Cross-coverage testing of functionally equivalent programs

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

Guglielmo De Angelis
IASI-CNR
Rome, Italy
guglielmo.deangelis@iasi.cnr.it

Felicita Di Giandomenico
ISTI-CNR
Pisa, Italy
felicita.digiandomenico@isti.cnr.it

Francesca Lonetti
ISTI-CNR
Pisa, Italy
francesca.lonetti@isti.cnr.it

Abstract—Cross-coverage of a program P refers to the test coverage measured over a different program Q that is functionally equivalent to P. The novel concept of cross-coverage can find useful applications in the test of redundant software. We apply here cross-coverage for test suite augmentation and show that additional test cases generated from the coverage of an equivalent program, referred to as cross tests, can increase the coverage of a program in more effective way than a random baseline. We also observe that -contrary to traditional coverage testing- cross coverage could help finding (artificially created) missing functionality faults.

Index Terms—Code-based testing, cross-coverage, functionally equivalent programs, test suite augmentation.

I. INTRODUCTION

Code coverage testing has been around for decades. Many criteria of increasing sophistication have been defined in hundred academic publications [1], [2], [3]. On the other side, basic (and less costly) techniques as statement or branch coverage are routinely used in industry [4], [5], seconding Beizer's early recommendation that covering all branches should be *the minimum mandatory test requirement* ([6], p. 75).

Although their actual effectiveness in failure detection is debated [7], [8], coverage testing techniques remain attractive because they promote a systematic and measurable test process [9], [10] and allow for automated test generation [11]. One shared argument between detractors and supporters of code-based testing is that achieving full coverage should not be by itself the goal for testing; rather, measuring coverage of otherwise conceived tests (e.g., functional test cases) can be invaluable to identify parts of the program that have not been executed and hence could hide faults that would otherwise go unnoticed. In fact, one well known limit of code-based test techniques is that by construction they cannot detect missing functionality problems, because they can only test what is in the code.

Basically, then, a schematic *how-to* for code-based testing would be *(i)* to monitor the coverage percentage achieved on the software under test (SUT) by the current test suite, and, until this is not satisfactory, *(ii)* to identify (manually or automatically) additional test cases (i.e., test suite augmentation) that can execute the yet untested parts of the code. In such a process, it is understood that the SUT code is available for instrumentation, and it is also tacitly assumed that additional tests not contributing to increase coverage are less, if at all, useful.

In the current literature, the above-described process concerns one single implementation at a time, i.e., the SUT is one program (unit, method, class, . . .) implementing a given function: this SUT is monitored for measuring coverage, and the additional test cases are identified looking at which parts of the SUT have not been exercised. In this work, we introduce the novel concept of *cross-coverage testing* for the case that multiple implementations of a same function have to be tested. Intuitively, this term refers to testing one program using the tests obtained for covering the code of a different, but functionally equivalent, implementation than the one under exam. Said in different words, we investigate if and how the well-established notion of code-coverage testing could be usefully deployed to the case of redundant software.

In literature, software redundancy has been leveraged in different contexts and with differing aims. For instance, multi-version programs have been used for decades in safety-critical applications as a means to tolerate residual faults [12], [13]. In web service applications it can often be the case that multiple services exposing a same API are found and before one is selected they need to be tested to validate their functionality [14]. Similarly, in the many available open source libraries several different functions can be retrieved that implement a same utility, e.g., a mathematical function. In general, we consider here the case that two or more programs[1] exist that are functionally equivalent.

The overarching question we address is: given two functionally equivalent programs P and Q, can we leverage the information relative to code coverage of Q to improve the testing of P (or the vice versa)? In particular, with reference to the above schematic *how-to* process, if we derive additional test cases to increase, say, the coverage of Q, do such additional test cases, which we call *cross tests*, provide useful insights if used instead to test P? While the problem of testing functionally equivalent programs has been investigated from different perspectives, as we will discuss in the Related Work section, we are not aware of any previous work that has addressed the above questions.

---

[1]We will use the generic term of "program" throughout, whereby the notion of cross-coverage can be adapted for the testing of methods, classes or whole systems.

In this paper we define the concept of cross-coverage of functionally equivalent programs, and empirically study its application in test suite augmentation. We conducted an evaluation over a benchmark of 336 programs collected into 140 functionally equivalent groups [15]. To assess the potential thoroughness of cross-coverage test suite augmentation, we first tested all programs in a group using a common test suite derived by the well-known tool Randoop[2]. Then, on each program in turn, we measured the (own) coverage increase achieved by the additional cross tests derived from each equivalent program in the same group. We observed an average increase of $\sim 26\%$ in statement coverage and of $\sim 33\%$ in branch coverage. The gains have been also confirmed when we restricted the study to a subset of programs from 7 groups for which we automatically found additional random test cases to compare as a baseline against the cross tests: for them the cross-tests achieve a coverage increase that is on average $\sim 21\%$ higher for statement and $\sim 22\%$ higher for branch than the increase obtained by the random augmentation with an equal number of tests. We also conducted an empirical study for assessing the efficacy of cross coverage in finding missing functionalities: i.e., we deleted some code parts and used the cross tests to see whether the tests were able to raise a failure. Over 169 programs where the injected fault remains undetected, our results show that cross-coverage test suite augmentation founds $\sim 73\%$ of them. Thus, we could automatically conceive test cases covering missing functionality faults for a program P from the throughout coverage of a functionally equivalent program Q.

The paper is structured as follows. In the next section we present two motivating examples. Then, in Section III we propose a formal definition of cross-coverage testing and related concepts. In Section IV we introduce the empirical evaluation study, whose results are then described in Section V, while in Section VI we discuss some aspects threatening the validity of this study. We contrast our work against existing literature in Section VII, and finally draw conclusions in Section VIII.

II. MOTIVATION

In simplified words, what we investigate here is whether the test cases that are generated based on the code-coverage of a given program can be leveraged to also test one or more other programs that are (supposed to be) functionally equivalent to the former. Before formally defining in the next section the notion of using coverage measures across functionally equivalent programs, we discuss here how and why this notion could be useful using a couple of examples.

The very first motivating example (illustrated in Figure 1) that we can think of belongs to the realm of N-version programming, in which more versions of a same program are executed in parallel for ensuring fault tolerance [16]. As well known, to mitigate the occurrence of common mode failures, it is important to ensure that the redundant versions run in parallel are independently designed and developed, e.g.,

---

2https://randoop.github.io/randoop/

by different teams. Let us assume in particular that we are developing a fault-tolerant system that can tolerate up to one fault, for which case we need to use triple-redundancy [17]. Hence we assume we have three programs P_1, P_2 and P_3 that have been developed by three independent teams from a same functional specification Sp (see Figure 1) Before deploying them to the redundant architecture, we aim at ensuring that the three programs properly implement Sp. Therefore we derive a test suite T_0 using some specification-based approach.

Let us assume that all three programs successfully pass the T_0 test suite. However in monitoring their coverage we observe that they achieve quite differing measures. We wonder why three programs that should be in principle equivalent, when exposed to the same inputs achieve so different coverage measures. Hence, we analyze the program that has achieved the lowest measure, say, e.g., P_1, and find some additional test cases T_1 that can raise its coverage. We execute these same tests on P_2 and P_3, and analyze the results: if either of them fails on the additional test cases, it probably does not fully implement the specified functionality, e.g., it might lack a proper check of the input variables. We can repeat this test suite augmentation process, finding further test cases that cover more unexecuted entities of the program that yields the lowest coverage so far, until we have all three programs adequately covered and no further failure is found.

In this example, to ensure that all three versions are adequately tested, we suggest that the coverage of one version is leveraged to augment the test suite of another version, and we expect that such approach could help finding missing functionalities.

A second motivating example (illustrated in Figure 2) concerns the reuse of existing code pieces in software development. Indeed, nowadays many different programs can be found in open-source code repositories such as GitHub, in web service directories, in app stores, as well as among proprietary solutions, for implementing one same desired function. As a matter of fact, several studies [18], [19], [20] confirm that developers increasingly perform code search to retrieve and reuse code snippets, methods or entire projects. Among the many proposed approaches to make code search more effective and accurate [21], Lemos et al. have presented a Test-Driven Code Search (TDCS) approach [22]. In brief, inspired by test-driven development, they propose to first write a set of test cases that define the desired functionalities, and then to use such test cases both for searching the code repository for suitable matching implementations and for testing the found code pieces. However the authors conclude the cited work by observing that in order "*to enhance confidence in the retrieved code, we could design test cases in a more systematic way*" [22]. Thus, they suggest that other testing criteria should be considered to integrate the initial design of test cases.

Our idea of cross-coverage can support this recommendation. Let us assume that thanks to TDCS we have found and retrieved three candidate programs P_1, P_2 and P_3 that all pass our initial test set T_0 (see Figure 2). So far we have no further hint about which program to choose among P_1, P_2 and P_3.
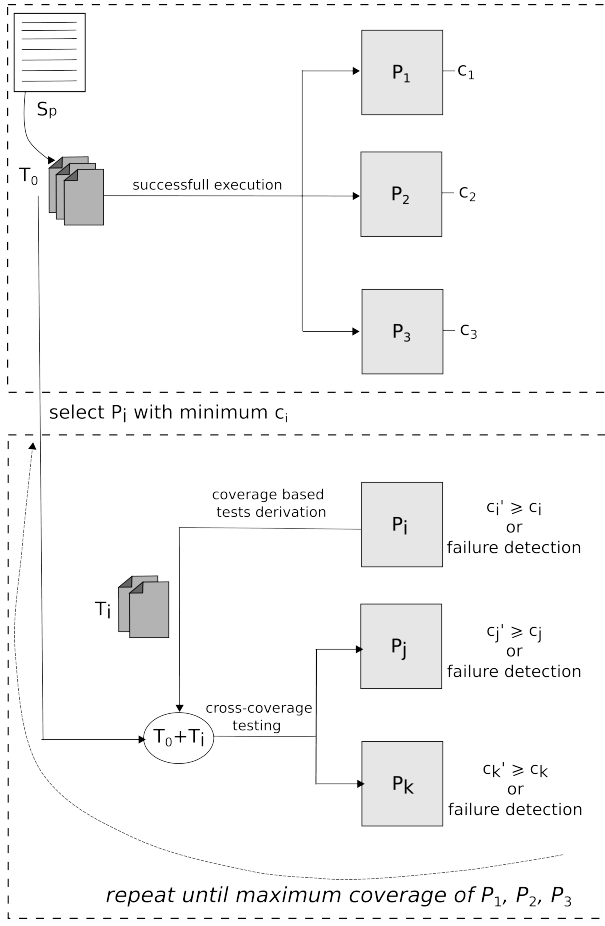
Fig. 1: *Cross-coverage testing* for *N-version* programming.

Fig. 2: *Cross-coverage testing* for TDCS.

We analyze the respective coverage achieved by the same test set T_0 over each of the three programs, which we denote as c_1, c_2, and c_3, respectively. For each program, we then define an additional set of test cases, T_1, T_2 and T_3 so to maximize its coverage (applying some traditional coverage-based test generation technique). Now we obtain an augmented test suite T_{aug} by combining the initial test set T_0 with all the additional test cases, and execute this combined test suite on all three programs. Doing so we *(i)* may possibly detect some failures and *(ii)* also obtain the increased coverage measures c'_1, c'_2, and c'_3, respectively. We can now use T_{aug} test results for choosing one program among P_1, P_2 and P_3; in particular we might prefer the one that yields the highest coverage, because this method implements our desired functionalities with less unused code.

In both the above examples, the usage of cross-coverage information helps to augment the test suite in systematic and automatable way. In the remainder of the paper, we formalize the notion of cross-coverage testing and provide a concrete evaluation of its usefulness.

III. CROSS-COVERAGE TESTING

In this section we present our cross-coverage testing approach and formally define the concepts of *cross-coverage*
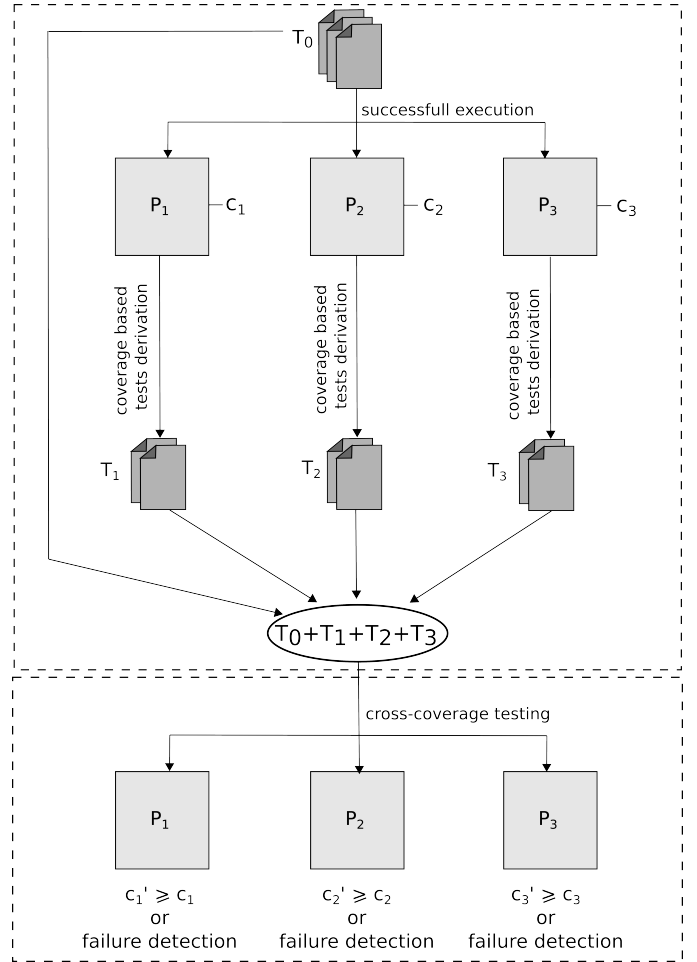
testing and *cross-coverage test suite augmentation* that are at the basis of our proposal.

First, for functional equivalence between programs, we rely on the same definition provided in [23], i.e., we say that two programs are *functionally equivalent* if given a same input, they provide the same output, without considering the intermediate program states. Please refer to [23] for further details. We can now introduce the concept of cross-coverage testing.

Definition 1 (Cross-coverage testing): Let $P_1, P_2, \ldots, P_n$ be *n* functionally equivalent programs. Given a test case t_i derived to cover program P_i, we define *cross-coverage testing* as the approach of executing t_i on P_j with $i, j = 1, \ldots, n$ and $i \neq j$. The test case t_i is said a *cross-test* for P_j, and the cross-coverage test execution is denoted as $t_i \checkmark P_j$.

With the above definition we have introduced the concept of using coverage-driven test cases for testing a different program than the one from which they were derived. In other terms, by cross-coverage we mean in some sense that the test coverage for a program P_j could be monitored over another program P_i.

In the following, with reference to the schematic *how-to*

process outlined in the introduction, we focus on the usage of coverage for improving an initial test suite T_0, for example derived from the common specification (as in Figure 1).

Definition 2 (Cross-coverage test suite augmentation): Let $P_1, \ldots, P_n$ be n functionally equivalent programs. Let T_0 be a common initial test suite for $P_1, \ldots, P_n$. Chosen a program P_i, let T_i be a test suite of cross tests for P_j, with $i, j = 1, \ldots, n$ and $i \neq j$. *Cross-coverage test suite augmentation* for P_j consists of adding the tests in T_i to T_0, obtaining the *augmented cross-test suite* $(T_0 \bigcup T_i)$.

Referring to the above definition, we can have that equivalent programs $P_1, \ldots, P_n$ represent different implementations of the same functional specification. Then, we can suppose to have a common test suite T_0 for this set of equivalent programs, derived for instance from an abstract functional specification of these programs. Then, we can suppose to already have a specific test suite T_i for the program P_i derived to cover the specific implementation P_i. The main idea of the proposed approach is to leverage the coverage of the program P_i, and then the *cross-tests*, namely the coverage-based test cases derived for P_i, for augmenting the existing test suite T_0 with additional test cases aiming to improve the testing of P_j. The intuition behind our approach is that test cases generated to cover a program, for instance P_i, could have good chances to improve the testing of another program P_j that is equivalent to P_i and that implements the same functionalities of P_i.

In the above definition we purposely leave the characterization of a cross test quite generic. Indeed, differing approaches could be taken for deriving the cross tests, for instance one could use tools such as EvoSuite [3] or Klee [4] to derive test cases that cover the not yet covered entities.

Note that this definition matches the first motivating example in the previous section: looking to Figure 1, the common initial test suite T_0 of the definition corresponds to the one derived from Sp, and we augment it in iterative way, choosing at each iteration to apply the cross-coverage test suite augmentation approach by covering the program that has reached so far the least coverage.

While in the above example we consider that coverage of one program is used to drive the testing of all the other equivalent programs, we can also consider the complementary case that all equivalent programs cumulatively contribute to augment the test suite of a selected program, as in the following definition.

Definition 3 (Cumulative cross-coverage test suite augmentation): Let $P_1, \ldots, P_n$ be n functionally equivalent programs. Let T_0 be a common initial test suite for $P_1, \ldots, P_n$. Chosen a program P_j, let T_i be a test suite of cross tests for P_j, with $i, j = 1, \ldots, n$ and $i \neq j$. *Cumulative cross-coverage test suite augmentation* for P_j consists of adding $\forall i \neq j$ the tests in T_i to T_0, obtaining the *augmented cumulative cross-test suite* $(T_0 \bigcup \{T_i\})$.

---

[3] https://www.evosuite.org/
[4] http://klee.github.io/

The idea of the *Cumulative cross-coverage test suite augmentation* of Definition 3 is to consider all the *augmented cross-test suites* derived from all programs P_i equivalent to P_j and then leverage these test suites of all programs P_i equivalent to P_j for augmenting the initial test suite T_0.

Cumulative augmentation recalls the case illustrated in the second motivating example (see Figure 2). However, note that in our example we have considered that one common test suite T_{aug} is obtained as $(T_0 \bigcup T_1 \bigcup T_2 \bigcup T_3)$, i.e., it embeds all the additional test cases, whereas the above definition of the augmented cumulative cross-test suite would correspond to tree test suites, e.g., $(T_0 \bigcup T_2 \bigcup T_3)$ when testing P_1, $(T_0 \bigcup T_1 \bigcup T_3)$ when testing P_2, and $(T_0 \bigcup T_1 \bigcup T_2)$ when testing P_3. In other terms, for this example we consider to test each program not only considering its cross tests but also the additional test cases that enhance its own coverage, which seems a sensible way to go when we can also monitor the coverage of the program considered.

The presented cross-coverage testing approach is not tailored to a specific test adequacy criterion [1]. As we will show in Section IV, for the aim of simplicity, in this paper we validated the proposed approach considering two coverage criteria that are: *statement coverage* and *branch coverage*, i.e, the fraction of the total number of statements and the fraction of the total number of branches that have been exercised by a given test suite, respectively. However, the proposed approach is generic and can address other common coverage measures including *path coverage*, *C-use*, *P-use* coverage and others [24].

IV. STUDY DESIGN

This section describes our study introducing the research questions (RQs) that guided our validation (see Section IV-A), the subjects used for the empirical evaluation (see Section IV-B), and the methodology followed to answer the RQs (see Section IV-C).

A. Research Questions

Overall our study aims at evaluating whether *cross-tests* contribute to improve the quality of a test suite when groups of functionally equivalent programs have to be tested. For addressing this general goal, we formulate two more specific research questions as follows.

RQ1: Does cross-coverage test suite augmentation contribute to improve the thoroughness of a test suite?
By RQ1 we aim at assessing whether, given a program P, the additional cross tests that are derived based on the coverage of functionally equivalent alternatives, provide effective test cases also when executed on P.

In the scope of this study, as a proxy indicator for the effectiveness of a cross test we use the coverage it can achieve on the program under test, i.e., we observe if the cross tests actually help to explore parts of P that have not yet been considered. Besides, our experiment will focus on cumulative cross-coverage test suite augmentation.

However, if the goal were just that of increasing the coverage of P, we could as well, or even better, use one of the many existing code-based test generators directly on P, without having to resort to the coverage of other programs. In this sense, the usefulness of cross-coverage test suite augmentation as evaluated by RQ1 would be better appreciated in situations in which the code of P is not available. More in general, we foresee that cross-coverage can be useful also, or above all, to detect missing functionality faults among a group of supposedly equivalent programs. Thus, we formulate the second research question as follows.

RQ2: Does cross-coverage test suite augmentation contribute to expose missing functionalities?
Overall, RQ2 aims to study if cross-coverage testing is a valid approach for detecting undesired behaviour in a program P due to missed functionalities in its implementation. Our intuition here is that the cross tests that are derived from the extensive coverage of a functionally equivalent program could also include test cases exercising functionalities that have been "forgotten" by the developers of P, and could certainly not be detectable using traditional coverage testing.

B. Study subjects

We conducted our evaluation over 140 groups G_k, each one consisting of functionally equivalent programs (precisely Java methods), which have been selected from the large dataset built by Higo et al. [15][5].

In turn, the authors constructed this dataset from Borges et al.'s collection of GitHub projects [25]: they first collected in groups those methods having same return and parameter types, and derived for each method a set of coverage-based test cases using EvoSuite [26]. Then they mutually tested each method within a group using test cases automatically generated for all the other methods in the same group. Finally, all pairs of methods that passed each other test cases were subjected to visual inspection. As a result their dataset includes a total of 276 groups of functionally equivalent methods (728 methods in total). The dataset provides an adequate benchmark for our study, because we get both groups of functionally equivalent programs and their test suites that are derived based on coverage (using EvoSuite on each method).

As each group refers to two or more equivalent programs exposing the same abstract functionality, we could also derive an additional test suite common to all the programs in each group. Specifically, we generated such a common black-box test suite using the feedback-directed random test generation approach by Randoop [27]. In the generation process we followed the typical configuration procedure as suggested by the Randoop user manual.

From the 276 available groups, in the scope of this experimentation we excluded those groups for which:
- we were not able to generate the common test suite; or,
- for all the methods within the group, their specific test suite already included in Higo et al.'s benchmark could not be improved (i.e., in terms of coverage).

[5]https://zenodo.org/record/5912689

Eventually, our benchmark consists of 140 groups including a total of 336 programs, their relative 336 test classes obtained from the benchmark in [15] that are derived on the specific implementation of each program, plus 140 test classes that are common for each equivalent program in a group. Notably, while many groups only include two methods, there are also groups collecting 10 or more methods. An archive with all the data we referred, the whole set of results, and the scripts for replicating our experiments is available at: http://saks.iasi.cnr.it/tools/cct/CCT-ReplicationPackage-AST2023.zip.

C. Procedure and evaluation metrics

To answer RQ1, we planned a two steps procedure, both steps referring to: a group of functionally equivalent programs $P_1, \ldots, P_n$, their respective coverage-driven test suites $T_1, \ldots, T_n$ generated using EvoSuite, and the common and implementation-independent test suite T_0 derived using Randoop.

In the first step, for each group we execute the common tests in T_0 on all programs of the group and measure for each program the coverage achieved (T_0 coverage). For coverage analysis, we use the tool JaCoCo, from which we referred the provided measures of statement and branch coverage. Then, for each program P_i in turn, we launch in exhaustive way all the Tj (with $i \neq j$): if for a fixed i and for any j, the cross test execution $t_c \checkmark P_i$ of any $t_c \in Tj$ improves over the current coverage of P_i (indifferently among either statements or branches), we iteratively add T_j to the test suite for P_i. Eventually, at the end of this first step, for each program P_i we obtain its augmented cumulative cross-test suite $CCT_i = (T_0 \bigcup \{T_j\})$ and the final coverage achieved by this augmented test suite (CCT coverage). To answer RQ1, we analyse the size of CCT_i and the gains in coverage that it scores against the initial T_0 coverage.

Nevertheless, as by construction the size of CCT_i is greater than or equal to the size of T_0, it can appear as obvious that by adding more tests the coverage of P_i increases. For a meaningful evaluation, it is important to assess the coverage gains against those that would be obtained by extending T_0 without taking into account any information on the programs equivalent to P_i. Thus, in the second step of the evaluation, for each P_i, we extended T_0 to the same size of CCT_i (denoted by T_0^{+i}); then we run T_0^{+i} against P_i and measure the coverage metrics it scores. The comparison of the coverage gains achieved by the baseline T_0^{+i} against those from CCT_i complements the answer to RQ1.

To answer RQ2, we consider all the P_i recording some coverage improvement from RQ1. Among these, chosen a program P_i, we create a faulty version MP_i where one of P_i's original functionality has been dropped. To do this, we blindly and systematically modify P_i by removing where possible either a branch of the first decision statement or one atom of its conditional expression. We verify that such modifications do not affect the compilation process. After that, for each MP_i we run its relative T_0 (passing against P_i) and we remove all those MP_i programs for which the modification is detected.

Programs	Cases Without Augmentation	Cases With Augmentation
336	44 (13,10%)	292 (86,90%)

TABLE I: Cumulative cross-coverage test suite augmentation

For the remaining MP_i, we separately run each T_j against MP_i counting each time the injected failure is spotted. The comparison between the times MP_i have been detected as faulty by T_j (but not T_0) against the whole set of cross-tests analysed gives an estimation of the efficacy in revealing missing functionalities of the cross-coverage testing across equivalent programs.

V. Study Results

In the following we report the outcome of the empirical evaluation and we provide answers to the RQs given in Section IV-A.

A. Capability to Improve Test Suite Thoroughness

In the first step of the empirical evaluation we employed 336 programs from [15], as explained in Section IV-B. On these we first applied the general procedure planned to answer RQ1 (Section IV-C).

On each program P_i^k within a group G_k, we executed: the implementation-independent test suite we generated with Randoop and that it is common to all the programs in a group (i.e., T_0^k); plus all the implementation-dependent test suites $\{T_j^k\}$, $j \neq i$, available from [15] and generated using EvoSuite from the code of the other programs in the same equivalence group G_k (i.e., when considering P_i^k we did not execute the test suite T_i^k EvoSuite generated from the code of P_i^k).

We provide below for each P_i^k the measures achieved for coverage after applying cumulative cross-coverage test suite augmentation starting from the initial suite T_0^k, and obtaining the augmented cumulative cross-test suite CCT_i^k. For simplicity in this study we add to the initial test suite only those T_j^k that actually bring some coverage increase.

Table I reports the overall results of this first step of our experiment: across all the considered programs, in most of the cases (i.e. 86,90%) we recorded that P_i improved some of its coverage metrics when tested with CCT_i, while only for a minor part (i.e. 13,10%) CCT_i did not perform better than the initial implementation-independent test suite generated with Randoop (i.e, T_0).

Figure 3 further details such results by plotting the average gains we measured in terms of both statements (Figure 3a) and branches (Figure 3b) coverage. The horizontal axis reports the group number (as referred in the original benchmark [15]), while the vertical axis reports the average percentage gain within a group; the latter is measured for each program P_i^k within a group G_k as the difference between the final percentage of own coverage achieved after executing CCT_i^k and the initial percentage of own coverage achieved by T_0^k. Overall across the 336 considered programs we register an average increase of $\sim 26\%$ in statement coverage and of $\sim 33\%$ in branch coverage.

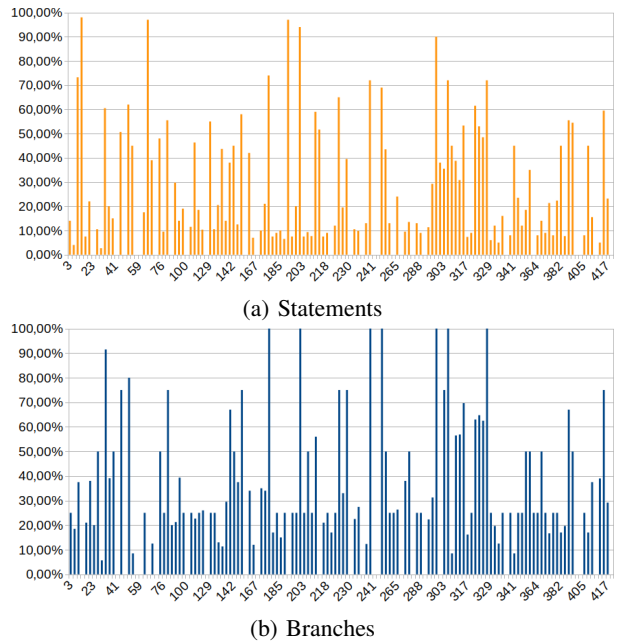(a) Statements

(b) Branches

Fig. 3: Average gain in coverage per groups: CCT vs. T_0

Specifically, concerning statement coverage (i.e., Figure 3a), for 22 groups we recorded an average gain between $[25 - 50\%)$, for 21 groups the average gain in statements coverage is between $[50 - 75\%)$, for 5 groups such average gain scores between $[75 - 100\%]$, while the other 66 have an average gain lower than 25%. For the remaining 26 groups there is no benefit from the cumulative cross-coverage test augmentation process.

Similar results have been recorded for branch coverage (i.e., Figure 3b): for 50 groups the average gain is between $[25 - 50\%)$, 20 groups score between $[50 - 75\%)$, 16 groups have an average gain in the range $[75 - 100\%]$, while 28 groups report an increase lower than 25%. The remaining 26 groups did not register any positive gain from the augmentation. Notably, among them 21 groups overlaps with those scoring 0.00% gain in statements coverage, while in 10 cases (i.e., $5 + 5$) we find a positive increase only in one of the two coverage metrics. Those groups that present a gain only on statements coverage refer to programs with no or few (e.g., only one) decisional statements where: some exception is explicitly thrown, the tests in T_0^k force some unchecked exception, or a `return` statement is met before the last instruction of the program. The groups presenting only an increase in branches coverage again refer to simple programs with few decisional statements but with a complex boolean formula as conditional expression.

In Table II we provide an excerpt of more detailed results for a sample of seven groups (we anticipate these correspond to the groups that were identified in the second step of this experiment, as explained below), whereas the detailed results for all the 336 programs can be found in the additional online

material[6]. The first and second column of the table report the Id of the groups and of the methods. Then we include two sets of three columns each, which report the specific coverage metrics (statements for columns 3-5, and branches for columns 6-8) scored for each program by CCT_i^k and T_0^k, and their difference (i.e., Gain). Also, column 9 shows the size of the test suite resulting from the cross-coverage augmentation procedure (i.e., T_0^k plus some T_j^k with $j \neq i$): we see that in most cases CCT_i^k adds to T_0^k only one implementation-dependent test suites T_j^k from some program equivalent to P_i^k. Overall the average CCT size for those test programs that increase some coverage metric is $2, 05$ test suites. In our intuition this result was somehow expected for the considered benchmark: the tests in [15] were generated by means of EvoSuite which (within the given budget) looks for specific configurations maximising the coverage of the considered program. Indeed, the test in T_j^k are selected so that to force the activation of several branches and conditions in P_j^k with the objective of highlighting multiple usage scenarios for the referred functionality. In addition, the programs in the benchmark are relatively small, and the scenarios explored by T_j^k could overlap with those covered by other test suites in G_k. Thus, limited to this benchmark, we found reasonable that on average a bit more than one additional test suite (i.e., $CCT_i^k = T_0^k \bigcup \{T_j^k\}$ for $j \neq i$) is the best solution for P_i^k.

While these results are encouraging, we also assessed what coverage gains could be scored by a test suite having the same size of CCT_i^k but generated without taking into account any cross-coverage information. We hence took as a baseline an augmented test suite T_0^{k+i} obtained from T_0^k by adding the proper set of additional random test cases (which depends on the resulting size of CCT_i^k), generated by using again Randoop. For this purpose we selected all the groups that from the previous step got a resulting average score in statements coverage lower than 85%. The rationale was to give higher chances to the random baseline to beat our approach (because it is well known that the higher is the current coverage the more difficult is to find test cases that can increase it). Such first selection returned with 67 programs. From them we excluded all those programs (i.e., 28) with no actual augmentation (i.e., $CCT_i^k = T_0^k$), and then we also removed all those programs (i.e., 17) where the typical configuration procedure suggested by the Randoop user manual produces a test suite lower in size than CCT_i^k. The resulting set of equivalent test programs (i.e., 22) is reported in Table III[7]. In the third and fourth columns we report for each program the difference between the increase of statement and branch coverage achieved by CCT and T_0^+. Then, columns 5 and 6 provide the average values of such differences.

As shown, the results from the comparison between the cumulative cross-coverage test suite augmentation process and the baseline clearly confirm that cross-coverage positively contributes to improve a test suite thoroughness, better than random generation. From the table, we see only one case where random augmentation scores higher coverage than cross-coverage (program 67630 of Group 37). In all the other cases CCT reaches always higher or much higher coverage than the baseline. Specifically, for these program cross-tests achieve an average coverage increase that is $\sim 21\%$ for statement and $\sim 22\%$ for branch more than the increase obtained by the baseline.

Thus we can answer RQ1 by reporting one of the outcome from our study: implementation-dependent tests from equivalent programs contribute to improve the test suites thoroughness. Moreover, the gain from such cross-coverage process is more effective than a random augmentation of the numbers of the tests.

B. Capability to Detect Missing Functionalities

About RQ2, we referred all the 292 programs P_i resulting from the first validation step we performed for RQ1 (see Table I). As introduced in Section IV-C, for each of these P_i we considered: MP_i, which is a faulty version of P_i where one of its functionality has been removed; the initial (implementation-independent) test suite T_0; and the set of T_j (with $i \neq j$), i.e., the implementation-dependent test suites of the programs equivalent to P_i. After removing all MP_i whose modification is detected by the initial test suite T_0, we got 187 modified programs. We compared these remaining MP_i against P_i to double-check the applied modification is actually subsuming a missed functionality: we further excluded 18 MP_i. Overall we are left with 169 valid modified versions of the original programs. For each of them, within their respective group G_k we separately considered all the T_j^k in the CCT_i^k resulting from RQ1 and added to T_0^k. As some equivalent test programs have multiple items in CCT_i^k, the final set of valid modified programs undetected by T_0^k to be considered for the cross-check analysis counts 183 items.

Table IV summarises the results of this second study: for 132 items the missing functionality is revealed by at least one of the implementation-dependent test suites of equivalent programs, while for the remaining 51 items, no specific implementation-dependent test suite of the programs equivalent to P_i is able to detect the missed functionality. More in detail about these last cases, the results report that for cases 2 the missing functionality is actually revealed when the whole CCT_i^k is run, thus the actual number of programs for which the modification is detected counts 124, while the modifications are never detected for 45 programs.

In conclusion, also our answer to RQ2 is positive: cross-coverage testing from equivalent programs is a valuable strategy for exposing issues related to missing functionalities. The outcomes from our study show that, over 274 (i.e., $292 - 18$) programs, the implementation-dependent tests from equivalent programs help to detect missing functionalities for $73, 37\%$ out of the 169 cases in which the initial implementation-independent test suites do not spot them.

---

[6]http://saks.iasi.cnr.it/tools/cct/CCT-ReplicationPackage-AST2023.zip

[7]As anticipated, the excerpt reported in Table II refers to the same set of equivalent programs.

GroupId	MethodId	STMS			BRANCHES			CCT Size	Group AVG		
		CCT	T_0	Gain	CCT	T_0	Gain		Gain STMS	Gain BRANCHES	CCT Size
37	10881	91,00%	65,00%	26,00%	87,00%	50,00%	37,00%	2			
	54076	17,00%	15,00%	2,00%	16,00%	12,00%	4,00%	2			
	61760	100,00%	78,00%	22,00%	100,00%	50,00%	50,00%	2			
	67630	29,00%	28,00%	1,00%	32,00%	26,00%	6,00%	3			
	101187	76,00%	61,00%	15,00%	83,00%	50,00%	33,00%	2			
	181606	100,00%	78,00%	22,00%	100,00%	50,00%	50,00%	2	20,00%	39,08%	2,25
	181644	100,00%	88,00%	12,00%	100,00%	66,00%	34,00%	2			
	184888	100,00%	60,00%	40,00%	100,00%	25,00%	75,00%	2			
	191816	100,00%	55,00%	45,00%	100,00%	37,00%	63,00%	3			
	277587	100,00%	68,00%	32,00%	100,00%	50,00%	50,00%	3			
	296201	88,00%	77,00%	11,00%	83,00%	50,00%	33,00%	2			
	296211	100,00%	88,00%	12,00%	100,00%	66,00%	34,00%	2			
40	267714	76,00%	61,00%	15,00%	100,00%	50,00%	50,00%	2	15,00%	50,00%	2,00
	292418	76,00%	61,00%	15,00%	100,00%	50,00%	50,00%	2			
80	242345	69,00%	23,00%	46,00%	62,00%	12,00%	50,00%	2	55,50%	75,00%	2,00
	242357	100,00%	35,00%	65,00%	100,00%	0,00%	100,00%	2			
106	225177	65,00%	65,00%	0,00%	50,00%	50,00%	0,00%	1	11,50%	25,00%	1,50
	250021	100,00%	77,00%	23,00%	100,00%	50,00%	50,00%	2			
225	137547	81,00%	62,00%	19,00%	66,00%	33,00%	33,00%	2	19,50%	33,00%	2,00
	154527	80,00%	60,00%	20,00%	66,00%	33,00%	33,00%	2			
343	20346	100,00%	53,00%	47,00%	100,00%	50,00%	50,00%	2	23,50%	0,50%	1,50
	251573	63,00%	63,00%	0,00%	50,00%	50,00%	0,00%	1			
369	77210	46,00%	46,00%	0,00%	33,00%	33,00%	0,00%	1	21,33%	16,67%	2,00
	102898	100,00%	70,00%	30,00%	100,00%	50,00%	50,00%	3			
	114340	100,00%	66,00%	34,00%	100,00%	100,00%	0,00%	2			

TABLE II: Thoroughness: Excerpt of few gains from the cross-coverage test suite augmentation

GroupId	MethodId	Gain: $(CCT - T_0^+)$		Group AVG	
		STMS	BRNS	STMS	BRNS
37	10881	26,00%	37,00%		
	54076	2,00%	4,00%		
	61760	11,00%	13,00%		
	67630	-10,00%	-10,00%		
	101187	7,00%	17,00%		
	181606	11,00%	13,00%	11,58%	19,83%
	181644	12,00%	17,00%		
	184888	14,00%	25,00%		
	191816	32,00%	38,00%		
	277587	11,00%	17,00%		
	296201	11,00%	33,00%		
	296211	12,00%	34,00%		
40	267714	15,00%	50,00%	15,00%	50,00%
	292418	15,00%	50,00%		
80	242345	46,00%	50,00%	55,50%	75,00%
	242357	65,00%	100,00%		
106	250021	23,00%	50,00%	23,00%	50,00%
225	137547	19,00%	33,00%	19,50%	33,00%
	154527	20,00%	33,00%		
343	20346	47,00%	50,00%	47,00%	50,00%
369	102898	30,00%	50,00%	32,00%	25,00%
	114340	34,00%	0,00%		

TABLE III: Comparison with the baseline: CCT vs. T_0^+.

Modified Programs Undetected by T_0	187
Invalid Modifications	18
Valid Modified Programs	169
Valid Cases For Cross-Check	183
Valid Cases Detected by CCT	132 (72,13%)
Valid Cases Undetected by CCT	51 (27,87%)
Valid Modified Programs Detected by CCT	124 (73,37%)
Valid Modified Programs Undetected by CCT	45 (26,63%)

TABLE IV: Results on detection of missing functionalities

VI. THREATS TO VALIDITY

Even though the outcomes from our study lead to positive answers for both the RQs, we highlight some threats that may potentially affect the validity of our empirical evaluation and the conclusions we have drawn.

A. Threats to External Validity

This class of threats to validity address those concerns limiting the generalisation of the observed results to other studies or contexts.

Considered Subject: A first class of concerns come from the referred benchmark by Higo et al. [15]. Indeed, the programs in [15] are actually methods extracted from wider Java classes, thus: their complexity is relatively limited, and they implement state-less functions only. Our work does not cover yet the study of complex software where the implementations of system-level features require the interaction across several (possibly state-full) software bundles. In this sense, even thought we are confident that tests from specific usage scenario of a whole system could be also valuable for checking missing functionalities in equivalent software, our conclusions do not have a general validity for any kind of software system.

Functional Equivalence: We are aware that the definition of functional equivalence referred from [23] and also used in [15] may not be accepted in all the contexts. In this sense our conclusions are strictly related to such definition, and they cannot be referred elsewhere if such definition does not hold.

B. Threats to Internal Validity

This class of threats to validity refers to those aspects that could have influenced the observed outcome.

Generation of T_0: As reported in Section IV-B, for the generation of the common implementation-independent test suites we followed the typical configuration procedure reported by the Randoop user manual. We are aware that we used only the more basic functionalities by Randoop. Thus, we do not exclude that a deeper tuning of its parameters may have led to the generation of more powerful T_0 for each of the considered programs. Probably in this case our experiment would have been resulted with different outcomes. However, our decision is motivated by both our limited confidence in tuning Randoop, and also the limited complexity of the programs in the subject.

Construction of CCT: About the cumulative construction of the cross-coverage test suite, for each program P_i we randomly selected one of its equivalent program P_j (with $i \neq j$) and we launched its T_j against P_i: if the cross testing improves the current coverage metrics, then T_j is added to CCT_i. Indeed, the specific order the equivalent programs P_j are selected can impact the construction of CCT_i: the resulting CCT_i cannot be considered the only admissible solution for P_i. Thus, some of the indexes presented in Table II (e.g., CTT Size), in Table IV (e.g., Valid Cases For Cross-Check), or discussed in Section V may result with small but different values across next replications of the experiment.

Programs modification: In the experiment that answers to RQ2, a potential threat to validity could also come from the specific procedure followed for creating the modified version of the each P_i. Indeed, in order to get the faulty version MP_i we first agreed on a common strategy without taking into account the actual implementation of P_i then we blindly applied it on the code base. We did our best to avoid biases in both the selection of the strategy, and its application; nevertheless we cannot exclude that our procedure did not affect any outcome or our conclusions neither.

VII. RELATED WORK

The related work of our proposal spans over two main research areas that are: testing of equivalent programs and test suite augmentation.

a) Testing of equivalent programs: Code fragments having functionally equivalent behavior, known also as *software redundancy*, are largely used in software development, for fault tolerance purposes (as in the case of *N-version* programming [16]) but also in self-healing systems [28] or self-adaptive services [29]. The problem of finding or checking semantically equivalent code has been largely investigated in the literature since decades [30]. Some of the existing studies focus on software testing for checking functional equivalence of two programs. For instance, the work in [31] aims to check functional equivalence of refactored code segments by generating test programs to detect variable values inequivalence whereas that in [23] applies random testing on arbitrary pieces of code and detects their equivalence if they show the same output behavior with the same input. Other studies address the problem of measuring the redundant software in order to predict the effectiveness of different redundancy techniques [32]. The goal of this paper is totally different from such previous studies. Our goal is to provide a new testing approach that we call *cross-coverage testing* to improve the testing of two or more programs that are (assumed to be) functionally equivalent.

Few studies address the problem of effective testing of equivalent or redundant programs. In fault tolerant systems, *back-to-back testing* is used to check the outputs obtained from functionally equivalent versions of the same program. If the outputs are equal, the versions are correct, if they are different, further investigations are made to locate faults. In *back-to-back testing*, test cases are usually randomly generated and include boundary or special values. Besides *back-to-back testing* is unable to detect identical incorrect outputs [33], it has been proven to be effective in detecting correlated faults [34]. The authors of [35] provide the ATAC coverage analysis tool and show its application to detect a correlation between the number of faults and the coverage of 12 program versions of a critical application developed into a *N-version* software project. The tool also allows to generate new tests aiming to improve coverage by examining not covered code. In our approach the aim is not to improve the test suite according to the uncovered code of a given program under test but to leverage the mutual coverage analysis of existing test suites derived from equivalent versions of a program for improving the testing of the overall redundant system.

b) Test suite augmentation: Existing test suites augmentation proposals aim to identify the code elements affected by changes, typically during regression testing, and then generate test cases to cover those elements [36]. These solutions rely on genetic algorithms [37], symbolic execution [38] or the combination of different test cases generation algorithms [39] for augmenting the test suite. Other approaches proposed in literature for test suite augmentation leverage search based techniques for test data regeneration starting from pre-existing test cases [40] or aim to derive integration tests from existing unit tests [41]. The idea of improving a test suite by adding variants of existing test cases (for instance manually written by developers) is also known as test amplification [42]. The concept of test amplification is a general concept that includes test augmentation. Along with mutation analysis [43], coverage analysis represents one of the main engineering goals guiding test amplification [44] and test augmentation [45]. Coverage analysis has been used for decades for different purposes, i.e., for providing an indication of the thoroughness of the executed test cases [7], for test case prioritization [46], for prediction of software reliability [47] or for selection of effective test cases [48]. In test augmentation, given an initial test suite, additional test cases are added in order to maximize the coverage of the program code. However, to the best of our knowledge, no previous work leverages the coverage metrics of equivalent program versions for augmenting a test suite and improving testing of the program under test. Differently from all the existing test suite augmentation techniques, in our approach the generation of new test cases is guided not by the coverage analysis of a single program under test but by the coverage analysis of a set of functionally equivalent programs.

Finally, few approaches address test augmentation considering multiple versions of a program as in our proposal. For instance, Osei-Owusu et al. [49] perform test suite augmentation for assessing the correctness of multiple implementations of the same specification. Additional test cases are derived such that they are able to detect incorrect programs according to a defined concept of behavioral-equivalence approximation. Gaston and Clause [50] propose a statical analysis technique that allows to identify missing unit tests by analysing sibling groups, i.e. groups of methods in different applications that accomplish the same goal in different ways, and their respective test suites defined in Junit. However, these works do not focus on coverage analysis of equivalent programs to identify additional tests.

VIII. Conclusion and Future Work

Although largely investigated in the literature, research related to coverage analysis still attracts interest within the software engineering community. In this paper, we defined the new concept of *cross-coverage testing* of equivalent programs. The main idea is to leverage the coverage driven test cases derived for a program to improve the testing of another program that is equivalent to the first one. Building on the foundational concept of *cross-coverage testing* we defined *cross coverage test suite augmentation* and *cumulative cross coverage test suite augmentation* concepts, discussing their application in two representative contexts dealing with N-version programming and test-driven code search for software reuse, respectively. Experimental results on a benchmark of 336 real programs collected into 140 functionally equivalent groups have demonstrated the effectiveness of *cumulative cross coverage test suite augmentation*, as cross tests reached an average increase of $\sim 26\%$ in statement coverage and $\sim 33\%$ in branch coverage.

Moreover, for a subset of 22 programs, selected among those for which random tests have higher chances to increase coverage (i.e. those showing average statement coverage lower than 85%), cross tests showed an average improvement of $\sim 21\%$ and $\sim 22\%$ of statement and branch coverage respectively, with respect to randomly generated test suites with the same cardinality, chosen as a baseline.

In our experiment we also showed the ability of the augmented test suite to discover $\sim 73\%$ of missing faults introduced by systematically modifying test programs. Therefore, the proposed cross coverage testing brings gains both in thoroughness of a test suite and in exposing missing functionalities.

Future work is foreseen in several directions. A set of more extensive experiments based on larger benchmark programs is a first planned extension. We also want to compare the effectiveness of our proposals of *cross coverage test suite augmentation* and *cumulative cross coverage test suite augmentation* with other existing test augmentation approaches, based for instance on mutation analysis. In addition, cross coverage testing application in the two representative contexts considered in the paper would deserve deeper investigation,

considering for instance a real redundancy based fault tolerant system.

From the perspective of demonstrating the potential generality of cross coverage testing approach and its usefulness in a variety of applications, we have already identified the usage of this technique for guiding the systematic testing for a program whose code is not available. For instance, cross coverage testing could be applied to test multiple programs when the code of one or more of them is proprietary (only the specification could be available), meaning that their coverage analyzer is not available. In this case, cross tests could be useful for improving black-box testing of these programs.

References

[1] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *ACM Computing Surveys (CSUR)*, vol. 29, no. 4, pp. 366–427, 1997.

[2] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *Comput. J.*, vol. 52, no. 5, pp. 589–597, 2009. [Online]. Available: https://doi.org/10.1093/comjnl/bxm021

[3] B. Miranda and A. Bertolino, "Testing relative to usage scope: Revisiting software coverage criteria," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 3, pp. 18:1–18:24, 2020. [Online]. Available: https://doi.org/10.1145/3389126

[4] G. Nativ, S. Mittennaier, S. Ur, and A. Ziv, "Cost evaluation of coverage directed test generation for the ibm mainframe," in *Proceedings of International Test Conference 2001 (Cat. No. 01CH37260)*. IEEE, 2001, pp. 793–802.

[5] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at google," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 955–963.

[6] B. Beizer, *Software Testing Techniques (2nd Ed.)*. USA: Van Nostrand Reinhold Co., 1990.

[7] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 435–445.

[8] G. Gay, M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "The risks of coverage-directed test case generation," *IEEE Trans. Software Eng.*, vol. 41, no. 8, pp. 803–819, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2015.2421011

[9] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in *Proceedings of 3rd international symposium on empirical software engineering and measurement*. IEEE, 2009, pp. 291–301.

[10] S. Fischer, D. Rigoni, and N. Obrenović, "itest: Using coverage measurements to improve test efficiency," in *Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 1155–1158.

[11] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, aug 2013.

[12] M. Verhaegen, S. Kanev, R. Hallouzi, C. Jones, J. Maciejowski, and H. Smail, "Fault tolerant flight control-a survey," in *Fault tolerant flight control*. Springer, 2010, pp. 47–89.

[13] U. Voges, *Software diversity in computerized control systems*. Springer Science & Business Media, 2012, vol. 2.

[14] L. D. Ngan and R. Kanagasabai, "Semantic web service discovery: state-of-the-art and research challenges," *Personal and ubiquitous computing*, vol. 17, no. 8, pp. 1741–1752, 2013.

[15] Y. Higo, S. Matsumoto, S. Kusumoto, and K. Yasuda, "Constructing dataset of functionally equivalent java methods using automated test generation techniques," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 682–686.

[16] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on software engineering*, no. 12, pp. 1491–1501, 1985.

[17] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and analysis of hardware- and software-fault-tolerant architectures," *Computer*, vol. 23, no. 7, pp. 39–51, 1990.

[18] R. Holmes and R. J. Walker, "Systematizing pragmatic software reuse," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, pp. 1–44, 2013.

[19] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github," in *Proceedings of IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 291–301.

[20] M. Suzuki, A. C. de Paula, E. Guerra, C. V. Lopes, and O. A. L. Lemos, "An exploratory study of functional redundancy in code repositories," in *Proceedings of IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2017, pp. 31–40.

[21] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, and J. Grundy, "Opportunities and challenges in code search tools," *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–40, 2021.

[22] O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes, "A test-driven approach to code search and its application to the reuse of auxiliary functionality," *Information and Software Technology*, vol. 53, no. 4, pp. 294–306, 2011.

[23] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 81–92.

[24] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, 1988.

[25] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 334–344.

[26] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[27] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 2007, pp. 75–84.

[28] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè, "Automatic recovery from runtime failures," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 782–791.

[29] M. Sikri and R. Gell, "Adaptive soa stack and discovery framework for redundant services," in *Proceedings of 9th International Conference on Industrial and Information Systems (ICIIS)*. IEEE, 2014, pp. 1–6.

[30] G. Cousineau and P. Enjalbert, "Program equivalence and provability," in *Proceedings of International Symposium on Mathematical Foundations of Computer Science*. Springer, 1979, pp. 237–245.

[31] M.-C. Jakobs and M. Wiesner, "Peqtest: Testing functional equivalence," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham, 2022, pp. 184–204.

[32] A. Carzaniga, A. Mattavelli, and M. Pezzè, "Measuring software redundancy," in *Proceedings of IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 156–166.

[33] S. S. Brilliant, J. C. Knight, and P. Ammann, "On the performance of software testing using multiple versions," in *Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*. IEEE Computer Society, 1990, pp. 408–409.

[34] M. A. Vouk, "On back-to-back testing," in *Computer Assurance, 1988. COMPASS'88*. IEEE, 1988, pp. 84–91.

[35] M. R. Lyu, J. Horgan, and S. London, "A coverage analysis tool for the effectiveness of software testing," *IEEE transactions on reliability*, vol. 43, no. 4, pp. 527–535, 1994.

[36] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 257–266.

[37] Z. Xu, M. B. Cohen, and G. Rothermel, "Factors affecting the use of genetic algorithms in test suite augmentation," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, 2010, pp. 1365–1372.

[38] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 218–227.

[39] Z. Xu, Y. Kim, M. Kim, and G. Rothermel, "A hybrid directed test suite augmentation technique," in *Proceedings of IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE, 2011, pp. 150–159.

[40] S. Yoo and M. Harman, "Test data regeneration: generating new test data from existing test data," *Software Testing, Verification and Reliability*, vol. 22, no. 3, pp. 171–201, 2012.

[41] M. Pezzè, K. Rubinov, and J. Wuttke, "Generating effective integration test cases from unit ones," in *Proceedings of IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 11–20.

[42] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *Journal of Systems and Software*, vol. 157, p. 110398, 2019.

[43] B. H. Smith and L. Williams, "On guiding the augmentation of an automated test suite via mutation analysis," *Empirical software engineering*, vol. 14, pp. 341–369, 2009.

[44] R. Tzoref-Brill, S. Sinha, A. A. Nassar, V. Goldin, and H. Kermany, "Tackletest: A tool for amplifying test generation via type-based combinatorial coverage," in *Proceedings of IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 444–455.

[45] R. Bloem, R. Koenighofer, F. Röck, and M. Tautschnig, "Automating test-suite augmentation," in *Proceedings of 14th International Conference on Quality Software*. IEEE, 2014, pp. 67–72.

[46] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74–93, 2018.

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

[48] A. Bertolino, B. Miranda, R. Pietrantuono, and S. Russo, "Adaptive test case allocation, selection and generation using coverage spectrum and operational profile," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 881–898, 2019.

[49] J. Osei-Owusu, A. Astorga, L. Butler, T. Xie, and G. Challen, "Grading-based test suite augmentation," in *Proceedings of 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 226–229.

[50] D. Gaston and J. Clause, "A method for finding missing unit tests," in *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 92–103.