CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Mining and comparing software testing metrics and evaluating the use of different test quality metric tools

Bachelor of Science Thesis in Software Engineering and Management

Mattias Ekdahl

Simon Engström

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

The Author grants to University of Gothenburg and Chalmers University of Technology the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let University of Gothenburg and Chalmers University of Technology store the Work electronically and make it accessible on the Internet.

Mining and comparing software testing metrics and evaluating the use of different test quality metric tools

© Mattias Ekdahl, June, 2022.
© Simon Engström, June, 2022.

Supervisor: Francisco Gomes de Oliveira Neto
Examiner: Richard Berntsson Svensson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
University of Gothenburg
Chalmers University of Technology
Gothenburg, Sweden 2022

Mining and comparing software testing metrics and evaluating the use of different test quality metric tools

1st Mattias Ekdahl
University of Gothenburg
(Software Engineering and Management)
Gothenburg, Sweden
gusekdmad@student.gu.se

2nd Simon Engström
University of Gothenburg
(Software Engineering and Management)
Gothenburg, Sweden
gussimen@student.gu.se

***Abstract*—[Context] Software test metrics are often compared with one another to find the most accurate predictor of test quality, however little is written about the connections between types of metrics. Moreover, open-source software is more commonly used to extract these test metrics, but the issues researchers face when attempting to use them is under-reported. [Objective] This paper aims to investigate the correlation between coverage, mutation and diversity-based test metrics, as well as to describe the challenges involved in automating the use of test quality metric extraction tools. [Method] We conducted an MSR study on 22 open-source software repositories from Github which were mined in order to extract test quality metrics. To do this we developed a tool which combined several metric extraction tools to provide code coverage, mutation coverage and average test suite diversity values. [Results] The challenges we faced when conducting this MSR study stemmed mostly from trying to automate the test quality metric extraction process, such as finding repositories that were compatible with all of our metric extraction tools, Maven plugins interfering, costly execution time and tools not working out of the box. We then applied Spearman's Rho to calculate the correlation between the metrics. [Conclusion] Our results show that code coverage, mutation coverage and test suite diversity all have a positive correlation with each other. We also found that automating the extraction of test metrics limited the availability of repositories due to stricter requirements.**

***Index Terms*—test quality metrics, mutation testing, mining software repositories, code coverage, test diversity**

I. Introduction

Testing is a crucial part of software development which ensures the delivery of functioning and stable products. As a result of software testing's importance a lot of research is naturally conducted about it. Open-source software has long been used to research testing techniques and their efficiency or effectiveness [3]. In order to extract meaningful data from repositories different mining tools are used. There are many different tools available both free and commercial for both industry and research [5].

Test metrics are used to verify the quality of the testing and for measuring the completeness of software testing activities. Test metrics are essential for quantifying how well a software has been tested by helping testers determine whether the requirements for reliability and functionality have been met [1]. There are currently many different test quality metrics used in research and industry. Common types of metrics are coverage-based which measure the percentage of the code covered by the test suite such as class, statement, method, or branch coverage. Coverage-based metrics can show how well certain parts of the code is tested which can help direct further testing [5]. However, code-based are limited in conveying about how well the tests relate to the specification (e.g., validation of requirements).

One typical technique to evaluate the quality of a test suite is to use mutation testing. This method introduces new artificial faults to the software, called mutants, which the test suite is then supposed to discover. Mutation testing is judged by a fault-based analysis, which takes into account all the newly introduced mutants and the discovered ones. Thus indicating for testers how well the tests catch new faults [6], [7]. However, mutation tools have many limitations such as long execution costs, or they are hard to scale to large software projects [6]. Mutation testing tools use a variety of mutation operators to make slight changes in the code, such as arithmetic operators which change mathematical operations like additions to subtractions as an example.

Another metric is test diversity that aims at identifying very similar tests. By comparing all tests pair-wise and placing the results in a distance matrix, similar tests can be found. The amount of tests in a test suite can be lowered by removing tests which are similar while still retaining similar fault detecting capabilities as before [8], [9]. This is because similar tests cover the same code and find the same faults which makes them redundant. Extracting diversity can however become a bottleneck as the distance matrix's size is the square of the number of tests $O(n^2)$ [16].

Research has so far been used to evaluate different metric extraction tools based on what features they possess [1]. However, research studies rarely report on additional challenges when applying and using those tools for mining repositories, challenges such as unexpected build failures due to plugins added in the repository's configuration or the project architecture not being supported.

Moreover, metrics are often represented in research in the context of replacing other metrics to gain some advantage. As the code under testing grows larger it becomes difficult to maintain previous levels of coverage since more tests have to be made to cover the increased code [2]. Similarly, if the number of tests increase, measuring diversity becomes more costly due to the bottleneck mentioned earlier. Mutation testing also suffers in performance when the code under test grows. But little research is done on how these metrics are correlated to each other. Knowing how they are connected could guide future testing efforts on how they could be combined to possible achieve even greater test quality.

The problem this thesis project aims to investigate is the correlation between coverage, mutation and diversity-based test metrics. In addition this thesis project intends to examine the challenges in gathering the data required for evaluating these test metrics from open-source projects. We intend to achieve all of this by creating our own MSR tool that will automatically run other test quality metric extraction tools on a downloaded repository, and then output a report containing the outcomes from each of the test quality metric tools. This will allow us to efficiently extract test quality metric data from open-source repositories, hence supporting our and future research to understand the relationship between test metrics. In this study we will be examining open-source repositories on Github.

In this study, our technical contribution is the tool we create for retrieving and running test data. This tool can be used in future mining software repositories (MSR) studies to extract testing metrics. Our scientific contribution is the increased understanding of the ways different test metrics relate. We also use and compare multiple existing tools for extracting test metrics and catalog the strengths and weaknesses along with related challenges of using said tools.

II. Related Work

In order to conduct the research for this thesis project, a lot of data is needed and mining software repositories is a field that has gained a lot of traction in recent years. As a research method it aims to utilise the vast amount of data stored in software projects in order to discover important information that can be generalizable on a large scale [4]. Conducting MSR on large scale commercial systems is often infeasible due to companies not generally being willing to share such information. But thanks to open-source software systems, vast amounts of data is available, free to be mined by researchers.

We are mainly interested in investigating the test cases of each project, in order to extract test quality metrics. Previous research has been done to evaluate different tools for test quality metric extraction, such as the study conducted by Shahid et al. In their study [1], the researchers analysed and compared 19 different test coverage tools and measured them against 3 different criteria: language support, instrumentation, and coverage measurement. The end result is a set of tables that give a clear view of some of the strengths and weaknesses of each tool. Our study will also compare the code coverage measurements of newer versions of Jacoco and Clover, along with a third coverage tool called JMockit which is not covered in their study.

Similar research has been done with mutation testing tools, such as in the study conducted by Kintis et al. In this study [10] the researchers studied and compared three different mutation tools: MUJAVA, MAJOR and PITest. The researchers compared the tools in terms of their effectiveness, and found that none of them were superior to the others. They concluded this by seeing if the tests of one tool are able to kill all the mutants of the others, and found that this was not the case for any of the tools. For our study we will investigate potential challenges in using both PITest and LittleDarwin, as well as any differences we find in their features and behaviour.

Similarly, in the case of diversity-based testing, there are many different methods to determine the diversity of a set of tests. For testing, researchers typically use string distances. Some string distance methods are Jaccard and Levenshtein, another type of method is Normalised Compression Distance (NCD) that works on the byte representation level which allows for more types of objects to compare, such as numerical. Diversity between two test cases can be calculated by representing the test cases as strings which then is supplied to the diversity methods [15]. By comparing all tests pairwise a diversity matrix can be made of the entire test suite [8], [9].

III. Research Methodology

The aim of this study is to investigate potential connections between coverage, mutation and diversity-based metrics and see how they correlate with one another. We will also explore the challenges in extracting test metric data from open-source projects on Github. We intend to describe what artefacts need to be mined in order to extract the test metrics and to use and compare the tools available in order to conduct the mining. We have formulated the following research questions:

A. Research questions

RQ1: What are the challenges in mining different open-source projects to extract test metrics?

RQ1.1: How do our chosen test metric extraction tools compare in terms of different features and behaviour?

RQ1.2: What are the key artefacts needed to extract the different metrics?

RQ2: What is the correlation between our chosen test metrics?

B. Research methodology to be used

In order to answer our RQ's we will conduct an MSR style study using Java projects on Github. We use an MSR methodology because we need metric data from many repositories in order for our results to be statistically significant [17]. We have chosen to focus on Maven [1] projects using JUnit [2], because we wish to automate the metric extraction process. Maven makes this possible by allowing us to execute the

[1]https://maven.apache.org/

[2]https://junit.org/

metric extraction tools directly via the terminal. This will give us a general representation of open-source software. We will use different open-source tools for extracting code coverage, mutation coverage and average diversity of test suites to collect the test metrics needed to answer our RQ's.

In order to conduct this research we developed a tool [3] that assists with mining open-source repositories from Github, see Fig. 1. It works by first cloning the git repository, then using Maven from the terminal in the operating system, our tool runs the code coverage tools and then the mutation testing tools and finally the diversity tool. It then parses the outcome of each tool, and presents the user with a table containing the combined reports from the six tools.

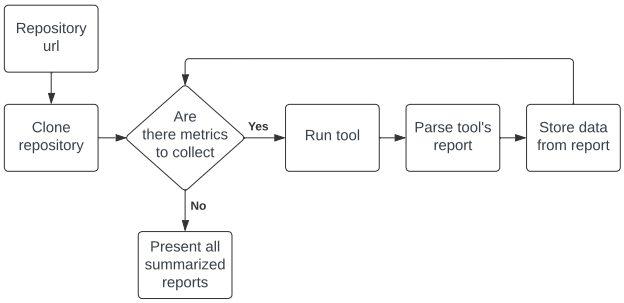

Fig. 1. Process diagram for the tool we develop for this study

To answer RQ1 we will document the challenges we encounter through out the course of conducting this study. Furthermore, the challenges met while working towards answering RQ1.1 and RQ1.2 are included as well.

In order to answer RQ1.1 we will look at existing tools and frameworks that can be automated by being able to be run by our own tool. The types of features we compare vary per type of tool. For instance, some measure code coverage on different levels, or have different mutation operators. Additionally we will look at their ease of use, which includes the documentation and the requirements they have of the projects in order to work. As a starting point for finding code coverage tools we will utilise the work done by Shahid et al. [1], and look at tools presented in their research and check whether they comply with our specifications. In addition to that paper we will conduct our own search for tools.

To answer RQ1.2 we will look at what artefacts are necessary in order to extract our selected test quality metrics, and how easily accessible these artefacts are. These may include the Java files in which the unit tests are stored, and how they are structured. This will help future researchers when deciding on what test quality metrics to study. For our research we also require the projects to be Maven projects and structured accordingly.

To answer RQ2 we will use our software repository mining tool in order to extract test quality metrics from the mined projects. Our tool will generate a table of metric data consisting of different code coverage types, mutation coverage and diversity ratings produced by the chosen tools. We will conduct statistical analysis on these in order to find any significant connections between the extracted metrics.

1) Data collection: For this study we developed a tool which can mine open-source repositories from Github and run test metric extraction tools on those repositories. Entire repositories have been mined in order to gain access to all necessary artefacts for RQ1.2. Since we want to collect coverage independently of type of tests, we do not distinguish between functional and nonfunctional ones. We have chosen to use small repositories, that is projects with less than 2000 lines of code, in order to have enough time to run the mutation testing tools.

The coverage metrics are measured on the test suite level, so that we do not look at individual tests. We have mined a total of 22 repositories, see Table 2, with a variety of purposes. Some of these projects are Java-string-similarity which is a software library containing several algorithms for comparing distances and similarities of strings, Maven BuildTime Profiler which help with finding causes to long build times and thirdly Maven Incremental Module Builder ensure that for larger builds only changed modules are being built. We show that the repositories range in lines of code, see Fig. 2, and the size of the test suites, see Fig. 3. The lines of code of each repository was found using the code coverage tool Jacoco, which counts the amount of lines with actual code on them rather than the total number of lines in every file, see Fig. 4.

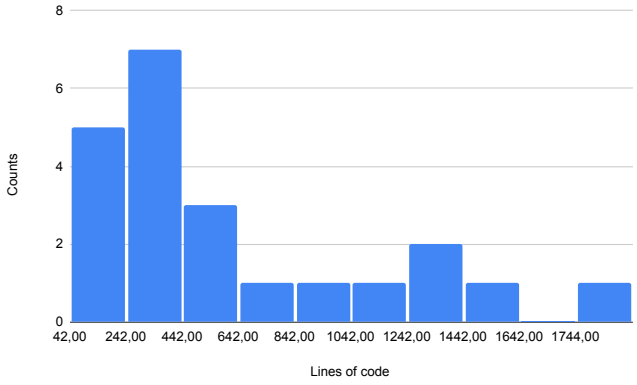

Fig. 2. Histogram showing the spread of lines of code for the repositories

In order for our tool to work the repository under test and the tools used to evaluate the tests needs to be compatible, therefore we require both the repositories and tools to support Maven version 3 or above. The projects need to be Maven projects and structured with only one module, furthermore their test suites must be written with JUnit. The tools must exist as Maven plugins or support Maven projects. Both tools and repositories must run on Java 9 or higher in order for them to compile and work properly with our tool, see Table 1 for all

[3]https://github.com/Simonster1999/RepoMiner

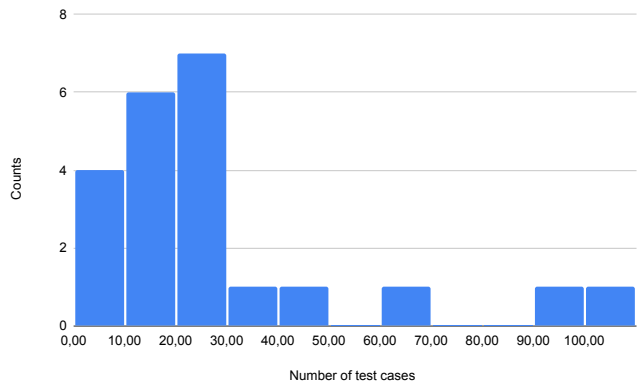

Fig. 3. Histogram showing the spread of test suite size among the repositories

```
87.       for (int i = 0; i < s1.length(); i++) {
88.           // calculate v1 (current row distances) from the previous row v0
89.           // first element of v1 is A[i+1][0]
90.           //   edit distance is delete (i+1) chars from s to match empty t
91.           v1[0] = i + 1;
92.
93.           int minv1 = v1[0];
94.
95.           // use formula to fill in the rest of the row
96.           for (int j = 0; j < s2.length(); j++) {
97.               int cost = 1;
98.               if (s1.charAt(i) == s2.charAt(j)) {
99.                   cost = 0;
100.              }
```

Fig. 4. Image showing actual lines code highlighted in green

tools looked at. Data will only be collected from repositories where all tools run successfully.

The different tools are individually run on each repository, while our own tool will parse the generated reports, summarise and present them in a table. The code coverage measuring tools, Jacoco [4], Clover [5] and JMockit [6], combined give multiple different types of code coverage, see Table 3, while the mutation testing tools, LittleDarwin [7] and PITest [8], mutate the code in several different ways, see Table 4. To measure the diversity of test suites we used a modified version of a tool developed for a previous study [15]. The tool offers three algorithms for calculating the diversity of individual tests using Jaccard index, NCD and Levenshtein distance, however for this study we exclude Levenshtein distance due to its costly execution time. The outcome of running the diversity tool is a matrix containing the diversity score for each test case pair. Diversity is often measured between pairs of test cases, and since we want to measure the diversity of a test suite, we calculate the average diversity of all pairs of tests. However, since the diversity of the test suite is an average we will also measure the standard deviation for each suite.

2) Data analysis: We use a few qualitative and quantitative criteria to analyse and compare testing tools. Some of these criteria, such as the supported programming languages and

[4]https://www.eclemma.org/jacoco/
[5]https://openclover.org/
[6]https://jmockit.github.io/
[7]https://littledarwin.parsai.net/
[8]https://pitest.org/

TABLE I
LIST OF TEST QUALITY METRIC EXTRACTION TOOLS EVALUATED BASED ON MAVEN 3 AND JAVA 9+ SUPPORT

Tool	Maven 3 support	Java 9+
Jacoco	X	X
Clover	X	X
JCoverage		
Emma		
Groboutils		
Serentity DBB		
Cobertura	X	
PITest	X	X
LittleDarwin	X	X
JMockit	X	X

corresponding coverage metrics, are answered by the tools' documentation and verified when extracting the test metrics. The diversity tool is not part of RQ1 because we only have a single tool that yields both distance measures used in our study. To answer RQ2 we use Spearman's Correlation Coefficient, also known as Spearman's Rho, to calculate the pairwise correlation between the test metrics. Our data being non-parametric also points us to use Spearman's Rho.

IV. RESULTS

This section covers the results from our study, and aims to answer our research questions.

A. *RQ1 - What are the challenges in mining different open-source projects to extract test metrics?*

When conducting our MSR study we have mined and gathered test quality data from 22 software projects. The challenges we encountered have originated to a large extent from trying to automate the metric extraction process, as well as integrating the extraction tools to our build scripts. We have identified three main challenges when mining test metrics from open-source repositories:

- **Maven plugins interfering**
- **Costly execution time**
- **Tools not working out of the box**

Maven plugins interfering with our metric extraction tools has been a challenge when trying to automate the metric extraction process. We found that some plugins present in a projects POM.xml file, such as Maven Rat and Checkstyle plugin that check the syntax of source code, caused build failures mistakenly when used together with our coverage tools. This resulted in our coverage and mutation tools not generating any reports. These plugins were present in 5 of the 22 repositories we included in this study, and we were able to solve the issue for these repositories by adding a skip variable to these plugins (e.g, -Drat.skip and -Dcheckstyle.skip) when running our tools from the terminal. Although this was not an option for all of the plugins, the repositories that did not work after applying this solution were discarded.

TABLE II
ALL REPOSITORIES USED IN THIS STUDY WITH VERSION, LINES OF CODE AND THE NUMBER OF TESTS

Repository	Commit/Tag	Lines of code	Number of tests
maven-clean-plugin	maven-clean-plugin-3.1.0	311	8
maven-checkstyle-plugin	maven-checkstyle-plugin-3.1.2	1744	28
rpm-builder	rpm-1.8.0	1261	14
maven-site-plugin	maven-site-plugin-3.11.0	1299	5
junit-test-example	5621b06	42	12
gitflow-maven-plugin	v1.18.0	1569	63
java-string-similarity	v2.0.0	662	24
Simpriser	42d9643	276	5
console-arena	1.0.1	210	20
configuration-parser	v1.5	186	21
incremental-module-builder	6cc82b5	109	4
maven-dependency-analyser	d963e2f	491	90
babel-maven-plugin	2d2d11b	234	22
maven-invoker	c53341d	553	100
maven-source-plugin	maven-source-plugin-3.2.1	303	10
maven-install-plugin	76fc2fe	337	15
maven-verifier	maven-verifier-1.8.0	980	10
plugin-repeater-maven-plugin	8931776	258	20
properties-maven-plugin	e470d9d	420	10
depmeter-maven-plugin	b9b1ebd	369	25
swaggerhub-maven-plugin	1.0.9	604	41
ph-javacc-maven-plugin	ph-javacc-maven-plugin-4.1.5	1205	38

Note: Commit hashes are used when tags are unavailable.

Costly execution time of extracting test metrics is another challenge we faced when mining test metrics from software repositories. Since the tools are run separately in order to not interfere with each other it means that the test suites are executed once for every coverage tool, which if the test suite is large or contains more complex test cases results in longer extraction times. This is further increased by the mutation testing tools which run the test suites multiple times in order to try to find every mutation. The number of mutants is dependent on how much of the code can be mutated using each tools' mutation operators. Examples of recorded execution times have been recorded for three repositories: maven-verifier, maven-invoker and maven-checkstyle-plugin, see Fig. 5.

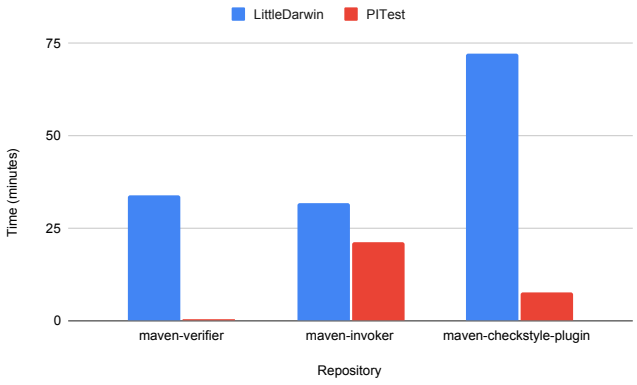

Fig. 5. Examples of recorded execution time for LittleDarwin and PITest on three different repositories

Tools not working out of the box was another challenge we faced. For LittleDarwin we were required to modify the source code in order for the tool to work on some repositories. This was due to LittleDarwin crashing when trying to decode exceptions sometimes thrown by a project. This problem only occurred for a few repositories, however the exceptions could appear partly through the mutation detection stage meaning the problem was not always spotted quickly.

While we had no problems running JMockit, trying to parse its coverage report proved to be challenging. The HTML report it generated did not make it clear what type of coverage it showed, therefore we used the second option of generating a Serial file. Using the JMockit java library we could parse this file and validate that we received line coverage by calling the methods included in the library.

As previously mentioned, when conducting this MSR study we have utilised three coverage-based tools Jacoco, Clover and JMockit, two mutation-based tools LittleDarwin and PITest and one diversity tool. We compare the coverage tools in terms of the types of coverage they support, and likewise for the mutation tools in terms of the mutation operators they support, see Tables 3 and 4.

After running and using JMockit we found that the doc-

TABLE III
COVERAGE TYPES SUPPORTED BY OUR CHOSEN COVERAGE TOOLS

Tool	Branch	Instruction	Line	Method	Class	Statement
Jacoco	X	X	X	X	X	
Clover	X			X		X
JMockit			X			

TABLE IV
MUTATION OPERATORS SUPPORTED BY OUR CHOSEN MUTATION TOOLS

Mutation operator	PITest	LittleDarwin
AOR	X	X
SOR	X	X
ROR	X	X
COR		X
COD		X
SAOR		X
LOR	X	X
PRIMITIVE_RETURNS	X	
NULL_RETURNS	X	X
TRUE_RETURNS	X	
FALSE_RETURNS	X	
EMPTY_RETURNS	X	
VOID_METHOD_CALLS	X	
RETURN_VALS	X	
NEGATE_CONDITIONALS	X	
NullifyInputVariable		X
NullifyObjectInitialization		X

umentation does not match the functionalities it possesses in practice. The creators state that JMockit supports line, path and data coverage, but after looking at the report JMockit creates it only supports line coverage. Investigating the source code also supports our findings of JMockit only providing line coverage in the current version.

We also found that projects consisting of multiple modules can not be analysed by Jacoco or JMockit when run via the command line, they require configuration which should be tailored to each project, thus not allowing for a general automatic solution. However, Clover does not have this limitation.

Further, we found that PITest has exceedingly faster execution time than LittleDarwin, see Fig. 5. The faster execution time of PITest is due to the difference in how it prioritises tests. PITest first checks line coverage information per test case, then discards all test cases that do not exercise the mutated line of code. The remaining test cases are then sorted by execution time, starting from low to high. Mutants which are not covered by any test are automatically considered to have survived. Any of the remaining test cases that fail will kill the mutant, and PITest moves on to the next mutated line of code.[9] Furthermore, PITest also mutates on bytecode level, further aiding in decreasing execution time. LittleDarwin does not prioritise test cases in the same way PITest does, rather it runs the entire test suite on every mutant. Additionally, LittleDarwin also mutates on source code level, further increasing execution time by compiling the source code for each mutant [14].

The artefacts needed to collect these test quality metrics is the test suite, as well as the source code of a project. In addition, the projects' configuration files was used to check for clashing plugins.

Main findings for RQ1:

- Main challenges we faced were interfering Maven plugins, costly execution time and tools not working out of the box.
- Comparing the coverage tools we found that Jacoco supports the most types of coverage, and JMockit the least.
- PITest has a far faster execution time compared to LittleDarwin, and they differ slightly in the types of mutation operators they support.
- The artefacts needed to conduct this MSR study is the test suite of the repository being mined, along with the source code.

B. RQ2 - To what extent are our chosen metrics connected

From mining the repositories using our tool, we extracted code and mutation coverage of the test suites, see Table 5. In addition, the average diversity of each test suite using both Jaccard and NCD was calculated, see Table 5. Finally, the lines of code of each repository and the size of their test suite was collected, see Table 2.

In the end we decided not to include coverage data from Clover and JMockit in our correlation analysis because their coverage types overlapped with Jacoco, see Table 3, apart from statement coverage which was discarded due to by definition being very similar to line coverage. Jacoco was chosen over JMockit and Clover because it provides more code coverage metrics.

The data from Tables 2 and 5, apart from the standard deviations, was used for pairwise correlation testing using Spearman's Rho. The resulting correlation analysis matrix is presented in a heatmap, see Fig. 6.

The results of this MSR study shows both negative and positive correlations between our chosen metrics, see Fig. 6. The number of test cases per test suite has a moderate positive

[9]https://pitest.org/faq/

TABLE V
CODE AND MUTATION COVERAGE RESULTS FOR THE 22 REPOSITORIES SHOWN IN PERCENTAGE AND THE AVERAGE DIVERSITY OF EACH TEST SUITE

Repository	Branch	Instruction	Line	Method	Class	PITest	LittleDarwin	Jaccard	NCD
maven-clean-plugin	0.33	0.37	0.37	0.59	0.77	0.34	0.56	0.27 (±0.02)	0.45 (±0.03)
maven-checkstyle-plugin	0.42	0.58	0.66	0.73	0.92	0.25	0.70	0.66 (±0.01)	0.68 (±0.01)
rpm-builder	0.05	0.03	0.05	0.06	0.11	0.04	0.12	0.57 (±0.02)	0.56 (±0.01)
maven-site-plugin	0.07	0.07	0.08	0.13	0.13	0.06	0.32	0.46 (±0.01)	0.48 (±0.01)
junit-test-example	1.00	0.85	0.83	0.68	0.88	0.82	0.87	0.65 (±0.03)	0.58 (±0.02)
gitflow-maven-plugin	0.04	0.03	0.04	0.07	0.18	0.03	0.26	0.43 (±0.01)	0.46 (±0.01)
java-string-similarity	0.88	0.79	0.77	0.80	0.77	0.75	0.88	0.56 (±0.03)	0.63 (±0.04)
Simpriser	0.05	0.14	0.23	0.25	0.57	0.07	0.00	0.56 (±0.06)	0.63 (±0.07)
console-arena	0.01	0.07	0.12	0.25	0.33	0.07	0.01	0.42 (±0.01)	0.33 (±0.01)
configuration-parser	0.78	0.90	0.88	1.00	1.00	0.77	0.81	0.60 (±0.01)	0.55 (±0.01)
incremental-module-builder	0.22	0.19	0.15	0.16	0.25	0.15	0.20	0.21 (±0.03)	0.35 (±0.04)
maven-dependency-analyzer	0.45	0.53	0.57	0.67	0.93	0.43	0.45	0.52 (±0.00)	0.54 (±0.00)
babel-maven-plugin	0.75	0.85	0.87	0.89	0.85	0.49	0.50	0.53 (±0.01)	0.60 (±0.01)
maven-invoker	0.78	0.82	0.84	0.82	0.76	0.66	0.66	0.65 (±0.00)	0.65 (±0.00)
maven-source-plugin	0.21	0.25	0.30	0.51	0.62	0.15	0.78	0.54 (±0.03)	0.54 (±0.02)
maven-install-plugin	0.25	0.32	0.40	0.46	0.75	0.22	0.55	0.36 (±0.01)	0.54 (±0.01)
maven-verifier	0.18	0.26	0.25	0.26	0.62	0.10	0.27	0.71 (±0.03)	0.69 (±0.03)
plugin-repeater-maven-plugin	0.28	0.27	0.30	0.31	0.50	0.27	0.28	0.57 (±0.01)	0.56 (±0.01)
properties-maven-plugin	0.21	0.31	0.33	0.50	0.53	0.20	0.60	0.54 (±0.02)	0.57 (±0.03)
depmeter-maven-plugin	0.31	0.46	0.52	0.75	0.90	0.36	0.77	0.62 (±0.01)	0.61 (±0.01)
swaggerhub-maven-plugin	0.69	0.91	0.91	0.95	0.94	0.74	0.66	0.62 (±0.01)	0.67 (±0.01)
ph-javacc-maven-plugin	0.26	0.21	0.29	0.44	0.34	0.22	0.57	0.63 (±0.00)	0.59 (±0.00)

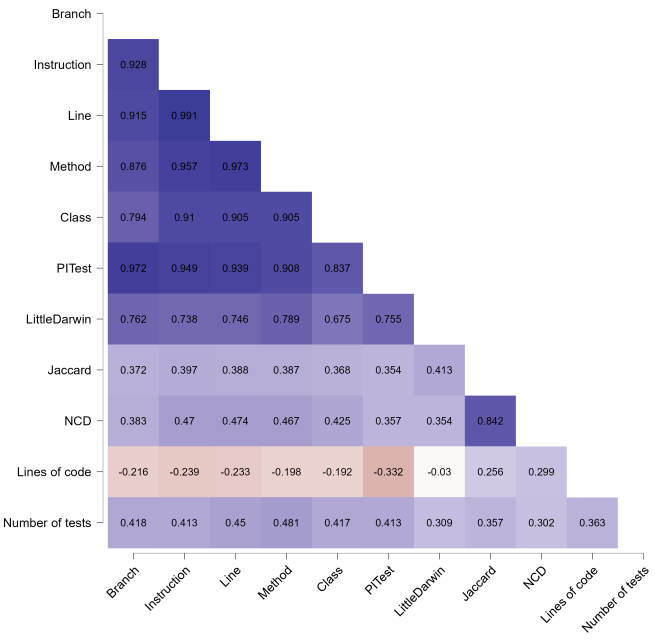

Fig. 6. A heatmap of the pairwise correlation analysis matrix between the mined data.

correlation with code coverage, while the size of the source code represented in lines of code has a minor negative effect.

Our results also showcase a strong positive connection between PITest and all code coverage metrics while LittleDarwin has a slightly weaker positive correlation between code coverage. The repository java-string-similarity has both high code and mutation coverage, see Table 5, while maven-dependency-analyser has a more moderate code coverage percentage and a moderate mutation coverage to match.

The slightly weaker correlation of LittleDarwin and code coverage can be seen in the maven-source-plugin repository where LittleDarwin's mutation operators could only modify one class which was more thoroughly covered by the test suite resulting in a mutation coverage much greater than PITest.

Diversity from both Jaccard and NCD have a large positive correlation with each other, but a lesser correlation to code and mutation coverage. Similarly diversity has a minor positive correlation to lines of code and the number of test cases.

Main findings for RQ2:

- Code coverage types have strong positive correlations with each other.
- Mutation testing is closely correlated with code coverage.
- Diversity has a minor positive connection to code and mutation coverage.

V. Discussion

This section contains discussions about the results we presented in the previous section.

*1) **Main challenges found**:* The challenges we have faced while conducting this MSR study have mostly stemmed from trying to automate the extraction of test quality metrics, and trying to adjust for multiple different tools.

This created a big hurdle due to the fact that many repositories would require manual adjustments in order to work properly. This is compounded by the issue of interfering Maven plugins. For many of the repositories we could skip plugins that caused issues from executing, but this requires us to reconfigure our tool continuously to account for each plugin that is causing an issue. This is a problem when trying to automate the process, as we can not know beforehand what plugins will interfere.

The extraction time for mutation coverage data has been a big bottleneck. As stated in the results section, this has for the most part been due to LittleDarwin. After running both PITest and LittleDarwin on relatively small repositories, we have shown that PITest has a far faster execution time. It can be assumed that PITest would also run faster on larger repositories, due to the way it prioritises test cases, as explained in results. This makes PITest a more favourable tool to use in future studies. This difference in execution time would nonetheless be interesting to investigate in a separate study to see if it also holds for larger repositories.

*2) **Implications of our result for metrics**:* Despite mutation testing and code coverage being seen as different options for test suite evaluation we have found that they are in fact closely positively correlated. While our results also show the exceptions to this connection from LittleDarwin, the mutation operators for PITest always generate mutation coverage close to that of the code coverage. Mutation operators do not mutate all code rather they find all instances of the code they can perform on, like the ROR operator which only replaces binary relational statements. As such, if a test suite covers all or most of these code snippets which are susceptible to mutation it can achieve great mutation coverage. This was observed in the repository maven-source-plugin which covered many of the mutations (78%) generated by LittleDarwin, while having a low branch coverage of 21%. The opposite is then also possible of a test suite not killing many mutants due to coverage being focused on other parts despite high code coverage.

An increase in code and mutation coverage can be seen from increasing the number of tests according to their positive correlation, similarly an increase in diversity has such a correlation. Thus, adding tests which also are diverse to a test suite can give better performance as measured by code and mutation coverage.

Mutation testing was found to be either too costly to run, as in the case for LittleDarwin, or highly correlated with code coverage, like PITest. Thus, based on these results code coverage should be the metric type to focus on between the two.

A. Threats to validity

Internal validity: Knowing whether or not the tools we use will give us correct and proper results is a threat to our internal validity since we have used third party software such as Jacoco, JMockit, Clover, PITest and LittleDarwin, as well as a Diversity tool we received from a prior bachelor student. However, this threat is mitigated by the fact that all of these tools have been used in prior research studies. [11]–[14]

External validity: The projects used in this study are all written in Java and built using Maven, which poses a threat to generalisability. This is so that our tool could build and test the projects more easily and automatically. Having these common requirements made it more straightforward to find test quality metric extraction tools which were compatible with the projects. In order to mitigate this threat we have made sure that the projects we include in this study are consistent with one another. Only projects with less than 2000 lines of code, that is actual code rather than comments or formatting, were used in this study.

Construct validity: A threat to construct validity is that our choices in what metrics and tools we investigate might not be representative. We will mitigate this by looking at and choosing tools used in previous test-related studies. We will also keep in mind the development status of said tools, i.e. whether or not they are actively being developed.

Conclusion validity: Our conclusions are limited by how correlation was calculated. Ideally we would compare metrics on a test case level, however this was not supported by our tooling so we compared on a test suite level. We mitigate this by selecting repositories with test suites containing 100 or less tests.

For this study we mined 22 repositories, however ideally we would have wanted more than 100 in order to have better statistical significance. Future work aims to expand the analysis, as our tool also works for larger projects, but more time is required to do the study on a larger sample.

VI. Conclusion

Test quality metrics are used to verify the quality of the testing and for measuring the completeness of software testing activities. Test metrics are essential for quantifying how well a software has been tested, however, little research has been done to investigate the correlation between different types of test metrics.

We have conducted an MSR style study where we have mined and gathered test quality data from 22 open-source projects from GitHub. We have created a tool that allows us to mine test quality metrics by running other test quality metric extraction tools. Furthermore, we have utilised and compared 6 different test quality metric extraction tools, as well as documented the challenges we have encountered during the course of this study.

For RQ1 we have documented the main challenges we faced during the course of conducting this study, being Maven plugins interfering, costly execution time and tools not working out of the box.

We have also compared features and behaviour of multiple different test quality metric extraction tools, as can be seen in table 3 and 4.

For RQ2 we have been able to show the correlation between several of the test quality metrics'. Most significant is the strong positive correlation between code coverage and mutation testing, while diversity as measured using Jaccard and NCD has a moderate positive effect on both previous metrics.

Future research aims to extract test coverage metrics on a test case level. This would require developing a new tool, as none of the coverage extraction tools we have utilised allow for extracting coverage on a per test case level. Our research has been limited to Java projects using Maven, with further limitations on the properties of these projects, mainly the structure being single module projects. Expanding the set of allowed repositories could yield new and interesting results. However, this may come at the cost of having to configure tools and time to run mutation testing on larger projects. Furthermore, research can be expanding to other programming languages.

REFERENCES

[1] M. Shahid; S. Ibrahim. An evaluation of test coverage tools in software testing. In: 2011 International Conference on Telecommunication Technology and Applications Proc. of CSIT. 2011.

[2] V. Antinyan, J. Derehag, A. Sandberg and M. Staron, "Mythical Unit Test Coverage," in IEEE Software, vol. 35, no. 3, pp. 73-79, May/June 2018, doi: 10.1109/MS.2017.3281318.

[3] P. S. Kochhar, T. F. Bissyandé, D. Lo and L. Jiang, "An Empirical Study of Adoption of Software Testing in open-source Projects," 2013 13th International Conference on Quality Software, 2013, pp. 103-112, doi: 10.1109/QSIC.2013.57.

[4] A. E. Hassan, "The road ahead for Mining Software Repositories," 2008 Frontiers of Software Maintenance, 2008, pp. 48-57, doi: 10.1109/FOSM.2008.4659248.

[5] K. Alemerien, K. Magel, "Examining the Effectiveness of Testing Coverage Tools : An Empirical Study," International Journal of Software Engineering and Its Applications, vol.8, no.5, pp.139-162, 2014.

[6] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, M. Harman, Chapter Six - Mutation Testing Advances: An Analysis and Survey, Editor(s): Atif M. Memon, Advances in Computers, Elsevier, Volume 112, 2019, Pages 275-378, ISSN 0065-2458, ISBN 9780128151211. https://doi.org/10.1016/bs.adcom.2018.03.015.

[7] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. 2014. Are mutants a valid substitute for real faults in software testing? In ¡i¿Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering¡/i¿ (¡i¿FSE 2014¡/i¿). Association for Computing Machinery, New York, NY, USA, 654–665. DOI:https://doi.org/10.1145/2635868.2635929.

[8] R. Feldt, R. Torkar, T. Gorschek and W. Afzal, "Searching for Cognitively Diverse Tests: Towards Universal Test Diversity Metrics," 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, 2008, pp. 178-186, doi: 10.1109/ICSTW.2008.36.

[9] P. M. S. Bueno, W. E. Wong, and M. Jino. 2007. Improving random test sets using the diversity oriented test data generation. In Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007) (RT '07). Association for Computing Machinery, New York, NY, USA, 10–17. DOI:https://doi.org/10.1145/1292414.1292419.

[10] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis and N. Malevris, "Analysing and Comparing the Effectiveness of Mutation Testing Tools: A Manual Study," 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2016, pp. 147-156, doi: 10.1109/SCAM.2016.28.

[11] R. P. A. d. Araujo and M. L. Chaim, "Data-Flow Testing in the Large," 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, 2014, pp. 81-90, doi: 10.1109/ICST.2014.19.

[12] T. L. Alves and J. Visser, "Static Estimation of Test Coverage," 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, 2009, pp. 55-64, doi: 10.1109/SCAM.2009.15.

[13] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. Jiang. 2018. An automated approach to estimating code coverage measures via execution logs. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/3238147.3238214

[14] A. Parsai, A. Murgia, S. Demeyer. LittleDarwin: A Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems. 7th International Conference on Fundamentals of Software Engineering (FSEN), Apr 2017, Teheran, Iran. pp.148-163, ff10.1007/978-3-319-68972-2-10ff. ffhal-01760857f

[15] M. C. Denke. "Using diversity measures to investigate the trade-off of redundant test cases: An experimental study," B.S. thesis, Dept. Comp. Sci., Uni. Gothenburg, Chalmers Uni., Gothenburg, Sweden, 2021.

[16] F. G. De Oliveira Neto, R. Feldt, L. Erlenhov and J. B. D. S. Nunes, "Visualizing Test Diversity to Support Test Optimisation," 2018 25th Asia-Pacific Software Engineering Conference (APSEC), 2018, pp. 149-158, doi: 10.1109/APSEC.2018.00029.

[17] A. E. Hassan, "The road ahead for Mining Software Repositories," 2008 Frontiers of Software Maintenance, 2008, pp. 48-57, doi: 10.1109/FOSM.2008.4659248.