Can Testedness be Effectively Measured?

Iftekhar Ahmed
Oregon State University, USA
ahmedi@oregonstate.edu

Rahul Gopinath
Oregon State University, USA
gopinatr@oregonstate.edu

Caius Brindescu
Oregon State University, USA
brindesc@oregonstate.edu

Alex Groce
Oregon State University, USA
agroce@gmail.com

Carlos Jensen
Oregon State University, USA
cjensen@oregonstate.edu

ABSTRACT

Among the major questions that a practicing tester faces are deciding where to focus additional testing effort, and deciding when to stop testing. Test the least-tested code, and stop when all code is well-tested, is a reasonable answer. Many measures of "testedness" have been proposed; unfortunately, we do not know whether these are truly effective.

In this paper we propose a novel evaluation of two of the most important and widely-used measures of test suite quality. The first measure is statement coverage, the simplest and best-known code coverage measure. The second measure is mutation score, a supposedly more powerful, though expensive, measure.

We evaluate these measures using the actual criteria of interest: if a program element is (by these measures) well tested at a given point in time, it should require fewer future bug-fixes than a "poorly tested" element. If not, then it seems likely that we are not effectively measuring testedness. Using a large number of open source Java programs from Github and Apache, we show that both statement coverage and mutation score have only a weak negative correlation with bug-fixes. Despite the lack of strong correlation, there *are* statistically and practically significant differences between program elements for various binary criteria. Program elements (other than classes) covered by any test case see about half as many bug-fixes as those not covered, and a similar line can be drawn for mutation score thresholds. Our results have important implications for both software engineering practice and research evaluation.

CCS Concepts

•Software and its engineering → Software testing and debugging; Empirical software validation;

Keywords

test suite evaluation, coverage criteria, mutation testing, statistical analysis

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950324

1. INTRODUCTION

The quality of software artifacts is one of the key concerns for software practitioners and is typically measured through effective testing. While it is widely held that "you cannot test quality into a product," you *can* use testing to detect that the Software Under Test (SUT) has faults, and to estimate its likely overall quality. Moreover, while testing itself does not produce quality, it leads to the discovery of faults. When these faults are corrected, software quality improves.

Testing software poses questions. First, how much testing is needed? Has "enough" testing been done? Second, where should future test efforts be applied in a partially tested program? The typical approach to answering these questions is to measure the quality of the test suite, not the SUT. Numerous measures, primarily focused on code coverage [20–22] have been proposed, and numerous organizations set testing requirements in terms of coverage levels [37]. Both code coverage and mutation score measure the "testedness" of an SUT, using the dynamic results of executing a test suite[1]. However, it is not established that using such testing requirements, or suite quality measures in general, translates into an effective practice for producing better software.

While test driven development, in particular, has pushed testing to new prominence, practicing software programmers often balk at having to satisfy what they see as arbitrary coverage requirements. Some go so far as to suggest that "code coverage is a useless target measure"[2]. Some studies seem to support this conclusion, at least in part [25]. The utility of testing itself has even come under attack[3].

Our aim in this paper is to place the evaluation of test suites (and thus decision-making in testing) on a firmer footing, using a measure that translates directly into practical terms. There is, of course, no end to studies measuring the effectiveness of test suite evaluation techniques. However, these studies tend to either cover only a few subject programs or faults, not use real faults, not measure what developers directly care about, or assume the validity of mutation testing, which is itself a relatively unproven method

[1]Most studies consider coverage as measuring the testedness of the entire SUT for a given suite, but it is also reasonable to project this concept onto program elements; most practical applications of coverage assume this.

[2]For examples, see http://martinfowler.com/bliki/TestCoverage.html and http://blog.ploeh.dk/2015/11/16/code-coverage-is-a-useless-target-measure/.

[3]For examples, see https://pragprog.com/magazines/2012-10/its-not-about-the-unit-tests and http://www.rbcs-us.com/documents/Why-Most_unit-Testing-is-Waste.pdf

for evaluating a test suite. The methodology of such studies also often involves using tool-generated or randomly-chosen subset-based test suites to measure correlation. Such suites may not resemble real-world test suites, and thus be of little relevance to most developer practice.

We propose a simpler and more direct method of evaluation that eliminates some of the concerns mentioned above. It can be argued that a correlation between measure of suite quality and fault detection is meaningless if the faults detected are unimportant, trivial, or artificial. As a recent paper evaluating the ability of automated unit test generators to detect real faults phrased it "Just because a test suite... [is effective] does not necessarily mean that it will find the faults that *matter*" [38]. It is usually hard to argue that a bug that has been discovered and fixed did not matter: fixing bugs is difficult and resource-intensive, requiring developers (and possibly testers) to devote time to implementing a correction (and, hopefully, validating it). In many cases, the bug was detected because it caused problems for users. There is evidence that problems in code, such as those identified by code smells, that do not have significant immediate consequences are often never corrected, even at the price of design degradation [1]. Bug-fixes therefore usually indicate important defects: in general, developers thought these were the problems worth addressing. The most practical goal of testing therefore, is to *prevent future bug-fixes*, by detecting faults before release, avoiding impact on users, and usually lowering development cost.

We should then evaluate test suite quality measures by a simple process: does a higher measure of testedness for a program element (in our work, a statement, block, method or class) correlate with a smaller number of future bug-fixes? By avoiding whole-project measures and focusing on individual program elements, we avoid the confounding effects of test suite size [25]. While a large test suite can produce higher coverage and detect more faults, even if coverage and fault detection are not themselves directly related, it cannot (at least in any way we can imagine) cause statements that are covered to have fewer faults than those that are not covered, *unless coverage itself is meaningful*. Similarly, using individual program entities as the basis of analysis mitigates some of the possible effects of, e.g., test suites with good assertions also having better coverage. This can cause a test suite with high coverage to perform better than one without high coverage, on average, but it cannot, we claim, plausibly result in covered entities having fewer bug-fixes than ones not covered, unless *coverage itself* matters.

The core argument for our analysis is as follows: if a particular program element is *fully* tested to conform to its specification, then that program element *should have no* bug-fixes applied (until the element ceases to exist as a result of a non-bug-fix modification, e.g. adding new functionality potentially invalidating the old specification). Similarly, a program element that is not tested at all should on average have a higher chance of seeing future bug-fixes applied for the simple reason that a fault had no chance of being caught through testing. This, on its own, should result in a strong negative correlation between testedness and future bug-fixes, if our fundamental assumption about the utility of testing is correct and our measure of testedness is effective. *In general, a well-tested program element should require fewer bug fixes than a less well-tested program element.* In this paper we assume (and later, based on empirical data, argue) that testing itself is beneficial. We therefore primarily aim to evaluate the measurement of test suite quality/testedness. We focus on two important, widely studied, measures of suite quality. First, statement coverage is the simplest, most widely used, and easiest to understand coverage measure, and has some support as an effective measure of test suite quality in recent work [21]. Second, mutation score is not only commonly advocated as the best method for evaluating suite quality, but is essential to most other studies of coverage method effectiveness [22, 27].

We evaluate these test suite evaluation methods empirically using a large representative set of real world programs, real world test suites, and bug-fixes, and find that while there is a small (but statistically significant) negative correlation between our testedness measures and future bug-fixes for program elements, the effect is so small as to be practically insignificant. There is very little *useful* continuous relationship between measures of testedness and actual tendency to not have bugs detected and fixed, and while it is reasonable to bet that a more-tested element will have fewer faults, the size of the effect is very small.

However, we do find that there *is* a consistent and practically (as well as statistically) significant difference in the mean number of bug-fixes for code, if we apply selected binary measures of "well-testedness" based on coverage or mutation score. For example, *program elements with at least a 75% mutation score see, on average, only about half as many future big-fixes (normalized[4]) as program elements with a lower mutation score.*

One intuitively appealing explanation for the low correlation of testedness to bug-fixes is that, even if "poorly" tested, unimportant pieces of code are likely to see few future bug fixes. If very few users execute a program element, or if its effects have very limited impact, then the bug will likely not be fixed (even if reported). However, the problem of varying program element importance is *unlikely* to be the root cause for the lack of a useful continuous correlation for suite evaluation measures. If it were, we would expect the effects of importance to also prevent binary testedness criteria based on mere coverage from predicting future bug-fixes (since no one will bother to test program elements that are unlikely to ever exhibit important bugs). However, like program elements with $< 75\%$ mutation score, program elements that are not covered are also likely to see *nearly twice as many future bug-fixes*[5].

Nonetheless, perhaps a suite quality measure should reflect the importance of program elements. However, forcing developers to annotate code by its importance is impractical; we need a static measure of importance. One approach is to say that complex elements are more likely to be important, since developing complex code with many operators and conditionals, but low importance, is an unwise use of development resources. In this case, in addition to its other advantages, mutation testing may help take importance of code into account, in that complex program elements produce more mutants than simple elements (e.g., a simple logging statement will seldom perform any calculations, and

[4]By normalized bug-fixes, we mean bug-fixes per statement/line for elements larger than a single statement or line; unless we indicate otherwise, we always normalize bug-fixes.

[5]We only demonstrate this result for statements and methods; there were too few classes that were not covered by any tests in our data to show a significant relationship.

so often only produce a single statement-deletion mutant). We therefore also measure whether the number of mutants (as a measure of code complexity) predicts the number of bug-fixes applied to a program element, and whether the number of mutants predicts the mutation score for an element. Both effects are significant but small. Surprisingly, more complex code sees slightly fewer bug-fixes than simple code. As might be expected if complexity is associated with importance, more complex code is also slightly more tested, according to mutation score. Both effects are too weak to be of much practical value, however.

Our findings with respect to correlation of testedness measures and bug-fixes are potentially troubling for the research community. Software testing researchers often use a difference of a few percentage points in mutation score or a coverage measure as a means to assert that one test generation or selection technique is superior to another. However, our data shows that relying on a few percentage points is dangerous, as such small differences may not indicate real impact in terms of defects that are worth fixing. On the other hand, our data seems to support the types of "arbitrary" adequacy criteria often imposed by managers or governments (if not the precise values used). Indeed, our data suggests that while a continuous ranking of testedness for program elements is not currently possible, using various empirically-validated "strata" of testedness (not covered, covered but with poor mutation score, covered with high mutation score) may provide a simple, practical way to direct testing efforts.

The contributions of this paper are:

- A novel approach to examining the utility of test suite quality measures that is based on direct practical consequences of testing.
- Analysis of relationships between bug-fixes, test suite quality (testedness) measures, and code complexity and importance metrics for 49 sampled projects from Github and Apache.
- Evidence that there is small negative correlation between the number of mutants (normalized) and the number of future bug-fixes to a program element, indicating that complexity alone does not predict importance well; in fact, more complex program elements seem to be changed less often than simple ones. However, this may partly be due to the fact that more complex elements are also somewhat more well-tested (in terms of mutation score).
- Evidence that the negative correlation between testedness (by our measures) and number of future normalized bug-fixes is statistically significant, but far too small to have much practical impact.
- Evidence that well-chosen adequacy criteria (e.g.,is the mutation score above 75%) can be used to predict future normalized bug-fixes in a practical way (leading to differences of a factor of two in expected future bugs), and can serve to distinguish untested, poorly tested, and well-tested elements of an SUT.

Our data is available for inspection and further analysis at http://eecs.osuosl.org/rahul/fse2016/.

2. RELATED WORK

Ours is not the first study to attempt to evaluate measures of testedness [22]. Researchers have often attempted to prove that mutation score is well correlated with real world faults. DeMillo et al. [13] empirically investigated the representativeness of mutations as proxies for real faults. They examined the 296 errors in TEX and found that 21% were simple faults, while the rest were complex errors. Daran et al. [11] investigated the representativeness of mutations empirically using error traces. They studied the 12 real faults found in a program developed by a student, and 24 first-order mutants. They found that 85% of the mutants were similar to real faults.

Another important study by Andrews et al. [2], investigated the *ease* of detecting a fault (both real faults and hand seeded faults), and compared it to the ease of detecting faults introduced by mutation operators. The ease was calculated as the percentage of test cases that killed each mutant. Their conclusion was that the ease of detection of mutants was similar to that of real faults. However, they based this conclusion on the result from a single program, which makes it unconvincing. Further, their entire test set was eight C programs, which makes the statistical inference drawn liable to type I errors. We also note that the programs and seeded faults were originally from Hutchins et al. [24] who chose programs that were subject to certain specifications of understandability, and the seeded faults were selected such that they were neither too easy nor too difficult to detect. In fact, the study eliminated 168 faults for being either too easy or too hard to detect, ending up with just 130 faults. This is clearly not an unbiased selection and cannot really tell us anything about the ease of detection of hand seeded faults in general (because the criteria of selection itself is confounding). A follow up study [3] using a large number of test suites from a single program, space.c, found that the mutation detection ratio and fault detection ratio are related linearly, with similar results for other coverage criteria (0.83 to 0.9). Linear regression on the mutation kill and fault detection ratios showed a high correlation (0.9).

The problems with some of these studies were highlighted in the work of Namin et al. [34] who used the same set of C programs as Andrews et al. [2], but combined them with analysis of four more Java classes from the JDK. This study used a different set of mutation operators and fault seeding by student programmers for the Java programs. Their analysis concluded that we have to be careful when using mutation analysis as a stand-in for real faults. The programming language, the kind of mutation operators used, and even the test suite size all have an impact on the relation between mutations introduced by mutation analysis and real faults. In fact, using a different mutation operator set, they found that there is only a weak correlation between real faults and mutations. However, their study was constrained by the paucity of real faults, which were only available for a single C program (also used in Andrews et al. [2]). Thus, they were unable to judge the ease of detection of real faults in Java programs. Moreover, the students who seeded the faults had knowledge of mutation analysis which may have biased the seeded faults (thus resulting in high correlation between seeded faults and mutants). Finally, the manually seeded faults in C programs, originally introduced by Hutchins et al. [24], were again confounded by a selection criteria which eliminated the majority of faults as

being either too easy or too hard to detect. Just et al. [27], using 357 real faults from 5 projects, showed that 1) adding more fault-detecting tests to a test suite led to the mutation score increasing more often (73%) than either branch (50%) or statement coverage (30%) and 2) mutation score was more positively correlated with fault detection than either of the other measures. Multiple studies provide evidence that mutation analysis subsumes different coverage measures [8, 30, 35], and it is on this basis that mutation score is often regarded as the "gold standard" for test suite quality measures.

Besides mutation score, the other metric that is commonly used to measure the adequacy of testing is code coverage, that is, a measure of the set of program elements or code paths that are executed by a set of tests. A large body of work considers the relationship between coverage criteria and fault detection — however, the analysis is often "mediated" by assuming the validity of mutation analysis (this is why we discussed mutation above, before turning to code coverage). Mockus et al. [32] found that increased coverage leads to a reduction in the number of post-release defects but increases the amount of test effort. Gligoric et al. [19,20] used the same *statistical* approach as our paper, measuring both Kendall τ and R^2 to examine correlations, for realistically non-adequate suites. Gligoric et al. found that branch coverage does the best job, overall, of predicting the best suite for a given SUT, but that acyclic intra-procedural path coverage is highly competitive and may better address the issue of ties, which is important in their research/method comparison context. Inozemtseva et al. [25] investigated the relationship of various coverage measures and mutation score for different random subsets of test suites. They found that when test suite size is controlled for, only low to moderate correlation is present between coverage and effectiveness, for all coverage measures used. Frankl and Weiss [16] compared of branch coverage and def-use coverage, showing that def-use was more effective for fault detection and there is stronger correlation to fault detection for def-use. Namin and Andrews [33] showed that fault detection ratio (non-linearly) correlated well with block coverage, decision coverage, and two different data-flow criteria. Their research suggested that test suite size was a significant factor in the model. Wei et al. [44] examined branch coverage as a quality measure for suites for 14 Eiffel classes, showing that for randomly generated suites, branch coverage behavior was consistent across many runs, while fault detection varied widely. In their experiments, early in random testing, when branch coverage rose rapidly, current branch coverage had high correlation to fault detection, but branch coverage eventually saturated while fault detection continued to increase; the correlation at this point became very weak.

Gupta et al. [23] compared the effectiveness and efficiency of block coverage, branch coverage, and condition coverage, with mutation kill of adequate test suites as their evaluation metric. They found that branch coverage adequacy was more effective (killed more mutants) than block coverage in all cases, and condition coverage was better than branch coverage for methods having composite conditional statements. The reverse, however, was true when considering the efficiency of suites (average number of test cases required to detect a fault). Li et al. [29] compared four different criteria (mutation, edge pair, all uses, and prime path), and showed that mutation adequate testing was able to detect the most hand seeded faults (85%), while other criteria performed similarly to each other (in the range of 65% detection). Similarly, mutation coverage required the fewest test cases to satisfy the adequacy criteria, while prime path coverage required the most. Therefore, while there are no compellingly large-scale studies of many SUTs selected in a non-biased way to support the effectiveness of mutation testing, it is at least highly plausible as a better standard than other criteria.

Cai et al. [9] investigated correlations between coverage criteria under different testing profiles: whole test set, functional test, random test, normal test, and exceptional test. They investigated block coverage, decision coverage, C-use and P-use criteria. Curiously, they found that the relationship between block coverage and mutant kills was not always positive. Block coverage and mutant kills had a correlation of $R^2 = 0.781$ when considering the whole test suite, but as low as 0.045 for normal testing and as high as 0.944 for exceptional testing. The correlation between decision coverage and mutation kills was higher than statement coverage, for the whole test suite (0.832), ranging from normal test (0.368) to exceptional test (0.952). Frankl et al. [17] compared the effectiveness of mutation testing with all-uses coverage, and found that at the highest coverage levels, mutation testing was more effective. Kakarla [28] and Inozemtseva [26] demonstrated a linear relationship between mutation detection ratio and coverage for individual programs. Inozemtseva's study used machine learning techniques to come up with a regression relation, and found that effectiveness is dependent on the number of methods in a test suite, with a correlation coefficient in the range $0.81 \leq r \leq 0.93$. The study also found a moderate to high correlation, in the range $0.61 \leq \tau \leq 0.81$, between effectiveness and block coverage when test suite size was ignored, which reduced when test suite size was accounted for. Kakarla found that statement coverage was correlated to mutation coverage in the range of $0.73 \leq r \leq 0.99$ and $0.57 \leq \tau \leq 0.94$. Gopinath et al. [21] found that statement, out of branch, and path coverages, best correlated with mutation score, and hence may best predict defect density, in a study that compared suites and mutation scores across projects, rather than using multiple suites for the same project.

The study by Tengeri et al. [42] provided a simple (essentially non-statistical) assessment of how statement coverage, mutation score, and reducibility predicted project defect densities for four open source projects, using a limited set of mutation operators.

None of these studies, to our knowledge, adopted the method used in this paper, where rather than investigate faults and their detection, we look at whether being "well tested" has predictive power with respect to *future* defects[6]. Most also consider a smaller, less representative (at least of open source projects) set of programs, and the majority are based on programs chosen opportunistically, rather than by our more principled sampling approach. The programs used are often small but well-studied benchmarks such as the Siemens/SIR [41] suite, partly for purposes of comparison to earlier papers, and partly due to the lack of easily available realistic projects with test suites and defects, before the ad-

[6]It is remotely possible that Tengeri et al. [42] are using a similar method, but this is not clear from their description, and the reasoning behind our approach is not elaborated in their work.

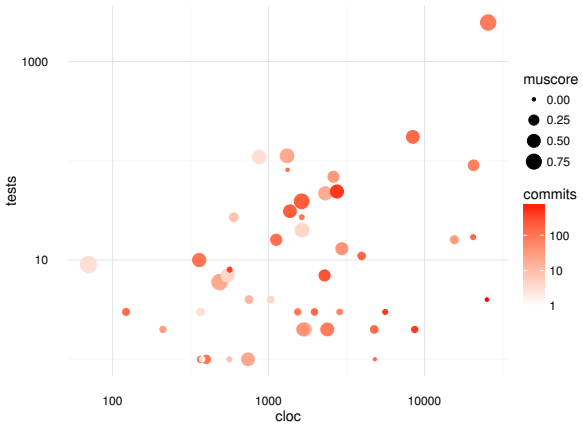

Figure 1: CLOC vs. tests for our projects.

vent of very large open source repositories. Unfortunately, as noted by Arcuri and Briand, not at least attempting to randomize selection of programs to study can greatly reduce the generalizability of results [6].

3. METHODOLOGY

Our goal was to evaluate various approaches to assessing the testedness of a program element, using future bug-fixes.

3.1 Collecting the Subjects

For our empirical evaluation, we tried to ensure that the programs chosen offered a reasonably unbiased representation of modern software. We also tried to reduce the number of variables that can contribute to random noise during evaluation. With these goals in mind, we chose a sample of Java projects from Github [18] and the Apache Software Foundation [4]. All projects selected used the popular maven [5] build system. We randomly selected $1,800$ projects. From these, we eliminated aggregate projects that were difficult to analyze, leaving $1,321$ projects, of which only 796 had test suites. Out of these, 326 remained after eliminating projects that did not compile (for reasons such as unavailable dependencies, or compilation errors due to syntax or bad configurations). Next, the projects that did not pass their own test suites were eliminated as mutation analysis requires a passing test suite. Finally, we eliminated projects our AST walker could not handle. This resulted in 49 projects selected. The distribution of project size vs. test suite size, and the corresponding mutation score is given in Figure 1.

3.2 Mutant Generation

In the next phase of our analysis, we used PIT [10] for our mutation analysis. PIT has been used in multiple previous studies [12, 21, 25, 40]. We extended PIT to provide the full matrix of test failures over mutants and tests. Mutants can basically be divided into three groups based on their runtime behavior: not covered, killed, and live mutants. We used this basic categorization in our analysis.

3.2.1 Computing Complexity

In order to evaluate the effect of *complexity* on testedness, it is necessary to find a reasonable measure for complexity. While previous research has used cyclomatic complexity [31] as a measure of complexity, the measure can not be used for single assignment statements. Further, it was found that cyclomatic complexity was strongly correlated with the size of code [39] and provided little extra information. We argue that a better measure of complexity is the average number of mutants generated from each statement. When a piece of code is highly complex, we expect to see a larger number of mutants compared to simpler code.

3.3 Tracking Program Elements

We started our investigation from an arbitrarily determined recent, but not too recent, point in time deemed the "epoch" — December 1, 2014. This was done to provide a point from which testedness (mutation score and statement coverage) could be calculated, and with respect to which bug-fixes could be considered to be "in the future". For the source code and test suite at epoch, we computed mutation score and statement coverage for each statement, block, method, and class in each project.

In order to determine when a program element (statement, block, method, or class) was changed, and track its history, we used the GumTree Differencing Algorithm [14]. For each element of interest, we considered it changed if the corresponding AST node was changed, or had any children that were added, deleted or modified. The algorithm maps the correspondence between nodes in two different trees, which allowed us to accurately track the history of the program elements.

Using AST differencing gives us three advantages over simple line based differencing. The first is that the algorithm ignores any whitespace changes. Second, we are able to track a node even if its position in the file changes (e.g. because lines have been added or deleted before our node of interest). Third, we are able to track nodes across refactorings, as long as the node stays in the same file. For example, we can track a node that has been moved because of an extract method refactoring.

When considering which statements to track, we used the version of the source code at epoch to determine which AST node resided at that particular line. We filtered only the commits that touch the file of interest. We then tracked that AST node forward in time, taking note of the commits that changed that particular node. For Java, it is possible for multiple statements to be in the same line (for example, a local variable declaration statement inside an if statement). In this case, we considered the innermost statement, as this gives the most precise results.

The epoch is (and can be) arbitrary. Our basic assumption is is that test coverage increases monotonically (people do not remove tests very often, and tests don't lose coverage). We checked this assumption for 5 random projects, at 1-5 random points (depending on history length) before epoch, and confirmed: once covered, always covered, in every case.

3.4 Classifying Commits

In order to answer our research questions, we needed to categorize the code commits. For each program element, we computed the number of commits that touched that element starting from the epoch. For our purpose, code commits can be broadly grouped into one of two categories: (1) bug-fixes and improvements (modifying existing code), and (2) *Other* — commits that introduced new features or functionality (adding new code) or commits that were related to documentation, test code, or other concerns. Two key problems are that it is not always trivial to determine which cate-

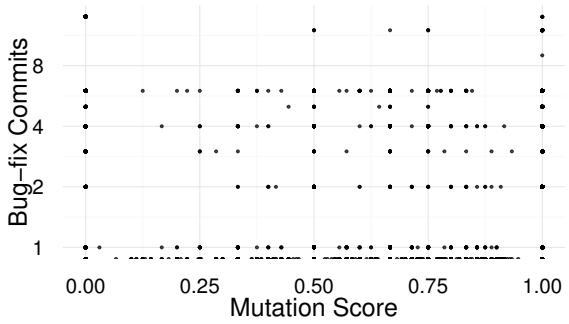

Figure 2: Mutation score vs. bug-fix commits for covered lines.

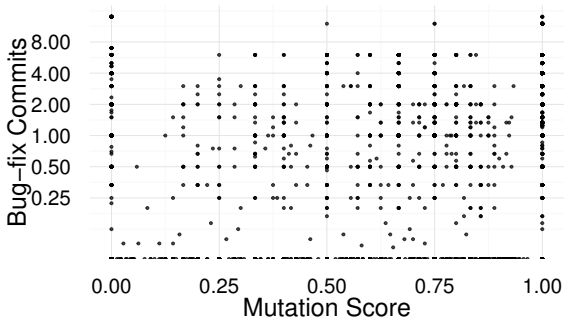

Figure 3: Mutation score vs. bug-fix commits for covered blocks.

Table 1: Naive Bayes classifier details

	Precision	Recall	f1.score	Support
Bug-fix	0.63	0.43	0.51	75.00
Other	0.74	0.86	0.80	140.00

gory a commit falls under, and that larger projects see a huge amount of activity. Manual classification of all commits was therefore not an option, and we decided to use machine learning techniques for this purpose, rather than limit the statistical power of our study (especially as arbitrarily dropping the most active subjects would clearly potentially introduce a large bias into our results).

3.4.1 Manual Classification of Fix-inducing Changes

In order to build a classifier for bug-fixing commits, we randomly sampled commits and manually labeled fix-inducing commits. Some keywords indicating bug-fixes were *Fix*, *Bug*, and *Resolves*, along with their derivatives. We should mention that not all bug-fixing commit messages include the words *bug* or *fix*; indeed, commit messages are written by the initial contributor of a patch, and there are few guidelines as to their contents. A similar observation was made by Bird et al. [7], who performed an empirical study showing that bias could be introduced due to missing linkages between commits and bugs. Improvements were manually identified based on the following keywords: *Cleanup*, *Optimize*, and *Simplify* or their derivatives. Commits were placed into the *Other* category if they had the keywords *Add* or *Introduce*. The number of lines modified was also compared with the lines added. Those commits with more lines added than modified were considered more likely to be associated with new features and were placed in the *Other* category. Anything that did not fit into this pattern was also marked as *Other*. We manually classified a set of 1,500 commits.

3.4.2 Training the Commit Classifier

We used the set of manually classified commits as the training data for the machine learning classifiers. Two evaluators worked independently to classify the commits. Their datasets had a 33% overlap, which we used to calculate the inter-rater reliability. This gave us a Cohen's Kappa of 0.90. In our training dataset the portion of bug-fixes was 46.30%, with 53.70% of the commits assigned to the *Other* category.

We trained a *Naive-Bayes* (NB) classifier and a *Support Vector Machine* (SVM) for automatically classifying the commits, using the scikit [36] platform. We applied the classifiers to the training data with 12-fold cross-validation. Our goal was to achieve high precision and recall, so we used the F_1-score to measure and compare the performance of the models. The F_1-score considers precision and recall by taking their harmonic mean. The NB classifier outperformed the SVM. Tian et al. [43] suggested that for keyword based classification the F_1 score is usually around 0.55 which also happened in our case. We used the classification identified by the NB classifier to classify 11566 commits. Table 1 has the quality indicator characteristics of the NB classifier. While our classifier is far from perfect, it is comparable to "good" classifiers for this purpose in the literature (over a larger set of projects), and we believe it is likely that any biases do not have confounding interactions with the goals of our project. That is, while we may only analyze about 43% of bug-fixes, it would be surprising if the missed bug-fixes relate in some systematic way to the dynamic testedness measures of program elements, given that the classifier only sees code commits. Since our analysis only relies on relative counts of bug-fixes for elements, so long as we do not systematically undercount bug-fixes for only some elements, our results should be valid.

The bug-fixes associated with each program element in our analysis are based on the classifier results in a simple way. For each element, we count commits that affect that element that are classified as *Bug-fix* up to the first commit that is classified as *Other*. This is because once an element has had a change that is not a bug-fix, it is often no longer valid to assume tests at the epoch apply to that element, or that it even still exists with the same functionality. However, so long as only bug-fixes are applied, we assume the tests still apply to the program element, so all bug-fixes count as missed by the tests at epoch. Note that our classifier for *Other* commits is highly effective.

4. ANALYSIS

We analyze the impact of testedness on program element bug-fixes using both mutation score and statement coverage in increasing lexical scope for each statement (except for statement coverage), block, method, and class.

4.1 Correlation Results

We answer this question in increasing scope from statement, smallest block, method, and then class. In each scope, we evaluate how the degree of adequacy in both mutation score and statement coverage affects the total number of bug-fix commits.

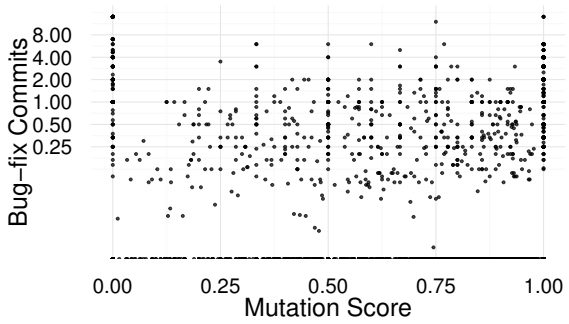

Figure 4: Mutation score vs. bug-fix commits for covered methods.

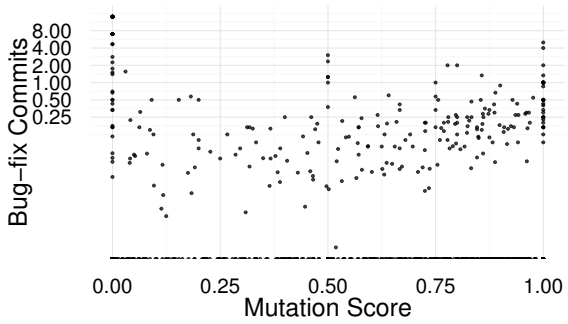

Figure 5: Mutation score vs. bug-fix commits for covered classes.

Table 2: Correlation between total number of bug-fixes per statement and mutation score

	(a) R^2				(b) Kendall τ_b	
	Mean	Low	High	p	Mean	p
Statements	-0.12	-0.13	-0.11	0.00	-0.13	0.00
Blocks	-0.14	-0.15	-0.12	0.00	-0.19	0.00
Methods	-0.16	-0.18	-0.14	0.00	-0.14	0.00
Classes	-0.13	-0.18	-0.08	0.00	-0.10	0.00

Table 3: Correlation between total number of bug-fixes per statement and statement coverage

	(a) R^2				(b) Kendall τ_b	
	Mean	Low	High	p	Mean	p
Statements	-0.11	-0.12	-0.10	0.00	-0.13	0.00
Blocks	-0.13	-0.14	-0.12	0.00	-0.21	0.00
Methods	-0.14	-0.16	-0.12	0.00	-0.13	0.00
Classes	0.09	0.04	0.13	0.00	-0.04	0.07

4.1.1 Mutation Score (μ)

The correlation between number of bug-fixes per statement and mutation score is given in Table 2.

For statements and methods, there is a statistically significant small negative linear correlation between number of bug-fixes per statement and mutation score. A similar effect is observed with Kendall τ_b correlation, where a small but statistically significant negative correlation is observed for statements, blocks, methods and classes.

The plot of mutation score vs. normalized bug-fixes for statements is given in Figure 2, for blocks in Figure 3, for methods in Figure 4, and for classes in Figure 5.

4.1.2 Statement Coverage (λ)

The correlation between number of bug-fixes per statement and statement coverage is given in Table 3.

For statements, blocks, and methods, there was a small but statistically significant negative linear correlation between coverage and bug-fixing commits. Surprisingly, the classes had a weak but significant positive correlation. A similar small but statistically significant negative correlation is also observed using Kendall τ_b, where even classes showed a small negative (but statistically insignificant) correlation. These correlations (for mutant score and for statement coverage) are much lower than those seen in recent studies showing good correlation between coverage metrics and mutation scores [19, 21] (these studies are measuring a different property, but in some sense aiming for similarly strong correlations). These correlations are not so small as to be completely devoid of value, but they do make the use of these measures dubious when comparing program elements or test suites with only small testedness differences. Unfortunately, this is a common practice in the evaluation of software testing experiments. Worse still, these results might be thought to suggest that testedness cannot be effectively measured, leaving the practicing tester without useful guidance.

Table 4: Difference in bug-fixes between covered and uncovered program elements

	Covered	Uncovered	p
Statement	0.68	1.20	0.00
Block	0.42	0.83	0.00
Method	0.40	0.87	0.00
Class	0.45	0.32	0.10

4.2 Binary Testedness: Is It Covered?

However, using testedness as a continuous valuation, where we expect slightly more tested program elements to have fewer bug-fixes, is not the only way to make use of testedness. Instead of trying to separate very similarly tested elements, we could simply draw a line between tested and not-tested program elements. For example, common sense suggests that if testing is useful at all, then code that is not covered should probably have more bug-fixes that code that has at least some test covering it. This rationale is the intuition behind ideas like "getting to 80% code coverage," though it does not justify any particular target value. Code that isn't executed in tests is surely less tested than code that executes in even very poor tests (since even very badly designed tests with a weak oracle may catch crashes, uncaught exceptions, and infinite loops, for example).

We compared the mean number of bug-fixes for covered vs. uncovered program elements using a t-test. The results are shown in Table 4. By covered element we mean a program element which has *at least* a single statement exercised by some test case. While this is a reasonable binary distinction up the method level, a class with only a single statement covered may not be much more tested than a class that does not have any statements covered. This may account for the difference seen for classes in Table 4. We also note that there is insufficient data for statistical significance in classes (most classes are covered by at least some test).

4.3 Binary Testedness: Mutation Score and Coverage Thresholds

While measuring testedness based on mutation score or statement coverage as a continuous value of limited value, we can do much better than just drawing a meaningful dividing line between covered and not-covered program elements.

We can instead evaluate whether the mean number of bug-fixes differs significantly when the tests reach a given adequacy threshold. Table 5 and Table 6 tabulate the mean number of normalized bug-fix commits per statement for both above and below the thresholds $\mu = \{0.25, 0.5, 0.75, 1.0\}$ and $\lambda = \{0.25, 0.5, 0.75, 1.0\}$. We find that there is a statistically and practically significant difference between the mean number of bug-fixes for both measures at all thresholds selected (though with classes perfect statement coverage strangely becomes a predictor of more faults). Note that for individual statements, all thresholds based on statement coverage are equivalent (coverage is always 0 or 1).

Table 7 shows mutant threshold results if we first remove all program entities that are not covered. This has little impact on the ability of thresholds to predict bug-fixes.

4.4 Complexity and Change

We compare the number of mutants, normalized by the size of the program element (e.g. number of lines), to the number of post-epoch bug-fixes for that element.

Statements: Comparing the *number of bug-fixes* to the *number of mutants* per statement, we find that the 95% confidence interval is $\{-0.004697, 0.013204\}$ $p > 0.01$.

Methods: Comparing the *number of bug-fixes* to the *number of mutants* per method, we find that the 95% confidence interval is $\{-0.087117, -0.048715\}$ $p < 0.01$.

Classes: Comparing the *number of bug-fixes* to the *number of mutants* per class, we find that the 95% confidence interval is $\{-0.096285, -0.000863\}$ $p > 0.01$.

Summary: Most results are not statistically significant. Further, there is a weak correlation between the number of mutants (normalized) and the number of bug-fixes. More "complex" code as measured by number of mutations has slightly fewer bug-fixes, but the correlation is even weaker than between testedness measures and bug-fixes. However, the difference in correlation is not very large, so another way to interpret this is that as a continuous measure, simple number of mutants, normalized, is only slightly worse as a predictor of bug-fixes than "testedness." However, unlike testedness measures, the number of mutants does not provide a useful binary predictor for bug-fixes. Binary splits based on a threshold using the mean normalized mutants (2.79) do not produce significantly different populations. Setting a threshold of 5 or more normalized mutants does produce significant differences ($p < 0.0001$), but the means are very similar, e.g., 1.1 bug-fixes for less complex statements vs. 0.95 bug-fixes for statements with more mutants.

4.5 Complexity and Testedness

Comparing the *normalized number of mutants* to the *mutation score* per program element:

Statements: The 95% confidence interval is $\{0.008016, 0.025912\}$ $p < 0.01$.

Methods: The 95% confidence interval is $\{0.005755, 0.044311\}$ $p > 0.01$.

Classes: The 95% confidence interval is $\{-0.049426, 0.046223\}$ $p > 0.01$.

Summary: At the statement level (only) there is a statistically significant but very weak correlation between the number of mutants (normalized) and the mutation score. More complex statements are (very slightly) more well-tested.

5. DISCUSSION

Our empirical results have some potentially important consequences for testing research and practice.

5.1 The Danger of Relying on Small Testedness Differences

First, there is only a weak correlation between either statement coverage or mutation score and future bug-fixes. This indirectly suggests that research efforts using coverage or mutants to evaluate test suite selection, generation, or reduction algorithms may draw unwarranted conclusions from small, significant differences in these measures. In particular, it may suggest that using mutation to evaluate testing experiments can potentially fail to reflect the ability of systems to detect the types of faults that are detected by practitioners and worth correcting in real-life. Given that the literature supporting the value of code coverage as a predictor of fault detection mostly relies on the ability of mutation testing to reflect real fault detection, and that mutation testing's effectiveness is validated by only a small number of studies, none of which present overwhelming evidence over a large number of programs, we strongly suggest that testing experiments, whenever possible, should rely on the use of some real faults in addition to coverage or mutation-score based evaluations. In some contexts, where detecting all possible faults is the goal (e.g., safety critical systems) and the oracle for correctness is known to be extremely good, mutation-based analyses may be justified, but even in those cases data based on real faults would be ideal.

5.2 Practical Application of Thresholds

On the other hand, our results show that numerous simple percentage thresholds for statement coverage and mutation score can, in a statistically significant way, predict the number of bug-fixes (with mean differences between populations of about 2x). This suggests a simple method for prioritizing testing targets in a program. The entities with the highest bug-fix counts were, unsurprisingly, those not even covered by any tests. As a first priority, covering uncovered program elements is likely to be the most rewarding way to improve testedness, since these elements can be expected to have the most potential undetected bugs that will be revealed in the near future. Surviving mutants of entities with low mutation scores can then be used to guide further testing. One obvious question is, which threshold should be used, since many thresholds seem effective? Our data shows that it really does not matter much — the significance and even average bug-fixes are not radically different for different thresholds. The simplest answer is to start with low thresholds, keep improving testing until there are no remaining interesting elements below the current threshold, then move on to a higher threshold. Setting a particular threshold for project-level testing is not supported by our data, however, as there is no clearly "best" dividing line, only a number of ways to define "less tested" and "more tested" elements, most of which equate to more bug-fixes for less tested elements.

Table 5: Mutation score thresholds

	(a) 0.25			(b) 0.5			(c) 0.75			(d) 1.0		
	$\mu \geq 0.25$	$\mu < 0.25$	p	$\mu \geq 0.5$	$\mu < 0.5$	p	$\mu \geq 0.75$	$\mu < 0.75$	p	$\mu \geq 1$	$\mu < 1$	p
Statements	0.60	1.20	0.00	0.60	1.19	0.00	0.58	1.16	0.00	0.58	1.14	0.00
Blocks	0.39	0.81	0.00	0.39	0.79	0.00	0.39	0.71	0.00	0.39	0.67	0.00
Methods	0.32	0.87	0.00	0.33	0.85	0.00	0.34	0.81	0.00	0.41	0.75	0.00
Classes	0.11	0.55	0.00	0.12	0.51	0.00	0.13	0.46	0.00	0.20	0.40	0.00

Table 6: Statement coverage score thresholds

	(a) 0.25			(b) 0.5			(c) 0.75			(d) 1.0		
	$\lambda \geq 0.25$	$\lambda < 0.25$	p	$\lambda \geq 0.5$	$\lambda < 0.5$	p	$\lambda \geq 0.75$	$\lambda < 0.75$	p	$\lambda \geq 1$	$\lambda < 1$	p
Statements	0.68	1.20	0.00	0.68	1.20	0.00	0.68	1.20	0.00	0.68	1.20	0.00
Blocks	0.42	0.83	0.00	0.42	0.83	0.00	0.42	0.82	0.00	0.42	0.82	0.00
Methods	0.40	0.87	0.00	0.41	0.86	0.00	0.42	0.84	0.00	0.46	0.80	0.00
Classes	0.48	0.31	0.04	0.51	0.30	0.01	0.59	0.28	0.00	0.90	0.24	0.00

5.3 Complexity, Bug-Fixes, and Testedness

There does not seem to be any very strong or interesting relationship between complexity (as measured by number of mutants) and bug-fixes, or between complexity and testedness. More complex code is (very slightly) less fixed, perhaps because it is (very slightly) more tested. The main take-away from the complexity analysis is that the number of mutants is almost as good a predictor of lack of bug-fixes as testedness, if used as a simple correlation, but it does not support useful binary distinctions in likely bug-fixes.

5.4 Testing is Likely Effective

One final point to note is that our data provides fairly strong support for the idea that *testing is effective in forcing quality improvements in code*. Our measures of testedness are, essentially, based purely on the dynamic properties of a test suite, not on static properties of program elements (the number of mutants for an entity depends on static properties, but all statements with any mutants can achieve or fail to achieve a score of any particular threshold). This means that, without using the static properties of code, the degree to which code is exercised in a test suite can often be used to predict which of two entities will turn out to require more bug-fixes. As far as we can determine, there are only a few potential causes for this ability to use the dynamics of a test suite to predict bug-fixes:

1) Some unknown property not related to code quality results in both a tendency to write tests that cover code and in fewer bug-fixes for that code. 2) A known property results in both a tendency to write tests that cover code and in fewer bug-fixes for that code: namely, good developers write tests for their already more correct code. Testing itself is more a sign of good code than a cause of good code. 3) Tests covering code often detect bugs, and developers fix the bugs, so the code has fewer bugs to fix.

The first possibility is, in our opinion, unlikely — it is difficult to imagine such an unknown factor. Some obvious candidate factors do not really bear up on examination. For example, perhaps code with many bug-fixes is new code, and so has not yet had tests written for it. If the act of writing tests for the new code makes it less buggy, however, then testing is in fact effective. Moreover, the predictive power of mutation score being over a threshold is present even if we restrict our domain to entities that have at least one covering test. New code might be expected to be completely untested, removing most truly new (no tests) code from this population. The second possibility is more plausible, and may well be true to some extent. The third possibility seems most plausible, and we believe is likely to be the main cause of the observed effects. However, even if we assume that the second explanation is the primary cause for the relationships we observed, notice the peculiar consequences of this claim: developers who believe testing is worthwhile, and devote more time to it, are "wrong" in that testing itself is useless, but on the whole produce statistically better code than those who do not value testing. This may not be an appealing argument to those dubious about testing's value.

While it could be argued that other measures of testedness such as warnings generated by static analysis tools could be an even better indicator, we believe that the number of bugs fixed is the most *direct* measure of testedness available.

6. THREATS TO VALIDITY

Threats Due to Sampling Bias: To ensure representativeness of our samples, we opted to use search results from the Github repository of Java projects that use the `Maven` build system. We picked *all* projects that we could retrieve given the Github API, and selected from these only based on necessary constraints (e.g., the project must build, and tests at epoch must pass). However, our sample of programs could be biased by skew in the projects returned by Github. Github's selection mechanisms favoring projects based on some unknown criteria may be another source of error. We also handpicked some projects from Apache, such as commons-lang. As our samples only come from Github and Apache, this may be a source of bias, and our findings may be limited to open source programs. However, we believe that the large number of projects more than adequately addresses this concern.

Bias Due to Tool Used: For our study, we relied on PIT. We have done our best to extend PIT to provide a reasonably sufficient set of mutation operators, ensuring also that the mutation operators were non-redundant (and have checked for redundancy in past work using PIT).

Secondly, we used the Gumtree algorithm discussed earlier for tracking program elements across commits. However, the

Table 7: Mutation score thresholds with uncovered program elements filtered out

	(a) 0.25			(b) 0.5			(c) 0.75			(d) 1.0		
	$\mu \geq 0.25$	$\mu < 0.25$	p	$\mu \geq 0.5$	$\mu < 0.5$	p	$\mu \geq 0.75$	$\mu < 0.75$	p	$\mu \geq 1$	$\mu < 1$	p
Statements	0.60	1.16	0.00	0.60	1.11	0.00	0.58	0.95	0.00	0.58	0.89	0.00
Blocks	0.39	0.72	0.00	0.39	0.64	0.00	0.39	0.50	0.00	0.39	0.47	0.00
Methods	0.32	0.90	0.00	0.33	0.71	0.00	0.34	0.53	0.00	0.41	0.39	0.68
Classes	0.11	1.66	0.00	0.12	1.13	0.00	0.13	0.75	0.00	0.20	0.50	0.00

algorithm used is unable to track program elements across renames or movement to another folder. Further, refactoring that involves modification of scope, such as moving the code out of the current compilation unit also causes the algorithm to lose track of the program element after refactoring.

Further, In this study we did not apply a systematic method for the detection and removal of equivalent mutants. This might have an impact on the mutation score of some projects.

Bias Due to Commit Classification: Our determination of commits as bug-fixes or not and of commits that "end the history" of a program element both depend on a learned classifier. While our results do not require those results to be anywhere near perfect, it may be that some unknown bias in the failures unduly influences our results, or gives rise to the weakness of observed correlations.

Bias Due to Lack of High Coverage: Some researchers have found that a strong relationship between coverage and effectiveness does not show up until very high coverage levels are achieved [15,17,24]. Since the coverage for most projects rarely reached very high values, it is possible that we missed the existence of such a dependent strong relationship.

Bias Due to Confounding Factors: Numerous confounding factors exist. For example, we assume that there is no specific skew in the individuals responsible for the bug fixes, and other personality factors in projects does not come into play. However, this cannot be guaranteed. Next, bug fixes may be done under various circumstances. For example the quality of a bug fix under time pressure may be very different from the quality of a bug fix under more leisure. Finally, we do not consider the changes to the test cases themselves. However, we believe that the impact of these factors are limited due to the large number of subjects considered.

7. CONCLUSION

This paper uses a novel method to evaluate the effectiveness of test suite quality measurements, which, we suggest essentially aim to capture the "testedness" of a program or program elements. Much of previous research attempting to evaluate such measures operates by a procedure that, at a suitably high level of abstraction, can be described as first collecting a large set of tuples of the form `(testedness measure for suite, # faults found by suite)`, then applying some kind of statistical analysis. Details vary, in that suites may all be for one SUT, or for multiple SUTs (though seldom for more than 5-10 SUTs), and in most cases "actual faults" are either hand-seeded or "faults" produced by mutation testing (which is assumed to measure real fault detection on a largely recently established and still limited empirical basis [27]). These studies have produced a variety of results, sometimes almost contradictory [22]. Is coverage useful? Is mutation score (more) useful?

We propose a different approach. Measuring fault detection for a suite can be extremely labor-intensive; worse, depending on the definition of faults, we may give too much credit for detecting faults that are of little interest to most developers. Instead, our evaluation chooses a point in time, collects testedness measures for a passing test suite from that date, and then examines whether these measures predict actual future bug-fixes for program elements. If "well tested" elements of a program require no less effort to correct, then either we are not measuring testedness effectively, or testing itself is ineffective.

We assume that testing is effective. Under this assumption, we show that there *is* the expected negative correlation between testedness and number of future bug-fixes. However, this correlation is so weak that it makes using it to compare testedness values in the continuous fashion, where slightly more tested code is assumed to be slightly better, or slightly higher scoring test suites are assumed to be better than slightly lower scoring test suite, a dubious enterprise. This suggests that the evaluation method in many software testing publications may be of questionable value. On the other hand, when we use testedness measures to split program elements into simple "more tested" and "less tested" groups, the population differences are typically significant and the mean bug-fixes are sufficiently different (usually about a factor of 2x) to provide practical guidance in testing.

So, is (statement) coverage useful? Is mutation score (more) useful? The answers, we believe, may be that *it depends on what you expect to achieve using these methods.* Testing is an inherently noisy and idiosyncratic process, and whether a suite detects a fault depends on a large number of complex variables. It would, given this complexity of process, be very surprising if any simple dynamic measure computable without human effort for any test suite produced strong correlations like those often shown between code coverage and mutation score (0.6-0.9). The correlations between these measures are often high because both result from regular, even-handed, automated analysis of the dynamics of a test suite. Actual faults are apparently (unsurprisingly) produced and detected by a much more complex and irregular process. However, when used to draw the line between less tested and more tested program elements, testedness measures can provide a simple automated way to prioritize testing effort, and recognize when all the elements of an SUT have passed beyond a high threshold of testedness, and are thus likely to have fewer future faults. In short, while we cannot (at present) measure testedness as precisely as we (software engineering researchers) would like, we can measure testedness in such a way as to provide some practical assistance to the humble working tester.

8. REFERENCES

[1] I. Ahmed, U. A. Mannan, R. Gopinath, and C. Jensen. An empirical study of design degradation: How software projects get worse over time. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 31–40, 2015.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering*, pages 402–411. IEEE, 2005.

[3] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8), 2006.

[4] Apache Software Foundation. Apache commons. http://commons.apache.org/.

[5] Apache Software Foundation. Apache maven project. http://maven.apache.org.

[6] A. Arcuri and L. C. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test., Verif. Reliab.*, 24(3):219–250, 2014.

[7] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM, 2009.

[8] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, USA, 1980.

[9] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[10] H. Coles. Pit mutation testing. http://pitest.org/.

[11] M. Daran and P. Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 158–171. ACM, 1996.

[12] M. Delahaye and L. Bousquet. Selecting a software engineering tool: lessons learnt from mutation analysis. *Software: Practice and Experience*, 2015.

[13] R. A. DeMillo and A. P. Mathur. On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software. Technical Report SERC-TR92-P, Software Engineering Research Center, Purdue University, West Lafayette, IN,, 1991.

[14] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

[15] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 153–162. ACM, 1998.

[16] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19:774–787, 1993.

[17] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.

[18] GitHub Inc. Software repository. http://www.github.com.

[19] M. Gligoric, A. Groce, C. Zhang, R. Sharma, A. Alipour, and D. Marinov. Guidelines for coverage-based comparisons of non-adequate test suites. *ACM Transactions on Software Engineering and Methodology*, 24(4):4–37, 2014.

[20] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2013.

[21] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *International Conference on Software Engineering*. IEEE, 2014.

[22] A. Groce, M. A. Alipour, and R. Gopinath. Coverage and its discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 255–268, New York, NY, USA, 2014. ACM.

[23] A. Gupta and P. Jalote. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *International Journal on Software Tools for Technology Transfer*, 10(2):145–160, 2008.

[24] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *International Conference on Software Engineering*, pages 191–200. IEEE Computer Society Press, 1994.

[25] L. Inozemtseva and R. Holmes. Coverage Is Not Strongly Correlated With Test Suite Effectiveness. In *International Conference on Software Engineering*, 2014.

[26] L. M. M. Inozemtseva. Predicting test suite effectiveness for java programs. Master's thesis, University of Waterloo, 2012.

[27] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 654–665, Hong Kong, China, 2014. ACM.

[28] S. Kakarla. An analysis of parameters influencing test suite effectiveness. Master's thesis, Texas Tech University, 2010.

[29] N. Li, U. Praphamontripong, and J. Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 220–229. IEEE, 2009.

[30] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1):9–31, 1994.

[31] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.

[32] A. Mockus, N. Nagappan, and T. T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 291–301. IEEE, 2009.

[33] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 57–68. ACM, 2009.

[34] A. S. Namin and S. Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 342–352, New York, NY, USA, 2011. ACM.

[35] A. J. Offutt and J. M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical report, Technical Report ISSE-TR-96-01, Information and Software Systems Engineering, George Mason University, 1996.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[37] RTCA Special Committee 167. Software considerations in airborne systems and equipment certification. Technical Report DO-1789B, RTCA, Inc., 1992.

[38] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 201–211, 2015.

[39] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[40] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing trade-offs in test-suite reduction. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 246–256, New York, NY, USA, 2014. ACM.

[41] SIR: Software-artifact Infrastructure Repository. Sir usage information , accessed at mar 8, 2016. http://sir.unl.edu/portal/usage.php.

[42] D. Tengeri, L. Vidacs, A. Beszedes, J. Jasz, G. Balogh, B. Vancsics, and T. Gyimóthy. Relating code coverage, mutation score and test suite reducibility to defect density,accepted paper. In *mutationworkshop*, 2016.

[43] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 386–396. IEEE, 2012.

[44] Y. Wei, B. Meyer, and M. Oriol. *Empirical Software Engineering and Verification*, chapter Is branch coverage a good measure of testing effectiveness?, pages 194–212. Springer-Verlag, Berlin, Heidelberg, 2012.