Relating Code Coverage, Mutation Score and Test Suite Reducibility to Defect Density

Dávid Tengeri* László Vidács*, Árpád Beszédes†, Judit Jász†,
Gergő Balogh†, Béla Vancsics† and Tibor Gyimóthy†
*MTA-SZTE Research Group on Artificial Intelligence
†Department of Software Engineering
University of Szeged, Szeged, Hungary
{dtengeri,lac,beszedes,jasy,geryxyz,vancsics,gyimothy}@inf.u-szeged.hu

***Abstract*—Assessing the overall quality (adequacy for a particular purpose) of existing test suites is a complex task. Their code coverage is a simple yet powerful attribute for this purpose, so the additional benefits of mutation analysis may not always justify the comparably much higher costs and complexity of the computation. Mutation testing methods and tools slowly start to reach a maturity level at which their use in everyday industrial practice becomes possible, yet it is still not completely clear in which situations they provide additional insights into various quality attributes of the test suites. This paper reports on an experiment conducted on four open source systems' test suites to compare them from the viewpoints of code coverage, mutation score and test suite reducibility (the amount test adequacy is degraded in a reduced test suite). The purpose of the comparison is to find out when the different attributes provide additional insights with respect to defect density, a separately computed attribute for the estimation of real faults. We demonstrate that in some situations code coverage might be a sufficient indicator of the expected defect density, but mutation and reducibility are better in most of the cases.**

***Index Terms*—Mutation analysis, code coverage, defect density, test adequacy criteria, test suite reduction.**

I. Introduction

Measurement and prediction of test suite adequacy is a constant debate topic both among testing practitioners and researchers. The level of adequacy for a particular set of test requirements might be seen as a quality indicator of the underlying test suites, hence it is of high importance. In white-box testing, a simple yet highly useful quality attribute is code coverage, but it has also been shown that code coverage is not always sufficient to predict the defect detection capabilities of test suites [1]. As a fault-based technique, mutation analysis [2] has been introduced with the hope to offer additional insight in this respect. However, it is still not precisely known in what situations would mutation analysis provide more reliable estimate of the ability to find real faults, or in fact, when is code coverage a good enough estimator itself. Furthermore, mutation analysis is still widely seen as a very expensive technique and that there are a lot of barriers to its wider adoption in industrial practice [2], [3].

At the same time, the demand from the industry for more reliable mutation testing methods and tools is growing, and fortunately there are promising trends to satisfy this need [4]. For instance, in the Agile High Assurance methodology, Binder proposes a 3-way Coverage Definition of Done, which includes code coverage, clean report from static analyzers, and minimum mutation score [5]. Also, there are mutation tools which can handle bigger industrial projects as well, for instance the PIT tool [6].

However, the testing community still needs to find out in what situations is mutation testing worth the effort. And we can go further: can we use further adequacy criteria in addition to code coverage and mutation scores to get better predictors of test suite effectiveness in finding real faults? Our test suite assessment methodology [7], [8], [9] tries to involve different adequacy criteria and combine them into a model for describing different quality aspects of test suites. The model currently mostly relies on code coverage based information, but involvement of additional criteria is an ongoing effort.

This paper is an early report on a combined use of different test suite adequacy criteria in an experiment which we conducted on the test suites of four non-trivial open source systems. The goal was to compare the criteria to the actual faults reported for the projects. For the criteria we used code coverage ratio, mutation score, and test suite reducibility. For the latter, we considered the amount the test suite can be reduced without degradation in code coverage and mutation score. For the estimation of the actual faults in the systems we used a Defect Density metric computed from the respective defect reporting systems (as the confirmed number of defects per system size). In this phase, we used two mutation operators with manually verified mutants. Our results are promising: we identified indications that code coverage alone is rarely a good indicator of the expected number of defects (hence test suite quality), and that the other two aspects provide useful additional insights.

The next section of the paper gives more details about the goals of the study and the methods we used, while Section III reports on the results. In Sections IV and V, we overview relevant related work and threats to validity, before concluding in Section VI.

II. Goals and Method

Our long term goal is researching test suite quality assessment methods, which involve various test adequacy criteria, and verify them on existing test suites. Our goal is to compare

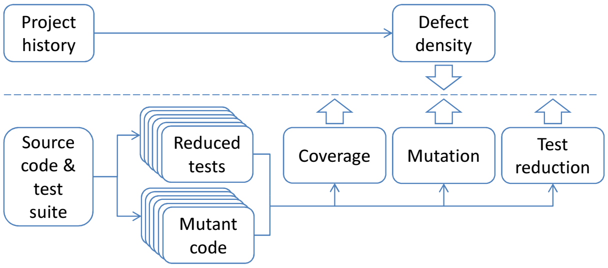


Fig. 1: Summary of test suite assessment process

the test suites, identify differences between them and to find out any relationships of these differences to other indicators, most importantly the test suite's ability to find real faults.

In particular, our question is when does mutation analysis (in terms of the *mutation score*) provide additional information to *code coverage* during the assessment and comparison of test suites. In addition, we experiment with *test suite reducibility*, a concept used to describe to what extent a test suite can be reduced without degrading a given adequacy criterion (such as code coverage or mutation score), or—viewed from a different angle—, given a reduced subset of the test suite, how much the adequacy is worse than the adequacy of the full test suite. If a small reduced part of the test suite is not- or only little less adequate than the full one, we say the test suite is more reducible (more redundant) than in the opposite case. In our research, our question is if reducibility provides additional insights over code coverage and mutation score.

We compare these three criteria to the *Defect Density – DD* [10] of the projects in question, which is in essence treated as the proxy for the actual test suite quality. Defect density is expressed as the number of confirmed defects in the system divided by the system size, and is typically used to characterize the project or the process itself. We apply this concept to post-release defects to characterize the test suite: we use the assumption that if a lower quality test suite is used in the project, fewer defects will be found during testing, hence higher defect density will be observed after release. Clearly, other aspects may contribute to actual DD values, not only test suite quality, but we currently rely on this estimation.

The method to measure these viewpoints is overviewed in the following. Figure 1 shows the process used in our experiments with the viewpoints on the right hand side. Code coverage ratio, mutation scores and test reducibility can be directly computed by running and analyzing the source code and tests. On the other hand, we consider defect density from project issue management history, which serves as separate dimension in the comparison.

a) Coverage ratio: Test coverage ratio is a basic metric, which is natural to measure and understand during test suite assessment. Our view is that any deeper analysis needs to be interpreted in the light of coverage ratio, since low test coverage has impact on many other factors: e.g., if a mutant is not covered by the test cases it may never be killed. We will investigate code coverage mostly in terms of how the other factors can extend this adequacy criterion.

b) Mutation analysis: Our approach to mutation analysis is to randomly sample a limited number of mutation points and manually implement mutations by source code annotation and script-based post-processing of the code. We did not use mutation tools so that we could manually control and verify the mutant generation process. We also limit our mutation operators to two which are expectedly likely to be killed by the test suite. Similar operators are referred to as "trivial" operators by some authors, however we think that if a test suite does not kill such mutants while covering them it may indicate serious deficiencies. The first operator mostly interferes with the data flow of the program, while the other one affects control flow. Similar approaches were proposed by Wong and Mathur [11], who also advocated a very low number of operators. The first operator we defined negates the result of `return` statements in the case of boolean and number return types (called the *return negation*, `RN`), while the second operator negates the whole conditional expression in `if` statements (called *if negation*, `IN`). The annotated source code is used to generate two types of outputs. First, an instrumented code is generated and run for coverage measurement for each mutation point. Second, all mutants are generated and run to obtain test results. Based on coverage data and test results, a detailed mutation analysis is possible.

c) Test suite reduction: Test suite reduction methods seek to minimize the size of a test suite by eliminating test cases with respect to an adequacy criterion such as code coverage. We use a traditional code coverage based reduction, where a possibly minimal subset of the test suite is computed that achieves the code coverage of the full test suite. The reduced subset is computed by a heuristic algorithm based on greedy addition of test cases with highest additional coverage [12]. Then, the obtained reduced test suite is assessed for mutation score. As indicators, we use the relative size of the reduced set and the difference of the resulting score to the score of the unreduced test suite. In the first case, a greatly reduced test suite might indicate redundancy from code coverage point of view, while the intuition behind the second concept is that if the score difference is big, maximal coverage alone does not indicate that the test suite is redundant because the eliminated test cases are able to kill some mutants.

d) Defect density: Defect density is computed using historical data of the subject projects from the respective issue tracking systems. Our open source subject projects are available on GitHub, which allows us to process the status and the history of issue reports. Defect density is not directly dependent on failing tests, but on problems found and reported by project collaborators and users. Although we did not verify each reported defect individually, they are usually reported for the released versions of the systems, meaning that the defects were not found during testing. Hence we assume that higher defect density rate is an indicator of unsuccessful testing and lower quality of the test suite.

III. Experiment Results

A. *Properties of Subjects*

For the experiments, we used four medium size open source Java programs, which are actively developed and which include regularly maintained test suites. Table I shows basic data about the subject programs. They belong to various domains: MapDB is an embedded database engine, Netty is an event-driven network application framework, OrientDB is a distributed graph database project, while Oryx is a real-time large scale machine learning engine. The total lines of code of the projects is about 450K, varying between 31K and 229K LOC. In the table, the number of methods and the number of test cases are also shown, together with the total coverage ratio of the whole test suite on method level. Here, the Clover tool [13] was used which computes the coverage based on source code instrumentation. The coverage ratio varies among projects, which makes them appropriate for our goals.

TABLE I: Subject programs

Program	LOC	Methods	Tests	Cover.	Domain
mapdb	53K	1 582	1 784	77.67%	database
netty	140K	8 133	4 079	48.88%	networking
orientdb	229K	13 052	1 058	39.38%	database
oryx	31K	1 557	208	27.51%	machine learning

B. *Mutant Generation*

As mentioned, mutations were manually added using source code annotation. Given the large effort required for manual mutant generation, only a randomly selected part of possible mutation points was implemented. Table II reports the number of candidate mutation points in the second and fourth columns for the two operators, while columns three and five contain the number of the actually annotated mutants. We will use the notation M_0 for the base set of generated mutants.

For determining the possible mutation points, the source code was scanned using the grep tool (for `RN` the `return` keyword and for `IN` the `if` keyword was used in the search). The order of the obtained list was then randomized. During the manual annotation process the random list was followed with the exclusion of false positives such as when the keyword was found in a comment. In case of `RN` annotation the return type of a given method was also manually checked, since only boolean and number types were negated. Finally, all mutants were successfully built, but in 12 cases they failed to produce test results because the test framework stopped with timeout. These cases were excluded from the final list, which is shown in the last column of the table. Overall, for the experiments we used 667 mutations. In the remaining part of the paper we present summarized results of the two operators.

C. *Mutant Classification*

Mutants can basically be divided into three groups based on their runtime behaviour: not covered, dead and live mutants. In this study, we distinguish between two subgroups of the not covered mutants: mutants whose method is not covered at all and mutants whose method is covered but the mutant itself is not covered. The set of all mutants M_0 is thus divided into the following four groups:

TABLE II: Total number of generated mutants

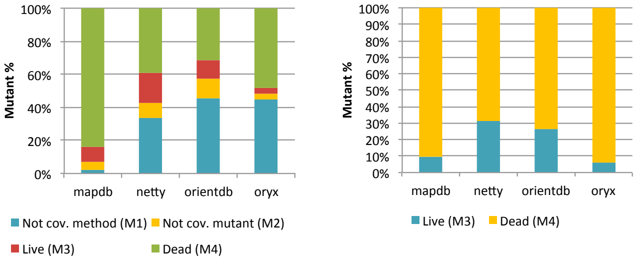
	`IN`		`RN`		All
Program	Candidates	$\lvert M_0\rvert$	Candidates	$\lvert M_0\rvert$	$\lvert M_0\rvert$
mapdb	1 002	50	1 539	53	100
netty	6 623	101	8 766	90	188
orientdb	15 765	150	14 009	134	278
oryx	1 143	51	2 038	50	101

M_1: set of mutants whose methods are not covered,
M_2: set of mutants whose methods are covered but the mutant is not covered,
M_3: live – set of mutants which are covered and test results did not change,
M_4: dead – set of mutants which are covered and at least one previously passed test failed.

TABLE III: Mutant coverage and liveness

	All	Not cov. meth.	Not cov. mut.	Live	Dead
Program	$\lvert M_0\rvert$	$\lvert M_1\rvert$	$\lvert M_2\rvert$	$\lvert M_3\rvert$	$\lvert M_4\rvert$
mapdb	100	2	5	9	84
netty	188	63	17	34	74
orientdb	278	126	33	31	88
oryx	101	45	4	3	49

Table III shows the associated numbers for each project, while the same data is shown in a normalized form in Figure 2a. The first thing to observe from the data regards the not covered mutants. Separating them to M_1 and M_2 gives insights into the mutation process. It can be observed that the $|M_1|/|M_2|$ ratio is larger in the case of the last 3 projects, the projects for which the total coverage is low. In other words, an overall low coverage has a negative effect on the interpretation of other data. This observation leads to a refined mutation score definition discussed later in this section.



(a) (b)

Fig. 2: Mutant coverage and liveness of covered mutants

The crucial part in mutation analysis is recording the ratio of killed mutants, which will then lead to computing the mutation score. This data can be read from the last two columns of Table III and from Figure 2a. If we observe the overall number of dead mutants among the total number of mutants, it becomes clear that mapdb is best performing in this respect, and the other three are much worse but comparable to each other. This information is misleading though, because in this basic calculation a mutant may be live for two reasons: either it is never covered or it is covered but the test suite failed killing it. mapdb has a high coverage compared to the other subjects, so this might explain the mentioned relationships. To clean the data from the bias of total coverage, we present the dead/live ratio of covered mutations in Figure 2b. From this viewpoint the test suite of the oryx project is the best in killing mutations, despite it has the lowest total coverage.

D. Mutation Score

Mutation score is the traditional adequacy measurement of the test suite in mutation analysis. It is the ratio of the number of killed mutants over the total number of non-equivalent mutants, which includes the not covered mutants too [2]. There are two notable issues with the denominator of this fraction. First one is the exclusion of equivalent mutants. In this study we did not apply a systematic method for the detection of equivalent mutants because we specifically picked mutation operators that severely modify the program behaviour, and hence unlikely generate equivalent mutants. Nevertheless, we manually verified all not dead mutants of two of the subjects (mapdb and oryx) and found that less than 10% of them are possible equivalent mutants. Hence, we define the basic mutation score as follows:

$$S = |M_4|/|M_0|$$

The other issue of the score computation regards the inclusion of not covered mutants, which is the approach followed by most mutation methods and tools. However, to eliminate the dominance of not covered mutants in subjects with low coverage, we define a modified version of mutation score by leaving out the not covered mutants:

$$S_{COV} = |M_4|/|M_3 + M_4|$$

Columns 2–3 of Table IV summarize mutation score values of the subject programs. It can clearly be seen how the low coverage distorts the mutation score, only mapdb has a similar value for the two score types. In fact, the relative order of the subjects according to their score values changes significantly.

TABLE IV: Mutation scores, suite size and score decrease when the test reduction reaches maximum coverage

Program	S	S_{COV}	Reduced size	S decrease
mapdb	0.84	0.90	10%	8%
netty	0.39	0.69	11%	11%
orientdb	0.32	0.74	26%	7%
oryx	0.49	0.94	37%	14%

E. Test Reduction

The test reduction algorithm has been applied to the subjects with fixed reduction sizes from one single test case to the whole unreduced test suite. This way, we could compare the characteristics of the gradual increase of the coverage ratios and the mutation scores of the reduced test suites. Figure 3 shows the changes of the coverage and S score values for oryx. We can observe that mutation score is still increasing after coverage reached its maximum. In this case, maximum coverage was achieved by selecting 37% of the test cases, at the same time the reduction of S was 14%.

The last two columns of Table IV show the reduced size and the mutation score decrease compared to the full test suite for all subjects. We use both values as the reducibility indicators. The increase curves of the other programs showed similar characteristics to oryx, except that for the subjects with lower score decrease values they were more flat.

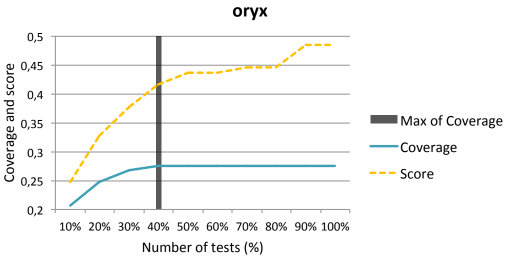


Fig. 3: Coverage and mutation score changes during test reduction for oryx

F. Defect Density

We computed DD values as the number of confirmed and closed issue reports marked as defects divided by the actual size of the system at the time of the analysis. The defects were obtained from the respective issue tracking databases on GitHub taking into account the whole lifetime of the projects. Table V shows the number of the identifed defects, the labels used to determine the defects among all issue reports, and the calculated defect density values. For the normalization we used the ratio per one thousand lines of program code, which is also shown in the table.

TABLE V: Defect density details

Program	Filter label	Closed defects	KLOC	DD
mapdb	'bug'	176	53	3.3
netty	'defect'	811	140	5.8
orientdb	'bug'	388	229	1.7
oryx	'bug'	44	31	1.4

G. Analysis of Results

In this study we used code coverage, mutation score and test reducibility as different indicators of test suite adequacy, which we aim to compare with defect density, an independent dimension acting as a proxy for test suite quality. In Figure 4, all these viewpoints are presented side by side, where the subjects are listed in an ascending order by defect density. Except defect density, the data take values from $0 - 1$, but the actual values are not very important in this summarization, only their relative order. An ideal case would be if all indicator curves would show an opposite steepness to the defect density, because in each case a bigger value indicates higher quality, which we mirror to low defect density.

We summarize our observations as follows:

1) Coverage ratio in itself is not enough to reason about defect density: only between subjects mapdb and netty can we observe the desired relationship.

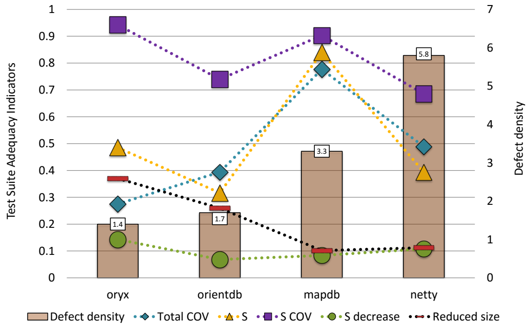


Fig. 4: Test suite adequacy indicators and defect density

2) Considering traditional mutation score S improves the relationship to defect density. It gives oryx a higher rank, which is more aligned with what is expected, but the characteristic of this indicator is still very much aligned with the overall code coverage.
3) Elimination of the coverage-bias mentioned for Figures 2a and 2b greatly improves defect density estimation. Except for the orientdb-mapdb pair, the order is aligned with defect density for S_{COV}.
4) Finally, test reducibility underlines the usefulness of additional adequacy criteria: it categorized mapdb correctly, which none of the other indicators managed.

The summarization of these results can be seen in Table VI. It includes a pairwise comparison of the subjects according to whether their relative order for to the respective indicator follows the relative order given by DD. The table contains the pairs in each row and there is a 1 in the column for an indicator if the order is aligned. As can be seen, code coverage is a good indicator only in one of the six cases, the traditional mutation score matches in half of the cases, while the score for covered mutants matches in 5 cases out of 6. It is also interesting to observe that the reduction size is aligned with DD in all but one case, and this indicator is not dependent on mutation analysis. A particular case to highlight is subject oryx, which has the lowest code coverage (only 27.51%), still has the lowest defect density ratio. All adequacy criteria except, of course, coverage are able to predict this attribute.

A promising direction of future research may be to find models to predict in which cases would be a specific indicator be suitable for test suite quality assessment; in other words, when would code coverage be enough in itself and when would mutation or reducibility analysis be required.

TABLE VI: Pairwise comparison of indicators to DD

Program-pair	Total COV	S	S_{COV}	Reduced size	S decrease
oryx–mapdb	0	0	1	1	1
oryx–orientdb	0	1	1	1	1
oryx–netty	0	1	1	1	1
mapdb–orientdb	0	0	0	1	0
mapdb–netty	1	1	1	0	0
orientdb-netty	0	0	1	1	0
Total match	1	3	5	5	3

IV. Related Work

Despite decades of research in the field, the concept of mutation analysis is still too stubborn to leave the academic research labs and become an everyday technique in industrial testing practice [3]. Researchers identified several key barriers for the adoption, including the identification of equivalent mutants, test case generation, reliable tooling, and the very high resource need compared to the expected benefits. Jia and Harman presented an excellent literature review of the field [2].

So, a very important research area is to find out in what situations can results from mutation analysis be used as successful test adequacy criteria. Previous studies used various kinds of mutation-based adequacy criteria like control-flow path coverage for the evaluation of so called class mutations [14], simulation-based models of distributed systems [15] and to explore alternatives of mutation to reduce its cost without significantly deteriorating its strength and effectiveness [16]. In our research, we seek for combined test adequacy criteria, which use code coverage, mutation score, and also test suite reducibility. Essentially, we are working on a general test suite assessment methodology, in which mutation analysis would play an important role. Other approaches to test quality include applying general source code measurement on test suite code [17], or evaluating tests from the testing process perspective [18].

There are several positive evidences that mutants are a valid substitute for real faults (following the *coupling effect*) [19], [20], however this is very hard to verify in a general context, and generalize to different systems, test suites, mutation techniques and tools. Higher order mutants are another approach to increase the coupling effect [21], [22]. In our work, we used a specific kind of *defect density* measurement to verify the adequacy of mutation analysis.

Selective mutation is a popular approach to cost reduction [2]. Our approach was to use only two mutation operators, which were expected to be able to sufficiently indicate the adequacy of the test suites. Delamaro et al. [23] found that using only deletion operators is a cost-effective approach. Another direction is to find a set of “sufficient operators,” which are statistically not too much worse than the full set, while being much more limited [24]. Mathur and Wong, for instance, used only two operators [11], while Navanti et al. [25] worked with a small set of specific ones. Mutation faults have been used in test case prioritization. Usaola et al. [26] present a test reduction method using mutation score. Methods for reducing regression test suites based on mutation have been presented by Offut et al. [27]. The problems related to equivalent mutants have been studied extensively (for a review, see [28]). While there are some very innovative approaches (see for instance the works by Papadakis et al. [29], Kintis and Malevris [30]), the problem is still largely unsolved.

V. Threats to validity

This experiment used only two mutation operators and four subject programs, so our findings should be generalized with caution. Generating mutations manually is very time consuming and there are tools that can solve this efficiently, however in this preliminary study we wanted to control and validate each step of the mutation generation. We did not perform a systematic analysis of equivalent mutants due to their unlikely occurrence as mentioned earlier, however this may influence the validity of our results.

VI. Conclusion

The experiment presented in this paper underlined that code coverage alone is rarely an indicator of the expected number of defects in the system. On the other hand, the other two adequacy criteria we used greatly improve the prediction; mutation score and test suite reducibility. Since we used defect density with real faults reported for the systems after release, we treat the combination of these indicators as indicators of test suite quality, hence a step towards a more general test suite assessment model. At present, we rely on a small sample of programs, and the experiment had other limitations as well. Hence, we are not drawing too much general conclusions, but the trends we observed are promising.

Unfortunately, merely knowing that code coverage is sometimes a good indicator in itself for test suite quality is not enough. Our results do not explain yet in what situations is mutation analysis superfluous, and what does reducibility actually add to the big picture. These topics are open for future research, because we believe that more complex analyses should be avoided when not necessary, but should be used whenever indicated. Approaches similar to the heuristic prediction of mutation score by Jalbert and Bradbury [31] might help constructing models to decide on these questions. We plan to continue experimentation with more subjects, more mutation operators and also using a dedicated mutation tool.

References

[1] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 2014, pp. 435–445.

[2] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649–678, Sept 2011.

[3] P. Ammann, "Transforming mutation testing from the technology of the future into the technology of the present," 2015, Mutation 2015 Keynote.

[4] M. P. Usaola and P. R. Mateo, "Mutation testing cost reduction techniques: A survey," *IEEE Softw.*, vol. 27, no. 3, pp. 80–86, May 2010. [Online]. Available: http://dx.doi.org/10.1109/MS.2010.79

[5] R. V. Binder, "Agile High Assurance: Testing re-imagined," in *Hungarian Software Testing Forum, 5th Annual International Conference*, Nov. 2015.

[6] "PIT homepage," http://pitest.org/, last visited: 2016-01-13.

[7] D. Tengeri, Á. Beszédes, T. Gergely, L. Vidács, D. Havas, and T. Gyimóthy, "Beyond code coverage - an approach for test suite assessment and improvement," in *8th IEEE ICST Workshops (ICSTW'15); 10th Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART'15)*, Apr. 2015, pp. 1–7.

[8] F. Horváth, B. Vancsics, L. Vidács, Á. Beszédes, D. Tengeri, T. Gergely, and T. Gyimóthy, "Test suite evaluation using code coverage based metrics," in *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15)*, Oct. 2015, pp. 46–60.

[9] L. Vidács, F. Horváth, D. Tengeri, and Á. Beszédes, "Assessing the test suite of a large scale system based on code coverage and derived metrics," in *1st International Workshop on Validating Software Tests (VST'16) – accepted paper*, Mar. 2016, pp. 1–4.

[10] "Standard glossary of terms used in software testing," International Software Testing Qualifications Board, 2014.

[11] W. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *Journal of Systems and Software*, vol. 31, no. 3, pp. 185 – 196, 1995.

[12] G. Rothermel, R. J. Untch, and C. Chu, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001.

[13] "Clover java and groovy code coverage tool homepage," https://www.atlassian.com/software/clover/overview, last visited: 2015-11-03.

[14] S. Kim, J. Clark, and J. McDermid, "Assessing test set adequacy for object oriented programs using class mutation," in *28 JAIIO: Symposium on Software Technology*, 1999.

[15] M. J. Rutherford, A. Carzaniga, and A. L. Wolf, "Evaluating test suites and adequacy criteria using simulation-based models of distributed systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 452–470, 2008.

[16] W. E. Wong, A. P. Mathur, and J. C. Maldonado, "Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness," in *Software Quality and Productivity*. Springer, 1995, pp. 258–265.

[17] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *Software Engineering, IEEE Transactions on*, vol. 40, no. 11, pp. 1100–1125, Nov 2014.

[18] B. Marick, J. Bach, and C. Cem Kaner, "A manager's guide to evaluating test suites," in *13th International Software Quality Conference (Quality Week)*, Jun. 2000, pp. 1–16.

[19] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 402–411.

[20] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 654–665.

[21] Y. Jia and M. Harman, "Higher order mutation testing," *Inf. Softw. Technol.*, vol. 51, no. 10, pp. 1379–1393, Oct. 2009.

[22] E. Omar, S. Ghosh, and D. Whitley, "Comparing search techniques for finding subtle higher order mutants," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '14. New York, NY, USA: ACM, 2014, pp. 1271–1278.

[23] M. Delamaro, J. Offutt, and P. Ammann, "Designing deletion mutation operators," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, March 2014, pp. 11–20.

[24] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *30th International Conference on Software Engineering*. ACM, 2008, pp. 351–360.

[25] J. Nanavati, F. Wu, M. Harman, Y. Jia, and J. Krinke, "Mutation testing of memory-related operators," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, April 2015, pp. 1–10.

[26] M. P. Usaola, P. R. Mateo, and B. P. Lamancha, "Reduction of test suites using mutation," in *Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 425–438.

[27] J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *Proceedings of International Conference on Testing Computer Software*, 1995, pp. 111–123.

[28] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *Software Engineering, IEEE Transactions on*, vol. 40, no. 1, pp. 23–42, 2014.

[29] M. Papadakis, Y. Jia, M. Harman, and Y. LeTraon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *37th International Conference on Software Engineering (ICSE)*, 2015.

[30] M. Kintis and N. Malevris, "Using data flow patterns for equivalent mutant detection," in *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, 2014, pp. 196–205.

[31] K. Jalbert and J. S. Bradbury, "Predicting mutation score using source code and test suite metrics," in *Proceedings of the First International Workshop on Realizing AI Synergies in Software Engineering*. IEEE Press, 2012, pp. 42–46.