Testing relative to usage scope: revisiting software coverage criteria

BRENO MIRANDA, Federal University of Pernambuco and ISTI - CNR
ANTONIA BERTOLINO, ISTI - CNR

Coverage criteria provide a useful means to guide software testing, however indiscriminately pursuing full coverage may not always be convenient or meaningful. We revisit the definition of coverage measures, to account for the scope in which the software is used. Entities that are not going to be exercised by the user should not contribute to the coverage ratio. According to this notion, in this paper we provide a definition and a theoretical framework of relative coverage, and discuss its implications on testing theory and practice. We also report the results from some examples of applications of this notion in different scenarios and hint at interesting directions for future research.

CCS Concepts: • **Software and its engineering** →**Software testing and debugging;** • **General and reference** → *Verification;* Metrics;

Additional Key Words and Phrases: Coverage Testing, Relative Coverage

ACM Reference format:
Breno Miranda and Antonia Bertolino. 2020. Testing relative to usage scope: revisiting software coverage criteria. *ACM Trans. Softw. Eng. Methodol.* 0, 0, Article 0 (July 2020), 22 pages.
DOI: 0000001.0000001

1 INTRODUCTION

Testing is a fundamental part of software engineering: it is a practical means to reveal faults, and revealing and removing faults ultimately contribute to improve the quality of software systems. Given its paramount importance, software testing has received much attention from researchers since the early days of the software engineering discipline. As testing usually involves a very costly process, a great amount of this research effort has focused on devising cost-effective testing techniques.

Closely related with cost reduction, one major problem of testing is deciding when to stop, *i.e.*, when one has maximized the chances of revealing faults on a reasonable effort. In principle, there is always a gain in carrying out new tests because of the possibility of revealing further faults. However, this gain reduces as testing proceeds, and so does the cost-effectiveness of testing.

Coverage testing is a widespread approach: it aims at guaranteeing that the tests execute relevant parts of the software code, motivated by the intuition that if a fault is not executed, it cannot be revealed. On the one side, coverage measures can support the decision of when to stop testing by defining a priori some coverage target (test adequacy). On the other side, when the coverage achieved is not sufficient, the coverage report identifies the parts of the software system that have not been exercised, which can be useful for test suite enhancement (test selection).

The history of coverage testing dates back to the 70's when Goodenough and Gerhart [17, 18] introduced the notion of an ideal test criterion, *i.e.*, the criterion that defines what constitutes an adequate test set: a test criterion should be *reliable* and *valid*. To be considered reliable, a test criterion should guarantee the selection of tests that are consistent in their ability to reveal faults. To be considered valid, a test criterion should ensure that for every fault in a program, there exists a complete set of test data capable of revealing that fault. This definition, however, is not practically applicable: showing that a test criterion is reliable and valid would require to know a priori the faults in the program [21, 51].

In 1988, Frankl and Weyuker proposed the *applicable family of data flow testing criteria* [13]. This seminal work represented a shift from theoretical to practical in dealing with test adequacy criteria. The criteria proposed prior to that work could not achieve 100% coverage, because of infeasible paths. After Frankl and Weyuker redefined existing adequacy criteria to cover the entities that could be reached, achieving full coverage (100%) was now possible (under the assumption that path executability is known).

In the years that followed, countless new coverage criteria have been proposed in the literature [53] and, even though they may target widely different entities, they are all based on the same underlying principle: *the kind of entities to be covered is identified (they could be statements, branches, paths, functions, and so on), and a program is not considered to be adequately tested until all entities (or a given percentage thereof) have been executed at least once.* Thus coverage is measured as the ratio or percentage of covered entities with respect to the total number of entities in the program under test.

Indeed, a large part of research on coverage testing has focused on identifying increasingly demanding and sophisticated criteria, motivated by the need to do more comprehensive testing. In our work, we depart from such thread of literature, and take a completely different stand: since testing resources are usually limited, we observe that the effort spent to cover all entities indiscriminately may be not well employed. We propose instead that coverage adequacy requirements should apply only to those entities that *are relevant in the usage scope*.

As a matter of fact, while the purpose of coverage is to ensure that all parts of code are exercised, this could not be a proper objective in all contexts. Due to the constantly increasing complexity of software systems, achieving full coverage becomes more difficult and, at the same time, less meaningful for some development paradigms (*e.g.*, component-based development, service-oriented architecture, cloud computing), because it is not always the case that all entities are of interest in every context. Consider, for example, the case of black-box reuse [38] with in-house component development. In such case, existing software artifacts found in in-house repositories are used to build new software, but modifications (*e.g.*, refactoring) are not allowed. The reuse asset must be used "as is" because other in-house software could depend on it. Assuming that the reused component is being tested in the new context where not all of its methods are of interest, does it make sense to aim for *full* coverage of the component's code?

Because testing is costly and the number of tests that can be executed is limited, we propose to opportunistically revise the definition of coverage testing to account for the *usage context*. More precisely, our proposal is to take into account the way the software system is going to be used to identify the relevance of each entity to the specific *scope*. We use the term "Relative Coverage" to refer to this customized usage-centric way of measuring coverage. To clarify, relevant is different from infeasible (as considered in Frankl and Weyuker's applicable criteria [13]): an entity could be perfectly executable because there exist some inputs that cover it, but not relevant because such inputs are never invoked.

Relative coverage is a generic notion: indeed, the usage context can be declined into many different testing scenarios and induce as many different implementations of the relative coverage measure. In previous works [28, 31, 32], we have in fact defined different instances of relative coverage and shown the validity of the specific approaches. In this paper, we aim at providing the broad theoretical framework of relative coverage and at showing how the previous approaches, individually defined and validated, are actually specific instantiations of one same concept that we considered in each work from different perspectives. It is only by merging together previous works into a unified presentation that we can properly explain the potential of the "relative coverage" concept. We believe that the concept of relative coverage constitutes an important building brick of the software testing discipline: both in theory, because it allows to account for usage context in testing foundations (thus further extending current formulation as in [45]), and in practice, because it supports the development of more cost-effective testing techniques and tools.

In summary, the contributions of this paper include:

- A formal (re)definition of coverage (of which traditional coverage is a specific instantiation);
- A revisitation of testing foundations highlighting the importance and implications of including usage context as one additional dimension of the theory;
- A comparative summary of empirical evaluations of relative coverage approaches collected from previous work, plus a discussion of other possible benefits from further applications of the concept.

The paper is organized as follows: In the next section we review related work; then in Section 3 we present three scenarios to motivate the need for relative coverage measures. In Section 4 we provide the formal definition of relative coverage, and in Section 5 we revise the foundations of testing in light of usage scope. Applications of the new coverage definition are hence presented both in Section 6, where we summarize some results already achieved in previous own work that correspond one-to-one to the three scenarios of Section 3, and in Section 7, where we hint at further potential applications worth of future study. Finally, Section 8 concludes the work.

2 RELATED WORK

In their already cited seminal work "An Applicable Family of Data Flow Testing Criteria" [13], Frankl and Weyuker proposed to circumvent the problem of non-applicability of the data flow testing criteria by requiring the test data to exercise *only* those definition-use associations that are executable. In other words, they proposed to change the denominator of the coverage equation to exclude the infeasible paths. Although determining the executable paths is, in general, an undecidable problem, the prospect of being able to achieve the usually much-desired 100% coverage, encouraged the proposal and evaluation of a multitude of adequacy criteria in the years to follow. Our work is similar to that of Frankl and Weyuker in the sense that our proposal involves another change in the denominator of the coverage equation. This time, we propose the exclusion of entities that are *not relevant* to a given testing context, even if they are perfectly feasible.

Using a different terminology, a similar notion of "testing scope" was introduced in 1997 by Rosenblum [41] in his definition of a formal model of test adequacy for component-based software. His model was defined in terms of subdomain-based test adequacy criteria as proposed in [14], and includes the concepts of adequate *unit* testing and adequate *integration* testing. Considering the case of a program $\mathcal{P}$ that invokes a component $\mathcal{M}$, he divided the input domain of $\mathcal{M}$ between the input subdomains that are "relevant" for $\mathcal{P}$ (*i.e.*, containing inputs that would be invoked by executing $\mathcal{P}$) and those that are irrelevant. Our work is more closely related with his notion of adequate *integration* testing of components, whereby a component $\mathcal{M}$ should be tested with inputs that are relevant for $\mathcal{P}$ (*i.e.*, with inputs of $\mathcal{M}$ that $\mathcal{P}$ uses for its traversals of $\mathcal{M}$), but not with inputs that would never invoked by $\mathcal{P}$. In contrast, traditional coverage of $\mathcal{M}$ would require anyhow to cover all entities, regardless their relevance. Rosenblum's notion of adequacy is very similar to relative coverage. However, he focused on defining an appropriate adequacy criterion for component-based software only. In our work, instead, we revisit in general the topic of coverage testing and propose that coverage criteria should be redefined to account for the way the software, component-based or otherwise, is used in order to provide more meaningful coverage metrics.

Staats et al. [45] introduced a framework that allows the investigation of the interrelationships between four testing artifacts: specifications, programs, tests, and oracles. The novelty of the proposed framework, which is an extension of Gourlay's functional description of testing [19], was the introduction of *oracles* as an artifact that should be considered while conducting research on software testing. The authors claim that most researchers consider pairs of artifacts (programs and tests in the majority of the cases), but the interrelationship between oracles and programs, or that of oracles and tests, have received little attention. In the same work, they also defined the notion of "*complete adequacy criterion*", which is a criterion specified in terms of both the test set and the oracle used. Thus another way of introducing our work could be by extending their framework, which we do later in Section 5.

The very term *relative coverage* has been used in other works before ours. In the work by Bartolini et al. [4] the term is used with the same meaning of our work, *i.e.*, test coverage is measured considering the portions of the system that a relevant to a particular user or user profile. In fact, the work presented in [4] inspired part of the research presented here. Interestingly, we found that relative coverage was also early used by Majumdar and Sen [24] for the empirical evaluation of hybrid concolic testing: they explain that – as a test subject they used a part of a large data structure library – "the absolute branch coverage" would have been very low and would not reflect the true branch coverage in relation to the part they only used. This is precisely our motivation in proposing relative coverage measures.

On the other hand, most previous works [1, 7, 11, 25] use the term of *relative coverage* in a sense that is different from the one we adopt in our work. Precisely they define coverage measures that are relative, but to some aspect (e.g., fault severity, coupling pairs) other than the usage scope. The authors of [7], for example, assume the existence of a fault model that assigns weights to each potential error in an implementation, and their relative coverage computes the error weight revealed by a test suite T as a fraction of the weight of all traces in T.

In another thread of literature, instead, there are works that do not explicitly use the term *relative coverage* [36, 37] but that could be presented in terms of the framework we will introduce in Section 5. In [36], for example, the authors introduced a testing criterion called *field-exhaustive* testing that requires a user-provided limit on the size of data domains, which would fit our definition of usage scope. In [37], the authors introduced value-based coverage and suggested that test cases should contribute differently to the coverage measure depending on the relevance of the input data they exercise. Our definition of *relative coverage* can deal with different weights for different groups of entities and could be easily applied in the context depicted in [37]. According to our framework, [36] and [37] could be considered as different instantiations of relative coverage.

3 MOTIVATION

To better motivate the *relative coverage* we will introduce in Section 4, here we present some examples of testing scenarios where having *usage* as another test dimension is important to compute meaningful coverage metrics.

3.1 Testing of Reused Code in a New Context

When a program uses reused code or third-party components in a context that is different from the original one, some of their entities (*e.g.*, branches) might never be exercised. Consider, for example, the small *example_function* displayed in Listing 1.

```
 1 def example_function(x, y):
 2     if x > 0:
 3         ...  # fault 1
 4     else:
 5         ...
 6         ...  # fault 2
 7     if y > 0:
 8         ...
 9     else:
10         ...
```

Listing 1. *example_function* with two faulty statements

As highlighted in Listing 1, we assume two faulty statements in the *example_function*. The first one can only be triggered if *line 3* is executed: hence only test cases with x bigger than 0 are likely to reveal this fault. The second fault, on the other hand, can only be triggered when *line 6* is traversed: hence only test cases with x equal to or less than 0 are likely to expose this fault. Let us now assume that the aforementioned code is going to be reused in a scope in which x is guaranteed to be *always* bigger than 0. In that case, *line 6* would never be reached and so *fault 2* would never be triggered. In our terminology, *fault 2* is said to be an *out-of-scope fault*, whereas *fault 1* that can be triggered under the known constraint is an *in-scope fault*. We will provide more formal definitions of *in-scope fault* and *out-of-scope fault* in the next section.

In a context where best practices for reusability can be adopted, the natural path to follow would be to refactor the code illustrated in Listing 1 and remove its unused parts. However, for a particular kind of software reuse, namely black-box reuse [38], the reused asset (*e.g.*, a component or a method) must be used *as is* because other in-house software could depend on it. In that case, the application of traditional testing techniques might result in the generation of unnecessary test cases or in misleading coverage measures. In Section 6.1, we provide a brief summary of how we leveraged usage information (the reuse context) to apply our notion of relative coverage and provide meaningful coverage information for the context of black-box reuse.

3.2 Testing in the Absence of Code Coverage Metrics

Service Oriented Architecture (SOA) is a very attractive architectural pattern as it allows pieces of software developed by independent organizations to be dynamically composed to provide richer functionality [4]. However, the same reasons that enable flexible compositions also prevent the application of some traditional testing approaches. Web services usually expose just an interface, which is enough to invoke the service's operations and to develop some general (black-box) test cases. Such an interface, however, is not sufficient for testers willing to evaluate the integration quality between their applications and the independent web services.

For a better illustration of this scenario, let us consider the example of a service provider S implementing a Travel Reservation System (TRS). The TRS includes several operations relative to flight and hotel booking, car reservation, user registration, login and payment. These operations are made available through the service public interface and many operations can be associated with a same group (e.g., the flight booking group contain the operations *FlightLookup*, *CheckSeatAvailability*, *getFlightTicketPrice*, and so on). The services made available by the TRS can be used by many consumers, and each consumer can use a different combination of operations. *Service Consumer A*, for example, could use operations from the flight booking, payment, user registration, and login groups, suggesting that it is a service being used in a travel agency. A different consumer, say *Service Consumer B*, could use operations from car reservation, user registration, login, and payment, suggesting that the service is being used by a car rental company. Service consumers A and B are different clients making use of the same service provider.

Let us now assume that a new service, say *Service Consumer N*, is developed. It is used in a travel agency and it is implemented to invoke the operations available from: flight booking, hotel booking, user registration, login, and payment. Operations from the car reservation group are *never* used. To diligently test the integration of the new service N with the TRS, a tester creates a test suite to exercise the operations implemented by the new service N as well as the operations used from the service provider S.

Because of the separation of concerns, a core principle of SOA, the tester of service N cannot have access to the code of service provider S. For clients to receive code coverage information, the service provider S should be willing to implement an intermediate coverage service (such as the one proposed by Bartolini et al. in [4], for example) that would still adhere to the SOA requirements. Black-box coverage criteria (*e.g.*, operation coverage) should also be implemented through the intermediate coverage service.

However, even if S is willing to make this effort, the simple adoption of traditional coverage metrics would not be helpful for its clients. The reason is that it is not the case that clients would be interested in thoroughly assessing *all* the operations provided by S. Assuming operation coverage, for example, traditional coverage would be calculated by dividing the amount of operations that N invokes by the total amount of operations available in the service provider S, including the ones related to the car reservation group which are *never* invoked by N. If the tester is deliberately not interested in some operations, which, in fact, is the case in this example, this coverage information would not be meaningful.

The testing scenario illustrated here is similar to the one previously depicted for software reuse in the sense that some operations from S are integrated with new code and (re)used in a different context. For the testing of N, we would like to measure coverage over the actually relevant operations (in-scope entities), and not over the whole set of operations available. In Section 6.2, we briefly summarize our proposal of a coverage criterion that customizes coverage information for a given user by leveraging usage information from similar users in a collaborative testing approach.

3.3 Testing of Systems with Heterogeneous Customer Base

The user's perceived reliability of a system can vary across users depending on how they use the system [33]. To illustrate that, let us consider the example of a publication management system used by different types of users. Table 1 lists the different roles and how frequently they use the various operations provided by the system. There are the *authors*, who use the system mainly to add new publications and to browse the existing ones. The *librarians*, who make sure that all the publications added to the database meet the library guidelines. Librarians are also in charge of making weekly backups to make sure that the database containing the publications of their respective institutes is safe. And there are the *system administrators*, who are in charge of adding and removing users, granting permissions according to the user role, and making daily backups of the whole system to make sure that all the publications are safe.

Table 1. Operational Profiles for a Publication Management System

Operations	Occurrence Probability		
	Authors	Librarians	System Administrators
Add publication	0.30	0.15	0.00
Browse publication	0.70	0.45	0.00
Add user	0.00	0.20	0.20
Remove user	0.00	0.10	0.10
Set/Update user permissions	0.00	0.05	0.28
Database backup	0.00	0.05	0.42

Now suppose that, after the last update, the publication management system started presenting intermittent issues during the backup operation. These issues go unnoticed by the authors as they do not make use of the backup operation. This group of users feels that the system is reliable as it effectively fulfills the users' needs. The librarians notice occasional problems while performing the backup. For them, the system is fairly reliable as it meets their needs most of the time. The system administrators, on the other hand, will notice the failures in the backup operation very frequently as they need to perform this action on a daily basis. Their opinion regarding the system's reliability is certainly very different from the other groups of users. This is a classic scenario of a system with heterogeneous customer base. In Section 6.3, we briefly discuss our proposal of a coverage criterion for deciding when to stop testing and for selecting test cases when the testing activities are carried out with a specific group of users in mind.

4 RELATIVE COVERAGE

Coverage is generally calculated as the ratio between the entities covered and the total number of entities (see Equation 1). In this traditional way of measuring coverage, the number of entities expected to be covered always embraces the full set of available entities of a given kind.

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

The notion of *Relative Coverage*, presented here, focuses on having a flexible and context-dependent number of targeted entities by taking into consideration the way a given program is going to be exercised by its users, i.e., by taking the *testing scope* into consideration. We define *testing scope*, or simply *scope*, as follows:

Definition 4.1 ((Testing) Scope). A subset of the (testing) input domain. More formally, given the input domain $\mathcal{D}$ of a program P, and given a set C of constraints over $\mathcal{D}$, a (testing) scope $\mathcal{S}$ is defined by the set of (test) input values to program P that satisfy the constraints C.

In the above definition, the constraints can be as formal as algebraic expressions over P input variables, or general properties delimiting the input domain $\mathcal{D}$. Notice that in this work we are not providing a general definition of an approach to identify the testing scope. Rather, we assume that the information regarding the specific testing context is available, and provide some examples in the developed case studies.

The basic equation used to calculate relative coverage (see Equation 2) is analogous to that of traditional coverage. Indeed, the only difference is that one should first identify the set of *in-scope* entities for a given testing scope before computing coverage metrics. We use the term *in-scope entities* to refer to the entities from the system under test that are relevant to a given testing scope. The remaining ones are referred to as *out-of-scope entities*. *In-scope entities* and *out-of-scope entities* are more formally defined as follows:

Definition 4.2 (In-scope entities). The set of entities relevant to a given scope. More formally, given a program P with entities $\{e_1, e_2, ..., e_n\}$ and a scope $\mathcal{S}$, the set of in-scope entities with regards to S is $\mathcal{E}_s = \{e_{i_1}, e_{i_2}, ..., e_{i_n}\}$ such that $\forall e_{i_j}$ there exists some input $v \in S$ that covers them.

Definition 4.3 (Out-of-scope entities). The set of entities that are not relevant to a given scope (they are not covered by any input $v \in S$).

$$\text{Relative coverage}_{basic} = \frac{\textit{number of covered entities}}{\textit{number of in-scope entities}} \cdot 100(\%) \tag{2}$$

Equation 2 improves on the traditional coverage equation as it focuses on the entities that are important for the testing scope only. However, similarly to the traditional coverage, it considers that all the entities contribute equally for the coverage measure, i.e., all the entities are considered to have the same importance for the testing scope. This is not true for contexts where entities have different importance depending on how frequently they are expected to be invoked in operation (*e.g.*, in the context of operation profile based testing). Thus, we provide a more generic definition for relative coverage (Equation 3) that can account for different weights for the different groups of entities.

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

where:

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

Notice that when only one group of entities exist or when all the groups have the same weight, Equation 3 reduces to Equation 2.

Computing coverage metrics taking into account only the entities that are relevant to a given testing scope is an attractive idea. However, it brings the difficult challenge of having to identify *what are the in-scope entities for a given testing scope*, *i.e.*, what entities go in the denominator of the coverage ratio? There is not a unique answer, as it depends on the meaning attributed to scope. However, the good news is that different techniques can be leveraged to automate the process of identifying the set of in-scope entities once a testing scope is defined. In the following we provide a summary of different approaches we have proposed and the results of evaluating the usefulness of relative coverage when used as adequacy and selection criteria.

Disclaimer. Having introduced the notion of relative coverage, two things need to be made clear. First, the motivation behind the adoption of relative coverage is *not* in merely achieving a higher coverage score (by getting rid of out-of-scope entities). Rather, it aims at providing a more *realistic estimate* of what could be achieved by augmenting the test suite. On this regard, we stress that coverage metrics are useful to guide developers and testers to find areas in the program that have not been exercised and as a stopping criterion for the testing activities if a given target is defined in advance. However, they are *not* a measure of quality or correctness of the program being tested. Achieving 100% of statement coverage is not necessarily correlated to the quality of the testing performed as some paths may be missed, and even reaching 100% path coverage, which is impractical in most of the cases (because of infeasible paths or because the program may contains cycles, which would cause the number of paths to be infinite or too large, for example), is not necessarily a guarantee of no remaining bug. If test cases are created just to increase the coverage rate without the objective of exercising error-prone areas and possibly revealing faults, achieving high levels of coverage does not help in gaining confidence about the level of quality of the software being tested.

Second, by proposing to focus the testing efforts on the in-scope entities we do *not* mean that less testing should be carried out. It is a well-known fact that exhaustive testing is impractical in the vast majority of the cases. As a result, typically some testing strategy is defined to make sure that the available resources are used efficiently. However, it is also well known that the testing activities are often penalized in the case of project time shortening. In a survey conducted by Torkar and Mankefors [47], 60% of the developers claimed that verification and validation was the first thing that was neglected in case of time shortage during a project. When feasible, covering even out-of-scope entities could provide enhanced confidence, but *if testing time and resources are limited*, then it seems sensible to target first those entities that are in-scope, because doing so we improve test effectiveness on *relevant* faults.

In Section 6 we will present the application of relative coverage to different testing contexts. We will introduce new adequacy criteria and, for each one of them, we propose a different definition for what in-scope means, which may depend on the context, on the user, or on another factor. We baptized each one of the new adequacy criteria introduced with mnemonics for ease of association with the explored testing scenario, but all of them are simply different instantiations of the relative coverage equation. More precisely, we will introduce: (i) "*relevant coverage*", a coverage criterion tailored for the context of software reuse; (ii) "*social coverage*", a coverage criterion that customizes coverage information to given user by leveraging coverage data from similar users; and (iii) "*operational coverage*", a coverage criterion for test adequacy and selection for operational profile based testing.

5 TESTING FOUNDATIONS RE-REVISITED

In the previous sections we have proposed a novel formulation of test coverage. In this section we look at our proposal from a theoretical perspective and explain that the usage scope, which we have introduced here to refine test coverage measures, should be added to the factors that impact the testing process.

The theoretical foundations of software testing have been studied since the early 80's [19]. More recently, Staats and coauthors [45] observe that existing formalizations of software testing suffer of two important issues: namely, they overlook the role of test oracles, and do not consider all relevant interrelationships among the factors involved in the testing process. Precisely, while previous frameworks consider three factors as having a major role in testing: the program, the specification, and the tests themselves, Staats and coauthors propose to also include the oracle as a fourth primary factor, and discuss all mutual relationships among such four factors (which they also refer to as "testing artifacts").

With the aim to provide a uniform and coherent support to empirical research works that reason on testing methods, Staats and coauthors continue by revisiting the widely referred Gourlay's framework [19]: the latter established a mathematical relation among sets of specification S, programs P and tests T, and used an *ok* predicate over a test $t \in T$, a specification $s \in S$, and a program $p \in P$ as the only possible oracle. More formally, Gourlay's framework defined a theoretical predicate $corr(p, s)$ over specifications and programs implying that a program p is correct with respect to a specification s, and postulated that $\forall p \in P, \forall s \in S, \forall t \in T, corr(p, s) \implies ok(t, p, s)$. Note that the above formula provides a foundation that justifies the usual approximation we do in testing, i.e., drawing conclusions about the ideal (but not observable) notion of correctness of a program p with respect to a specification s based on a sample of observations of the behavior of p (the executed tests).

In revisiting the above formal framework, Staats and coauthors: introduced a set O of test oracles (in place of the unique oracle *ok*), where a test oracle o is a predicate over programs and tests; defined a new $corr_t$ predicate over tests, program and specifications that holds if and only if when running test t, specification s holds for program p; and discussed the mutual relationships among oracles, tests and program correctness. For clarity, in the following we refer to such revisited formulation as the SWH framework (from the initials of authors).

Our notion of scope-based testing considers that the effectiveness and completeness of a testing process should also depend on how the software is used. In fact, we argue that Staats and coauthors [45] still missed another important factor of software testing theory that is the usage scope. As we have discussed in the previous sections, a same program can be used in many different ways and this should be taken into account to discuss about a program "correctness". In this regard, the revisited SWH framework is still incomplete, as it does not allow to include in the assessment of test techniques the many possible user profiles, and does not support reasoning about user-centered measures of correctness: though, such notion of the user-profile is important in several testing approaches, such as for instance testing for reliability improvement or evaluation [34].

Therefore, abstracting the notion of scope-based testing here introduced, we also propose a revised conceptual framework for the testing process as illustrated in Figure 1, which expands the SWH one with usage scope U as a fifth factor.

In SWH the authors provided an intuitive diagrammatic vision of the testing factors and their relationships, as a graph with four nodes: program P, specification S, tests T and oracle O, and all logical one-to-one relations between such nodes as annotated edges. We add to that graph a fifth node U for usage scope, and drawing all possible connections to account for all possible logical relations we obtain an hexahedron. Instead of annotating its edges as in SWH framework, for readability the relations between nodes are then reported in Table 2.

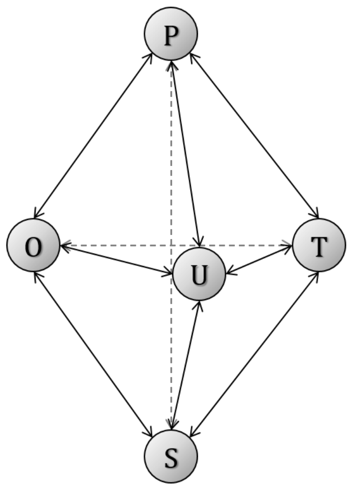

Fig. 1. Relationships between testing factors including the usage scope U)

We also re-revisit the formal testing foundations by Staats and coauthors as follows: we define the usage scope $u \in U$ as a predicate over tests and specifications, such that $u(t, s)$ holds if the test t is in scope. In contrast, existing frameworks all considered u as always true. Then we replace the $corr_t$ predicate in the revisited framework with a further modified $corr_r$ predicate over tests, programs and specifications, such that: $\forall p \in P, \forall s \in S, corr(p, s) \implies \forall t \in T : u(t, s), corr_r(t, p, s)$. That is to say, in our re-revisited framework we approximate the ideal notion of correctness of p with respect to s with a sample of test observations relative to a user's point of view, formalized by the set U.

In plain words, Staats and coauthors expanded Gourlay's original framework by taking into account that for a same set of tests different oracles could provide different approximations of correctness. On top of this, our further reformulation can allow to also take into account that even using a same oracle, different usage scopes can provide different observation views (*i.e.*, different test sets), and hence again different correctness approximations.

Table 2. Relationships between testing factors

	Specification (S)	**Program** (P)	**Tests** (T)	**Oracles** (O)	**Usage** (U)
(S)		S may guide P implementation	S may guide T selection	S may guide O derivation	S may guide U definition
(P)	P attempts to implement S		Syntactic structure may guide test selection	Observability of P limits information available to O	Usefulness of P parts is determined by U
(T)	Tests designed to distinguish incorrect P from S	Semantics determines propagation of errors of each test		Tests suggest variables worth observing	T can be weighted according to U
(O)	O approximates S	O may predicate over P paths	Combination of O and T determines efficacy of testing process		Combination of O and U determines efficacy of testing process
(U)	U captures a subset of S	U determines the parts of P that are relevant	U impacts on test process effectiveness	U limits O scope	

It is beyond the scope of this paper to fully expand the theoretical implications of usage scope in testing foundations. We only briefly and informally discuss here a couple of examples: how our formulation can more comprehensively support reasoning about the testability of a software program (also discussed in SWH [45]) and how it allows for modeling operational testing (not discussed in SWH [45]).

Testability. Among other works discussing testing theory, Staats and coauthors revisit Voas's definition of software testability, defined as the *probability that the program will fail if faulty* [48, 49]. In [45], the authors focus most of their discussion on showing how such definition is incomplete because it lacks to consider the impact of the oracle in revealing a fault, in particular in fault propagation. This observation is true, and indeed the incompleteness of Voas's original definition of testability and the need to also take into account the oracle coverage was already recognized and addressed in [6]. In the latter, Bertolino and Strigini defined a program's testability as *the probability that a test on an input drawn from a specified input distribution is rejected, given a specified oracle and given that the program is faulty.* However, what we want to notice here is that both Voas's (incomplete) definition of testability, and the following (revised) one are also conditional on the input distribution: indeed, a program may yield different testability measures depending on the input tests. This can be discussed in our framework, but not in SWH.

Operational Testing. In [12], Frankl and coauthors propose an analytical framework in which they model and compare Operational testing vs. "Debug'" testing. To distinguish the two approaches, they observe that the probability of failure for a randomly selected test input (and hence a program's reliability) depends on two factors: *i)* the *operational profile* that assigns different selection probabilities to the points in the input domain, and *ii)* an hypothetical labeling of all such points as failing or successful. Based on such concept, operational testing focuses on developing an input profile that approximates as closely as possible the distribution by which the software will be used in the field (mostly regardless of input labeling). Debug testing instead applies methods or heuristics to maximize the probability to find inputs labeled as failing (mostly regardless of the operational profile). Our re-revisited framework can support reasoning of operational testing properties properly taking into account how it is used.

6 APPLICATIONS OF RELATIVE COVERAGE

We have discussed so far a conceptual idea of a customized usage-centric way for measuring coverage. The idea is not only important from a theoretical viewpoint, but can also find useful applications in practical testing: in some previous works we have applied it to real-world case studies and quantitatively evaluated the results of adopting relative coverage in different domains.

In the next sections we provide, for each one of the scenarios illustrated in the Motivation (Section 3), a brief summary of how we applied the notion of relative coverage for the given context and the main results achieved. For a quick overview, Table 3 summarizes and compares the application of relative coverage for the different contexts.

6.1 Relevant Coverage

For the scenario illustrated in Section 3.1, where source code is reused "as is" and refactoring is not an option, an ideal coverage criterion should be capable of measuring the extent to which the portions of the reused code that are relevant to new context are exercised.

In [29] we introduced relevant coverage: a coverage criterion tailored for the context of black-box reuse. For the relevant coverage, the in-scope entities are those that are relevant for the new (reuse) context. Entities from the legacy code of the reused assets that are not expected to be invoked in the new (reuse) context, on the other hand, are considered to be out-of-scope.

For identifying the entities that are relevant to the new context (i.e., the in-scope entities), the testing scope must be known. Once the input domain constraints are defined, the testing scope can be mapped into in-scope entities. Different strategies could be adopted for performing this mapping, *e.g.*, dynamic symbolic execution (DSE) [8]; program slicing [9]; a reachability algorithm applied on top of the static call graph of the target program [40]; among others. In [29] we used DSE guided by the input domain constraints for exploring the source code.

We conducted experiments to evaluate the usefulness of relevant coverage as both adequacy and selection criterion. When used as an adequacy criterion, the main result observed was that relevant coverage was capable of achieving high levels of coverage with much smaller tests suites, when compared with traditional coverage, and with little impact on the fault detection capability [27]. As for selection criterion, besides using the relevant coverage for test case selection, we used it also to minimize and prioritize test suites [31]:

when applied to test case selection and minimization our approach considerably reduced the test suite size, with small to no additional impact on fault detection capability (considering both in-scope and all faults); when applied to test case prioritization, our approach improved the average rate of faults detected when considering faults that are in scope, while remaining competitive when considering all faults. Refer to [27, 31] for detailed results.

6.2 Social Coverage

For the scenario illustrated in Section 3.2, where the user does not have access to the source code, we introduced social coverage [26, 28], a coverage criterion that customizes coverage information in a given context based on coverage data collected from similar users.

Social coverage was conceived for black-box environments having some notion of testing community (i.e., several users/programs using/testing the service under test). The notion of testing community is important to allow coverage information to be leveraged in a collaborative testing approach. For social coverage to work, the service under test should be willing to collect and to share usage information with its users.

For social coverage the in-scope entities are those that are invoked by similar users, whereas the entities that are never invoked by similar users are considered to be out-of-scope. Thus, for identifying the entities that are in-scope to a given user $\mathcal{U}$, the service under test first identifies which users are similar to $\mathcal{U}$ and then it suggests which entities could be of interest to user $\mathcal{U}$.

For evaluating the social coverage we applied it in the context of a real-world service-oriented application and we were able to predict the entities that would be of interest for a given user with an average precision of 97% and average recall of 75% [28]. As part of our future work, we plan to evaluate the effectiveness of social coverage for test adequacy and selection.

6.3 Operational Coverage

For the scenario illustrated in Section 3.3 we introduced in [30] a coverage criterion for operational profile-based testing 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 profile-based testing and coverage-based testing provide two quite diverse software testing approaches. The former is a black-box approach: the test cases are selected from the input domain, trying to reproduce how the software will be used in practice, with the aim of rapidly detecting those failures that would occur most frequently in operation. The latter 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. By mapping operations (black-box) into code entities (white-box), operational coverage allows us to use these two approaches in combination.

Operational coverage assumes the existence of an operational profile, which is a quantitative characterization of how a system is used [33]. The operational profile could be derived by domain experts during the specification stage, or it could be obtained from real world usage, e.g., by monitoring field data by means of an infrastructure such as Gamma [35].

For relevant coverage and for social coverage presented in the previous sections, an entity is either in-scope or out-of-scope. Operational coverage further classifies the in-scope entities into different importance groups according to their frequency of usage. For doing that we make use of program spectra [20]. A program spectrum characterizes a program's behavior by recording the set of entities that are exercised as the program executes. More precisely, for operational coverage we use *count* spectrum, which indicates the number of times a given entity was executed. The *hit* spectrum, on the other hand, which is used by relevant coverage and by social coverage, only indicates whether or not a given entity has been executed (hit). Once the entities are classified into different importance groups we can assign the weights of each group and calculate operational coverage as in Equation 3.

We conducted experiments to evaluate the adoption of operational coverage 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. The results of our studies, reported in [32], showed that operational coverage is better correlated than traditional coverage with the failure probability of test cases derived according to the user's profile. 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 traditional coverage in terms of test suite size and fault detection capability when we look at the average results.

The importance groups as well as the weights assigned to them play a fundamental role in the operational coverage computation. In [32] we assigned more weight to the group of entities frequently exercised and less weight to the group of entities scarcely exercised, so to privilege those entities that are expected to be exercised more often 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 group of entities scarcely exercised [5]. As for any testing strategy, considering the context and the testing objectives is of fundamental importance when defining the operational coverage parameters.

Table 3. Examples of relative coverage applied to different contexts

	Relevant Coverage	Social Coverage	Operational Coverage
Coverage context:	A coverage criterion tailored for the context of black-box reuse	A coverage criterion that customizes coverage information to a given user based on coverage data from similar users	A coverage criterion for the context of operational profile based testing
White-box vs Black-box:	White-box	Black-box	Maps operations (black-box) into source code entities (white-box)
Type of program spectrum used by the coverage metric:	Hit spectrum	Hit spectrum	Count spectrum
Which entities are in-scope?	Set of entities that are relevant for the new (reuse) context	Set of entities that are invoked by similar users	Set of entities that are relevant to the user's operational profile

Table 3. Examples of relative coverage applied to different contexts

	Relevant Coverage	Social Coverage	Operational Coverage
Which entities are out-of-scope?	Set of entities from the legacy code of the reuse assets that are never invoked in the new (reuse) context	Set of entities that, although reachable, are never invoked by the target user or by similar users	Entities that are not relevant to the user's operational profile
Different weighting for the in-scope entities?	No. All the in-scope entities are considered to have the same importance, *i.e.*, they contribute equally to the coverage metric.	No. All the in-scope entities are considered to have the same importance, *i.e.*, they contribute equally to the coverage metric.	Yes. In-scope entities can be further classified into different groups according to their frequency of usage for a given operational profile. Each importance group contributes in a different way to the coverage metric.
Ways for identifying the set of in-scope entities that we have already explored:	Identification of the input domain constraints + guided dynamic symbolic execution	Identification of similar users based on their historical usage of the target program/code	Derived from the operational profile
Alternative ways for identifying the set of in-scope entities:	Reachability algorithm on the static call graph of the given program; Program slicing	Clustering techniques; Different similarity metrics can be adopted	Real usage data collected using profiling techniques (as in continuous testing, for example)
Main results achieved when used as an Adequacy Criterion:	Fewer test cases required (when compared with traditional coverage) to achieve a given coverage goal with little impact on the fault detection capability.	N/A	Operational coverage is better correlated with failure probability when compared to traditional coverage

Table 3. Examples of relative coverage applied to different contexts

	Relevant Coverage	Social Coverage	Operational Coverage
Main results achieved when used as a Selection Criterion:	When applied to test case selection and minimization our approach considerably reduced the test suite size, with small to no additional impact on fault detection capability (considering both in-scope and all faults). When applied to prioritization, our approach improved the average rate of faults detected when considering faults that are in scope, while remaining competitive when considering all faults.	N/A	Test suites selected using operational coverage were able to find more faults (when compared to traditional coverage) in the majority of the cases investigated.
Other results:		We evaluated the effectiveness of our approach in predicting the set of in-scope entities based on the analysis of similar users. Our approach achieved precision rates ranging from 73% to 97% (depending on the Jaccard similarity coefficient adopted) and recall ranging from 67% to 75%.	

7 FURTHER IMPLICATIONS FOR SOFTWARE ENGINEERING

7.1 Test Oracles

The availability of some mechanism for determining whether the execution of a test case has passed or failed is a fundamental assumption when conducting software testing. Such a mechanism is called a *test oracle* [3, 22]. Achieving high levels of coverage is not sufficient *per se* to guarantee that appropriate testing has been accomplished: if the test cases exercise some entities without having proper oracles, test success cannot guarantee that the expected behavior of the software under testing is satisfied. In other words, without strong and accurate test oracles, test cases are not helpful in revealing potential faults. This is in line with the work from Zhang and Mesbah [52] that found that test suite's effectiveness is strongly correlated with the number of test assertions and assertion coverage.

Test oracles generation as well as their quality evaluation is not in the scope of this work. If we were to describe our work based on the framework from Staats et al. [45], in all the experiments reported in Section 6 we changed the coverage criteria and kept the oracles constant as we reused the ones that were made available with the subjects that we investigated. However, as we propose to focus the testing effort on the entities that are relevant for a given testing context, one immediate impact that relative coverage might have concerning test oracles is that, by minimizing the number of entities to be covered, the expected number of test oracles required might also decrease. Naturally, the savings in test effort would be more pronounced in the cases where the test oracles need to be derived manually. Another possible impact is that some instantiations of relative coverage might require oracles capable of adapting the program's expected output based on the user (this is evident for operational coverage, for example). In the future we plan to investigate more deeply the influence of test oracles on the performance of relative coverage.

7.2 Software Instrumentation

Instrumentation is a technique that adds extra code to a program or environment for monitoring/changing some program behavior [15]. It can be used for many things, including simulation, performance analysis, software profiling, etc. When used for collecting coverage metrics, the current instrumentation techniques generally involve compiling and linking the application program along with instrumented code. Because this process usually implies some overhead on the program's runtime, a lot of research has been conducted in the pursuit of more cost-effective methods and techniques for collecting coverage data [2, 10, 46]. Relative coverage requires upfront identification of the set of in-scope entities (i.e., those entities that would be of interest in a given context). Thus, the adoption of our approach requires fewer points of instrumentation to be added in the code, which is reflected directly in the program's runtime. Further benefits could be achieved if our approach is combined with a lightweight instrumentation technique (e.g., [23]). We plan to investigate this in future work.

7.3 Test Case Generation

Given the high costs associated with the creation of test cases, many approaches and tools have been proposed for the automated generation of test cases targeting maximized coverage of some predefined criterion [16, 39, 43]. With relative coverage, instead of targeting maximized coverage, it is possible to guide the test case generation towards the parts of the software that are relevant for the testing scope. As relative coverage presupposes the upfront identification of the in-scope entities, such information could be used for guiding the generation task. To minimize the risks of coverage-directed test case generation as reported in [44], the set of in-scope entities could be used as a supplement — and not as a target per se — for the test generation. In our future work, we plan to further investigate the effectiveness of test case generation approaches guided by relative coverage.

7.4 Regression Testing

Most regression test selection (RTS) techniques select tests based on information about the code of the program and the modified version [42]. Coverage-based techniques aim at identifying code entities (e.g., functions, statements, etc) that have been modified, or may be affected by the modifications, with respect to the previous version of the program, and select tests from an existing test suite to exercise those entities. During the test execution phase, if coverage is computed to assess the selection or to track the progress of the test execution, the percentage of covered entities is meaningless if measured in the traditional way considering all the program entities. If, on the other hand, coverage is computed on the current version of the program based on the set of entities that have been affected (added/modified), what we would call *in-scope* entities, it might be possible to assess how the selected test cases cover the new/modified parts of the program. Assessing the coverage of a RTS test suite considering only the set of affected entities can, thus, be interpreted as an instantiation of relative coverage according to our framework and definitions.

8 CONCLUSIONS AND FUTURE WORK

We have proposed a conceptual framework that revisits the goals of coverage test criteria, from targeting indiscriminately all the entities in the software under test to instead concentrating the testing effort on those entities that are relevant in the usage context. Under such novel perspective, coverage testing assumes a different meaning, because it becomes relative to user, similarly to the notion of software reliability testing.

The notion of relative coverage changes the way coverage is measured, not the definition or identification of entities. To make the approach applicable, though, we need a technique for distinguishing relevant entities from not relevant ones. In the paper we have provided a few examples of instantiations of scope-based coverage, which span both white-box and black-box criteria. We have also instantiated a more sophisticated measure of relevance that not only takes into account whether an entity should be covered or not (hit-spectrum), but also considers if the entity should be covered more or less frequently (count-spectrum). Preliminary experimentation carried out in related works (summarized in Section 6) has shown that using relative coverage as a test adequacy or selection criterion can improve the cost-effectiveness of testing in comparison with traditional coverage testing, both by requiring less test cases before a desired coverage measure is reached, and by finding more in-scope faults with a same test effort.

We stress that the motivation behind relative coverage is not that of reducing test cases, but that of better targeting the available amount of test cases. In other words, scope-based coverage may leave parts of software untested, based on the knowledge that those parts are not going to be executed. Thus, a component that has been tested within some scope and is later tested in a different scope, needs to be tested again. This is not different from what Weyuker recommended in [50].

Many future research directions can be pursued from here. In fact, the notion of relative coverage is orthogonal to existing test criteria and in principle could be introduced to revise any existing coverage technique. Therefore, in addition to the approaches of Relevant, Social and Operation coverage exemplified in the paper, other criteria could be conceived. For example, we have studied control-flow based criteria, but also data-flow based criteria could be made relative to user's spanning over the data-input domain. Moreover, we would like to explore other possible techniques to identify the relevant entities. Other potential applications of scope-based coverage have been discussed in Section 7.

Finally, although we have carried out our evaluations using real-world programs, we would like to experiment the actual costs and benefit of relative coverage in industrial world.

ACKNOWLEDGMENTS

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

[1] Andrea Arcuri and Gordon Fraser. 2011. On parameter tuning in search based software engineering. In *International Symposium on Search Based Software Engineering*. Springer, 33–47.

[2] Matthew Arnold and Barbara G Ryder. 2001. A framework for reducing the cost of instrumented code. *Acm Sigplan Notices* 36, 5 (2001), 168–179.

[3] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2015), 507–525.

[4] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. 2011. Bringing white-box testing to service oriented architectures through a service oriented approach. *Journal of Systems and Software* 84, 4 (2011), 655–668.

[5] Antonia Bertolino, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2017. Adaptive Coverage and Operational Profile-based Testing for Reliability Improvement. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 541–551. DOI:http://dx.doi.org/10.1109/ICSE.2017.56

[6] Antonia Bertolino and Lorenzo Strigini. 1996. On the Use of Testability Measures for Dependability Assessment. *IEEE Trans. Software Eng.* 22, 2 (1996), 97–108. DOI:http://dx.doi.org/10.1109/32.485220

[7] Laura Brandán Briones, Ed Brinksma, and Mariëlle Stoelinga. 2006. A semantic framework for test coverage. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 399–414.

[8] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. DOI:http://dx.doi.org/10.1145/2408776.2408795

[9] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. 1998. Conditioned program slicing. *Inf. and Sw Technology* 40(11) (1998), 595–607.

[10] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming Jack Jiang. 2018. An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 305–316.

[11] Domenico Cotroneo, Roberto Natella, and Stefano Russo. 2009. Assessment and improvement of hang detection in the Linux operating system. In *Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on*. IEEE, 288–294.

[12] P.G. Frankl, R.G. Hamlet, Bev Littlewood, and L. Strigini. 1998. Evaluating testing methods by delivered reliability. *IEEE Trans. Softw. Eng.* 24, 8 (Aug 1998), 586–601. DOI:http://dx.doi.org/10.1109/32.707695

[13] P. G. Frankl and E. J. Weyuker. 1988. An Applicable Family of Data Flow Testing Criteria. *IEEE Trans. Softw. Eng.* 14, 10 (Oct. 1988), 1483–1498.

[14] Phyllis G. Frankl and Elaine J. Weyuker. 1993. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering* 19, 3 (1993), 202–213.

[15] Markus Geimer, Sameer S Shende, Allen D Malony, and Felix Wolf. 2009. A generic and configurable source-code instrumentation component. In *International Conference on Computational Science*. Springer, 696–705.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. DOI:http://dx.doi.org/10.1145/1065010.1065036

[17] J.B. Goodenough and S.L. Gerhart. 1975. Toward a theory of test data selection. *IEEE Trans. on Software Engineering* SE-1, 2 (1975), 156–173. DOI:http://dx.doi.org/10.1109/TSE.1975.6312836

[18] J.B. Goodenough and S.L. Gerhart. 1977. Toward a theory of testing: Data selection criteria. *Current Trends in Programming Methodology* 2, 2 (1977), 44–79.

[19] John S. Gourlay. 1983. A mathematical framework for the investigation of testing. *IEEE Transactions on software engineering* 6 (1983), 686–709.

[20] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. 1998. An Empirical Investigation of Program Spectra. In *Proc. PASTE '98*. 83–90. DOI:http://dx.doi.org/10.1145/277631.277647

[21] W. E. Howden. 1976. Reliability of the Path Analysis Testing Strategy. *IEEE Trans. Softw. Eng.* 2, 3 (May 1976), 208–215. DOI:http://dx.doi.org/10.1109/TSE.1976.233816

[22] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test Oracle Assessment and Improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 247–258. DOI:http://dx.doi.org/10.1145/2931037.2931062

[23] Baris Kasikci, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi. 2014. Efficient Tracing of Cold Code via Bias-Free Sampling.. In *USENIX Annual Technical Conference*. 243–254.

[24] Rupak Majumdar and Koushik Sen. 2007. Hybrid concolic testing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 416–426.

[25] Matthias Meitner and Francesca Saglietti. 2014. Target-specific adaptations of coupling-based software reliability testing. In *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*. Springer, 192–206.

[26] Breno Miranda. 2014. A proposal for revisiting coverage testing metrics. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 899–902. DOI:http://dx.doi.org/10.1145/2642937.2653471

[27] Breno Miranda. 2016. *Redefining and Evaluating Coverage Criteria Based on the Testing Scope*. Ph.D. Dissertation. University of Pisa, Italy.

[28] Breno Miranda and Antonia Bertolino. 2014. Social coverage for customized test adequacy and selection criteria. In *9th International Workshop on Automation of Software Test, AST 2014*. 22–28. DOI:http://dx.doi.org/10.1145/2593501.2593505

[29] Breno Miranda and Antonia Bertolino. 2015. Improving Test Coverage Measurement for Reused Software. In *41st Euromicro Conference on Software Engineering and Advanced Applications, EUROMICRO-SEAA 2015, Madeira, Portugal, August 26-28, 2015*. 27–34. DOI:http://dx.doi.org/10.1109/SEAA.2015.69

[30] Breno Miranda and Antonia Bertolino. 2016. 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)*. ACM, New York, NY, USA, 22–28. DOI:http://dx.doi.org/10.1145/2896921.2896934

[31] Breno Miranda and Antonia Bertolino. 2017. Scope-aided test prioritization, selection and minimization for software reuse. *Journal of Systems and Software* 131 (2017), 528 – 549. DOI:http://dx.doi.org/10.1016/j.jss.2016.06.058

[32] Breno Miranda and Antonia Bertolino. 2018. An assessment of operational coverage as both an adequacy and a selection criterion for operational profile based testing. *Software Quality Journal* 26, 4 (01 Dec 2018), 1571–1594. DOI:http://dx.doi.org/10.1007/s11219-017-9388-0

[33] John D Musa. 1993. Operational profiles in software-reliability engineering. *Software, IEEE* 10, 2 (1993), 14–32.

[34] John D. Musa. 1996. Software-Reliability-Engineered Testing. *IEEE Computer* 29, 11 (1996), 61–68.

[35] Alessandro Orso, Donglin Liang, Mary Jean Harrold, and Richard Lipton. 2002. Gamma System: Continuous Evolution of Software After Deployment. In *Proc. Int. Symp. on Sw. Testing and Analysis (ISSTA '02)*. ACM, 65–69. DOI:http://dx.doi.org/10.1145/566172.566182

[36] Pablo Ponzio, Nazareno Aguirre, Marcelo F Frias, and Willem Visser. 2016. Field-exhaustive testing. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 908–919.

[37] Rudolf Ramler, Theodorich Kopetzky, and Wolfgang Platz. 2012. Value-based coverage measurement in requirements-based testing: Lessons learned from an approach implemented in the tosca testsuite. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 363–366.

[38] T. Ravichandran and Marcus A. Rothenberger. 2003. Software Reuse Strategies and Component Markets. *Commun. ACM* 46, 8 (Aug. 2003), 109–114. DOI:http://dx.doi.org/10.1145/859670.859678

[39] Sanjai Rayadurgam and Mats Per Erik Heimdahl. 2001. Coverage based test-case generation using model checkers. In *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*. IEEE, 83–91.

[40] Thomas Reps. 1998. Program analysis via graph reachability. *Information and software technology* 40, 11-12 (1998), 701–726.

[41] D. S. Rosenblum. 1997. *Adequate Testing of Component-Based Software*. Technical Report UCI-ICS-97-34. Un. of California, Irvine, CA.

[42] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6, 2 (1997), 173–210.

[43] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. *SIGSOFT Softw. Eng. Notes* 30, 5 (Sept. 2005), 263–272. DOI:http://dx.doi.org/10.1145/1095430.1081750

[44] Matt Staats, Gregory Gay, Michael Whalen, and Mats Heimdahl. 2012. On the Danger of Coverage Directed Test Case Generation. In *Fundamental Approaches to Software Engineering*, Juan Lara and Andrea Zisman (Eds.). Lecture Notes in Computer Science, Vol. 7212. Springer Berlin Heidelberg, 409–424. DOI:http://dx.doi.org/10.1007/978-3-642-28872-2_28

[45] Matt Staats, Michael W Whalen, and Mats PE Heimdahl. 2011. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd international conference on software engineering*. ACM, 391–400.

[46] Mustafa M Tikir and Jeffrey K Hollingsworth. 2002. Efficient instrumentation for code coverage testing. In *ACM SIGSOFT Software Engineering Notes*, Vol. 27. ACM, 86–96.

[47] R. Torkar and S. Mankefors. 2003. A survey on testing and reuse. In *Software: Science, Technology and Engineering, 2003. SwSTE '03. Proceedings. IEEE International Conference on.* 164–173. DOI:http://dx.doi.org/10.1109/SWSTE.2003.1245437

[48] J. Voas, L. Morell, and K. Miller. 1991. Predicting where faults can hide from testing. *Software, IEEE* 8, 2 (March 1991), 41–48. DOI:http://dx.doi.org/10.1109/52.73748

[49] Jeffrey M. Voas. 1992. PIE: A Dynamic Failure-Based Technique. *IEEE Trans. Software Eng.* 18, 8 (1992), 717–727. DOI:http://dx.doi.org/10.1109/32.153381

[50] E.J. Weyuker. 1998. Testing component-based software: a cautionary tale. *Software, IEEE* 15, 5 (Sep 1998), 54–59. DOI:http://dx.doi.org/10.1109/52.714817

[51] E.J. Weyuker and T.J. Ostrand. 1980. Theories of Program Testing and the Application of Revealing Subdomains. *Software Engineering, IEEE Transactions on* SE-6, 3 (1980), 236–246. DOI:http://dx.doi.org/10.1109/TSE.1980.234485

[52] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*. 214–224.

[53] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software unit test coverage and adequacy. *ACM Comput. Surv.* 29, 4 (1997), 366–427.

Received 21st July, 2020