Code Coverage for Suite Evaluation by Developers

Rahul Gopinath
Oregon State University
gopinath@eecs.orst.edu

Carlos Jensen
Oregon State University
cjensen@eecs.orst.edu

Alex Groce
Oregon State University
agroce@gmail.com

ABSTRACT

One of the key concerns of developers testing code is how to determine a test suite's quality – its ability to find faults. The most common approach in industry is to use code coverage as a measure for test suite quality, and diminishing returns in coverage or high absolute coverage as a stopping rule. In testing research, suite quality is often evaluated by measuring a suite's ability to kill mutants, which are artificially seeded potential faults. Mutation testing is effective but expensive, thus seldom used by practitioners. Determining which criteria best predict mutation kills is therefore critical to practical estimation of test suite quality. Previous work has only used small sets of programs, and usually compares multiple suites for a single program. Practitioners, however, seldom compare suites — they evaluate one suite. Using suites (both manual and automatically generated) from a large set of real-world open-source projects shows that results for evaluation differ from those for suite-comparison: statement coverage (not block, branch, or path) predicts mutation kills best.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging Testing Tools

General Terms

Measurement, Verification

Keywords

test frameworks, evaluation of coverage criteria, statistical analysis

1. INTRODUCTION

The purpose of software testing is to improve the quality of software, and the primary route to this goal is the detection of faults. Unfortunately, the problem of finding all faults in a program (or proving their absence), for any meaningful program, is essentially unsolvable. Testing is therefore always a trade-off between the cost of (further) testing and the potential cost of undiscovered faults in a program. In order to make intelligent decisions about testing, developers need ways to evaluate their current testing efforts in terms of its ability to detect faults. The details of how to make use of such evaluation are likely to be highly project-dependent, but the ability, given a test suite, to predict whether it is effective at finding faults, is basic to most such approaches.

The *ideal* measure of fault detection is, naturally, fault detection. In retrospect, using the set of defects discovered during a software product's lifetime, the quality of a test suite could be evaluated by measuring its ability to detect those faults (faults never revealed in use might reasonably have little impact on testing decisions). Of course, this is not a practical method for making decisions during development and testing. Software engineers therefore rely on methods that predict fault detection capability based only on the suite itself and the current version of the software under test (SUT). The most popular such method is the use of code coverage criteria [1]. Code coverage describes structural aspects of the executions of an SUT performed by a test suite. For example, statement coverage indicates which statements in a program's source code were executed, branch coverage indicates which branches were taken, and path coverage describes (typically in a slightly more complex way, to account for loops) the paths explored in a program's control flow graph.

In software testing research, the gold standard for suite evaluation is generally considered to be actual faults detected, but this is, again, in practice difficult to apply even in a research setting [16]. The second most informative measure of suite quality is usually believed to be *mutation testing* [7, 2], which measures the ability of a test suite to detect small changes to the source code. Mutation testing subsumes many other code coverage criteria, and has been shown to predict actual fault detection better than other criteria in some settings, but never shown to be worse than traditional code coverage measures.

Unfortunately, mutation testing is both difficult to apply and computationally expensive, which has led to the search for "next-best" criteria for predicting suite quality by researchers [16, 19]. This effort is highly relevant to real software developers, who almost never apply mutation testing due to its complexity, expense, and the lack of tool support in many languages. From the point of view of actual software developers and test engineers, rather than researchers,

however, most studies of suite evaluation are not focusing on their actual needs.

First, researchers typically consider the effectiveness of criteria for predicting which of multiple suites for the same SUT will detect the most faults. For research purposes, this is the right focus: the primary use of coverage criteria is to compare test-generation methods, where the fundamental question is typically "Which of the following approaches should we apply to this program to detect the most faults?" In typical development settings, however, the question is more often "Should we devote more effort to improving this test suite?" and applying completely different testing methods is not an option. In fact, the suites being evaluated are usually produced by manual effort and the only mitigation for poor effectiveness considered is adding more manually developed tests. Testing research often focuses on comparing suites for one SUT; practitioners more often need to simply evaluate a single suite. It is not clear that the best criteria for multi-suite comparison, where suites are expected to be generated by competitive automated methods, are the most effective for evaluating a fixed suite.

Second, researchers are far more likely to apply novel coverage criteria than practitioners. Many recent papers comparing criteria either introduce a novel criteron [19] or implement a previously proposed but never-implemented criteria [16]. While most even moderately popular languages (e.g., Java, C, C++, Python, Haskell) have (multiple) tools for measuring statement and branch coverage, relatively few non-research tools even offer any variant of path coverage, much less data flow-based [20, 19] or more esoteric measures such as predicate-complete test coverage [5, 6]. This limitation is particularly important for open source development, where expensive commercial coverage tools are unlikely to be applied, and developers working together may lack a common high-end development environment. Even criteria as widely known as the MC/DC coverage required in aerospace code development [9] lack open source or free tools.

This paper examines the question of coverage criteria as suite quality predictors from the perspective of the non-researcher developer audience, interested in suite evaluation (rather than comparison) and relying on lightweight, widely available, tools and well-known coverage criteria. Given the constraints under which real software projects operate, which widely available coverage criteria provide the best estimation of fault detection? The data in this paper is drawn from the evaluation of hundreds of open source projects. While the results are based only on open source Java programs hosted on Github using the popular Maven build system, it is likely that they apply at minimum to many other Java projects using other build systems, and may well apply to other languages as well. As a "silver standard" for evaluating suite quality, mutation testing is used, as identifying real faults of hundreds of Java projects was clearly infeasible. Our findings show that, in contrast to the results of some studies in a research context, statement coverage is generally the most effective predictor of suite quality. This is not an accident of the nature of manually-produced test suites; the same relationship also holds for test suites generated by the Randoop tool [31], which uses feedback-directed random testing to generate suites. While branch coverage or some variant of path coverage may be most useful for many research contexts, in the context of typical Java open source projects, at least, a focus on the simple and easily understood measure of statement coverage is probably most useful for predicting suite quality, even if developers are using an automated testing tool.

The primary contributions of this paper are twofold. First, the existence of popular open source repositories makes it possible to investigate the effectiveness of coverage criteria in a more unbiased, large-scale, and systematic way than previous studies, using a large body of very different SUTs so as to limit selection effects. Using the actual test suites from typical real projects also ensures that results are relevant to actual testing practices as seen "in the wild." The additional availability of automated testing tools mature enough to apply to this set of projects enables us to draw conclusions about both human-generated and automatically-generated suites, and to show that results do not depend on this property of test suites. This enables our second contribution, a practical proposal to developers wishing to evaluate test suites for open source Java projects. Finally, by showing that the best criteria for research purposes differ from those for practitioners, this paper may indicate that the particular needs and abilities of software testing researchers may lead to less-than-optimal advice to developers whose focus is not on evaluating testing methods but on producing quality software.

M	Mutation score
S	Statement coverage
$\tilde{S}$	Block coverage
B	Branch coverage
P	Path coverage
K	Project size in LOC
T	Test suite size in LOC
C	Cyclomatic complexity

Table 1: Symbols used

2. RELATED WORK

A very large body of work considers the question of the relationship between coverage criteria and fault detection. The most closely related work to ours, which considers some of the same questions from a different perspective (that of researchers) is the recent work of Gligoric et al. [16]. Their work uses the same statistical approach as this paper, measuring both τ_β and R^2 to examine correlation to mutation kill for a set of criteria, and both studies consider realistically non-adequate suites. However, their work considers only a set of 15 Java programs and 11 C programs, selected not randomly but primarily from container classes used in previous studies and the classic Siemens/SIR subjects. Their larger projects (JodaTime, JFreeChart, SQLLite, YAFFS2) were chosen opportunistically. Our study is on a much larger scale in terms of subjects and uses a more principled selection process. Most importantly, however, we consider correlation of criteria across all SUTs, to answer the question "given a suite for an SUT, which criteria best predicts mutation kills for that SUT?" rather than to determine, within each SUT, which criteria best ranks various suites for that SUT. Gligoric et al. report 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/comparison context. Inozemtseva et al. [21] investigates the relationship of

various coverage measures and mutation score for different random subsets of test suites. They observe that when the test suite size is controlled, only low to moderate correlation is present between coverage and effectiveness. This conclusion holds for all kinds of coverage measures used. The difference in subjects and focus yields substantially different results than ours, as we discuss below.

Budd et al. [7] proposed mutation testing as a stronger criteria than other methods for evaluating test suites. Offut et al. showed that mutation coverage subsumes [29] many other criteria, including the basic six proposed by Myers [27].

Frankl et al. [13] compared the effectiveness of mutation testing with all-uses coverage, and found that at highest coverage levels, mutation testing was more effective. Andrews et al. compared [2] the fault detection ratio and the mutation kill ratio of a large number of test suites, finding the ratios were very similar, and hence the faults induced by mutation representative of the real faults in programs. 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 the fault detection ratio are related linearly, with similar results for other coverage criteria (0.83 to 0.9). Linear regression on the mutation kill ratio and fault detection ratio showed a high correlation (0.9). Li et al. [24] compared four different criteria (mutation, edge pair, all uses, and prime path). They found that mutation-adequate testing was able to detect the most hand seeded faults (85%), while other criteria were similar to each other and were 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 in the literature.

Frankl and Weiss [12] performed a comparison of branch coverage and def-use coverage, showing that def-use is more effective than branch coverage for fault detection and there is stronger correlation to fault detection for def-use than branch coverage.

Gupta et al. [18] 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 (average number of test cases required to detect a fault) of suites.

Kakarla [23] and Inozemtseva [22] 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 by Kendall's τ 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 $0.57 \leq \tau \leq 0.94$.

Wei et al. [35] 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. Early in random testing, where branch coverage was rising 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.

Cai et al. [8] investigated correlation 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 observed that the relationship between block coverage and mutant kills was not always positive. They found that 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).

Namin and Andrews [28] also 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.

In general, none of this work considered a large, representative, set of open source projects, and many studies considered the within-SUT suite comparison problem, not the problem of determining if a single suite provides effective testing for an SUT, as we do. The variety of reported rankings and correlations of criteria can be highly confusing to even a researcher wishing to compare suites, much less a typical (open source) developer seeking to simply decide if current testing for a project is likely effective for fault detection. Many studies do not even include all of branch, statement, and block coverage, the most readily available criteria. Our contribution over related work is a study that (1) uses a large set of open-source projects, (2) uses both manually and automatically generated tests, (3) includes all the criteria of most interest to developers, and (4) focuses on the critical question of single-suite evaluation correlation.

3. METHODOLOGY

Methodology was driven by two primary concerns. We wanted our results to be applicable to the largest set of real-world programs as possible, and should be based on a diverse set of actual test suites constructed by developers, not testing researchers. The second was to strive for a statistically significant result, preferring to keep as many experimental variables constant as possible. One result of this was to restrict this study to Java programs. Java is one of the most widely used programming languages [14, 34], and choosing a single language allows us to ensure a consistent definition for coverage criteria and avoid any difficulties due to variance in mutation operators. As a consequence, our results are only directly applicable to projects written in Java (still a large portion of the code written today). However, the results are also likely applicable to other programming languages with similar structures. Previous studies [16] do not show major differences between criteria effectiveness between Java and C programs, despite Java's object-oriented nature, inclusion

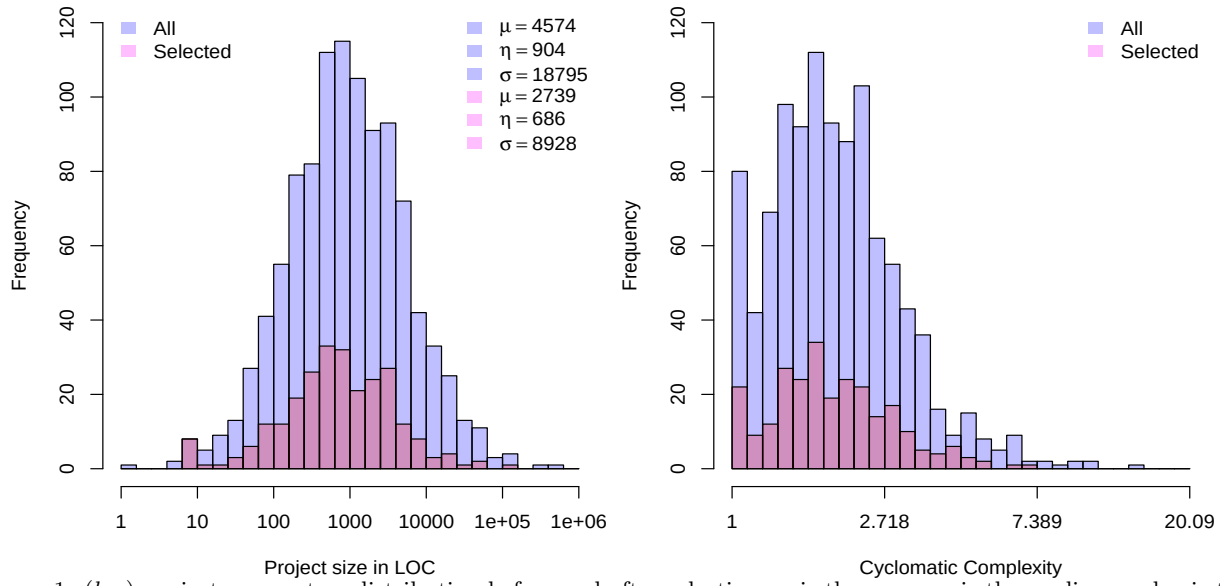

Figure 1: (*log*) project parameters distribution before and after selection. μ is the mean, η is the median, and σ is the standard deviation of the selected parameter.

of exceptions, the different kinds of programs that tend to be written in C and Java, etc. Inferring criteria effectiveness for projects not based on C-like languages such as Java, C, C++, and C# — e.g. for functional languages — would be less justified.

Projects were taken from Github [15], one of the largest public repositories of Java projects. As a concession to ease of analysis, only projects using the dominant Maven [4] build system were considered. Github provided an initial set of 1,733 projects meeting this criterion. While far from the entire set of Java projects hosted by Github, there is no reason it should be biased in terms of test suites. Note that it is also not the full set of Maven projects, since Github only returns 99 pages of search results. The process used by Github to select projects is not public, but we believe it is orthogonal to our concerns, and likely based on popularity and recency. After eliminating projects aggregating multiple projects (which are difficult to properly analyze), a set of 1,254 projects remained.

In order to ensure that our results remain free of systematic biases, we conducted our analysis in two phases. In the first phase, test suites present in the projects (mostly manually produced, though there may have been some automatically generated tests included) were used for coverage and mutation analysis. In the second phase, we generated test cases using Randoop [30], and performed the same statistical analysis using these suites. Finally, we compared the results of our first phase to that of the corresponding analysis in the second phase.

Our aim in conducting this cross-validation was to ensure that our results would not be affected by possible bias in manually generated suites. Automatically generating suites allowed us to have a second, independent measure.

In both phases, we gathered coverage metrics reported from Emma [32] (statement coverage), Cobertura [11] (statement coverage, branch coverage), CodeCover [33] (statement coverage, branch coverage), JMockit [25] (statement coverage, path coverage), and PIT [10] (mutation kills). PIT is a tool aimed at developers rather than researchers, actively developed and supported, with some penetration in open source testing. While it is not explicitly mentioned in the JMockit project page, the path coverage provided by JMockit is similar to the Acyclic Intra Method Path (AIMP) coverage that performed well in Gligoric et al. [16].

Out of our projects, only 729 projects had test suites. These were selected for the first phase of analysis. The analysis included running each of the coverage tools over the selected projects, with some effort expended for fixing trivial errors. In the places where the compilation did not succeed even after some effort, we discarded the project. Further, we specified a maximum timeout of 1 hour per project for a single tool. In the end, using original test cases we had 318 results from Cobertura, 286 results from Emma, 253 results from CodeCover, 361 from JMockit, and 259 from PIT.

As an example of the selection process, consider mutation testing. Starting from 729 projects, 273 had compilation errors, dependency resolution problems, language version problems, or other fundamental issues preventing a build. Of the remaining projects, 37 timed out, requiring more than an hour for mutation testing with PIT. An additional 102 projects had test failures that prevented PIT from running, and PIT failed to produce any output for 39 projects, possibly due to no coverage or mutants produced, even though the build completed successfully.

For the second phase of analysis, we used Randoop, a feedback-directed random testing tool, on each of the projects, and discarded those where it failed to complete successfully or timed out. This produced test suites for 437 projects. From these suites, following the previous procedure, we obtained 314 results from Emma, 323 results from Cobertura, 287 results from CodeCover, 329 results from JMockit, and 243 results from PIT.

From the 437 projects for which we were able to generate Randoop suites, 66 again had compilation, etc. errors preventing analysis. Of the remaining, 4 projects timed out after over an hour with PIT, and 84 had test failures that prevented PIT from running. PIT failed to produce output for 51 of the Randoop suites.

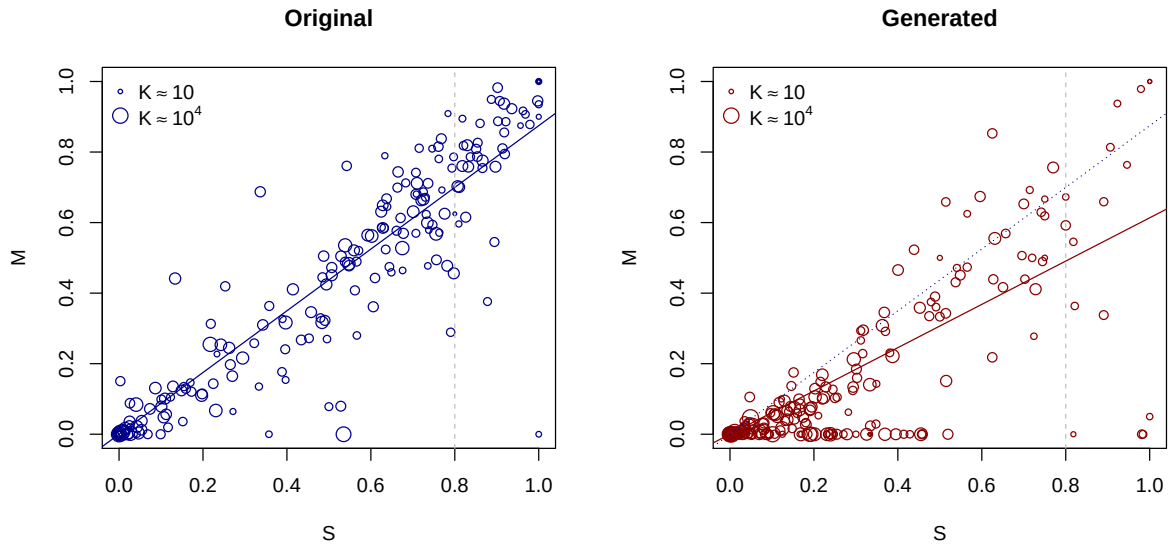

Figure 2: Relation between Statement Coverage and Mutation Kills. The circle represents the magnitude of project size.

The symbols used to indicate various metrics collected are given in Table 1. The use of $\tilde{S}$ for block coverage is motivated by the observation that block coverage is a kind of weighted statement coverage; given source code and a CFG, block coverage can be computed given statement coverage details and vice-versa.

Since the coverage and mutation process resulted in a rather drastic reduction in sample space, we compared the distributions for code size and complexity before and after selection to verify that our procedure did not inordinately skew the sample space in at least these dimensions. The size distribution histograms for both before and after selection is provided in Figure 1.

Another important dimension in which a bias could appear is the complexity of programs; perhaps rejection is much more common with more complex or simpler programs, which could bias results, since coverage metrics are intimately tied to code complexity (for very simple programs, e.g., statement and path coverage are quite similar). We measured McCabe cyclomatic complexity [26] which provides a measurement of program complexity by counting the number of linearly independent execution paths through a program. The distributions before and after selections are given in Figure 1. These graphs suggest that selection did not unduly bias the sample in these two key dimensions.

To account for effects of nondeterminism, we ran each coverage measurement 10 times, and computed the average. We also made use of multiple tools, as noted above, to verify that the coverage reported was accurate — e.g., Emma, Cobertura, CodeCover and JMockit all give statement coverage. Further, Cobertura and CodeCover provided branch coverage, and JMockit provided path coverage. Thus, we could compare most coverages provided by most tools and ensure they had high correlation to other tools. Further, we could ensure that the tools were processing all classes and test cases by examining statement coverage results. This was important because early on, we found that JMockit was not including classes that were not covered by any tests in its calculation of coverage[1]. Further, we have also removed a few observations (11 in original suites, 14 in Randoop suites) where the statement coverage reported by other tools was zero, and mutation or path coverage was non-zero, as these indicate some incorrect value from some tool. Our dataset, which includes metrics for all projects before elimination of outliers, is available for perusal in Dataverse [17].

4. ANALYSIS

The purpose of our analysis is to determine which coverage criteria that are likely to be used by real-world developers best predict mutation kill ratios. Our analysis also considers project and test suite size and cyclomatic complexity to determine if these factors affect the utility of coverage criteria.

The scatter-plots for mutation kills and statement coverage for both original test suites and Randoop-generated test suites are shown in Figure 2. It contains 232 pairs for original test suites, and 217 pairs for generated test suites. Similarly, the scatter-plots for mutation kills and branch coverage for both original test suites and generated suites is given in Figure 3, with 223 pairs for original test suites, and 191 pairs for generated test suites. The scatter-plots for path coverage in Figure 4 contain 214 pairs for original test suites, and 183 pairs for generated test suites. Finally, the scatter plots between *block* coverage and mutation kills are given in Figure 5. The diameter of the circles in all scatter plots correspond to the magnitudes of the project sizes ($log(K)$). The central result of these experiments is generally visible in these plots: statement coverage appears to give the best prediction of mutation kills of all criteria developers are likely to use, and this holds for both original and generated test suites.

We use regression analysis and significance testing to ascertain the contribution of different factors to test suite effectiveness. The correlation coefficient R^2 indicates the effectiveness of a model, i.e how much of the variation found in data is explainable by the parameters of the model. The

[1]http://code.google.com/p/jmockit/issues/detail?id=305

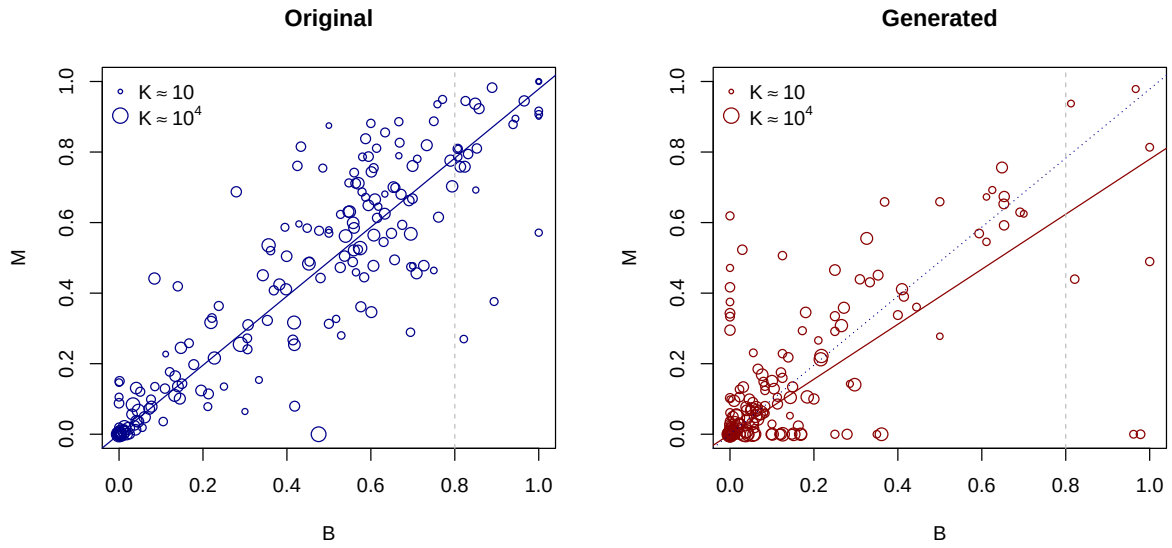

Figure 3: Relation between Branch Coverage and Mutation Kills. The circle represents the magnitude of project size.

factors that were found insignificant were eliminated to obtain reduced models.

4.1 Statement Coverage and Mutation Kills

In this section, we try to find the significant factors that in combination with statement coverage, are able to predict mutation kills to a high degree of confidence.

We begin with the saturated model consisting of all variables. These are mutation score, project size in LOC, test suite size, cyclomatic complexity, and statement coverage. We removed the test suite size to avoid multicollinearity with project size after noticing that it correlated very strongly with the project size. Performing the same analysis with test suite size in place of project size gave the same results as below, except that the weak effect for branch coverage become a stronger effect (but for other criteria, suite size did not matter). This gives the regression relation:

$$\mu\{M|K,C,S\} = \beta_0 + \beta_1 \times log(K) + \beta_2 \times C + \beta_3 \times S$$

β_0 was set to zero since we had sufficient coverage data near the zero point, and zero statement coverage should indicate zero mutation coverage too. This is given in Table 2

	Estimate	Std. Error	t value	Pr(>|t|)
$log(K)$	0.00	0.01	0.15	0.88
$log(C)$	-0.01	0.02	-0.64	0.52
S	0.88	0.02	40.88	0.00

Table 2: Saturated model (Original, Statement)

Further, we noticed that project size itself did not have a significant contribution to the response variable. Once we removed project size, our table was updated to Table 3

	Estimate	Std. Error	t value	Pr(>|t|)
$log(C)$	-0.01	0.01	-0.86	0.39
S	0.89	0.02	45.09	0.00

Table 3: Second model

Since cyclomatic complexity was also clearly not significant, removing it resulted in the equation

$$\mu\{M|S\} = 0 + \beta_1 \times S$$

and the result of this equation is in Table 4

	Estimate	Std. Error	t value	Pr(>|t|)
S	0.87	0.01	59.88	0.00

Table 4: Original: Mutation × Statement R^2= 0.9392

There is no significant effect of project size or program complexity on mutation coverage in a model based on statement coverage.

4.2 Branch Coverage and Mutation Score

In this analysis, we follow the same path we took for statement coverage with mutation coverage. Project size had (very) weak evidence of significance at $p = 0.0672$ when compared to statement coverage (the effect for suite size here was stronger at 0.0015). β_0 was set to zero since we had sufficient coverage data near the zero point, and zero branch coverage, again, should indicate zero mutation kills.

$$\mu\{M|B\} = 0 + \beta_1 \times B$$

	Estimate	Std. Error	t value	Pr(>|t|)
B	0.98	0.02	51.76	0.00

Table 5: Original: Mutation X Branch R^2= 0.9231

4.3 Path Coverage and Mutation Score

Following the same analysis steps, we removed code size and cyclomatic complexity as they were not significant. β_0 is again set to zero for the same reasons.

$$\mu\{M|P\} = 0 + \beta_1 \times P$$

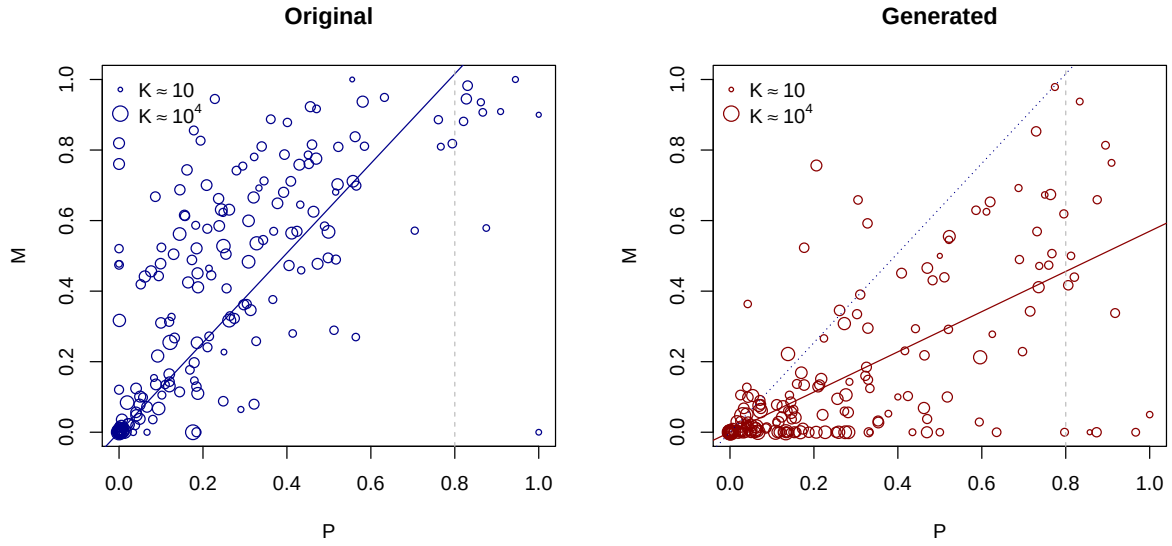

Figure 4: Relation between Path Coverage and Mutation Kills. The circle represents the magnitude of project size.

	Estimate	Std. Error	t value	Pr(>|t|)
P	1.27	0.05	25.33	0.00

Table 6: Original: Mutation X Path R^2= 0.7496

4.4 Comparing the Criteria

After determining that project size, suite size, and cyclomatic complexity were essentially ignorable for our purposes, we turned to comparing correlation statistics for all criteria, for both original and Randoop-generated tests. In keeping with the most recent and extensive studies [16] we report both R^2 and Kendall τ_β correlations. R^2 in our context is the most useful correlation measure, since ideally developers would like to predict the actual mutation killing effectiveness of a test suite. Kendall τ_β is a rank-correlation statistic that is non-parametric, and therefore should be reliable even if underlying relationships are not linear — in our context, it roughly answers the question: given that the ranking between two coverage criteria for suites for projects is such that C(X) > C(Y), what is the chance that the ranking of mutation kills is in agreement with this ranking?

The results from computing R^2_{adj} and τ_β for each of the coverage metrics with mutation coverage are given in Table 7. (O) indicates values for original test suites, and (R) indicates Randoop-generated suites.

	R^2(O)	τ_β(O)	R^2(R)	τ_β(R)
$M \times S$	0.94	0.82	0.72	0.54
$M \times \tilde{S}$	0.93	0.74	0.69	0.48
$M \times B$	0.92	0.77	0.65	0.52
$M \times P$	0.75	0.67	0.62	0.49

Table 7: Correlation coefficients (Mutation), $p < 0.001$

The results are clear: across both original and generated suites, statement coverage almost always has the best correlation for both R^2 and τ_β. For predicting mutation kills for test suites included with projects, branch, statement, and block coverage all provide a satisfactory method; predictions for Randoop-generated suites are more difficult, but statement coverage still performs relatively well, with sufficient power to be useful in practice. Path coverage performs poorly on project suites, but was the best method for randomly generated suites, a surprising result; however, its improvement over statement coverage was relatively small, and it does not seem to be nearly as reliable for manually produced suites.

5. WHY STATEMENT COVERAGE?

Some previous research (e.g., as recently as 2013 [16]) suggests the use of branch coverage as the best method for predicting suite quality. However, this study was conducted on a small set of programs, a majority of which were algorithms and data-structure implementations, and based on comparing suites for the same SUT rather than predicting the quality of testing for each SUT in isolation. Moreover, all previous studies tend to include somewhat artificial suites produced by testing researchers, rather than focusing on real developer-produced test suites for a large variety of projects.

That statement coverage performs so well in fact agrees with the conclusion of Gligoric et al. [16] that "for *non-adequate suites*, criteria that are stronger (in terms of subsumption for *adequate suites*) do *not necessarily* have better ability to predict mutation scores." The superiority of statement to branch coverage, however, requires some further examination. Figure 8 shows a simple portion of a CFG that may explain this result. A test suite that covers either of branches A or B would result in branch coverage of 50%. However, there are more mutations of branch B than branch A, and (under the assumptions that guide mutation testing) more chances for coding errors.

Modeling the potential impact of the fact that of the popular coverages only statement coverage takes into account the size of a code block predicts the observed results. Assume there are n lines of code in a SUT, and let $\mu(S_i)$ be the mutability (number of mutants) of the i^{th} statement. For the SUT there are then $\sum_{i=1}^{n} \mu(S_i)$ mutants. This linear relationship is also suggested by our data which shows a high

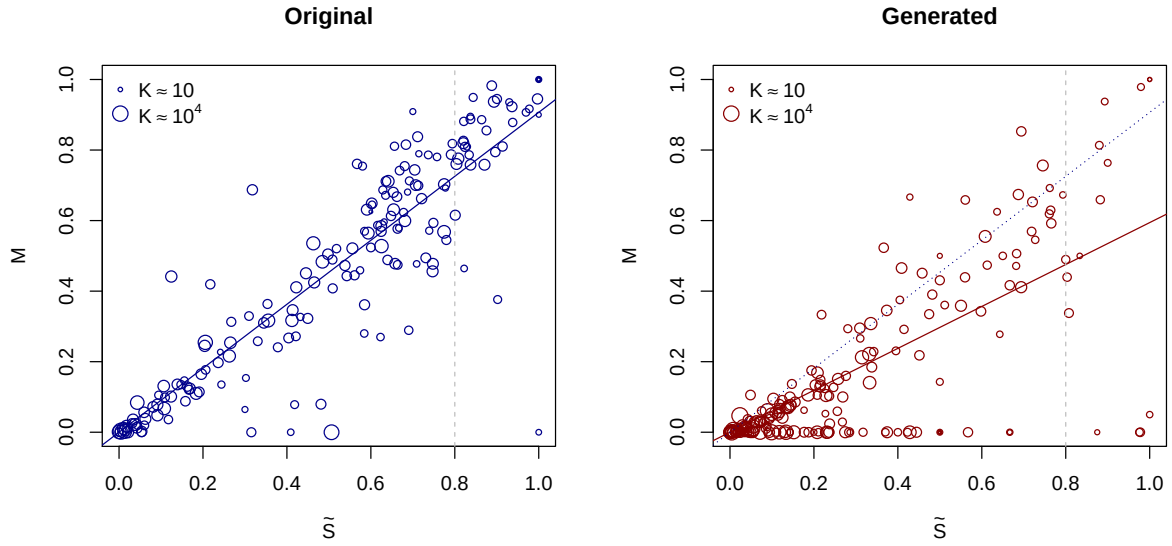

Figure 5: Relation between Block Coverage and Mutation Kills. The circle represents the magnitude of project size.

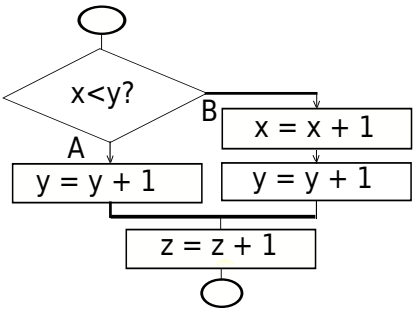

Figure 8: Unbalanced branching

correlation – $R^2 = 0.96$ between number of statements and number of mutants produced. If we assume that a mutation is detected every time a test suite covers the statement , and we have a constant mutability k, then we can see that of the $n \times k$ mutants produced, $n \times c \times k$ would be detected by a test suite with c as statement coverage ratio. Under ideal conditions, mutation kills and statement coverage share a simple relationship.

This formulation also suggests that branch coverage alone could not be as closely correlated with mutation kills as statement coverage unless a model includes some way to incorporate the difference in mutability of program segments, or the assumption that coverage usually results in detection is very far from reality, and branch execution is a major factor in actual detection ratios for most mutants. For the first possibility we draw the readers attention to the fact that there was a weak project size effect and fairly strong suite size effect when we considered branch coverage models.

This also suggests that if we consider basic block coverage, which is basically statement coverage without mutability information, the correlation should also be lower than the coverage reported by ordinary statement coverage. This is again borne out by the lower values of R^2 and τ_β for block coverage ($M \times B$) in Table 7. Why, though, is block coverage sometimes better correlated than branch coverage, when both criteria ignore mutability of code segments? Branch coverage can "compensate" for missing a block (which always contains at least one mutable statement) by taking a branch that contains *no* mutable code. In fact, "missing else" detection is what distinguishes block and branch coverage.

5.1 Statement Coverage and Path Coverage

One result undercutting this simple explanation for the superiority of statement coverage, however, is that statement coverage also better predicts *path* coverage than branch coverage.

Table 8 shows correlations between path coverage, and statement, branch, and block coverage. Scatter plots of statement coverage and branch coverage against path coverage are provided in Figure 6 and Figure 7.

These support the conclusion that statement coverage is better than branch for this purpose also (the winner between block and statement coverage is less obvious, since block coverage performs better for generated suites).

	R^2(O)	τ_β(O)	R^2(R)	τ_β(R)
$P \times S$	0.81	0.68	0.84	0.65
$P \times \tilde{S}$	0.80	0.59	0.87	0.67
$P \times B$	0.80	0.65	0.59	0.45

Table 8: Correlation coefficients (Path), $p < 0.001$

We do not have any explanation for this effect at this time, since path coverage, like branch and block coverage, ignores the size of code blocks. It is possible that executing more statements leads to producing more unusual execution states, which results in more covered paths, but this is hard to model or investigate.

5.2 Correlation at High Coverage Levels

An additional interesting question to consider is whether the superiority of statement coverage for our purposes is an

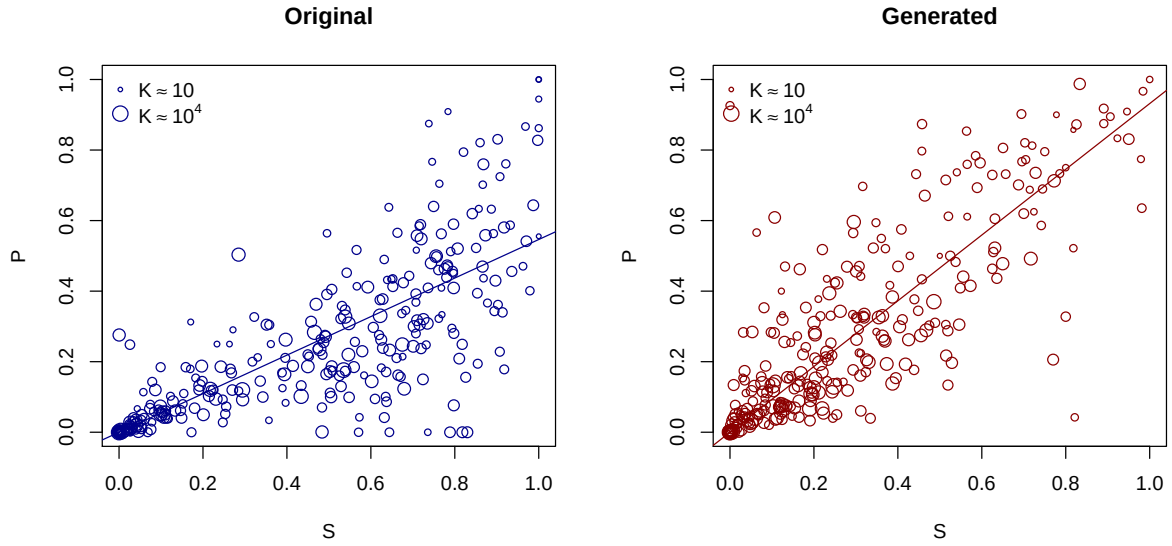

Figure 6: Relation between Statement Coverage and Path Coverage. The circle represents the magnitude of project size.

artifact of the fairly weak test suites for many SUTs. While statement coverage is most predictive for projects, perhaps its actual utility is lower than expected, in that it works best for suites where the question "Is the testing effective for finding faults?" is fairly obviously answered with "no." Perhaps the better results for branch coverage in previous studies are due to considering mostly effective test suites. In order to examine this possibility, we computed correlations for only those suites with at least 80% statement coverage, a reasonable threshold for "good" testing. We have less confidence in our results for this requirement, as only 41 SUTs have original suites with this level of coverage, and only 14 Randoop-generated suites manage it; however, this is larger than the set of subjects in many previous studies. Table 9 shows that even at high levels of statement coverage, for the original suites, statement coverage is a better predictor of mutation kills than branch coverage. In fact, statement coverage's R^2 value is slightly increased for the original suites. The ability to rank projects in terms of testing effectiveness, however, is (unsurprisingly) considerably diminished. and block coverage becomes a better predictor in this sense, but statement coverage remains better than branch coverage.

Table 10 shows correlations when the requirement is changed to branch coverage being at least 80%, which yields only 23 original suites and 7 Randoop-generated suites. Interestingly, in this case statement coverage has a very high R^2 for the original suites, and branch coverage has a negative rank correlation for Randoop suites. On the whole, given the small number of SUTs meeting strict coverage requirements, the claim that statement coverage is more useful in predicting quality than branch and block coverage, at least for original test suites, does not seem to depend on low coverage. Our scatter plots show a vertical line at 80% coverage to aid in visualizing the correlations at high coverage levels. We also show the best-fit lines for manual test suites in the Randoop test suite scatter-plots as dashed lines for easy comparison.

6. THREATS TO VALIDITY

	R^2(O)	τ_β(O)	R^2(R)	τ_β(R)
$M \times S$	0.95	0.46	0.64	0.30
$M \times \tilde{S}$	0.93	0.51	0.69	0.30
$M \times B$	0.92	0.36	0.65	0.11
$M \times P$	0.75	0.31	0.62	-0.17

Table 9: Correlation coefficients (S > 0.8)

	R^2(O)	τ_β(O)	R^2(R)	τ_β(R)
$M \times S$	0.98	0.53	0.58	0.00
$M \times \tilde{S}$	0.93	0.54	0.69	0.10
$M \times B$	0.92	0.33	0.65	-0.05
$M \times P$	0.75	0.25	0.62	0.00

Table 10: Correlation coefficients (B > 0.8)

Our research findings may be subject to the concerns that we list below. While we have taken all steps possible to be free of the impacts listed, it is possible that our mitigation strategies may not have been effective.

One of the concessions we were forced to make while measuring various coverage ratios was to restrict the amount of time each test suite was allowed to run to one hour. While this did not result in a significant elimination of projects, there is a possibility that it may have biased us against larger or more complex projects with extremely large and thorough test suites.

We selected only those projects that compiled and passed all tests. This may bias us against small one-time projects where the authors may not have sufficient time to do a thorough job of testing, or even against projects with active development, where some tests represent open bugs.

Our findings are dependent on the metrics reported by our tools. We have taken a number of steps, including cross verification and multiple runs, to ensure that we are not led astray by erroneous results. However, we rely on a single mutation testing tool, which is central to our results. Path coverage results are also somewhat less definite, as JMockit

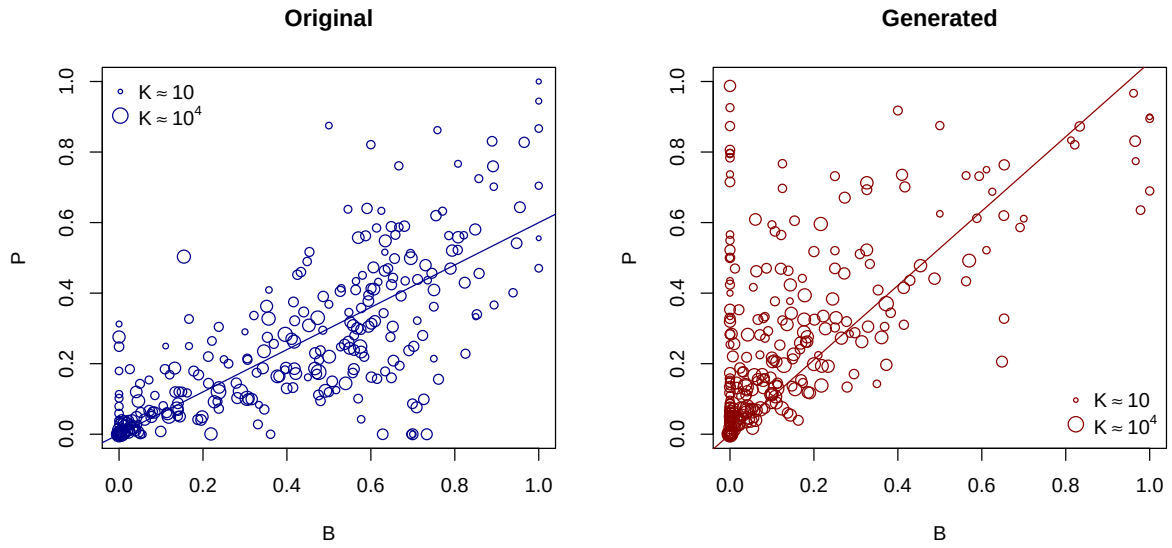

Figure 7: Relation between Branch Coverage and Path Coverage. The circle represents the magnitude of project size.

is less mature than the other tools, and we do not have a second path coverage tool to cross-validate results. However, hand examination of results on multiple simple examples seems to indicate the path coverage reported is correct, and path coverage is fairly trivial to produce, given the ability to produce other coverages.

Finally, our findings are restricted to projects using the Java language, using the Maven framework, and standard project layout. Further, the samples came from a single repository (Github), and are all open source projects. While none of these factors are obviously confounding, they may limit applicability of results in radically different settings, e.g. Haskell development or a commercial software project with an extremely large user base and a dedicated QA team operating independently of developers.

7. CONCLUSION

Mutation testing is one of the best predictors of test suite quality, in terms of ability to detect actual faults. However, it is also computationally expensive to run, complex to apply, and is generally not used by real-world developers with any frequency. A long-term goal of the testing community has therefore been to develop alternative methods for predicting suite quality, for software development purposes and (perhaps primarily) for use in evaluating competing testing techniques. Unfortunately, the large body of previous studies on this topic have largely considered only a small set of programs, selected opportunistically, sometimes focused on coverage criteria used as rarely as mutation testing in real-world projects, and often been constructed around the question of predicting the best among multiple suites for a single SUT. In reality, software developers seldom have the luxury of applying esoteric coverage criteria or choosing between competing test suites. Rather, given an existing test suite, they want to estimate whether that suite is likely effective at detecting faults, or if more testing effort may be justified given the cost of faults.

This paper reports the correlation between lightweight, nearly universally available coverage criteria (statement, block, branch, and path coverage) and mutation kills for hundreds of Java programs, for both the actual test suites included with those projects and suites generated by the Randoop testing tool. For both original and generated suites, statement coverage is the best predictor for mutation kills, and in fact does a relatively good ($R^2 = 0.94$ for original tests and 0.72 for generated tests) job of predicting suite quality. SUT size, code complexity, and suite size did not turn out to be important factors in our analysis. A simple model of mutation and mutation detection predicts the higher effectiveness of statement coverage, but does not explain why statement coverage even predicts path coverage better than branch coverage does, a highly counter-intuitive result. Moreover, while block coverage became more competitive at high statement coverage levels, statement coverage still appeared to be the best method for evaluating high-coverage (80%+ statement coverage) suites, with an R^2 of 0.98 for suites with branch coverage $\geq 80\%$.

The lesson for software developers is somewhat comforting: statement coverage is the most widely available and easily interpreted coverage criteria, and is also probably the best coverage criteria for predicting test suite quality in their context, over testing effort lifetime. The lesson for software testing researchers is that the question of how coverage correlates to suite effectiveness likely has no single correct answer, but must be answered with careful attention to the context of application, and the selection of a proper population of subjects and suites to examine.

8. REFERENCES

[1] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th 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):608–624, 2006.

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

[5] T. Ball. A theory of predicate-complete test coverage and generation, 2004. Technical report, Microsoft Research Technical Report MSR-TR-2004-28.

[6] T. Ball. A theory of predicate-complete test coverage and generation. In *Formal Methods for Components and Objects*, pages 1–22. Springer, 2005.

[7] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward. *Mutation analysis*. Yale University, Department of Computer Science, 1979.

[8] 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.

[9] J. J. Chilenski. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical report, DTIC Document, 2001.

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

[11] M. Doliner and Others. Cobertura - a code coverage utility for java. http://cobertura.github.io/cobertura.

[12] 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.

[13] 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.

[14] GitHub Inc. Github languages. http://www.github.com/languages.

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

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

[17] R. Gopinath. Replication data for: Code coverage for suite evaluation by developers. In *http://dx.doi.org/10.7910/DVN/24574*. Harvard Dataverse Network V1, 2014-01.

[18] 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.

[19] M. M. Hassan and J. H. Andrews. Comparing multi-point stride coverage and dataflow coverage. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 172–181. IEEE Press, 2013.

[20] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200. IEEE Computer Society Press, 1994.

[21] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 2014 International Conference on Software Engineering*, 2014.

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

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

[24] 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, 2009. ICSTW'09.*, pages 220–229. IEEE, 2009.

[25] R. Liesenfeld. JMockit - A developer testing toolkit for Java. http://code.google.com/p/jmockit/.

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

[27] G. J. Myers. The art of software testing. *A Willy-Interscience Pub*, 1979.

[28] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 57–68. ACM, 2009.

[29] A. J. Offutt and J. M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical report, 1996.

[30] C. Pacheco and M. D. Ernst. Randoop random test generation. http://code.google.com/p/randoop.

[31] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84. IEEE, 2007.

[32] V. Roubtsov and Others. Emma - a free java code coverage tool. http://emma.sourceforge.net/.

[33] R. Schmidberger and Others. Codecover - an open-source glass-box testing tool. http://codecover.org/.

[34] TIOBE. Tiobe index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

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