### THE EFFECTS OF TEST REDUNDANCY ON THE SYSTEM UNDER TEST

Mario Gonzalez Macedo

### Ortask

mgm@ortask.com

#### ABSTRACT

This paper presents results of new research that shows a link between test redundancy and bugs in their associated SUT¹. It also presents statistically significant evidence that test suites have 25% redundancy on average. Additionally, it provides evidence that test redundancy is a good supplementary metric to assess test quality, along with an equation that closely estimates the amount of bugs for each project.

# **Categories and Subject Descriptors**

D.2.5 [Software Engineering]: Testing and Debugging—Testing Tools; D.2.8 [Software Engineering]: Metrics—product metrics, process metrics

# **General Terms**

Algorithms, Experimentation, Management, Measurement, Reliability, Theory

# **Keywords**

Unit testing, quality metrics, code coverage, empirical software engineering, test redundancy, software faults

## 1. INTRODUCTION

Extensive research has focused on test redundancy detection and suite minimization from the perspective of the costs associated to the testing process [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. However, there has been minimal research done about the effects of test redundancy on the systems whose quality they assess. In other words, the motivation behind most research about test redundancy and suite minimization is based on improving the efficiency of the testing process by decreasing maintenance, runtime and energy costs of tests. Even though such motivation is important, no known research exists to investigate whether test redundancy affects the SUT in terms of quality and bugs. [12] is one of the few studies so far to consider test redundancy with respect to bugs in software, although it does so in an auxiliary, rather than centric, manner. This implies that its results on test redundancy cannot be generalized to other projects due to several limitations in the methodology used.

There are several reasons for the lack of research of this kind (i.e. research focused on bugs in the SUT as effected by test redundancy of their associated test suites). The mayor reason suspected by this author is due to the fact that the methods discussed in the aforementioned studies unfortunately do not scale well and, thus, cannot be used in real projects. In other words, the algorithms, mechanisms and methodologies introduced by the studies above

are themselves generally too costly or disruptive to implement, execute and maintain. For example, [6, 10] requires repetitively executing the test suites of the SUT in their entirety while [1, 2, 3, 7, 8, 9, 10, 11] also requires instrumenting the SUT. Furthermore, [2] proposes instrumenting both the SUT and the tests themselves. Keep in mind that the effort implied by the requirements above is simply to begin assessing the redundancy of the tests. Depending on the requirement, such instrumentation naturally bloats the SUT or the test suites, which leads to more processing required for the test redundancy detection process. This, in turn, translates into slower test redundancy detection.

There are two important implications for the requirements in the aforementioned papers. On one hand, their requirements imply that the maintainers of test suites would need to remember to instrument new tests as they are added during the evolution of the SUT. On the other hand, developers would need to remember to do the same to code that is added or modified in the SUT. However, as the same studies indicate, merely executing such tests within the development and testing process can be costly. Yet, the proposed solutions call for the introduction of inefficiencies into the development and testing activities - inefficiencies that run counter to their objective - merely to be able to achieve results.

For such solutions to be adopted by industry, they must run in alignment with the development and testing processes while providing low (or zero, if possible) friction to their execution. Doing so will allow such solutions to be assimilated as best practices and incorporated into the development life-cycle<sup>2</sup> without much, if any, disruption. Once they have been assimilated as best practices, they will be able to yield the intended results of improving the quality of the tests as well as of their associated SUT. As such, the contributions in [13] show strong evidence that a lighter-weight solution is highly accurate and fast enough to be applicable to real projects.

Therefore, this paper both extends and relies on [13] to present new contributions that address the missing area of research regarding test redundancy. Namely, it relies on the mathematical model presented in [13] to study more than 50 test suites of different sizes, spanning 10 popular open-source projects, and shows a strong relationship between bugs and test redundancy. Moreover, it extends the findings presented in that paper to show that its proposed mathematical model yields a fast, scalable solution that is practical in industry and real projects.

Therefore, as far as this author is aware, this paper is the first known study to focus primarily on the effects of test redundancy on their associated SUT. An additional motivation of this paper is to investigate whether test redundancy can be utilized as a supplementary criterion to reliably assess the quality of tests and whether

 $<sup>^{1}</sup>$ System Under Test

<sup>&</sup>lt;sup>2</sup>Much like static analysis tools like BEAM have been incorporated into Continuous Integration processes that run automatically throughout the evolution of a project.

a test redundancy metric is beneficial to the testing process.

### 2. RESEARCH GOALS

There are two types of test redundancy one of which is the focus of this paper. The first type is *partial test coverage* (i.e. partial test redundancy) as defined in [13]. The other type of redundancy, and the focus of this paper, is *total test coverage* as defined in [13]. Total test coverage is also known as *test duplication*.

The primary purpose of this study was to investigate the relationship between test duplication and the quality of tests. Another objective was to shed light on the amount of test duplication in open-source projects and whether such results can be generalized to other projects. An additional objective was to explore the use of a test redundancy metric as a supplementary criterion<sup>3</sup> to assess test quality. Therefore, the study was designed around the following research questions (RQs):

 $\it RQ1$ : What is the amount of test duplication in open-source projects?

**RQ2:** Is there a link between test duplication and the quality of tests?

**RQ3**: Does test redundancy, as a metric, provide a good supplementary assessment into the quality of tests?

### 3. METHODOLOGY

The sections below describe in detail each aspect of the methodology utilized to carry out the research, from object selection to the tools utilized for analysis of results. For each aspect of the research, the guidelines set forth by [15] were followed to achieve statistically sound results.

### 3.1. Object Selection

Ruby<sup>4</sup> is a popular multi-paradigm, high-level, dynamic programming language that has become prominent across several areas of technology due to its simplicity and robust support. It is used in areas ranging from AI [18] to web-development [17]. At the time of this writing, Ruby has also expanded in ways that parallel the maturity and adoption of other popular programming languages, like Python<sup>5</sup>. For instance, much like Jython<sup>6</sup>, JRuby runs Ruby on top of the Java Virtual Machine, which allows Ruby to be used for enterprise projects while taking advantage of the JVM's ability to scale [19].

Due to the reasons mentioned above together with Ruby's widespread (and increasing) adoption, test suites from Ruby projects were chosen for the experiment.

# 3.1.1. Selection Criteria

Selective mutation analysis [22, 23, 24] was one of the methods used to evaluate the impact of removing duplicate tests<sup>7</sup>. Therefore, the test suite selection process had two important qualifying criteria that allowed for easier mutation analysis:

- 1. All tests in a given suite had to execute without failures.
- 2. The code that is the target of a given test suite had to be loosely-coupled.

The first requirement implies that no calculations have to be applied *a posteriori* in order to correct the mutation score with respect to failing tests. This translates into faster, more accurate mutation analysis since the mutation score assigned when the analysis is complete is the final, effective score.

The second requirement stems from an observation that was made during the experiment. In particular, it was observed that moderately- to highly-coupled code tends to have a very low mutation score (below 0.05) because the code that is the target of a given test suite is also intertwined with code that is logically unrelated. The consequence of this is that the test suite naturally fails to detect mutants in the unrelated code when mutation analysis is applied, which results in a low, inaccurate (and therefore useless) mutation score for the purposes at hand. However, an exception to this requirement was made for the test suites associated to *csv* which is a project incorporated into the core of the Ruby project, *ruby-trunk*. The reason for this is because their mutation scores were high enough to be used for analysis, thus becoming usable for the research.

Therefore, this study used 51 test suites from 10 popular Ruby projects that satisfied the selection requirements (with the exception mentioned above). Table 1 summarizes the projects whose test suites were utilized.

### 3.2. Test Redundancy Detection and Removal

The implementation of the mathematical model presented and utilized in [13] was used in this study to detect duplicate tests. This paper calls that implementation the *Ortask Comparison Engine*, or *OCE* for short.

In contrast to previous work such as [5] which is based on Kripke structures, the mathematical model in [13] (and, hence, OCE) approaches test redundancy from the perspective of *Abstract States*. Abstract States are related to the theory of Abstract Interpretation introduced by Cousot [25, 26].

This has several advantages. First, as the results in [13] show, OCE is language-independent. In practice, this means that it can be easily extended for any particular testing style and programming language without changing the underlying comparison logic. Second, OCE takes the source code of tests directly as an approximation to real abstract states. Doing so bypasses the requirement to instrument either the SUT or the tests themselves, while still achieving high accuracy as demonstrated by the results in [13].

## 3.3. Tools Utilized

Two quality criteria were used simultaneously to measure the impact of each test suite's effectiveness before and after removing redundant tests. As discussed in section *Selection Criteria* above, selective mutation analysis [22, 23, 24] was performed on each test suite. The other criterion was code coverage analysis. Both are discussed next.

### 3.3.1. Mutation Analysis

During the early phases of the research, Ruby mutation gems<sup>8</sup> like Heckle<sup>9</sup> and Mutant<sup>10</sup> were explored. However, neither tools were utilized in the end for several important reasons.

<sup>&</sup>lt;sup>3</sup>Supplementary to criteria such as code coverage and mutation score.

<sup>4</sup>https://www.ruby-lang.org/

<sup>&</sup>lt;sup>5</sup>http://www.python.org/

<sup>6</sup>http://www.jython.org/

<sup>&</sup>lt;sup>7</sup>The other method was code coverage analysis, described in section 3.3.2

<sup>&</sup>lt;sup>8</sup>A Ruby gem is an encapsulated piece of functionality analogous to Python's and Perl's packages. http://rubygems.org/

<sup>&</sup>lt;sup>9</sup>http://rubygems.org/gems/heckle

<sup>&</sup>lt;sup>10</sup>https://github.com/mbj/mutant

**Table 1: Project summary** 

Name	contributors	bugs	suites in study	M	C (%)	D
highline	31	97	2	0.752184047	91.54	0.12797619
json	31	189	2	0.2524	77.31	0.2
mongrel*	9	172	2	0.488604456	88.665	0.107954545
net-ssh*	52	128	9	0.707648871	96.32	0.157894737
rake	59	229	3	0.326446281	76	0.307692308
rdoc	30	262	2	0.608757459	92.625	0.285377358
ruby-dbi*	14	107	6	0.699005614	83.595	0.303571429
rubygems	115	704	6	0.281297646	73.635	0.430555556
ruby-trunk**	33	875	13	0.584114727	85.87	0.285714286
ai4r	15	19	6	0.729585385	98.525	0.211111111

 $M = median \ mutation \ score; \ C = median \ code \ coverage \ percentage; \ D = median \ test \ duplication \ ratio.$  The values for contributors and bugs are as of November 2013 and originate from github, while stars indicate additional sources.

On one hand, Heckle relies heavily on a gem called Parse-Tree<sup>11</sup>, which is supported up to Ruby 1.8 [27]. Therefore, Heckle is not supported in later Ruby versions. Ruby 1.8, in turn, is no longer supported since June 2013 [28]. On the other hand, Mutant exhibited dependency conflicts with other gems that were required in the research machine, such as ice\_nine<sup>12</sup> and adamantium<sup>13</sup>. Further, since the version of Mutant was at that time a release candidate (0.3.0.rc1), instability in terms of functionality would be expected and would bring the research results into question.

Due to these reasons, selective mutation analysis was carried out via a home-grown script that employs the results from [22, 23, 24]. Further and in contrast to Heckle and Mutant, the mutation script applies mutation on the lexical (i.e. file) level, rather than at the parse stream. Thus, to achieve uniform mutation for each SUT, the script iteratively and pseudo-randomly selects small chunks from the SUT's target source code and globally applies a mutation within the chunk.

Moreover, to achieve a fine-level of granularity for the mutation order<sup>14</sup>, each chunk is initially only 12 bytes long. If no mutations can be applied to the current chunk, then the chunk grows arithmetically by adding 2 bytes and the pseudo-random selection process takes place once more. This is done iteratively until a mutation can be applied. In effect, the mutation script allows for low and higher order mutations to capture the wide spectrum of faults found in software [29, 30, 31].

In this way, 10000 mutations were applied to each SUT as per the guidelines established in [15] as well as to achieve a high level of statistical confidence with respect to the test suite's mutation score. In other words, each test suite was executed 10000 times, each time against a mutated version of its associated SUT.

Therefore, by employing the slow arithmetically-increasing chunk window, together with the pseudo-random selection of source code and the high amount of mutations, the Central Limit Theorem is enabled to ensure accurate mutation analysis.

#### 3.3.2. Code Coverage Analysis

To measure code coverage levels before and after removal of redundant tests, the Ruby gem simplecov<sup>15</sup> was used.

### 3.3.3. Statistical Analysis

The statistical software package  $R^{16}$  was used to analyze the results.

### 4. RESULTS

In answering RQ1, some differences in terms of code coverage and mutation scores were allowed and even necessary. In particular, it was necessary to allow differences in mutation score before and after removal of duplicate tests due to the stochastic nature of mutation analysis as described in section  $Mutation\ Analysis\$ above. For example, it was observed that some mutation scores were higher than the original scores after removing duplicate tests, while others where lower. For this reason, statistical equivalence testing was utilized to measure the effective difference in mutation scores prior to and following test duplication removal [32, 33, 34, 35] . To this end, a loss of at most 0.05 was considered equivalent to the original mutation score. In other words, if removing tests resulted in a mutation score lower than m-0.05, where m is the original score, then the removal was considered harmful and no tests were removed.

The non-parametric statistical test *Mann-Whitney U* was applied to the mutation scores to detect statistical differences [15]. Additionally, the Vargha-Delaney  $\hat{A}_{12}$  standardized measure was utilized to calculate the effect size of those differences [16]. Table 2 presents the mutation score means before and after duplicate test removal, along with the results of the statistical tests.

Table 2: Statistics for mutation scores prior and after removal of duplicate tests

$\mu$ before	$\mu$ after	p-value	$\alpha$	$\hat{A}_{12}$	
0.5552075	0.4959145	0.222	0.05	0.5697809	

As shown by the p-value of 0.22, there is no statistical difference with respect to the mutation scores before and after removal

<sup>\*</sup> rubyforge.org

<sup>\*\*</sup> ruby-lang.org

<sup>111</sup> http://rubygems.org/gems/ParseTree

<sup>12</sup>http://rubygems.org/gems/ice\_nine

<sup>&</sup>lt;sup>13</sup>http://rubygems.org/gems/adamantium

 $<sup>^{14}</sup>$ *k-order* mutation means *k* mutations per mutant. So, for instance, a 3-order mutation is a mutant with 3 mutations.

<sup>&</sup>lt;sup>15</sup>https://rubygems.org/gems/simplecov

<sup>&</sup>lt;sup>16</sup>http://www.r-project.org/

of duplicate tests. In other words, the *p-value* indicates there is a 22% probability (a high probability) that the variance in average mutation scores is simply due to chance. If the *p-value* had been equal to or less than the significance level,  $\alpha$ , then the mutation scores would have differed significantly in statistical terms.

The  $\hat{A}_{12}$  measure corroborates this result. The value of  $\hat{A}_{12}$  is a real number in the range between 0 and 1. A value of 0.5 means that the first and second parameters (in our case, the average mutation scores prior and following removal of duplicate tests) are statistically identical. As shown by the value of  $\hat{A}_{12}$  in table 2, the "before" and "after" mutation score averages are practically equivalent, with the original average mutation score being only slightly higher by 0.0697809. Since mutation score is a way to measure the regression detection capability of a test suite [6, 10], the tests indeed have the same regression detection capability before and after removal of duplicate tests.

In terms of code coverage, the average decrease produced by removing duplicate tests was 0.27%, with a median loss of 0% and standard deviation of 1.1%.

Figure 1 shows the test duplication ratios for each project. It is clear from the figure that the amount of test duplication is greater than 20% on average. In fact, the average test duplication is 24.74% while the median is 23.33% and standard deviation is 15%.

**RQ1**: On average, 1/4 of a suite's tests are totally redundant (i.e. duplicates)

Table 3 shows the 95% confidence interval along with the associated summary statistics for test duplication ratios. The table shows the values for the mean, median and standard deviation as described previously, as well as the error margin, low bound and high bound of the interval. As the confidence interval describes, there is a very high probability (95%, in fact) that the average test duplication amount for a given test suite will fall between 20% and 29%, with a very low likelihood of outliers (5%). Stated differently, while there is 5% probability that test suites might have less than 20% test duplication, test suites will most likely have an average test duplication of 25%. An analogous argument holds for the "greater than" case (i.e. greater than 29%).

Table 3: Test duplication statistics within 95% confidence interval

cci i ai					
$ar{\mu}$	$ ilde{\mu}$	$\sigma$	error margin	low	high
0.2474	0.2333	0.1547	0.0424	0.2049	0.2899

The aim of RQ2 was to find whether test duplication affects test quality in any way. Similar to studies where the widely-used code coverage criterion shows that the likelihood of bugs found in software decreases as code coverage increases [36, 37, 38, 39, 40], the number of bugs found in each project was used as a measure for the quality of the tests in the study. Therefore, the following reasonable assumption was made: the higher the number of bugs in each project, the lower the quality of their tests.

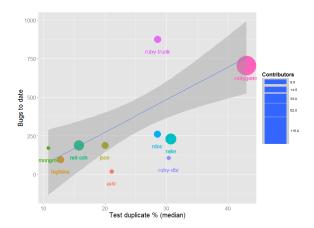
Using this assumption, this study found a high relationship between test duplication and test quality. In particular, the results show a linear (Pearson coefficient) correlation of 0.55452 between test duplication and the amount of bugs in their projects. In fact, this linear correlation was found to be slightly stronger than the correlation between code coverage and test quality, which was found to be -0.5329788. In other words, while tests with higher

code coverage tend to reduce the amount of bugs found in the final SUT (as indicated by the negative sign), projects having a higher amount of test duplication tend to have higher amounts of bugs.

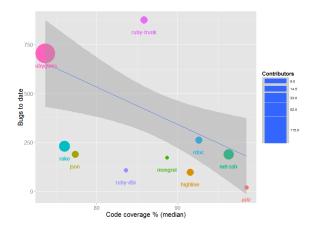
**RQ2**: There is an inverse relationship between the amount of test duplication in a suite and the quality of its tests

Applying linear regression corroborates this observation by yielding a *p-value* of  $2.42 \times 10^{-5}$ . This indicates that the relationship between test duplication and software bugs is statistically significant. Indeed, there is practically zero probability that the relationship is due to chance, indicating that test duplication does affect test quality in a negative way.

Figure 2 shows the relationship between the median test duplication ratios for each project and their associated number of bugs. Furthermore, figure 3 shows the same relationship between bugs and the widely-used code coverage criterion for the same tests.



**Figure 2:** Relationship between bugs in each project and their test duplication ratios. Shaded area shows the 99% confidence interval.



**Figure 3:** Relationship between bugs in each project and their code coverage ratios. Shaded area shows the 99% confidence interval.

Figures 2 and 3 illustrate important consequences of code coverage when it is used as the only quality criterion for tests. For ex-

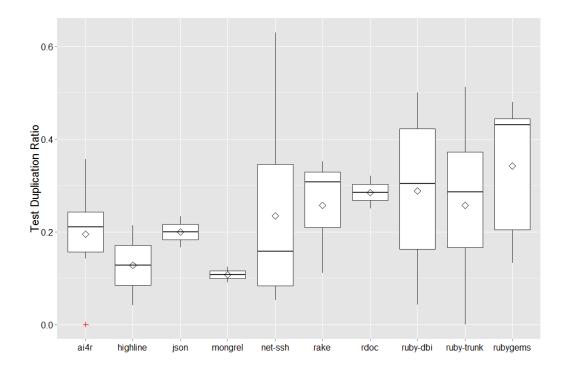


Figure 1: Test duplication ratios for each project (boxes span from 1st to 3rd quartile, middle lines mark the median, whiskers extend up to  $1.5 \times$  inter-quartile range, while plus symbols represent outliers and diamonds signify the mean)

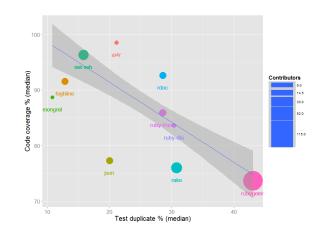
ample, the rdoc project had higher code coverage than json  $(1.2\times)$ , yet it had 1.38 times the number of bugs. It turns out the test duplication ratio for rdoc was 1.43 times higher than json's even though their regression detection capability (i.e. their mutation scores) was 2.4 times higher. Another example can be made with rake and rubygems. In particular, rubygems had only sightly lower code coverage than rake (by  $0.035\times$ ), yet it had 3 times as many bugs. It turns out the test duplication ratio for rubygems was 1.4 times higher than rake's, while their regression detection capability was almost identical (only 1.02 times higher).

Nonetheless, just like with code coverage, the effect of test redundancy is a probabilistic relationship rather than a deterministic one. In other words, the results do not imply causality but indicate a likelihood for the existence of a larger amount of bugs when test duplication is high. That is, these results do not suggest that test duplication directly causes faults in software just like [36, 37, 38, 39, 40] do not suggest that higher code coverage causes their absence. For instance, *ruby-trunk* had almost the same code coverage and test duplication ratios as *ruby-dbi* (only 1.02 and 1.06 times higher, respectively), yet *ruby-trunk* had 8.2 times as many bugs as *ruby-dbi*. It should be noted, however, that the regression detection capability (i.e. mutation score) of *ruby-dbi*'s tests was 1.2 times higher than *ruby-trunk*'s.

The aforementioned relationships between test redundancy, code coverage and mutation score suggest some kind of interplay between these quality criteria. That is, even though there might exist further criteria that need to be studied, perhaps a mathematical relationship developed between the currently existing criteria would allow for a more accurate estimate of the quality of tests. This, in turn, would allow for better predictions regarding the final quality

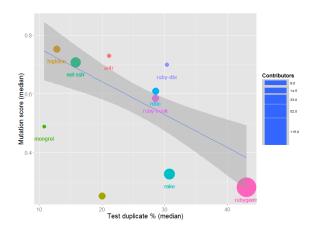
of their SUT.

In terms of the relationship between code coverage and test duplication, it is easy to see by figures 2 and 3 that test redundancy is closely related to code coverage. Indeed, the Pearson correlation coefficient of -0.802849 strongly corroborates that test suites with higher amounts of test duplication have a lower likelihood of having high code coverage, as shown by figure 4. This is further supported by the results of linear regression analysis, where the resulting p-value of  $1.54\times10^{-11}$  indicates a practically zero probability that code coverage and test duplication are unrelated.



**Figure 4:** Relationship between code coverage and test duplication. Shaded area shows the 99% confidence interval.

With respect to the suite's mutation scores (i.e. their regression detection capability) and test redundancy, figure 5 suggests a close relationship as well. In fact, the Pearson coefficient of -0.6022639 supports this finding. Moreover, applying linear regression analysis yields a p-value of  $2.93 \times 10^{-6}$ , supporting the observation that as test duplication increases, the regression detection capability of their suites tends to decrease.



**Figure 5:** Relationship between regression detection and test duplication. Shaded area shows the 99% confidence interval.

The fact that test redundancy, mutation score and code coverage are closely related indicate the possible existence of a relationship that might be grasped by a mathematical function whose parameters are at least such criteria. For instance, one such relationship might be the following linear combination:

$$\beta_0 + \beta_1 M + \beta_2 D + \beta_3 C + \beta_4 M D + \beta_5 M C + \beta_6 D C$$

where, D represents the median duplication ratio, C is the median code coverage percentage and M is the median mutation score of test suites for a single project, while each  $\beta_i$  represents a coefficient that varies depending on the values of each criterion for a given project. For example, using the values in table 4 as the coefficients for the linear combination, the estimated amount of bugs for ruby-trunk is 693.12, while the estimated amount of bugs for ruby-trunk is 229.83. Even though the difference in estimated bugs between ruby-trunk and ruby-trunk is only  $3 \times a$  opposed to the actual 8, the estimate is certainly better compared to simply considering each of the criteria alone.

Table 4: The values assigned to each coefficient

-12207.13				
30446.58				
-15228.48				
119.71				
-25869.35				
-308.56				
366.08				

In fact, applying this linear combination to the projects in this study yields a rather high  $R^2$  value (goodness of fit) of 0.7144, which is close to a perfect fit with a value of 1. However, more

studies are needed to corroborate and improve such a model, especially due to the fact that base cases (i.e. perfect values for each criterion) and extreme cases (i.e. very poor values for each criterion) result in inaccurate values. Nonetheless, the results show that measuring test quality is significantly more accurate when other quality metrics are used in conjunction with test redundancy. This is further evidenced by the fact that removing test redundancy from the model dramatically lowers its  $\mathbb{R}^2$  value to one that is at most 0.3493.

Given these findings along with their statistical significance, we are able to answer RQ3 with confidence.

**RQ3**: Test redundancy is an accurate supplementary metric into the quality of tests

#### 5. CONCLUSION

This paper has presented statistically significant results that show a strong relationship between test redundancy and lower quality of tests. In particular, the results in this paper show that, contrary to maximizing their effectiveness, test duplication tends to increase the likelihood of failures associated to their SUT.

Moreover, the results also illustrate that test redundancy is a useful metric that should be used in the software development lifecycle together with other criteria such as code coverage and mutation score. In other words, code coverage cannot be the only criterion used to asses test quality, as doing so does not provide a complete view into the health of tests. For a better assessment of test quality, therefore, this paper suggests expanding the view to include a test redundancy metric. This argument is further strengthened by the close link between the regression detection capability of test suites and their redundancy.

This paper also proposes a tentative mathematical model that appears to capture the relationship between test duplication and other quality criteria in a manner that is rather effective for estimating the amount of bugs for each project in this study.

Overall, the results presented in this paper together with the proposed model provide strong evidence that test redundancy is an intricate aspect of test quality. As such, test redundancy should be a relevant and important concern for test management in particular and the software development life-cycle in general.

## 6. FUTURE WORK

The linear combination presented in the answer for *RQ*4 necessitates further research. Preliminary assessments using Q-Q and residual error analysis has shown that the model predicts bugs in the SUT much better than each criterion taken alone. Further, it would be interesting to assess how well the model fits data from other projects, or whether the linear combination effectively overfits or under-fits the data. In such a case, machine learning algorithms might be able to discover a model that more accurately predicts the amount of bugs in the SUT. Nonetheless, the linear combination presented above is a starting point that improves our current understanding of test quality and promises the potential for more effective estimation of bugs in the final SUT, therefore unifying the known quality criteria into an effective estimation device that is certainly more accurate than taking each quality criterion by itself.

### 7. REFERENCES

[1] V. Vangala, J. Czerwonka, and P. Talluri, "Test Case Comparison and Clustering using Program Profiles and Static Execution," Microsoft, 2009

- [2] M. Greiler, A. van Deursen, and A. Zaidman, "Measuring Test Case Similarity to Support Test Suite Understanding," 2012
- [3] T. Xie, D. Marinov, and D. Notkin, "Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests," 2004
- [4] N. Koochakzadeh, V. Garousi, and F. Maurer, "A Tester-Assisted Methodology for Test Redundancy Detection," 2009
- [5] G. Fraser and F. Wotawa, "Redundancy Based Test-Suite Reduction," 2007
- [6] N. P. Singh, R. Mishra, R. R. Yadav, "Analytical Review of Test Redundancy Detection Techniques," 2011
- [7] M. J. Harrold, R. Gupta, and M. L. Soffa, "A Methodology for Controlling the Size of a Test Suite," 1993
- [8] D. Jeffrey and N. Gupta, "Test Suite Reduction with Selective Redundancy," 2005
- [9] H. Zhong, L. Zhang, and H. Mei, "An Experimental Comparison of Four Test Suite Reduction Techniques," 2006
- [10] J. von Ronne, "Test Suite Minimization, An Empirical Investigation," 1999
- [11] D. Li, C. Sahin, J. Clause, and W. G. J. Halfond, "Energy-Directed Test Suite Optimization," 2013
- [12] T. Gergely, Á. Beszédes, T. Gyimóthy, and M. I. Gyalai, "Effect of Test Completeness and Redundancy Measurement on Post Release Failures âĂŞ an Industrial Experience Report," 2010
- [13] M. G. Macedo, "Unit-Test Source Code is a Good Approximation For Abstract States," Ortask, 2013
- [14] R project, http://www.r-project.org/
- [15] A. Arcuri and L. Briand, "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," 2010
- [16] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of Mc-Graw and Wong," Journal of Educational and Behavioral Statistics, 2000
- [17] http://rubyonrails.org/
- [18] http://ai4r.org/
- [19] http://jruby.org/
- [20] J.H. Andrews, L.C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?," 2005
- [21] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov, "Evaluating the âĂIJSmall Scope HypothesisâĂİ," 2003
- [22] A. J. Offutt, and J. Pan, "Procedures for Reducing the Size of Coverage-based Test Sets," 1995
- [23] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators," 1996

- [24] A. J. Offutt, and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," 2001
- [25] P. Cousot, and R. Cousot, "Basic Concepts of Abstract Interpretation," 2005
- [26] P. Cousot, and R. Cousot, "Abstract Interpretation: A Unified Lattice model for Static Analysis of Programs by Constructions or Approximation of Fixpoints," 1977
- [27] http://rubygems.org/gems/heckle
- [28] http://www.ruby-lang.org/en/news/2013/06/30/we-retire-1-8-7/
- [29] M. Harman, Y. Jia, and W. B. Langdon, "Strong Higher Order Mutation-Based Test Data Generation," 2011
- [30] M. Harman, and Y. Jia, "Constructing Subtle Faults Using Higher Order Mutation Testing," 2008
- [31] M. Harman, and Y. Jia, "Higher Order Mutation Testing," 2009
- [32] M. J. Knol, W. R. Pestman, and D. E. Grobbe, "The (mis)use of overlap of confidence intervals to assess effect modification," 2010
- [33] R. Hubbard, and R. M. Lindsay, "Why P Values Are Not a Useful Measure of Evidence in Statistical Significance Testing," 2008
- [34] A. Knezevic, "Overlapping Confidence Intervals and Statistical Significance," 2008
- [35] J. Shtaynberger, and H. Bar, "Equivalence Testing," 2013
- [36] M. Harder, B. Morse, and M. D. Ernst, "Specification Coverage as a Measure of Test Suite Quality," 2001
- [37] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test Coverage and Post-Verification Defects: A Multiple Case Study," 2009
- [38] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software Unit Test Coverage and Adequacy," 1997
- [39] Y. K. Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe, "The Relationship Between Test Coverage and Reliability," 1994
- [40] S. Berner, R. Weber, and R. K. Keller, "Enhancing Software Testing by Judicious Use of Code Coverage Information," 2007