Noname manuscript No.
(will be inserted by the editor)

Hybrid is better: Why and how test coverage and software reliability can benefit each other

Antonia Bertolino · Breno Miranda · Roberto Pietrantuono · Stefano Russo

Received: date / Accepted: date

Abstract Functional, structural and operational testing are three broad categories of software testing methods driven by the product functionalities, the way it is implemented, and the way it is expected to be used, respectively. A large body of the software testing literature is devoted to evaluate and compare test techniques in these categories. Although it appears reasonable to devise hybrid methods to merge their different strengths - because different techniques may complement each other by targeting different types of faults and/or using different artifacts - we still miss clear guidelines on how to best combine them.

We discuss differences and limitations of two popular testing approaches, namely coverage-driven and operational-profile testing, belonging to structural and operational testing, respectively. We show *why* and *how* test coverage and operational profile can cross-fertilize each other, improving the effectiveness of structural testing or, conversely, the product reliability achievable by operational testing.

Keywords Software testing · Reliability · Structural testing · Operational testing

1 Introduction

Testing is an essential part of the software development and maintenance processes. It consists of the dynamic assessment of software behavior on a finite sample of executions. To make testing systematic and to measure progress while tests are executed, some strategy is necessary. It will help testers to keep costs within reasonable bounds and to identify those test cases deemed the most effective.

Antonia Bertolino
ISTI - CNR, Pisa, Italy
E-mail: antonia.bertolino@isti.cnr.it

Breno Miranda
Federal University of Pernambuco, Recife, Brazil
E-mail: bafm@cin.ufpe.br

Roberto Pietrantuono, Stefano Russo
Università degli Studi di Napoli Federico II, Napoli, Italy
E-mail: {roberto.pietrantuono,stefano.russo}@unina.it

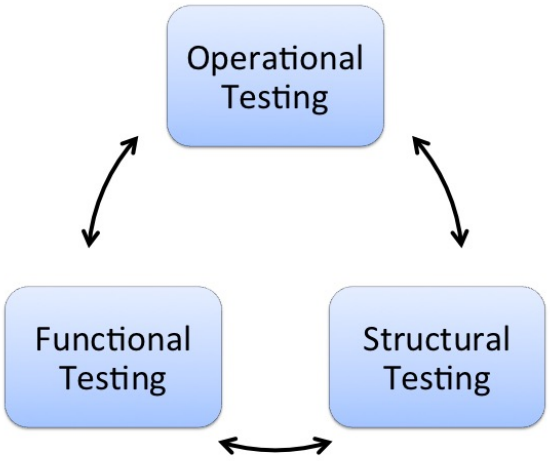

Fig. 1 Test strategies and their potential relations

Broadly speaking, systematic testing strategies are driven by three major aspects of the software under test (SUT): *i)* what it is expected to do, *ii)* how it is implemented, and *iii)* how it will be used. Such three aspects correspond to three major categories of software testing techniques, namely functional, structural and operational testing (Figure 1).

Each category relies on different assumptions and artifacts, and a broad variety of techniques and tools for each one has been proposed.

Since the early years of software testing discipline, researchers have conducted analytical and empirical studies to evaluate and compare the effectiveness of the different test techniques, in search for the most cost-effective approach.

From such studies we have learned that testing techniques may suffer from saturation effects and from various other limitations, and that there exist no one technique which best suits all circumstances. Different test techniques target different types of faults and thus may complement each other. For this reason, it is reasonable to invest resources by properly combining different techniques, rather than employing all the testing budget in only one selected strategy.

However, there are not many proposals for hybrid techniques merging the respective strengths of functional, structural and operational testing (examples are [7,8,10]), and no widely accepted guidelines on how different methods could be combined into one effective strategy are available. Further research is needed to understand how such strategies could be combined, depending on the testing purpose and the available artifacts.

As a step forward in this direction, we discuss the differences and respective limitations of two popular testing approaches: techniques driven by code coverage information, and techniques driven by the operational profile. Traditionally these two test approaches are adopted to address different purposes: coverage-driven testing aims at finding as many faults as possible, whereas operational-profile driven testing aims at improving software reliability. So, apparently, they seem to

belong to two worlds apart, and in fact there is little overlap between research progresses. However, we have found that on the one side coverage criteria can be made more effective if not all entities are considered equal, but software usage in operation is referred to assign them different weights. On the other side, software reliability testing can be made more effective as well if coverage information is considered alongside the operational profile in selecting the test cases.

Our reported results provide only an incomplete vision of several other potential "hybridizations". For instance, we have not considered yet the usage of functional strategies where software specifications or models are available. In presenting how coverage criteria and reliability improvement can benefit each other our contribution is one step towards unleashing the potential of many more useful combination of techniques.

The chapter is structured as follows. Section 2 describes the main concepts of test coverage and related measures in *debug testing*. Section 3 presents the rationale behind software *reliability testing* techniques. Section 4 discusses the relationship between coverage and reliability, and how these can benefit each other. Section 5 describes related work on combining white-box and operational testing. Section 6 concludes the chapter.

2 On test coverage measures

Software testing can pursue different goals. Along the development process, testing may aim at detecting as many faults as possible so that these can be removed before the software goes in production. For this reason, this type of testing is referred to in the literature as *debug testing* [16].

Measures of effectiveness of debug testing techniques are related with its faults finding capability. For example, a test technique would be evaluated more effective than another if it detects the first fault by executing a lower number of test cases, or otherwise if by executing an equal number of test cases the former finds a higher number of faults than the latter.

Along such line of reasoning, measuring the coverage of which and how many program elements are exercised during test execution is seen by many as an appealing proxy for assessing fault finding effectiveness. The intuition is that if a fault resides in a part of code that is never tested, such fault would never be activated and hence would survive testing, probably remaining undetected until the final user will eventually trigger it. In his seminal and highly-referenced book on "Software Testing Techniques" [3], Beizer defined leaving parts of code untested as "*stupid, shortsighted and irresponsible*".

Depending on which elements of code are targeted, in the years a broad variety of test coverage criteria have been proposed [17, 47]. All of them basically share the following scheme: an element of the program source code is identified as the type of entity to be covered. This element can be as basic as every statement or every branch of the program control flow, or become more sophisticated, such as for example every association between the definition of a variable and all its potential usages, for every variable in the program (all definition-use associations [17]). Then the source code of the SUT is parsed and instrumented, so that the coverage of the targeted elements can be monitored during testing. While test proceeds, a quantitative assessment of the thoroughness of testing is provided by the ratio

between the number of entities that have been already covered and the cardinality of the whole set of entities, expressed by the percentage:

$$\text{Test coverage} = \frac{\#\ \textit{of covered entities}}{\#\ \textit{of available entities}} \cdot 100(\%). \tag{1}$$

The underlying idea of coverage criteria is that until there remain entities that have not been exercised, the testing cannot be deemed complete, and more test cases have to be executed that can increase the above ratio. Therefore, coverage measures provide both a practical stopping rule (when a satisfying coverage is achieved), and a guide for the selection of additional test cases (*i.e.*, those covering yet uncovered entities).

There exist no proven direct relation, for any of the existing criteria, that when complete test coverage is achieved, then the SUT can be guaranteed to be defect-free. Since testing is essentially a sampling from a practically infinite set of executions [4], it is obvious to everyone that no finite test campaign can ensure correctness. Indeed, the most famous quotation about software testing is probably Dijkstra's aphorism that *software testing can only show the presence of bugs, but never their absence* [13]. In search for more effective testing strategies, the realistic goal is not to remove all faults, but rather to maximize the likelihood of revealing potential failures.

Coverage criteria can be considered as belonging to *partition testing* strategies that divide the input domain into equivalence classes (even though they generally create overlapping subdomains and not true partitions), and ensure to pick representative test cases (at least one) from each class. Theoretical analyses of partition testing strategies [44] have early shown that their effectiveness depends on how and where the failure-causing inputs are located, which is of course beyond testers' control and knowledge. The root of the problem is what Roper called the "*missing link*": we still cannot (will we *ever* be able to?) establish a logical or practical "*link between the adequacy criteria and attributes of the program under test such as its reliability or number of faults*" [37]. Thus, the only way to establish whether a relation exists between coverage of some entity type and fault finding effectiveness is through empirical studies, and in fact a series of such studies has been and continues to be undertaken by several researchers, *e.g.*, [43,23], but no definitive answers are available yet.

More properly, we must understand that what coverage measures provide us is an assessment of a *test suite thoroughness*. At the same time, some researchers have raised concerns against misusing coverage as the main goal of testing [18,27]. In such light, additional test cases that do not contribute to increase coverage would be considered "redundant" and not useful, however such test cases could indeed be able to catch still undetected faults. We should also never forget the cost in terms of time consumed in monitoring coverage, which makes white-box testing impractical on large scales [20].

In conclusion, coverage criteria provide a very useful and practical means towards systematic thorough testing. However, "*100% coverage should always be the result of good testing but it makes few sense as a goal in itself*"[36].

3 On software reliability

Testing to find as many faults as possible may seem a good strategy. However, in real-world production we have to face stringent time and budget constraints, which make Herzig note that "*There's never enough time to do all the testing you want*" [21]. Henceforth, this strategy could not be the best choice.

The point is that debug testing targets all faults indiscriminately, without considering the important difference between a *fault* (the cause) and a *failure* (its manifestation), nor the likelihood and potential impacts of the failure originating from a given fault. Indeed, *not all faults are created equal*. An early seminal study by Adams [1] showed, for example, that the 30% of the faults found in the systems he studied (at the time in IBM production) would each show itself less then once every 5,000 years of operational use. Clearly any testing effort spent to find these "tiny" faults would not be well employed.

This brings us to the fundamental concept of *software reliability*, which is "the probability of failure-free operation for a specified period of time in a specified environment" [24]. When the SUT is not safety-critical, testing to improve software reliability may be a more convenient aim than debug testing: in other words, we acknowledge that we would never be able to find all faults, and aim at focusing our efforts towards those ones whose removal mostly contributes to increase reliability.

Pioneered in the 70's by Musa [30], software reliability testing is based on the notion of the *operational profile* [31, 40], which provides a quantitative characterization of how a system will be used in the field. In operational profile-based testing (OP testing in the following), the SUT is thus tested by trying to reproduce how its final users will exercise it, so that the failures are detected with the same likelihood they would be experimented by those users in operation.

The operational profile is normally built by associating the points in the input domain D with values representing the probability to be invoked in operation. Making such association is a difficult task; the best case is when historical data are available, otherwise this can be done by domain experts. Usually, D is divided into M subdomains $D_1, \ldots, D_m$, so that the inputs within a partition are estimated as having the same probability of occurrence in operation. The operational profile is then defined by a probability distribution over the partitions D_i: a value p_i denotes the probability that in operation an input is selected from D_i, with $\sum_{i=1}^{M} p_i = 1$. The software reliability, R, can then be defined [16] as:

$$R = 1 - \sum_{t \in F} p_t \tag{2}$$

where F is the (unknown) set of failure-causing inputs and p_t is the expected probability of occurrence in operation of input t.

OP testing has been shown to be an effective strategy, both in theory [16] and in practice, *e.g.*, [42, 14]. With this strategy, when the test is stopped (for instance because of imperative schedule constraints) and the software released, testers are ensured that the most-frequently invoked operations have received the greatest attention, so that the delivered reliability is at the maximum level achievable under the given test resources [26].

However, OP testing faces difficult challenges that may hinder its broad take-up: first, an operational profile may not be readily available and its derivation

can be costly and complex [22]; second, as more frequent failures are detected and removed, the application of OP testing may progressively lose efficacy.

The latter problem is known as the *saturation effect* [22]. Actually, it is not a prerogative of OP testing, but could affect any test technique. To counteract saturation, research has shown that it is convenient to always consider a *combination* of different testing strategies, which target different types of faults and can together achieve higher effectiveness than the individual application of the most effective technique [25]. Considering specifically reliability improvement, the authors of [11] suggest that the combination of techniques should aim at exposing failures with high occurrence probability, but also as many *failure regions* as possible.[1]

4 How are coverage and reliability related?

4.1 Ways of combining coverage measures and operational profile

In the previous sections we have overviewed two widely used testing strategies, which employ different techniques and pursue different goals. Indeed, coverage testing and OP testing have formed two separate threads of the software testing literature, with little overlaps (see Section 5).

In recent work, we have addressed the question whether and how coverage and OP testing techniques could mutually benefit each other towards the goal of increasing software testing effectiveness for reliability improvement. Indeed, we have achieved encouraging results in either directions.

On the one hand, we have found that coverage testing can be made more cost-effective if not all entities are indiscriminately targeted, but a subset of entities is selected based on their relevance for the final user. In other terms, we have somehow embedded a notion of operational profile within the definition of coverage measures. This research has been presented in [29], and is summarized in Section 4.2.

On the other hand, we have found that using coverage information can help prevent the saturation effect of OP testing and achieve higher effectiveness in reliability improvement. In other terms, to further improve reliability beyond a certain point, within a selected input subdomain the testing should target those entities that are the most rarely covered. This research has been presented in [5], and is summarized in Section 4.3.

4.2 Mimicking operational profile by means of coverage count spectrum

The leading idea of OP testing is exercising the SUT in similar way to how their final users would do. OP testing is inherently a black-box technique, since it disregards the SUT internal structure. Conversely, in coverage testing, a tester tries to exercise the SUT thoroughly without leaving parts untested, no matter of whether and how final users will exercise them. One attractive feature of coverage testing is the availability of a simple and intuitive stopping rule, which is provided, as said, by the coverage measure. On its side, OP testing lacks such a straightforward adequacy criterion.

[1] A failure region is the set of failure points eliminated by a program change [16].

In traditional coverage testing, while testing proceeds each entity is marked as covered or not covered, *i.e.*, from monitoring code coverage testers derive the so-called *hit spectrum*. In general, a program spectrum [19] characterizes a program's behavior by recording the set of entities that are exercised as it executes. The hit spectrum, in particular, records if an entity is covered ("hit") or not. When used in operation, the different program entities will be covered with different frequencies. Some entities will never be exercised, others will be accessed only few times, and others will be covered very frequently. The hit spectrum does not give any information about this varied usage of program entities, beyond revealing that some entities have never been exercised and hence are probably "out-of-scope". Conversely, the *count spectrum* records how many times an entity is exercised: by referring during coverage testing to the count spectrum rather than to the normally used hit spectrum, we keep track of the frequency with which each entity is covered.

As an example, Table 1 displays the *branch-hit* and *branch-count* spectra of two test cases TC_1 and TC_2 exercised during a test campaign. Both TC_1 and TC_2 cover the same set of branches, thus their hit spectra are identical. If we look at their count spectra, we can notice that TC_1 and TC_2 exercise the SUT quite differently.

Table 1 An example of branch-hit and branch-count spectra

	Branch-hit spectrum		*Branch-count* spectrum	
Branch ID	TC_1	TC_2	TC_1	TC_2
b_1	1	1	5	23
b_2	0	0	0	0
b_3	1	1	1	1
b_4	0	0	0	0
b_5	1	1	85	394
b_6	1	1	9	42
b_7	0	0	0	0
b_8	1	1	28	129
b_9	0	0	0	0

Hence, the count spectrum could be used to obtain an approximate representation of how the final users behaviour impacts on the SUT code. Such intuition inspired us the idea of "*operational coverage*": using the count spectrum, it measures code coverage taking into account whether and how the entities are relevant with respect to a users operational profile.

In principle, the notion can be applied to any existing coverage criterion. In previous work [28, 29], we studied operational coverage for three types of entities, namely statements, branches and functions.

To measure operational coverage, we developed the following method. First, program entities are classified into different importance groups based on the count spectrum. Consider, for instance, three importance groups, denominated *high*, *medium*, and *low*. To cluster entities into these three groups, the list of entities is ordered according to their usage frequency; the first 1/3 entities are assigned 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. Of course, different grouping schemes could be adopted.

Then, different weights are assigned to the importance groups to reflect the operational profile. We gave the highest weight to entities in the *high* group, and the lowest weight to the *low* group. Entities that are never covered are assign a zero weight (they are out-of-scope).

Finally, the operational coverage is computed as the weighted arithmetic mean of the rate of covered entities according to the Equation:

$$\text{Operational coverage} = \frac{\sum_{i=1}^{3} w_i \cdot x_i}{\sum_{i=1}^{3} w_i} \cdot 100(\%) \tag{3}$$

where: x_i is the rate of covered entities from group i; w_i is the weight assigned to group i. Note that reducing the above formula to only one group we re-obtain the formula of traditional coverage as per Equation 1.

Operational coverage can be used both as an adequacy criterion and as a selection criterion. In the former case, we use operational coverage for deciding when to stop testing: intuitively, the coverage measure that we achieve during testing gives a weighted estimation of how many of the entities that are more relevant for the final users have been covered. The weights allow testers to take into account if the not yet covered entities may have a large impact on the delivered reliability. For the same reason, using operational coverage in test selection provides a criterion to prioritize the next test cases to be executed.

In [29], we performed some empirical studies to assess operational coverage and the results confirmed the above intuition. Precisely, operational coverage is better correlated than traditional coverage with the probability that the next test case will not fail while performing OP testing. Regarding test case selection, operational coverage on average outperforms traditional coverage in terms of test suite size and fault detection capability.

4.3 Boosting reliability improvement by targeting the lowest covered entities

As described in Section 3, in OP testing the test cases are selected from the operational profile, aiming at finding the failure-causing inputs that have the highest likelihood of being invoked in operation. However, as we already observed, due to the saturation effect [22], after some testing campaign in which the most frequent faults have been revealed and removed, continuing to perform OP testing will progressively lose its efficacy.

Saturation is a well-known problem, and advanced approaches have been proposed to counteract it. For example, Cotroneo and coauthors [11] have recently developed the RELAI technique that uses an adaptive scheme for redefining the operational profile, dynamically learning from the test outcomes. Indeed, to continue improving reliability, at a certain point it becomes necessary to find a proper strategy to move farther from the most frequently exercised operations and start "digging" in less frequent zones of the input domain.

In line with [25] that suggests to combine different testing approaches, we explored whether considering code coverage as an additional information to the

operational profile helps achieving higher reliability. The intuition is that coverage-driven selection can point to parts of the program that have not been exercised by the operational profile driven test cases and that may contain faults. However, even so, we would like to take into account the user's profile, because the aim remains to improve reliability.

Along such line of reasoning, we have recently developed a hybrid approach that relies on both operational profile and coverage information, the latter specifically considering the above introduced count spectrum [5]. The approach, called *covrel*, works in iterations: each iteration dynamically uses the test outcomes from previous iteration to re-arrange the operational profile. This adaptation is based on an inference method called *Importance Sampling* (IS) method [6], which was previously used in the already cited work [11].

Each iteration consists of two steps. First, a partition of the input domain into subdomains D_i is dynamically redefined. In line with traditional OP testing (see Section 3), this step allows to assign probability values to inputs. More precisely, at each iteration the output of the first step is the number of test cases to execute from within each partition (for more details we refer the reader to [5]). In the second step, among all the inputs within a partition (*i.e.*, having a same occurrence probability), *covrel* selects those that exercise the least covered entities according to the count spectrum. This is the novel aspect of *covrel*, in comparison with the more usual approach of selecting such test cases in random way. Of course, to do so *covrel* assumes that the SUT is instrumented and test traces are tracked, as in any white-box testing strategy.

Note that similarly to operational coverage (Section 4.2), the *covrel* strategy derives the count spectrum and classifies the entities into three different importance groups: *high*, *medium*, and *low*. However, differently from operational coverage, in *covrel* we are interested in covering the most "hidden" entities. Therefore, we assign the weights for the importance groups prioritizing the low group. Then, for each partition, we select the test cases with the highest ranks. The two steps are repeated until the available budget of test cases exhausts.

In [5] we have evaluated *covrel* against traditional OP testing with controlled experiments. The results showed that *covrel* can outperform OP testing and achieve faster a given reliability value. The performance of *covrel* is better considering high values of reliability, confirming the intuition that the extra costs it requires for coverage measurement do pay when a high value of reliability is required.

5 Related work

While a huge literature exists about the topics of coverage testing and OP testing considered individually, here we are concerned with the interplay between the two worlds. As anticipated in Section 4.1, there have been only few overlaps between the two research communities. These overlaps have interested mostly the investigation of the effectiveness of coverage testing in terms of reliability improvement instead of fault finding, as, *e.g.*, in [12, 15] and the usage of coverage information for refining software reliability growth models, as surveyed in [2].

Related approaches of interest are those exploring some direct or indirect knowledge derived from the program code (i.e., white-box information) or from the development process in order to either improve or assess reliability.

Smidts *et al.* consider operational testing as a means to *corroborate* (rather than to assess) an already assessed reliability, by complementing evidences gained in previous phases of the development process (e.g., by white-box testing) [39]. This is a problem particularly felt in ultra-reliable systems, where no failures are observed during testing, making operational testing not able, by itself, to give confidence about reliability.

Neil *et al.* propose to use Bayesian networks (BN) as a means to combine evidences: in their example, many pieces of information coming from development-time activities, including code coverage and operational profile, are used together with test results as evidence to assess reliability [32]. A Bayesian approach is also proposed by Singh *et al.*, who use reliability prediction obtained from UML models as the prior belief for reliability assessment in system operational testing [38].

In a PhD proposal by Omri [33], white-box information is used in combination with the operational profile, again with the aim of estimating reliability; the author applies symbolic execution combined with stratified sampling to derive the most favorable partitions for minimizing the variance of the estimate. We too have conjectured the usage of white-box information such as coverage as a means to modify the belief about the partitions' failure proneness, with the aim of driving the profile-based test generation process [34, 35].

All these approaches try to augment the profile-based testing with other pieces of information so as to expose more reliability-impacting failing inputs. None, however, directly embeds code coverage information into the test selection or generation process like *covrel* [5].

Our operational coverage and *covrel* approaches rely on the coverage count spectrum. The idea of using program spectra to help software validation tasks is not new: program spectra have been used, among others, for fault localization [45] and regression testing [46]. To the best of our knowledge, however, we are the first to compute coverage measures based on program count spectra, for the purpose of reflecting the importance of program entities.

One more feature of our approaches is adaptivity. Many authors have exploited adaptivity for improving testing. A noticeable example is the well-known family of *Adaptive Random Testing* (ART) techniques by Chen et *al.* [8], in which the intuition is to improve random testing by using test results online in order to evenly distribute test cases across the input domain. ART is aimed at debug testing; as such, it does not explicitly target reliability improvement and/or assessment like OP testing. *Adaptive testing*, proposed by Cai et *al.*, uses the operational profile for reliability assessment and foresees adaptation (via controlled Markov chains) in the assignment of test cases to partitions [7]. Both these approaches use neither coverage nor any other development-time information to boost reliability.

To implement adaptivity, we used *Importance Sampling*, a statistical sampling method to approximate the true distribution of a variable of interest [6]. We used it to approximate the unknown distribution of the number of test cases for each partition to maximize delivered reliability. While Importance Sampling is successfully used in many fields, its usage for testing is limited to few papers: Sridharan and Namin used it to prioritize mutation operators in mutation testing [41]; we ourselves used it for test techniques selection [9].

6 Conclusions

A large part of software testing literature evaluates the effectiveness of testing techniques based on the faults found, irrespectively of the potential likelihood and impact of such faults. In this way, among several test techniques the one that finds the highest number of faults would be considered the most effective, but this might not correspond to reality. If the faults found are never experienced in practice, the test technique would not be very effective.

In this work, considering that test effectiveness should be evaluated based on the delivered reliability [16], we have discussed some results from combining two usually separated test strategies: white-box coverage criteria and black-box operational testing. The former exploits knowledge of program internals, the latter of program usage.

We have overviewed two approaches that mix the two strategies following two different intuitions. In operational coverage, we have augmented coverage testing criteria with a notion of user's relevance. The intuition is that if an entity is rarely or never used in operation, coverage of this entity should contribute to coverage measure with lower weight. On the contrary, entities that, based on operational profile, are frequently covered, should be given higher weights. In covrel, we have augmented OP testing with coverage information, targetting the selection of test cases within a domain partition towards those entities that remain hidden, i.e. yielding a lower coverage count. The intuition here is that monitoring coverage along OP testing may help increasing faster the reliability.

The approaches we have developed are just a first attempt to implement what seems a very attractive perspective: by combining information from coverage and operational profile we can achieve a stronger testing technique that yields both a practical stopping rule and mitigates the inherent saturation problem.

Having opened a novel research thread, we are also aware that a myriad of other potential techniques could be devised, only limited by creativity. For example, we have considered coverage of only three more common entities, statement, branch and function. Other entities could have been considered. Moreover, as we hinted in the introduction, we could consider a model of software behaviour and different combinations also involving functional testing strategies.

Acknowledgements This work has been partially 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).

References

1. Adams, E.N.: Optimizing Preventive Service of Software Products. IBM Journal of Research and Development **28**(1), 2–14 (1984)
2. Alrmuny, D.: A Comparative Study of Test Coverage-Based Software Reliability Growth Models. In: Proc. 11th Int. Conference on Information Technology: New Generations, ITNG, pp. 255–259. IEEE (2014)
3. Beizer, B.: Software testing techniques (2nd ed.). Van Nostrand Reinhold Co., New York, NY, USA (1990)
4. Bertolino, A.: Software testing. In: P. Bourque, R. Dupuis (eds.) Software Engineering Body of Knowledge (SWEBOK), chap. 5. IEEE Computer Society (2001)

5. Bertolino, A., Miranda, B., Pietrantuono, R., Russo, S.: Adaptive coverage and operational profile-based testing for reliability improvement. In: Proc. 39th Int. Conference on Software Engineering, ICSE, pp. 541–551. IEEE (2017)
6. Bishop, C.: Pattern Recognition and Machine Learning. Information Science and Statistics. Springer-Verlag, New York, NY, USA (2006)
7. Cai, K.Y., Li, Y.C., Liu, K.: Optimal and adaptive testing for software reliability assessment. Information and Software Technology **46**(15), 989–1000 (2004)
8. Chen, T.Y., Leung, H., Mak, I.K.: Adaptive random testing. In: Proc. 9th Asian Computing Science Conference, *Lecture Notes in Computer Science*, vol. 3321, pp. 320–329. Springer (2004)
9. Cotroneo, D., Pietrantuono, R., Russo, S.: A Learning-based Method for Combining Testing Techniques. In: Proc. 35th Int. Conference on Software Engineering (ICSE), pp. 142–151. IEEE (2013)
10. Cotroneo, D., Pietrantuono, R., Russo, S.: Combining Operational and Debug Testing for Improving Reliability. IEEE Transactions on Reliability **62**(2), 408–423 (2013)
11. Cotroneo, D., Pietrantuono, R., Russo, S.: RELAI Testing: A Technique to Assess and Improve Software Reliability. IEEE Transactions on Software Engineering **42**(5), 452–475 (2016)
12. Del Frate, F., Garg, P., Mathur, A., Pasquini, A.: On the correlation between code coverage and software reliability. In: Proc. 6th Int. Symposium on Software Reliability Engineering, ISSRE, pp. 124–132. IEEE (1995)
13. Dijkstra, E.W.: Structured programming. In: J. N.Buxton, B. Randell (eds.) Software Engineering Techniques, NATO Science Committee (1970)
14. Donnelly, M., Everett, B., Musa, J., Wilson, G., Nikora, A.: Best Current Practice of SRE. In: Handbook of software reliability engineering, chap. 6, pp. 219–254. IEEE Computer Society Press and McGraw-Hill (1996)
15. Frankl, P.G., Deng, Y.: Comparison of delivered reliability of branch, data flow and operational testing: A case study. ACM SIGSOFT Software Engineering Notes **25**(5), 124–134 (2000)
16. Frankl, P.G., Hamlet, R.G., Littlewood, B., Strigini, L.: Evaluating testing methods by delivered reliability. IEEE Transactions on Software Engineering **24**(8), 586–601 (1998)
17. Frankl, P.G., Weyuker, E.J.: An applicable family of data flow testing criteria. IEEE Transactions on Software Engineering **14**(10), 1483–1498 (1988)
18. Gay, G., Staats, M., Whalen, M., Heimdahl, M.P.: The risks of coverage-directed test case generation. IEEE Transactions on Software Engineering **41**(8), 803–819 (2015)
19. Harrold, M.J., Rothermel, G., Wu, R., Yi, L.: An Empirical Investigation of Program Spectra. In: Proc. of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE, pp. 83–90. ACM (1998)
20. Herzig, K.: Let's assume we had to pay for testing. Keynote at the 11th IEEE/ACM International Workshop on Automation of Software Test (2016). URL `https://www.kim-herzig.de/2016/06/28/keynote-ast-2016/`
21. Herzig, K.: There's never enough time to do all the testing you want. In: Perspectives on Data Science for Software Engineering, pp. 91–95. Elsevier (2016)
22. Horgan, J., Mathur, A.: Software testing and reliability. The Handbook of Software Reliability Engineering pp. 531–565 (1996)
23. Inozemtseva, L., Holmes, R.: Coverage is not strongly correlated with test suite effectiveness. In: Proc. 36th Int. Conference on Software Engineering, ICSE, pp. 435–445. ACM (2014)
24. Institute of Electrical and Electronic Engineers: IEEE standard glossary of software engineering terminology - IEEE Standard 610.12 (1990)
25. Littlewood, B., Popov, P., Strigini, L., Shryane, N.: Modelling the effects of combining diverse software fault detection techniques. In: Formal methods and testing, *Lecture Notes in Computer Science*, vol. 4949, pp. 345–366. Springer (2008)
26. Lyu, M.R.: Software reliability engineering: A roadmap. In: Future of Software Engineering, FOSE, pp. 153–170. IEEE (2007)
27. Marick, B.: How to misuse code coverage. In: Proc. 16th Int. Conference on Testing Computer Software, pp. 16–18 (1999)
28. Miranda, B., Bertolino, A.: Does Code Coverage Provide a Good Stopping Rule for Operational Profile Based Testing? In: Proc. 11th Int. Workshop on Automation of Software Test, AST, pp. 22–28. ACM (2016)

29. Miranda, B., Bertolino, A.: An assessment of operational coverage as both an adequacy and a selection criterion for operational profile based testing. Software Quality Journal **26**(4), 1571–1594 (2018)
30. Musa, J.D.: A theory of software reliability and its application. IEEE Transactions on Software Engineering **SE-1**(3), 312–327 (1975)
31. Musa, J.D.: Operational profiles in software-reliability engineering. IEEE Software **10**(2), 14–32 (1993)
32. Neil, M., Fenton, N., Nielson, L.: Building large-scale Bayesian networks. Knowledge Engineering Review **15**(3), 257–284 (2000)
33. Omri, F.: Weighted statistical white-box testing with proportional-optimal stratification. In: WCOP'14 Proc. 19th Int. Doctoral Symposium on Components and Architecture, pp. 19–24. ACM (2014)
34. Pietrantuono, R., Russo, S.: On Adaptive Sampling-Based Testing for Software Reliability Assessment. In: Proc. 27th Int. Symposium on Software Reliability Engineering, ISSRE, pp. 1–11. IEEE (2016)
35. Pietrantuono, R., Russo, S.: Probabilistic Sampling-Based Testing for Accelerated Reliability Assessment. In: Proc. IEEE 18th Int. Conference on Software Quality, Reliability and Security (QRS), pp. 35–46. IEEE (2018)
36. Prause, C.R., Werner, J., Hornig, K., Bosecker, S., Kuhrmann, M.: Is 100% test coverage a reasonable requirement? lessons learned from a space software project. In: M. Felderer, D. Méndez Fernández, B. Turhan, M. Kalinowski, F. Sarro, D. Winkler (eds.) Product-Focused Software Process Improvement, *Lecture Notes in Computer Science*, vol. 10611, pp. 351–367. Springer (2017)
37. Roper, M.: Software testing—searching for the missing link. Information and Software Technology **41**(14), 991–994 (1999)
38. Singh, H., Cortellessa, V., Cukic, B., Gunel, E., Bharadwaj, V.: A Bayesian approach to reliability prediction and assessment of component based systems. In: Proc. 12th Int. Symposium on Software Reliability Engineering, ISSRE, pp. 12–21 (2001)
39. Smidts, C., Cukic, B., Gunel, E., Li, M., Singh, H.: Software reliability corroboration. In: Proc. 27th Annual NASA Goddard/IEEE Software Engineering Workshop, pp. 82–87. IEEE (2002)
40. Smidts, C., Mutha, C., Rodríguez, M., Gerber, M.J.: Software testing with an operational profile: OP definition. ACM Computing Surveys **46**(3), 39:1–39:39 (2014)
41. Sridharan, M., Namin, A.: Prioritizing Mutation Operators Based on Importance Sampling. In: 21st Int. Symposium on Software Reliability Engineering, ISSRE, pp. 378–387. IEEE (2010)
42. Tian, J., Lu, P., Palma, J.: Test-execution-based reliability measurement and modeling for large commercial software. IEEE Transactions on Software Engineering **21**(5), 405–414 (1995)
43. Wei, Y., Meyer, B., Oriol, M.: Is Branch Coverage a Good Measure of Testing Effectiveness? In: B. Meyer, M. Nordio (eds.) Empirical Software Engineering and Verification, *Lecture Notes in Computer Science*, vol. 7007, pp. 194–212. Springer (2012)
44. Weyuker, E.J., Jeng, B.: Analyzing partition testing strategies. IEEE Transactions on Software Engineering **17**(7), 703–711 (1991)
45. Wong, W., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A Survey on Software Fault Localization. IEEE Transactions on Software Engineering **42**(8), 707–740 (2016)
46. Xie, T., Notkin, D.: Checking inside the black box: regression testing by comparing value spectra. IEEE Transactions on Software Engineering **31**(10), 869–883 (2005)
47. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surveys **29**(4), 366–427 (1997)