2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)

Evaluating a test automation decision support tool

Kesina Baral
Computer Science, PhD
George Mason University
Fairfax,USA
kbaral4@gmu.edu

Rasika Mohod
Computer Science, MS
George Mason University
Fairfax,USA
rmohod@gmu.edu

Jennifer Flamm
The MITRE Corporation
McLean,USA
jflamm@mitre.org

Seth Goldrich
The MITRE Corporation
McLean,USA
sgoldrich@mitre.org

Paul Ammann, PhD
Computer Science
George Mason University
Fairfax,USA
pammann@gmu.edu

***Abstract*— Goldrich and Flamm developed the MITRE Automated Test Decision Framework (ATDF) to help MITRE government sponsors (and, via sharing on GitHub, development organizations in general) move from manually tested legacy software towards automated test, continuous integration, continuous deployment, and, ultimately, DevOps. Often such legacy systems comprise multiple components with manual test procedures. The objective of the empirical study described in this paper is to determine whether ATDF usefully ranks components with respect to Return on Investment (ROI) when introducing automated tests. ROI is simply the ratio of profit to cost. When adding automated tests, what will be the profit that these tests will carry? What is the cost or level of effort to engineer a sufficient set of automated tests? Our evaluation approach models ROI using static defect counts identified by SonarLint and estimated cost to complete testing. We found positive Pearson correlations between normalized ATDF rankings versus the normalized rankings of our evaluation approach. We reject the null hypothesis that there is no correlation between the two rankings.**

Keywords—test automation, legacy systems, software components

I. Introduction

MITRE developed ATDF [1] to aid project owners and technical staff in managerial decisions pertaining to the addition of test automation in a software project with limited resources. ATDF takes advantage of the fact that many projects are comprised of components and test automation can be implemented on a component by component basis. ATDF takes various system characteristics and software engineering metrics of components in a system and computes an ordering of components to automate based on expected ROI from test automation.

To investigate the validity of ATDF rankings, MITRE approached George Mason University to develop a validation approach. This paper reports on the results of that collaboration. The collaboration evaluated multiple legacy software projects to measure the consistency of ATDF's *predicted* ROI rankings by component with a model of *actual* ROI rankings by component. In a retrospective analysis of a code base, if test automation increases after some point, what is the measured value of those tests? What was the level of effort to engineer automated tests?

We selected a set of open-source projects and divided each project into components based on the evident structure, e.g., Java packages. Projects were filtered to only include those that displayed an increase in test automation across, at least a temporal subset of, the historical Software Development Life-Cycle (SDLC). After identifying the ATDF factors applicable to such open-source projects, we executed ATDF against the "before" version of each project, exploiting a limited, but robust, set of open-source tools to capture the necessary metrics. The actual ROI is the profit realized by the increase in test automation between the "before" and "after" versions of the project codebase. One tangible measure of the profit of automated software test is improved code *quality*. We measure quality with the SonarLint tool [16]. Given the introduction of automated test to a component, how much did the corresponding source code quality improve, and what was the cost borne to realize that quality increase?

A. Contributions

- Adapted ATDF to open source projects.
- Defined evaluation metric based "quality approach" based on static defect counts as independent ROI measure.
- Evaluated ATDF with quality approach metric over seven open source projects.
- Report positive correlations between ATDF rankings and our quality approach independent ROI measure.

978-1-7281-0888-9/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSTW.2019.00034

IEEE computer society

B. *Organization of the paper*

Section II overviews ATDF and defines its adaptation to open source projects. Section III defines the methodology. Section IV describes the data we collected. Section V gives results. Section VI details threats. Section VII lists related works. Section VIII concludes the paper.

II. Background

A. *ATDF Overview*

ATDF is an open source decision aid, developed by MITRE. ATDF is available on GitHub [1], including the tool itself and a complete description. We provide a summary here. When resources are constrained, ATDF is used to prioritize where to apply those resources when adding test automation to a software project. ATDF works as depicted in Fig. 1. First, facts regarding the software are fed as input to the ATDF, next ATDF performs its ROI computations, and finally the test automation ROI rankings are output.

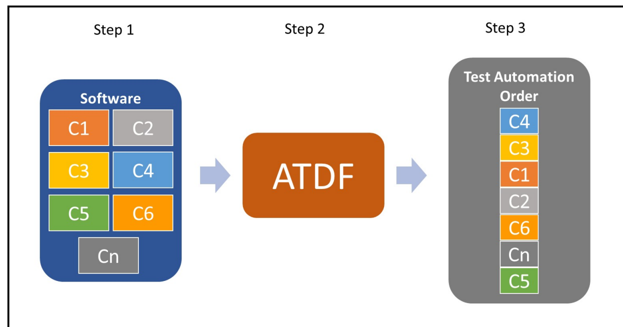

Fig. 1. ATDF Overview

The inputs provided to ATDF are: (1) a small set of system-wide characteristics, and (2) a larger set of measurable factors applied to each software component. The system-wide characteristics are somewhat subjective in nature, with half of these characteristics describing the dynamics of the development team, and the underlying ATDF computations are only minimally affected by these system-wide characteristics. Thus, we applied the "neutral" or "neither" answer for each of these characteristics. Per the larger set of component factors, ATDF prescribes to only consider those for which data is available. We applied ATDF in this study, using the seven empirically measurable characteristics that could be calculated against open source projects' source code. Those characteristics are described in TABLE I.

The output obtained from ATDF is a ranking of components by expected ROI value. ATDF suggests a prioritization, or ordering, by components, to introduce automated tests to maximize benefit within available resources. As ATDF output is always obtained in terms of ranking of software components, an assumption here is that any software project on which ATDF is executed can be decomposed into mutually exclusive components.

Funding provided by MITRE Innovation Program.

TABLE I. ATDF CHARACTERISTICS FOR OPEN SOURCE

Row#	Characteristic	Description
1	Test Coverage Metrics	Test Coverage measures the line or statement test coverage.
2	Volatility	Volatility is the rate of change over time. This can include the expected rate, density, and extent of changes.
3	Modularity	In a modular design, the functionality is divided into independent, typically small and simple, pieces or modules.
4	Self-Descriptiveness	Self-descriptive software provides the naming constructs, comments, and descriptions in the code to facilitate the analysis and understanding of the code.
5	Design Simplicity	Simplicity relates to the readability and traceability of the code.
6	Anomaly Control	Anomaly control measures the sufficiency of the error handling and exception processing.
7	Independence	Independence implies that the software is not tied to any specific host environment which would make it difficult or impossible to migrate, evolve, or enhance the software.

B. *Validation overview*

The ATDF is useful if its predictions are an improvement on the state-of-the-art, which right now is random component ranking. Therefore, we model ROI as the actual, realized profit-over-cost and compare ATDF predictions against a baseline early in the project history, with ROIs measured due to the automated test increases later in the project history.

III. Methodology

A. *Criteria for open source project selection*

The following criteria were applied to project selection:

1. The open source project must be decomposable into mutually exclusive components. We define components as pre-existing project division, e.g., packages or classes present.
2. The open source project should have a substantial SDLC history in terms of release versions to enable us to pinpoint the earliest release version (V_o), version before significant test introduction (V_b), version after significant test introduction (V_a) and latest version (V_n) as shown in Fig. 2.
3. The SDLC of the project should show an increase in test-code-to-source-code ratio from earliest release version (V_o) to latest release version (V_n) of software.

4. There must be at least 20% of SDLC history between V_o and V_b and between V_a and V_n, as shown in Fig. 3. The SDLC history between V_o and V_b is used to calculate Characteristic 2, Volatility, for ATDF. The SDLC history from V_a to V_n supported calculating defect density, a metric for which we do not report results.

.

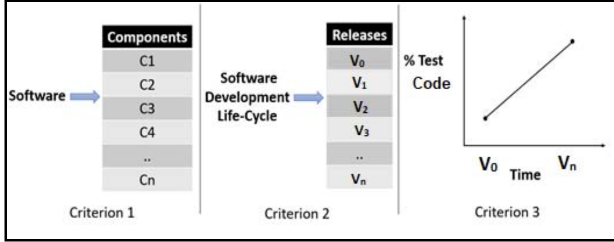

Fig. 2. Project Selection Criteria

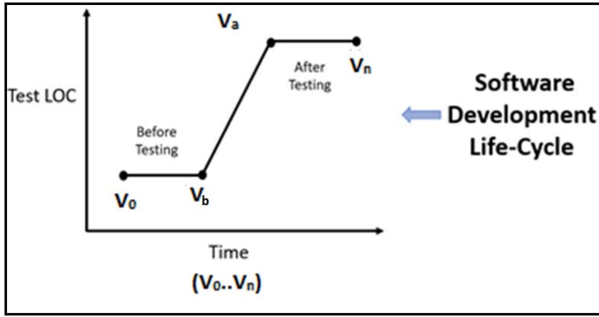

Fig. 3. Final Sample Project SDLC Behavior

B. Applying ATDF to open source projects

We considered twenty-four open source projects which are listed in Appendix A, iteratively refining the selection to those that met the criteria discussed above. The team completed data collection for seven open source software projects through the duration of this study.

To execute ATDF against a historical baseline of each project, each of the factors previously described in TABLE I. was specifically measured. TABLE II. gives details regarding the specific measures, calculations and tools that were used to capture the metrics. Regarding the Test Coverage metric, our goal was to capture actual test coverage—the percent of the source code that is covered by the existing automated test code in the project. That proved to be difficult for some early project versions given build dependencies on obsolete versions of compilers and other dependent open source libraries, so we also collected test-code-to-source-code ratio as a proxy for test coverage. This proved to be a valid proxy for ATDF's test coverage metric, and that discovery is an ancillary finding of this study.

TABLE II. ATDF CHARACTERISTICS AND METRICS

Row#	Characteristic	Specific Metric
1	Test Coverage Metrics	Line coverage percentage collected using Cobertura tool.
1'	Test Code Metrics	Git command was used to calculate SLOC and test LOC
2	Volatility	SLOC changed in all commits, including both addition and deletion, from V_o to V_b was collected using git command
3	Modularity	Calculated as ratio of coupling violations to SLOC. Coupling violation was collected using pmd tool.
4	Self-Descriptiveness	Ratio of Javadoc Comments to SLOC. Javadoc comment was collected using regular expression.
5	Design Simplicity	Calculated as mean cyclomatic complexity, which was collected using CAT Tool.
6	Anomaly Control	Calculated as ratio of anomaly word count to SLOC. Anomaly word count was collected using mgrep scripts for pattern matching.
7	Independence	Calculated as dependency word count per KSLOC. Dependency word count was collected using mgrep scripts for pattern matching .

C. Quality Approach to empirical ROI calculation

ATDF predicts relative ROI for automating the testing in each component. ATDF was applied, using the specific metrics presented in the previous section. To assess the validity of ATDF's rankings, we must evaluate each component later in the SDLC. How much profit (that is, value or benefit) was realized by any additional test automation that was added? What was the cost associated with that additional test automation?

To define the benefit, consider: how much better is the software after V_a than at or before V_b? Or, what is the delta-***goodness*** from V_b to V_a? For purposes of verifying ATDF, we define that goodness as software "quality". Software code components can be objectively assessed per their quality, based on the number of statically discovered defects or issues. We applied a static analysis tool, SonarLint, to identify issues in each component's V_b and V_n. The benefit is the reduction in issue density as discovered by the static analysis tool (We chose to take measurements at V_n, instead of V_a, because that is consistent with measuring defect density, a metric for which we gathered data, but for which we do not report results). This measure is independent of any of the component characteristics utilized by ATDF.

As ATDF ranks components by predicted ROI, we must capture the ***relative*** cost to engineer a sufficient set of automated tests for each component. That relative cost is a factor of size and difficulty. How big is the job to engineer the test automation for each component at V_b? How difficult or complex will it be? This size of the job is based on the volume of source code for which we must write automated tests (SLOC at V_b), but can be reduced by the amount of test code already in the baseline at V_b. We use a proxy for the relative difficulty. Apply an automated test "generator" to V_b,

then execute the auto-generated tests. Auto-generated tests are insufficient as actual test automation artifacts because they lack a known *truth* against which to assert correctness. We assert that the source code coverage that they achieve is a proxy for this relative difficulty. This study utilized the EvoSuite [15] tool to generate automated unit tests and Cobertura [19] to measure coverage.

For each component, relative ROI is measured as: (Benefit/Cost) * Factor

Benefit = Delta-quality, defined as [total SonarLint {BLOCKER, CRITICAL or MAJOR} issues at V_n] – [total SonarLint {BLOCKER, CRITICAL or MAJOR} issues at V_b] (issues scaled per KSLOC)

Cost = Estimated cost to complete code coverage at V_b
= [Size] * [Relative Difficulty]
= [(1 - %Test Coverage (V_b)) * SLOC (V_b)] *
[1 - % Test Coverage attained by EvoSuite (V_b)]

Factor = Fraction of the Test Cost that was borne from V_b to V_n, defined as %Test Coverage (V_n) - %Test Coverage (V_b)

We calculated the normalized measures of ROI from the model and compared it with the normalized rankings from ATDF using Pearson correlation.

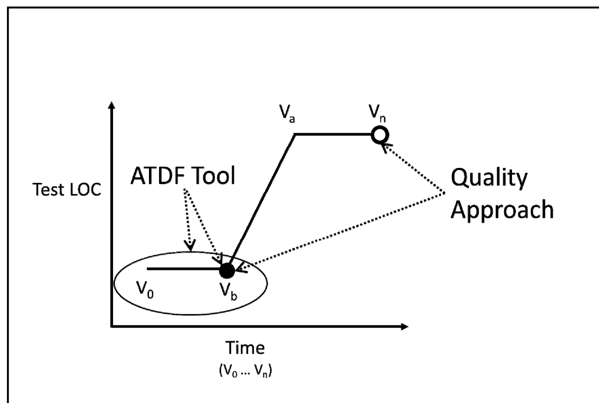

Fig. 4. Validation Approach over SDLC

1) ATDF Evaluation

We collected data for ATDF evaluation at V_b (shown as solid dot in Fig. 4) and considered software volatility over V_0 to V_b period of SDLC (shown as the ellipse in Fig. 4) to get a historical volatility as of the V_b snapshot in time.

2) Quality Approach for ROI Benefit

We collected data at version V_b and V_n (shown by outlined dot in Fig. 4). We used SonarLint [16], an open source platform to perform static analysis of code, which identifies issues at various levels of severity: blocker, critical, major, minor and warnings. We used the difference in major, blocker and critical issues between V_b and V_n as delta-quality value in the ROI computations. With these calculations, we obtained ordering of software components in terms of realized ROI obtained in after-testing period of SDLC, which was compared with ordering predicted by ATDF tool.

3) ROI Cost Calculation

We collected SLOC counts for all components via command-line git, generated tests for the components via EvoSuite [15], and measured coverage achieved by those EvoSuite-generated tests via Cobertura [19] at V_b. Where EvoSuite test generation failed for a given component at V_b, we used the average of the project's other components' EvoSuite-generated-test coverage for the component. Where EvoSuite test generation failed across an entire project at V_b, the relative Cost calculation for each component is reduced to the SLOC count only.

IV. Description Of Data

A. Sample Project - OpenNLP

OpenNLP [14] is decomposable into several components based on its Java package structure. Therefore, it satisfies the first criteria in Section III.A. It also has substantial release history and hence, meets the second criteria. Among the components, we considered the four that are present throughout the SDLC. Other components of OpenNLP were not considered because they weren't available for data collection at all the necessary points in the revision history. The project has a rise in %test code from 13% at V_o to 18% at V_n as presented in Fig. 5. This satisfies the third criteria.

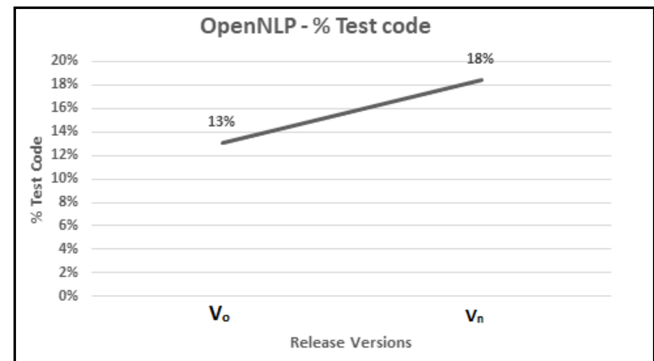

Fig. 5. OpenNLP Test Code Distribution

To satisfy the fourth criterion in III.A we chose release versions opennlp-1.7.1 as V_b and opennlp-1.8.0 as V_a. opennlp-1.7.0 is the earliest version that would successfully compile, and thus it was set as initial version (V_o), while the final version (V_n) was opennlp-1.8.4. Fig. 6 shows these release versions with their test LOC.

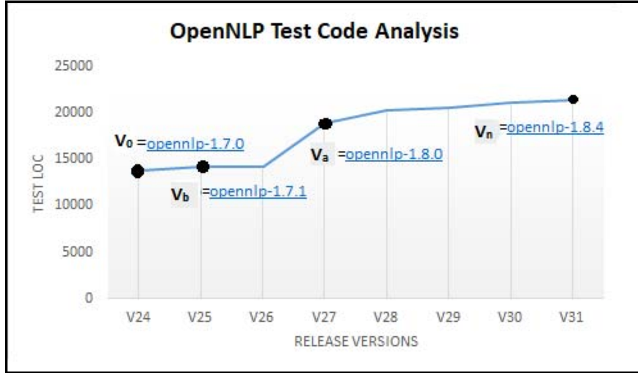

Fig. 6. OpenNLP Test Code Analysis

We computed ATDF at V_b and applied quality model of ROI to $V_b \rightarrow V_n$. The ROI predicted by ATDF, and that measured by the quality model, are both normalized to a range of [1,100], given in Fig. 7.

OpenNLP Components	ATDF	Quality Approach	Test Code Change
Component 1: opennlp-brat-annotator	88	97	67
Component 2: opennlp-morfologik-addon	1	1	44
Component 3: opennlp-tools	100	100	64
Component 4: opennlp-uima	16	99	25
Pearson correlation:		0.67	

Fig. 7. OpenNLP Rankings from ATDF tool and Quality Approach for ROI

V. RESULT

We analyzed ATDF ranking with respect to the quality approach to independently calculate ROI. The more similar the ATDF rankings are to the independent ROI rankings, the more valuable ATDF is. Hence, correlation between the rankings is a useful metric to judge the validity of ATDF. Because the normalized values provide information beyond simple ordinal rankings, Pearson is an appropriate correlation measure.

For some components, EvoSuite test generation failed, largely due to dependence on obsolete versions of other open source libraries and obsolete compiler versions. As described in III.C.3) ROI Cost Calculation, the cost calculation for the Quality Approach ROI measure was modified in these cases to use either: the average coverage for the projects' other components when that coverage data was available, or to not use coverage at all when unavailable.

As shown in Appendix B, ATDF correlations with the quality approach varied widely within each project (-0.76 to 0.8). But this analysis includes components with little or negative change in test code and hence is not suitable for drawing conclusions. The number of components within each project is also too small for the results to be significant.

To remedy these issues, we analyzed correlations across all projects with respect to the amount of increase in test code during their SDLC. Note again that both ATDF rankings and Quality Approach ROI rankings are normalized to a [1,100] range, which enables comparison across all projects. This analysis is shown in Fig. 8.

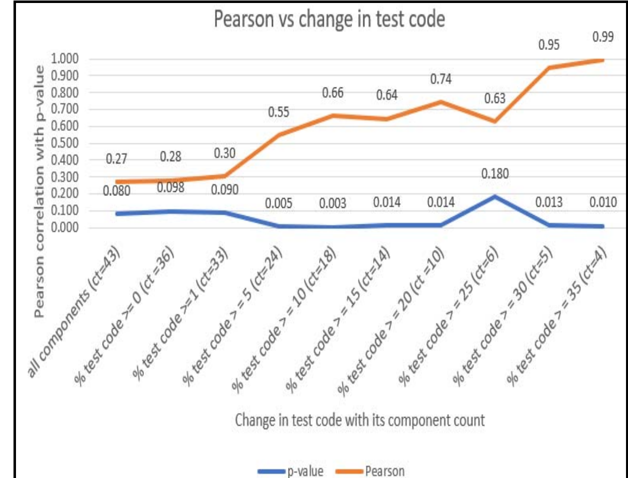

Fig. 8. Pearson Correlation vs Change in Test Code

The x-axis in Fig. 8 shows the change in test code and the number of components corresponding to the change. The y-axis shows Pearson correlation in orange and p-value in blue. For example, at the point on the x-axis "% test code >=10(ct=18)", the chart shows results for the 18 components whose change in test-to-source-code ratio was greater than or equal to 10%–Pearson value is 0.66 with p-value 0.003. To assess the impact of the missing EvoSuite coverage values, we computed all results with the simplified calculation for ROI Cost which depends only on SLOC. For that same data point, "% test code >=10(ct=18)", the resulting Pearson correlation reduces from 0.66 to 0.58 with 95% confidence. Thus, the missing EvoSuite coverages do not pose a threat to validity; the lower correlation is unsurprising given the lower fidelity cost calculation.

As shown in Fig. 8, the p-values are below 0.01 for increased test code ratios of greater than or equal to 5% or 10%. For smaller values of test code change, the Pearson correlation is simply too low for significance. For higher values, the number of components is too low for significance. However, for the two values where p-value is less than 0.01 we can reject the null hypothesis at 99% confidence level that ATDF and quality approach are independent. Hence, we conclude that the quality approach provides a positive evaluation of ATDF.

VI. THREATS

We collected data on only seven open source projects. Larger data sets would be required to reduce the uncertainty inherent in this noisy data.

In relating current version to prior version, there is a threat that the history may not be linear. We used TortoiseGit [18] to analyze version history and chose a linear path leading to the current version. Since the CAT [23] tool requires Java 8, it may undercount on project versions using Java 7 or older. We don't believe the undercounts introduced a bias between components of same version. In our experiment, we restricted our attention to maven-based projects, which may have introduced a bias.

To maintain independence between ATDF and our evaluation metric we did not use a standard measure (e.g., cyclomatic complexity) to estimate the difficulty of completing test automation. Instead we used the coverage achieved by the test code generated by EvoSuite.

Cobertura couldn't always produce coverage value due to Java versioning and compilation issues, hence in our experiment we used percentage of test code as a proxy for test coverage. To evaluate the consequence of this decision, we compared ATDF ranking with test code vs ATDF ranking with test coverage. For projects where test coverage collection is complete, the correlation is above 0.91. These strong correlations provide evidence that test code is a good proxy for test coverage when applying ATDF to open source projects. The quality approach metric has not been separately validated, either with or without the EvoSuite component.

VII. Related work

Garousi et al. [2] suggest qualitative factors to drive the decisions regarding when and what to automate at the test case level. Couto et al. [3] proposed a tool to investigate the relationship between internal quality metrics and bugs. Their tool, BugMaps-Granger, analyzed source code properties that are more likely to cause bugs. Similarly, Yamashita et al. [4] examined the relation of size and complexity of code to its bugginess. Li et al. [5] showed how the extent of modification of a source code file has a positive correlation with its defect density. Test automation has also been used to measure software quality in many instances. Bach et al. [6] analyzed real bug-reports and bug-fixes and found a strong signal that test-covered code contains smaller number of future bugs compared to code uncovered by tests. A strong relationship between test coverage and field related problems was found by Mockus et al. [7]. Through their multi-case study, Mockus et al. observed that higher test coverage leads to detection of more flaws and fixing them leads to better software quality. Similar test code and defect density relationship was found in research by Athanasiou et al. [8]. Their study introduced a model which assessed test code quality by combining three main benefits of automated testing: code completeness, effectiveness, and maintainability. Their results revealed that good test code quality positively influences throughput and defect handling performance. Gren et al. [9] tried to find a correlation between unit testing and number of defects in the codebase. Their research further aims to help developers in understanding how to best allocate their resources to testing. Swart [10] and his research group measured the effect of reengineering and unit testing code on the number of fixed bugs. They compared the predicted and actual number of bugs for a component, after reengineering and unit testing it. They predicted the bug-proneness of components, and successfully ranked them by feasibility. Their results indicated that the number of bug fixes decreases after a bug-prone component has been reengineered and covered with unit tests. Another experiment was conducted by Tengeri et al. [11] on four open source systems' test suites, to compare them with respect to code coverage and mutation score. Tengeri et al. demonstrated situations where code coverage and mutation score are sufficient indicators of expected defect density. The relation to our work is that ATDF relies in part on standard software metrics and our evaluation is in terms of defect. Hence, we expect to see the relationship between the two.

There has been research in identifying and categorizing defects through code changes and commit history logs. GitcProc a tool developed by Casalnuovo et al. [13] is based on regular expressions and source code blocks, which analyzes GitHub [17] project history, including fine-grained source code information and development time bug fixes. This tool searches for bug-fix related words such as error, bug, defect, and fix within the commit message to classify it as a bug-fix commit. We implemented a similar keyword-based approach for analyzing commit logs when computing the two ATDF metrics: Self-Descriptiveness and Anomaly Control.

Hoffman's research [12] measured computing cost and benefits from test automation. His research described some financial, organizational, and test effectiveness impacts observed when test automation is introduced in the system.

We examined research studies to extract different open source projects used in their case studies to assess defect density and other software metrics. All but one of the 24 projects that we considered were selected from references.

VIII. Conclusion

The purpose of this paper is to evaluate ATDF for ranking legacy components versus expected ROI from test automation. We used a set of open source projects and defined ROI as lint style defect count vs cost of test automation.

We calculated Pearson correlation between ATDF rankings and our independent ROI measure. We found moderate to strong correlation between the ATDF and the quality approach ROI measure, for those components with increase in test code greater than ten percent. Hence, the quality approach supports the ROI calculations produced by ATDF for the studied projects.

References

[1] ATDF, https://github.com/mitre/atdf/

[2] V. Garousi and M.V. Mäntylä, *When and what to automate in software testing? A multi-vocal literature review*, Information and Software Technology, 2016.

[3] Cesar Couto, Marco Tulio Valente, Pedro Pires, Andre Hora, Nicolas Anquetil and Roberto S Bigonha, *BugMaps-Granger: a tool for visualizing and predicting bugs using Granger causality tests*, Journal of Software Engineering Research and Development, 2014.

[4] Kazuhiro Yamashita, Changyun Huang, Meiyappan Nagappan, Yasutaka Kamei, Audris Mockus, Ahmed E. Hassa, and Naoyasu Ubayashi, *Thresholds for Size and Complexity Metrics: A Case Study from the Perspective of Defect Density*, IEEE International Conference on Software Quality, Reliability and Security, 2016.

[5] Zengyang Li, Peng Liang, Bing Li, *Relating Alternate Modifications to Defect Density in Software Development*, IEEE/ACM 39th IEEE International Conference on Software Engineering Companion, 2017.

[6] Thomas Bach, Artur Andrzejak, Ralf Pannemans, and David Lo, *The Impact of Coverage on Bug Density in a Large Industrial Software Project*, ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2017.

[7] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong, *Test Coverage and Post-Verification Defects: A Multiple Case Study*, Third

IEEE International Symposium on Empirical Software Engineering and Measurement, 2009.

[8] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser Member, and Andy Zaidman, *Test Code Quality and Its Relation to Issue Handling Performance*, IEEE Transactions on Software Engineering, 2000.

[9] Lucas Gren and Vard Antinyan, *On the Relation Between Unit Testing and Code Quality*, IEEE 43rd Euromicro Conference on Software Engineering and Advanced Applications. 2017.

[10] Jerry Swart, *Selecting Bug-prone Components to Study the Effectiveness of Reengineering and Unit Testing*, Masters Thesis - Software Engineering Research Group, Delft University of Technology

[11] Dávid Tengeri, László Vidács, Árpád Beszédes, Judit Jász, Gergõ Balogh, Béla Vancsics, and Tibor Gyimóthy, *Relating Code Coverage, Mutation Score and Test Suite Reducibility to Defect Density*, IEEE International Conference on Software Testing, Verification and Validation Workshops, 2016.

[12] Douglas Hoffman, Cost Benefits Analysis of Test Automation, Software Quality Methods, LLC, 1999.

[13] Casey Casalnuovo, Yagnik Suchak, Baishakhi Ray, and Cindy Rubio-González, *GitcProc: A Tool for Processing and Classifying GitHub Commits,* In Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, (ISSTA'17-DEMOS), 2017.

[14] OpenNLP, https://github.com/apache/opennlp

[15] EvoSuite, http://www.evosuite.org/

[16] SonarLint, https://www.sonarlint.org/

[17] GitHub, https://github.com/

[18] TortoiseGit, https://tortoisegit.org/

[19] Cobertura, http://cobertura.github.io/cobertura/

[20] F. Thung, X. D. Le and D. Lo, "Active Semi-supervised Defect Categorization," 2015 IEEE 23rd International Conference on Program Comprehension, Florence, 2015, pp. 60-70.

[21] J. Flisar and V. Podgorelec, "Enhanced Feature Selection Using Word Embeddings for Self-Admitted Technical Debt Identification," *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Prague, 2018, pp. 230-233.

[22] P. S. Kochhar, F. Thung, D. Lo and J. Lawall, "An Empirical Study on the Adequacy of Testing in Open Source Projects," *2014 21st Asia-Pacific Software Engineering Conference*, Jeju, 2014, pp. 215-222.

[23] CAT Tool https://www.mitre.org/research/technology-transfer/technology-licensing/software-quality-assurance-evaluation-sqae

[24] Greg Barish, Matthew Michelson, and Steven Minton, *Mining commit log messages to identify risky code,* Int'l Conf. Artificial Intelligence, 2017.

[25] J. Flisar and V. Podgorelec, "Enhanced Feature Selection Using Word Embeddings for Self-Admitted Technical Debt Identification," *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Prague, 2018, pp. 230-233.

[26] Defects4j https://github.com/rjust/defects4j

Appendix

A. Projects Considered.

	Project	Source	URL
1.	Accumulo	[5]	https://accumulo.apache.org/
2.	Closure-compiler	[26]	https://developers.google.com/closure/compiler/
3.	Cobertura	[6][7]	http://cobertura.github.io/cobertura/
4.	Commons Lang	[7]	https://commons.apache.org/proper/commons-lang/
5.	Commons Math	[26]	http://commons.apache.org/proper/commons-math/
6.	Eclipse JDT Core	[2]	https://www.eclipse.org/jdt/core/
7.	Guava	[21]	https://github.com/google/guava
8.	Hadoop	[22]	https://hadoop.apache.org/
9.	Jacoco	[22]	https://www.eclemma.org/jacoco/
10.	Jfreechart	[26]	http://www.jfree.org/jfreechart/
11.	JodaTime	[26]	https://www.joda.org/joda-time/
12.	Junit4	[6][7]	https://junit.org/junit4/
13.	Mahout	[20]	https://github.com/apache/mahout
14.	Mapdb	[11]	https://github.com/jankotek/mapdb
15.	Mockito	[26]	https://site.mockito.org/
16.	Netty	[11]	https://github.com/netty/netty
17.	OpenNLP	[20]	https://github.com/apache/opennlp
18.	Orientdb	[11]	https://github.com/orientechnologies/orientdb
19.	Oryx	[11]	https://github.com/OryxProject/oryx
20.	Spring Boot	[25]	https://spring.io/projects/spring-boot
21.	ZXing	[9]	https://github.com/zxing/zxing
22.	Tomcat	[24]	http://tomcat.apache.org/
23.	OpenMQ	-	https://javaee.github.io/openmq/
24.	Lucene	[20]	http://lucene.apache.org/

B. Data

	ATDF	Quality Approach	Test code change	Intra-project pearson correlation
Accumulo				**0.80**
core	83	84	1	
examples	79	84	-8	
minicluster	27	1	-26	
proxy	100	84	-3	
start	1	2	6	
trace	45	100	-7	
server	84	84	**11**	
Closure				**0.33**
sourcemap	1	82	0	
jscomp	100	100	5	
rhino	36	1	-4	
Lucene				**0.33**
highlighter	7	74	4	
memory	2	1	**24**	
misc	60	30	**16**	
queries	1	32	**19**	
queryparser	100	54	3	
spatial	54	100	**14**	
OpenNLP				**0.67**
Component 1: opennlp-brat-annotator	88	97	**67**	
Component 2: opennlp-morfologik-addon	1	1	**44**	
Component 3: opennlp-tools	100	100	**64**	
Component 4: opennlp-uima	16	99	**25**	
Oryx				**-0.76**
oryx-app	41	1	1	
oryx-app-api	1	100	4	
oryx-app-common	43	17	1	
oryx-app-mllib	100	8	2	
oryx-app-serving	76	9	4	
OrientDB				**0.43**
Component 1: Client	33	93	6	
Component 2: Core	56	88	4	
Component 3: Distributed	53	85	6	
Component 4: GraphDB	1	1	**21**	
Component 5: Object	89	100	**13**	
Component 6: Server	76	88	**12**	
Component 7: Tools	100	39	**15**	
Netty				**0.07**
buffer	21	100	6	
codec	1	96	5	
codec-http	100	97	**24**	
codec-socks	53	98	0	
common	32	81	**22**	
example	93	88	**50**	
handler	38	1	**32**	
transport	77	99	**15**	
transport-rxtx	61	100	0	
transport-sctp	63	89	0	
transport-udt	1	100	-2	
Inter-project pearson correlation				**0.27**