 \times 0)}{2500} = 0.0156$$

Because we assume that the existing faults are independent, when more faults exist, θ is the overall sum of the individual probability of failure for each fault. In the cases where coupled faults exist, however, this might not be true.

4.2 Study results

A summary of the output of the tasks described in Section 4.1 is displayed in Table 5. The second column shows the span variations in the number of test cases required for performing operational profile-based testing for each subject while the third column presents the average number of test cases among the 500 customized operational profiles created for our study. As we can see, for all the subjects, testing stopped after around 60 test cases. The biggest span variation happened for gzip with some operational profiles requiring as few as 13 test cases to complete testing, whereas other profiles required 136 tests; sed had the smallest variation with the number of test cases ranging from 15 to 98.

Table 5 also displays the average traditional and operational coverage achieved grouped by different adequacy criteria. In this table, "*trad.*" and "*oper.*" stand for traditional coverage and operational coverage, respectively. Operational coverage achieved higher coverage values in all the cases. This was expected because, by construction, operational coverage targets only a subset of the entities for each operational profile, which increases the chances of providing high coverage values.

Figure 2 shows, for all the subjects and coverage criteria investigated, the average traditional and operational coverage achieved as the number of test cases increases. The x-axes display the number of test cases while the y-axes represent the coverage achieved. In this figure, traditional coverage and operational coverage are represented by the continuous line and the dashed line, respectively. For each test case n, its equivalent point in the curve represents the average coverage (of the 500 customized profiles) achieved after n test cases. Notice that not all the profiles finished after the same amount of test cases. For this reason, it is possible to see, for gzip in particular, some fluctuation in the curve when the number of test cases gets close to the maximum number of tests required for that product.

From the graphs, the main observation is the fact that the curve of operational coverage rises sharply and achieves high coverage values even after just a few test cases. This happens due to the combination of two facts: (i) the most frequently exercised entities contribute more for the computation of the coverage achieved (because of the weight assigned to the high importance group) and (ii) the operational profile-based testing strategy selects test cases that cover the most frequent entities first.

4.3 Answer to the research question (RQ1)

To answer RQ1, we computed, for both traditional and operational coverage, the correlation between the coverage achieved and the probability that the next test casewill *not* fail ($1-\theta$, which is the reliability of the next invocation). For doing so, we adopted the Kendall τ

Table 5 Average traditional and operational coverage achieved per subject

Subject	#TCs span	Avg. #TCs	Branch		Statement		Function	
			trad.	*oper.*	*trad.*	*oper.*	*trad.*	*oper.*
grep	24 to 116	64	24.7	86.3	42.4	89.2	61.5	92.5
gzip	13 to 136	56	39.9	79.3	52.0	85.4	60.3	95.5
sed	15 to 98	51	29.5	95.5	48.3	96.2	71.3	98.1
	Average:	57	31.3	87.0	47.6	90.3	64.4	95.3

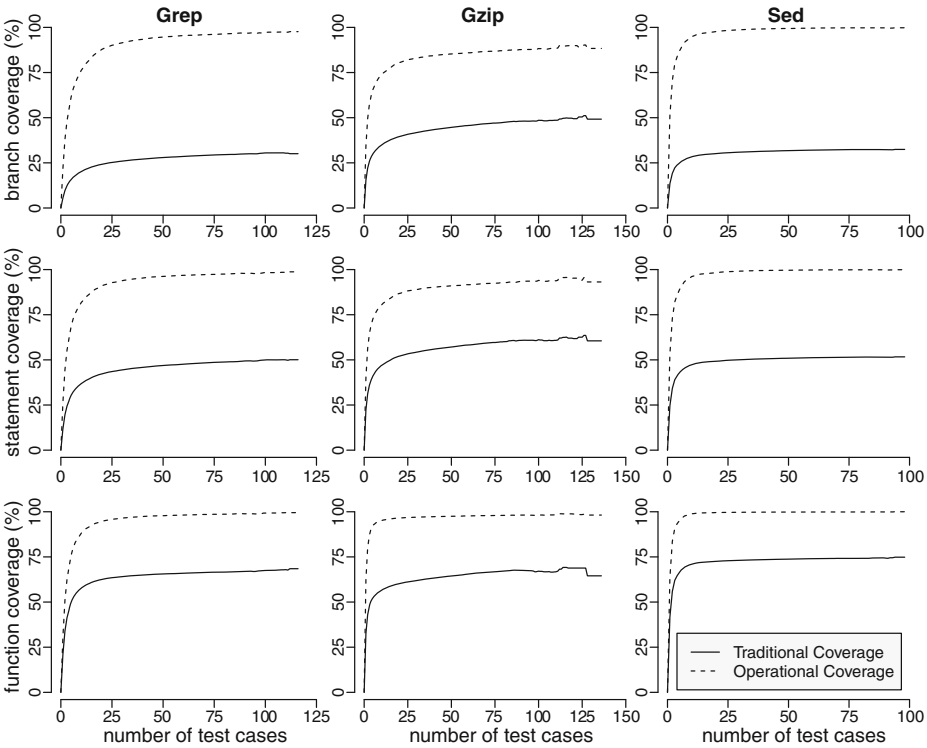

Fig. 2 Average traditional and operational coverage achieved as the number of test cases increases

correlation coefficient. Kendall τ is similar to the more commonly used Pearson coefficient but it does not require the variables to be linearly related or normally distributed. By using Kendall τ, we avoided introducing unnecessary assumptions about the distribution of the data.

As Kendall τ measures the similarity of the orderings of the data when ranked by each of the variables, a high correlation means that one can predict the rank of the importance of the faults revealed given the rank of the coverage achieved, which in practice is nearly as useful as predicting the absolute importance of the faults revealed.

Table 6 displays the Kendall τ correlation between the coverage achieved and the probability that the next test case will not fail, grouped by the different coverage criteria

Table 6 Kendall τ correlation between coverage and $1 - \theta$ (all entries are significant at the 99.9% level)

Subject	Branch		Statement		Function	
	trad.	*oper.*	*trad.*	*oper.*	*trad.*	*oper.*
grep	0.37	*0.40*	0.38	*0.41*	0.39	0.35
gzip	0.41	*0.45*	0.44	*0.46*	0.39	*0.44*
sed	0.39	*0.50*	0.40	*0.52*	0.35	*0.47*
Average:	0.39	*0.45*	0.41	*0.46*	0.38	*0.42*

investigated. For interpreting the data in accordance with Inozemtseva and Holmes (2014), we use the Guildford scale, in which correlations with absolute value less than 0.4 are described as "low", 0.4 to 0.7 as "moderate", 0.7 to 0.9 as "high", and over 0.9 as "very high".

As we can see, operational coverage yielded better correlation coefficients than traditional coverage for the vast majority of the cases (we set in italics the cases in which operational coverage performed better than traditional coverage). The only exception was for grep when considering the function adequacy criterion, in which traditional coverage achieved a correlation coefficient of 0.39 and operational coverage produced 0.35 (statistically, they were tied as both achieved low correlation). This was the only case in which operational coverage yielded low correlation. For the remaining cases, it always achieved moderate correlation.

Traditional coverage achieved moderate correlation three times, and in all the cases, it was statistically tied with operational coverage, if we consider the correlation group; if we consider the absolute correlation coefficient achieved, though, it was always defeated by operational coverage.

5 Selection study (RQ2)

After our investigations on the adoption of operational coverage for test case adequacy, we examined the usefulness of our approach for test case selection. Here, we report the specific tasks and procedures associated with the selection study as well as the results of our investigations.

5.1 Tasks and procedures

For our investigations on the adoption of operational coverage for test case selection, we used the 500 customized profiles per subject previously created. For each subject and for each customized profile, the following tasks are performed:

1. We randomly select a subset of 1k test cases from test pool (the 10k set).
2. We calculate the faults' importance based on the customized operational profile and the test pool derived for that iteration. This calculation is done in the same way as explained in Section 4.1.
3. Derive one test suite using the greedy additional algorithm targeting all the entities available in the subject under testing. We refer to it as the *traditional test suite*.
4. Derive a second test suite, the *operational test suite*, again using the greedy additional algorithm, but this time targeting the most important entities for the target customized operational profile.
5. We run both test suites against the subject under testing and we measure:

   (a) the size of the derived test suites;
   (b) the traditional coverage achieved by both test suites;
   (c) the operational coverage achieved by the operational test suite;
   (d) the remaining failure probability.

Performing step 1 is important for two reasons: first, as we use a greedy algorithm to derive the test suites, if we always considered the same test pool, it could be the case that the derived test suites would be very similar among different iterations. Some differences in the algorithm choices would still happen as the relevance of each entity changes on each iteration based on the customized operational profile. However, by randomly selecting a smaller set from the test pool, we guarantee that the greedy algorithm will always have a different set of test cases to choose from. Second, this step changes the relative importance of each fault with respect to the test pool on every iteration. In one iteration, the 1k set may contain all the test cases that would trigger the most important fault, whereas in a different round, it could contain only one test case that reveals the critical fault. This is important to observe whether or not the different approaches are able to (and to what extent) select the "best" test cases for each customized profile.

The traditional test suite is derived using a greedy additional selection heuristic depicted in Algorithm 1. Such heuristic repeatedly selects the test case that covers the maximum number of uncovered entities until all entities are covered. It receives, as input, a test suite from which test cases can be selected (T), the list of entities to be covered ($entities$), and information regarding which entities are covered by which test cases ($coverageInfo$). Then, on each iteration, the test case that yields the highest coverage is selected (line 3) and the coverage information of the remaining test cases is updated (line 5) to reflect their coverage with respect to the not yet covered entities. Algorithm 1 outputs a set of test cases (T') selected for testing the target program. When multiple test cases cover the same number of not yet covered entities, the function *getNextTestCase* (line 3) will select one test case randomly among the tied ones.

Algorithm 1 Greedy additional selection (traditional test suite)

```
   Input: T        /*the test suite from which test cases can be selected*/
          entities                        /*list of entities to be covered*/
          coverageInfo   /*list of entities covered by each test from T*/
   Output: T'    /*a subset of T, with which to test the target program*/
1  T' ⟵ [ ]                          /*T' is initialized as an empty list*/
2  while thereAreUncoveredEntities(T, entities, coverageInfo) do
3  |  selectedTestCase ⟵ getNextTestCase(T, entities, coverageInfo)    /*selects
   |  the test case that covers the highest number of uncovered
   |  entities*/
4  |  add(selectedTestCase, T')
5  |  updateUncoveredEntities(entities, selectedTestCase)  /*removes the entities
   |  covered by the selected test case from the list of uncovered
   |  entities*/
   end
```

By applying Algorithm 1 to the test cases from Table 2, we have that the first test case selected is TC_1 as it covers the highest number of uncovered entities (4 out of 9). Then, TC_3 is selected as it covers three uncovered entities (e_7, e_8, e_9). At the next iteration, TC_2 is the one that covers the highest number of uncovered entities (2 out of 9). At this point, 100% coverage has been achieved and the selection procedure stops, producing the final test suite $T' = [TC_1, TC_3, TC_2]$.

Algorithm 2 Greedy additional selection (operational test suite)

```
   Input: T        /*the test suite from which test cases can be selected*/
          entities                        /*list of entities to be covered*/
          coverageInfo    /*list of entities covered by each test from T*/
          entitiesImp             /*importance of each entity from the list
   entities*/
   Output: T'     /*a subset of T, with which to test the target program*/
1  T' ⟵ [ ]                          /*T' is initialized as an empty list*/
2  while thereAreUncoveredEntities(T, entities, coverageInfo, entitiesImp) do
3      selectedTestCase ⟵ getNextTestCase(T, entities, coverageInfo, entitiesImp)
       /*selects the test case that achieves the highest rank based on
       the importance of the entities it covers*/
4      add(selectedTestCase, T')
5      updateUncoveredEntities(entities, selectedTestCase)  /*removes the entities
       covered by the selected test case from the list of uncovered
       entities*/
   end
```

Algorithm 2, used to derive the operational test suite, differs from Algorithm 1 in the following ways: (i) it receives an additional input ($entitiesImp$) which maps each of the entities to be covered to their respective importance group and (ii) the function *getNextTestCase* (line 3) selects the next test case based on the importance of the entities covered.

On each iteration, the remaining test cases are ranked based on the list of uncovered entities and the weights assigned to the importance groups. For each low entity covered by TCi, its rank is increased by 1 (the weight assigned to the *low* group); for each medium entity covered, the rank of TCi is increased by 3 (the weight assigned to the *medium* group); and so on. The test case with highest rank is selected. In the case that multiple test cases achieve the same rank, the function *getNextTestCase* gets the one that covers the highest number of entities; if the tie persists, one test case is selected randomly from the list of tied ones.

When adopting Algorithm 2 for selecting test cases from Table 2, at the first iteration, TC_2 achieves the highest rank (19) as it covers two entities from the *high* group and one entity from the *low* group. After TC_2 is selected and the list of uncovered entities is updated, TC_3 scores the highest rank (13) among the remaining test cases. TC_1 is the third test case to be selected (with a rank of 7) and, at this point, Algorithm 2 stops because 100% coverage has been achieved. The final test suite produced by Algorithm 2 is T'=[TC_2,TC_3,TC_1]. Table 7 displays the ranks scored by TC_1 to TC_4 in the different iterations ($iter_j$) of the greedy heuristic applied by Algorithm 2.

In the case that the derived test suites contain a different number of test cases, for the sake of providing a fair comparison, we reduce the size of the bigger test suite down to the size of the smallest one before computing the metrics described in step 5.

5.2 Study results

Figures 3, 4, and 5 display the box plots of the sizes of the test suites derived for branch, statement, and function coverage, respectively. The results are grouped by the different

Table 7 Ranks scored by the test cases in the different iterations of the greedy heuristic applied by Algorithm 2

TC #	Entities Covered	Imp. Group	TC Rank at $iter_1$	TC Rank at $iter_2$	TC Rank at $iter_3$	TC Rank at $iter_4$
1	e_1	Low	$2w_l + 2w_m = 8$	$w_l + 2w_m = 7$	$w_l + 2w_m = 7$	Selected at $iter_3$
	e_2	Medium				
	e_3	Low				
	e_4	Medium				
2	e_3	Low	$w_l + 2w_h = 19$	Selected at $iter_1$	Selected at $iter_1$	Selected at $iter_1$
	e_5	High				
	e_6	High				
3	e_7	High	$w_l + w_m + w_h = 13$	$w_l + w_m + w_h = 13$	Selected at $iter_2$	Selected at $iter_2$
	e_8	Medium				
	e_9	Low				
4	e_1	Low	$2w_l + w_h = 11$	$2w_l = 2$	$w_l = 1$	$w_l = 1$
	e_5	High				
	e_9	Low				

subjects considered, and the *y*-axis displays the number of test cases in the devised test suites. The red dots indicate the average test suite size for each combination of subject and approach. For this metric, the lower the test suite size, the better.

Overall, the operational coverage approach defeated the traditional one in all the cases with lower median and average values. In all the cases observed, the upper quartile for the operational coverage approach is below (or very close to) the lower fence. This tells us that in about 75% of the cases, the size of the test suite derived targeting operational coverage was smaller than the smallest test suite derived targeting traditional coverage.

To complement the visual analysis of the box plots, we performed the Wilcoxon signed-rank test,[4] with significance level of 5%, to assess the null hypothesis that the difference between the test suite sizes for the two approaches follows a symmetric distribution around zero, i.e., the null hypothesis is that the median values are statistically equivalent. The *p value* returned by the Wilcoxon test was smaller than 2.2e−16 for all the cases, which means that the differences in the median values are statistically significant at least at the 95% confidence level.

While targeting branch coverage (Fig. 3), the number of test cases in the test suites derived according to the operational coverage approach varied from 1 to 31 for grep, from 2 to 38 for gzip, and from 2 to 21 for sed. For the test suites derived according to traditional coverage, the number of test cases ranged from 29 to 34 for grep, from 30 to 41 for gzip, and from 19 to 22 for sed.

With respect to the statement coverage criterion, displayed in Fig. 4, the average numbers of test cases in the test suites derived for operational coverage were 22.18, 18.79, and 12.01, for grep, gzip, and sed, respectively. For the test suites derived for traditional coverage, the figures were (following the same order) 28.76, 24.85, and 16.18.

[4]We adopted a non-parametric statistical hypothesis test because our data could not be assumed to be normally distributed.

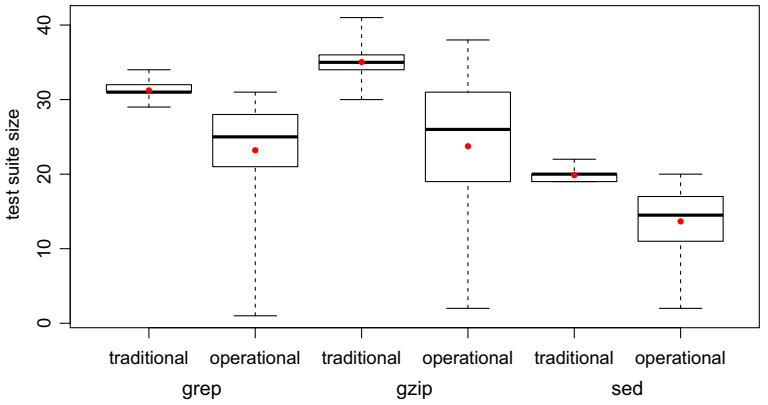

Fig. 3 Test suite reduction achieved when targeting Branch coverage

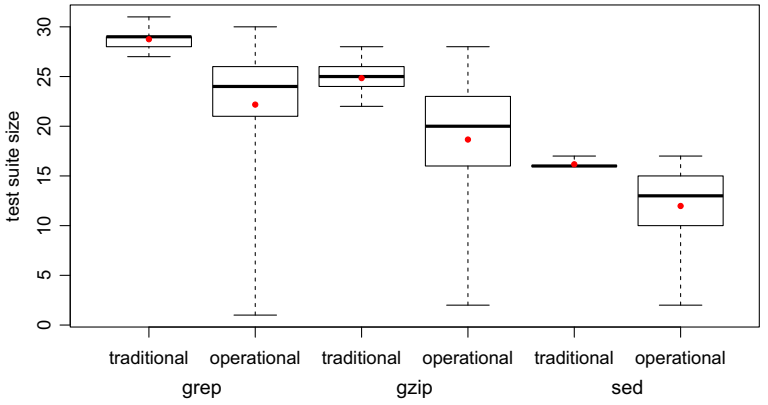

Fig. 4 Test suite reduction achieved when targeting Statement coverage

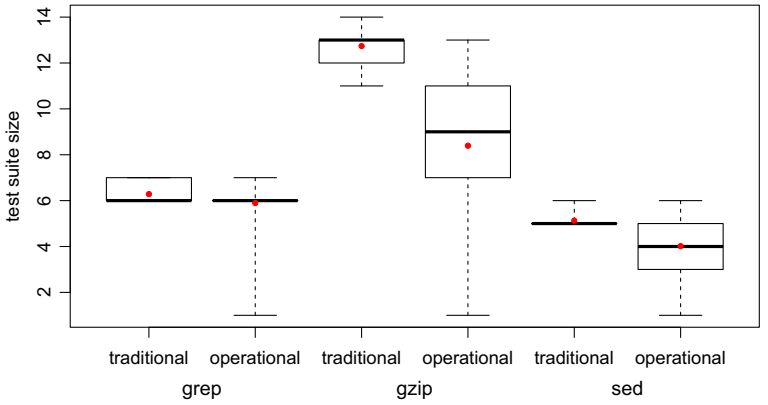

Fig. 5 Test suite reduction achieved when targeting Function coverage

For the test suites generated targeting function coverage (Fig. 5), the average number of test cases required by operational coverage for grep was 6.28, while 6.29 test cases were required for traditional coverage. For gzip, the figures were (following the same order) 8.39 and 12.74. Finally, for sed, the average numbers of test cases required were 4.02 and 5.13.

Besides measuring the size of the test suites generated by the greedy algorithm when targeting different selection criteria, we also computed the traditional and operational coverage achieved by them. The results are available in Table 8 and they are grouped by the different subjects and coverage criteria considered in this study. In this table, column "*trad.*" displays the traditional coverage achieved by the traditional test suite, while column "*oper.*" shows the operational coverage achieved by the operational test suite. As we were also interested in understanding how the test suite derived targeting the most important entities only would perform in traditional terms, we also computed the traditional coverage achieved by the operational test suite (this information is available in the column "*trad**").

As we can see in Table 8, the average operational coverage was very high in all the cases observed. As operational coverage targets only a subset of the entities from the program under testing, and considering the relatively big pool of test cases to select from, it was already expected that the operational test suites would achieve 100% operational coverage in many cases. Indeed, when targeting function coverage for sed, the operational test suites achieved 100% coverage of the in-scope entities in each one of the 500 observations.

When the coverage of the operational test suites was measured in the traditional way, column "*trad**", the average coverage achieved was always lower (but very similar) to the one obtained by the traditional test suites.

5.3 Answer to the research question (RQ2)

To answer RQ2, we computed, for the traditional and operational test suites, the remaining failure probability, i.e., the probability that the next test case would fail after finishing running the test suites. Reaching a 0 for this metric means that all the relevant faults for the customized operational profile have been revealed by the test suite. Thus, the lower the value, the better.

Table 9 displays the remaining failure probability grouped by subject and coverage criteria. The results achieved by the traditional and operational test suites are shown in columns "*trad.*" and "*oper.*", respectively. We set in italics the cases in which operational coverage outperformed traditional coverage.

Table 8 Average coverage (in %) achieved by the test suites derived according to different selection criteria

Subject	Branch			Statement			Function		
	trad.	*oper.*	*trad**	*trad.*	*oper.*	*trad**	*trad.*	*oper.*	*trad**
grep	32.43	99.69	30.86	51.79	99.82	50.43	68.33	99.71	66.47
gzip	55.59	99.82	50.40	69.04	99.87	63.82	69.26	99.87	66.23
sed	32.46	99.99	31.69	51.61	99.99	50.79	74.12	100.00	73.35
Avg.:	40.16	99.84	37.65	57.48	99.90	55.01	70.57	99.86	68.68

Table 9 Remaining failure probability (in %) after test suite execution (all entries are statistically significant at the 95% confidence level)

Subject	Branch		Statement		Function	
	trad.	*oper.*	*trad.*	*oper.*	*trad.*	*oper.*
grep	2.720	*0.907*	2.180	*0.804*	7.113	7.815
gzip	0.003	0.063	0.056	*0.043*	1.200	*0.966*
sed	0.205	*0.147*	0.306	*0.174*	15.125	*13.682*
Avg.:	0.976	*0.372*	0.847	*0.340*	7.813	*7.488*

Operational coverage was defeated by traditional coverage for grep when targeting function coverage, and for gzip when directing test selection for branch coverage. For all the other cases, operational coverage performed better than the traditional one. When considering the overall average, operational coverage exceeded traditional coverage for the three coverage criteria investigated in this study.

Despite the fact that operational coverage performed better in the majority of the cases when considering the average remaining failure probability, when we look at the individual results for each customized operational profile, operational and traditional approaches were tied in more than half of the cases. This was an unforeseen result as we were expecting that the test suites derived targeting a specific operational profile would had performed much better than the ones generated without further guidance regarding the relevance of the entities of the program under testing.

One possible explanation for that unexpected result is *collateral coverage*. It may have influenced the results in two ways: (i) for operational coverage, by mapping white-box entities to existing test cases, it is possible that the greedy algorithm selects *not relevant test cases* (a test case that covers some of the target entities but that is not related to the target operation). Traditional coverage (ii), on the other hand, may have "benefited" from this collateral coverage effect as the selection of a test case targeting a particular set of entities will most likely also satisfy the coverage of further entities by accident. Thus, even if the most important entities for a given operational profile are not the real target of the greedy algorithm aiming traditional coverage, many of them may be covered unintentionally by the first test cases selected.

6 Threats to validity

Beyond our best efforts in the accurate design and execution of the reported studies, our results might still suffer from validity threats. Thus, the results should be interpreted with the following potential threats to validity in mind.

As for *internal validity*, the operational profiles derived by ourselves are used in our study as proxies for *true* operational profiles, but if this approximation is not good, the observed results might have been impacted. Because the subjects' operational profiles were not readily available, we had to develop them ourselves. It is possible, then, that the set of operations we identified is not a complete list of operations that could be performed by real users. We mitigated this threat by carefully reading the subjects' documentation to understand well how they could be used before developing the operational profiles. However,

control for this threat can be achieved only by conducting additional studies using subjects with *true* operational profiles (preferably derived by experts).

A similar threat concerns the customized operational profiles. The customized profiles are derived by randomly selecting the target operations and their respective occurrence probabilities, which may have results into unrealistic operational profiles. To control this threat, we created 500 customized profiles for each subject investigated expecting that the effect of possible unrealistic profiles would be minimized with a big number of observations. Moreover, since both test suites (the traditional and the one based on operational coverage) are derived for each customized profile, we do not see how such threat could produce different impacts on our evaluations in a systematic way and, thus, influence the results biased to the benefit of one approach or another.

Another threat to internal validity might derive from the operational coverage calculation. The operational coverage can be influenced by many parameters that allow a high level of customization (e.g., the entities importance, the number of importance groups, the way the entities are assigned to different importance groups, the weight assigned to the importance groups). Control for this threat can be achieved only by conducting additional studies using different configurations.

Concerning threats to *external validity*, the study results may suffer from representativeness of the used subjects and faults. With reference to subject representativeness, our study covered three C programs from the SIR repository and additional studies using a range of diversified subjects that should be conducted before the results can be generalized.

With reference to faults representativeness, as said in our study, 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.

7 Related work

Code coverage criteria occupy a large part of software testing literature (Ammann and Offutt 2016; Bertolino 2007; Zhu et al. 1997). Many authors aim at finding novel criteria that subsume existing ones or that improve fault-finding effectiveness. Here, we do not propose yet another coverage criterion by identifying a new type of control-flow or data-flow entity to be covered. We propose instead that the entities to be covered (even basic ones, such as function, statement or branch) be weighted based on their relevance according to a usage profile.

Our research is inspired by the idea of relative coverage originally defined in Bartolini et al. (2009). In our previous work (Miranda 2014; Miranda and Bertolino 2014, 2015, 2016b), we distinguished between relevant and not relevant coverage, which amounted to give entities either a 1 or 0 weight, i.e., we used a hit spectrum. In this paper, we consider also the frequency of coverage and accordingly propose to weight entities using a count spectrum.

As discussed in the introduction, the effectiveness of coverage criteria is still a very active research topic (Gopinath et al. 2014; Inozemtseva and Holmes 2014; Kochhar et al. 2015; Staats et al. 2012). However, the studies evaluating the effectiveness of coverage measures consider the faults (or mutations) as having same importance and do not take into account their respective probability of failure. Considering the possible impact of faults detected, as we do here, provides a more meaningful picture when the software under test is going to be used under different profiles, and thus the effectiveness of coverage measures for the same software may vary depending on who is the future user.

Our work is mainly related to operational profile-based testing, as pursued in, e.g., Musa's SRET (Software Reliability-Engineered Testing) approach (Musa 1993). In SRET,

test cases are selected from the user's operational profile, thus those inputs that were forecasted to be more often invoked in use are also more stressed in testing. By classifying the entities according to their importance to the operational profile under testing and by calculating coverage based on the importance of the entities covered, we also aim at targeting the input subdomain that is the most relevant for the user, while giving lower priority to inputs that are not or seldom exercised. SRET uses a statistical approach for selecting test cases based on the user's operational profile. In our approach, we assume that such an operational profile would be available. In the case of its absence, we propose that field data could be exploited in the form of count spectra to capture the frequency at which entities are exercised by real users.

Harder et al. (2003) proposed a specification-based test case selection technique called "operational difference." Their technique dynamically generates a collection of logical statements (called "operational abstractions") that abstract the program's operation based on the execution of test cases. By comparing operational abstractions, their technique can be used for test suite generation, augmentation, and minimization. In our approach, however, we assume that the representation of the expected program's operation is already available in the form of operational profiles. We classify the code entities according to their importance to the operational profile and use that information to select test cases that would exercise the most relevant entities to that particular profile. Besides that, although the same term of *operational coverage* is defined in Harder et al., the meaning is different from the one we used here. In (Harder et al. 2003), operational coverage is defined in terms of precision and recall and the term is used to evaluate the quality of an operational abstraction when compared with an oracle or goal specification. In our work, operational coverage is a coverage criterion that can be used for both test adequacy and selection in the context of operational profile-based testing.

Program spectra (Harrold et al. 1998) have been used extensively in software analysis. Beyond the original application in program optimization (Ball et al. 1998), more recently code profiling information has been used to analyze the executions of different versions of code, e.g., in regression testing (Xie and Notkin 2005), and to compare traces of failed and successful runs in fault diagnosis (Wong et al. 2016). Here, we propose to exploit traces information to tune coverage measures onto the usage profile. This is a novel application of spectra in operational profile-based testing that has never been tried (except for our own antecedent work (Miranda and Bertolino 2016a)) and could be exploited in many ways.

There are similarities between operational coverage and the former notion of sensitivity by Voas and coauthors (Voas et al. 1991). Sensitivity was defined as the probability that a particular program "location" could reveal a possible fault under a specified input profile, where a location is a unit of code that can change a variable's value (it may roughly correspond to a statement, although some statements could also contain more locations). We do not consider specifically program locations; however, similar to sensitivity analysis, we profile program entities under the operational usage profile to understand their impact on probability of failure. So, motivations are similar, even though we then propose a different application of the notion.

Modern pervasive and interconnected networks and huge advances in potential to collect and analyze big data make it thinkable that developers continue testing and maintenance of deployed software by exploiting usage profiling information (Orso 2010). This requires the establishment of proper infrastructures (Orso et al. 2002), but can provide many opportunities for improved testing and analysis techniques (Elbaum and Diep 2005). For example, in Orso et al. (2003), field data are exploited for impact analysis and regression testing; in Jin and Orso (2013), field failures data are used to support in-house debugging and fault

localization. More in general, costs and benefits of profiling deployed software are empirically studied in Elbaum and Diep (2005), and the results confirmed their large potential. Hence, our approach is one among the many opportunities provided by field data to improve testing techniques.

8 Conclusions and future work

We have introduced operational coverage, which measures code coverage taking into account whether and how the entities are relevant with respect to a user's operational profile. We propose that the entities to be covered (even basic ones, such as function, statement or branch) should be weighted based on their relevance according to a usage profile.

The results of our studies, reported in Sections 4 and 5, showed that operational coverage is better correlated than traditional coverage with the probability that the next test case derived according to the user's profile will not fail. This result suggests that our approach could provide a good stopping rule for operational profile-based testing. With respect to test case selection, our investigations revealed that operational coverage outperforms the traditional one in terms of test suite size and fault detection capability when we look at the average results.

Concerning the costs of the proposed operational coverage, the cost of classifying the entities according to their importance (the first step of our approach) will depend on how the operational profile is derived. For the case advised in Fig. 1 in which the operational profile can be derived from real world usage by monitoring field data, the count spectrum (bullet 3, Fig. 1) required for defining the relevance of the entities might even be readily available, or otherwise can be obtained by using mining techniques to capture the frequencies of the entities being exercised by the users. Regarding the cost of computing the operational coverage itself (bullet 6, Fig. 1), as for any coverage criterion, operational coverage presupposes that the code is instrumented so to allow the identification of the entities exercised. So the cost of applying the operational coverage equation is comparable to any other coverage metric.

The proposed operational coverage may be particularly helpful for test suite augmentation. After the developer has derived the test suite to verify a given program, two main things could happen: (i) for all the code exercised by the program's users, there exist at least one test case in the test suite that covers that code; or (ii) there exist some code exercised by the program's users for which there does not exist any test case in the test suite. Operational coverage would acknowledge the former case by yielding 100% coverage. For the latter case, it would provide an assessment of the extent to which the derived test suite is adequate to that specific operational profile. Moreover, it would also provide the precise portions of code that are *actually exercised* by the program's users and that are not covered by the test suite, guiding the test suite enhancement process towards the real usage of the program. Regarding traditional coverage, for the case (i), if the test suite does not achieve 100% traditional coverage — probably the case — the developer cannot tell whether or not all the code that is relevant to the program's users has been covered. Similarly, for case (ii) traditional coverage does not help the developer unless they are willing to augment the test suite until 100% traditional coverage is reached, which may be impractical even for small programs.

Even though we believe that operational coverage may be a promising approach for operational profile-based test case selection, its adoption may not be as straightforward as it was for adequacy criterion. As anticipated in Section 5.3, one of the issues involved is that of collateral coverage. To overcome this effect, some fairly sophisticated selection approach

that takes into account not only the relevance of the entities to be covered, but also the way the candidate test cases cover those entities needs to be devised. We plan to investigate this as part of our future work.

The importance groups as well as the weights assigned to them play a fundamental role in the operational coverage equation as discussed in Section 2. When the weights are assigned as we did for our experiments, i.e., more weight to the "*high*" group and less weight to the "*low*" group, the operational coverage approach will privilege those entities that are expected to be exercised more frequently by the program's users. In different contexts, however, the testing objective could be different. For example, one could be interested in testing the areas of the program that are less frequently exercised in the attempt of finding possibly latent, difficult to find, faults. In that case, more weight should be assigned to the "*low*" group (Bertolino et al. 2017). As for any testing strategy, considering the context and the testing objectives is of fundamental importance when defining the operational coverage parameters.

Whatever strategy is adopted to assign weights to program entities, this is anyhow based on the count spectra and as such cannot consider the potential impact of failures, as is done in so-called *risk-based testing* (Erdogan et al. 2014; Felderer and Ramler 2014). In risk-based testing, test selection or prioritization is driven by the potential risk of failures, which is given by *both* their probability of occurrence (as in operational testing) *and* their impact. In our approach, we do consider the likelihood of failures, but not their consequence. Thus, parts of the system under test that are rarely used could not receive adequate testing and possibly lead to serious damages after time in operation. This limitation is common with SRET (Musa 1993) and any other reliability-driven testing approach. If the subject of test has safety-critical functions, then different test strategies that are risk-aware should be also considered.

Operational coverage is the very first attempt to tune coverage testing based on program count spectra. In particular, here, we used a simple approach to map count spectrum of branches, statements, and functions into a coverage measure. Of course more empirical studies are required to further assess the usage of operational coverage for test case selection and adequacy. However, we believe that the very idea introduced here paves the way to exploring many other powerful coverage measures.

Acknowledgements This research has been partly funded by the European Project ElasTest in the Horizon 2020 research and innovation program under Grant Agreement No. 731535, and by the MIUR National Project GAUSS under the PRIN 2015 program (Contract 2015KWREMX). Breno Miranda wishes to thank the postdoctoral fellowship jointly sponsored by CAPES (Coordination for the Improvement of Higher Education Personnel) and FACEPE (Foundation for Science and Technology Development of the State of Pernambuco) (APQ-0826-1.03/16; BCT-0204-1.03/17).

References

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge: Cambridge University Press.

Ball, T., Mataga, P., & Sagiv, M. (1998). Edge profiling versus path profiling: the showdown. In *Proceedings of the 25th symposium on principles of programming languages, POPL'98* (pp. 134–148). New York, NY, USA: ACM. https://doi.org/10.1145/268946.268958.

Bartolini, C., Bertolino, A., Elbaum, S., & Marchetti, E. (2009). Whitening SOA testing. In *Proceedings of the ESEC/FSE '09* (pp. 161–170): ACM.

Basili, V., & Selby, R. (1987). Comparing the effectiveness of software testing strategies. *IEEE Trans. Softw. Eng. SE*, *13*(12), 1278–1296. https://doi.org/10.1109/TSE.1987.232881.

Bertolino, A. (2007). Software testing research: achievements, challenges, dreams. In *2007 future of software engineering, FOSE '07* (pp. 85–103). Washington, DC, USA: IEEE Computer Society. https://doi.org/10.1109/FOSE.2007.25.

Bertolino, A., Miranda, B., Pietrantuono, R., & Russo, S. (2017). Adaptive coverage and operational profile-based testing for reliability improvement. In *Proceedings of the 39th international conference on software engineering, ICSE '17* (pp. 541-551). Piscataway, NJ, USA: IEEE Press. https://doi.org/10.1109/ICSE.2017.56.

Del Frate, F., Garg, P., Mathur, A., & Pasquini, A. (1995). On the correlation between code coverage and software reliability. In *Proceedings of the 6th international Symposium on sw reliability engineering* (pp. 124–132). https://doi.org/10.1109/ISSRE.1995.497650.

Do, H., Elbaum, S.G., & Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Software Engineering*, *10*(4), 405–435.

Elbaum, S., & Diep, M. (2005). Profiling deployed software: assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering*, *31*(4), 312–327. https://doi.org/10.1109/TSE.2005.50.

Erdogan, G., Li, Y., Runde, R.K., Seehusen, F., & Stølen, K. (2014). Approaches for the combined use of risk analysis and testing: a systematic literature review. *International Journal on Software Tools for Technology Transfer*, *16*(5), 627–642. https://doi.org/10.1007/s10009-014-0330-5.

Felderer, M., & Ramler, R. (2014). Integrating risk-based testing in industrial test processes. *Software Quality Journal*, *22*(3), 543–575. https://doi.org/10.1007/s11219-013-9226-y.

Frankl, P., Hamlet, R., Littlewood, B., & Strigini, L. (1998). Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, *24*(8), 586–601. https://doi.org/10.1109/32.707695.

Gopinath, R., Jensen, C., & Groce, A. (2014). Code coverage for suite evaluation by developers. In *Proceedings of the 36th international conference on software engineering, ICSE 2014* (pp. 72-82). New York, NY, USA: ACM. https://doi.org/10.1145/2568225.2568278.

Harder, M., Mellen, J., & Ernst, M.D. (2003). Improving test suites via operational abstraction. In *Proceedings of the 25th international conference on software engineering* (pp. 60–71): IEEE Computer Society.

Harrold, M.J., Rothermel, G., Wu, R., & Yi, L. (1998). An empirical investigation of program spectra. In *Proceedings of the PASTE '98* (pp. 83–90). https://doi.org/10.1145/277631.277647.

Inozemtseva, L., & Holmes, R. (2014). Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering* (pp. 435–445): ACM.

Jin, W., & Orso, A. (2013). F3: fault localization for field failures. In *Proceedings of the ISSTA* (pp. 213–223): ACM.

Kochhar, P., Thung, F., & Lo, D. (2015). Code coverage and test suite effectiveness: empirical study with real bugs in large systems. In *Proceedings of the SANER* (pp. 560–564). https://doi.org/10.1109/SANER.2015.7081877.

Lyu, M.R. et al. (1996). *Handbook of software reliability engineering*. CA: IEEE Computer Society Press.

Marick, B. (1999). How to misuse code coverage. In *Proceedings of the 16th international conference on testing comp. sw.* (pp. 16–18).

Miranda, B. (2014). A proposal for revisiting coverage testing metrics. In *ACM/IEEE International conference on automated software engineering, ASE'14* (pp. 899–902). https://doi.org/10.1145/2642937.2653471.

Miranda, B. (2016). *Redefining and evaluating coverage criteria based on the testing scope*. Ph.D. thesis, University of Pisa, Italy.

Miranda, B., & Bertolino, A. (2014). Social coverage for customized test adequacy and selection criteria. In *9Th international workshop on automation of software test, AST 2014* (pp. 22–28). https://doi.org/10.1145/2593501.2593505.

Miranda, B., & Bertolino, A. (2015). Improving test coverage measurement for reused software. In *41St euromicro conference on software engineering and advanced applications, SEAA 2015* (pp. 27–34). https://doi.org/10.1109/SEAA.2015.69.

Miranda, B., & Bertolino, A. (2016a). Does code coverage provide a good stopping rule for operational profile based testing? In *Proceedings of the 11th international workshop on automation of software test, AST '16* (pp. 22–28). New York, NY, USA: ACM. https://doi.org/10.1145/2896921.2896934.

Miranda, B., & Bertolino, A. (2016b). Scope-aided test prioritization, selection and minimization for software reuse Journal of Systems and Software. https://doi.org/10.1016/j.jss.2016.06.058, http://www.sciencedirect.com/science/article/pii/S0164121216300875.

Musa, J.D. (1993). Operational profiles in software-reliability engineering. *IEEE Software*, *10*(2), 14–32.
Orso, A. (2010). Monitoring, analysis, and testing of deployed software. In *Proceedings of the FSE/SDP workshop on future of software engineering research, FoSER '10* (pp. 263-268). New York, NY, USA: ACM. https://doi.org/10.1145/1882362.1882417.
Orso, A., Liang, D., Harrold, M.J., & Lipton, R. (2002). Gamma system: continuous evolution of software after deployment. In *Proceedings of the Int. Symp. on sw. Testing and analysis, ISSTA '02* (pp. 65–69): ACM. https://doi.org/10.1145/566172.566182.
Orso, A., Apiwattanapong, T., & Harrold, M.J. (2003). Leveraging field data for impact analysis and regression testing. In *Proceedings of the ESEC/FSE-11* (pp. 128–137): ACM. https://doi.org/10.1145/940071.940089.
Staats, M., Gay, G., Whalen, M., & Heimdahl, M. (2012). On the danger of coverage directed test case generation. In *FASE'12* (pp. 409–424): Springer.
Voas, J., Morell, L., & Miller, K. (1991). Predicting where faults can hide from testing. *IEEE Software*, *8*(2), 41–48. https://doi.org/10.1109/52.73748.
Wei, Y., Meyer, B., & Oriol, M. (2012). Is branch coverage a good measure of testing effectiveness? In *Emp. sw. eng. and verification* (pp. 194–212): Springer.
Wong, W., Horgan, J., London, S., & Mathur, A. (1994). Effect of test set size and block coverage on the fault detection effectiveness. In *Proceedings of the 5th international Symposium on software reliability engineering* (pp. 230–238). https://doi.org/10.1109/ISSRE.1994.341379.
Wong, W.E., Gao, R., Li, Y., Abreu, R., & Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, *42*(8), 707–740.
Xie, T., & Notkin, D. (2005). Checking inside the black box: regression testing by comparing value spectra. *IEEE Transactions on Software Engineering*, *31*(10), 869–883. https://doi.org/10.1109/TSE.2005.107.
Zhu, H., Hall, P.A.V., & May, J.H.R. (1997). Software unit test coverage and adequacy. *ACM Computing Surveys*, *29*(4), 366–427. https://doi.org/10.1145/267580.267590.

Breno Miranda (http://cin.ufpe.br/~bafm) earned his PhD in computer science at the University of Pisa in 2016 and his master degree in computer science at the Federal University of Pernambuco in 2011. Currently, he is a postdoctoral researcher at the Federal University of Pernambuco and he collaborates with the software engineering laboratory at the Italian National Research Council (CNR) in Pisa. His PhD research focused on Coverage Testing and he was advised by Antonia Bertolino. His current research interests include software testing, recommender systems, and mining software repositories.

Antonia Bertolino (http://www.isti.cnr.it/People/A.Bertolino) is a Research Director of the Italian National Research Council (CNR), in Pisa. She investigates approaches for software and services validation, testing, and monitoring, and on these topics has worked in several national and European projects, including the on-going H2020 ElasTest and the recently concluded FP7 Learn PAd, CHOReOS, and NESSOS. Currently, she serves as the Area Editor for Software Testing for the Elsevier Journal of Systems and Software, and as an Associate Editor of ACM Transactions on Software Engineering and Methodology, and of Springer Empirical Software Engineering. She serves regularly in the Program Committee of the most renowned conferences in the field of Software Engineering, such as ESEC-FSE and ICSE, and in software testing and analysis, as ISSTA and ICST. She has been the General Chair of the ACM/IEEE Conference ICSE 2015, May 2015, Florence (Italy). She has (co)authored over 100 papers in international journals and conferences.