On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later

Domenico Serra[1], Giovanni Grano,[2] Fabio Palomba,[2] Filomena Ferrucci,[1] Harald C. Gall,[2] Alberto Bacchelli[2]
[1]University of Salerno, Italy — [2]University of Zurich, Switzerland
dserra@studenti.unisa.it, grano@ifi.uzh.ch, palomba@ifi.uzh.ch, fferrucci@unisa.it, gall@ifi.uzh.ch, bacchelli@ifi.uzh.ch

***Abstract*—Good unit tests play a paramount role when it comes to foster and evaluate software quality. However, writing effective tests is an extremely costly and time consuming practice. To reduce such a burden for developers, researchers devised ingenious techniques to automatically generate test suite for existing code bases. Nevertheless, how automatically generated test cases fare against manually written ones is an open research question. In 2008, Bacchelli et al. conducted an initial case study comparing automatic and manually generated test suites. Since in the last ten years we have witnessed a huge amount of work on novel approaches and tools for automatic test generation, in this paper we revise their study using current tools as well as complementing their research method by evaluating these tools' ability in finding regressions. Preprint [https://doi.org/10.5281/zenodo.2595232], dataset [https://doi.org/10.6084/m9.figshare.7628642].**

***Index Terms*—Software Testing; Automatic Test Case Generation; Empirical Studies.**

I. Introduction

Software testing is widely recognized as a crucial part of any software development process especially in the context of Continuous Integration (CI), where developers run both unit and integration tests against new changes to promptly discover the presence of software faults [1]. Unfortunately, writing good tests—particularly unit tests [2]—represents one of the most difficult and time consuming testing activities. Therefore, an increasingly large amount of effort has been devoted to implement approaches and tools to *automatically generate unit test suites* [3].

The limited industrial adoption of tools for automatic test generation raises some questions about the actual effectiveness of these tools in reducing the burden of testing tasks for developers [4]. In fact, while it has been shown that tests automatically generated often reach a higher code coverage compared to their manual counterparts [4], the practical advantages of using automatically generated test suites versus manual ones are still unclear, as well as the to what extent these tests suites can complement each other.

In 2008, Bacchelli et al. [5] conducted a first exploratory study with the goal of empirically understanding the differences between manual and automatic unit test generation. The authors considered test effectiveness in terms of code coverage, mutation score, as well as error finding. In particular, Bacchelli et al. [5] investigated the FREENET project [6] as the subject of their study.

Since FREENET used to lack of a significant test suite, the authors first manually created tests for a subset of 15 classes. Through this process, 14 distinct defects were discovered and fixed. Afterwards, the authors applied automatic test generation tools to the same defective classes and compared the automatically generated tests against the manually created ones. They selected three tools available at that time, namely RANDOOP [7] and JUNIT FACTORY [8], designed for regression testing, and JCRASHER [9], designed for defect revelation. The study reported that automatically generated tests could achieve a higher code and mutation coverage than the manually generated ones, and the former could also generate unexpected scenarios that lead to the identification of faults, partially overlapping with those found manually.

The amount of research done in the last ten years in this field [3] calls for a re-assessment of the aforementioned findings: As an example, a tool like EVOSUITE [10]—which is widely recognized as the current state-of-the-art with respect to the generation of test suites [11], [12]—could not yet be considered by Bacchelli et al. [5]. Therefore, in this paper we report a replication of the reference study that compares (i) code coverage, (ii) mutation score, and (iii) fault detection ability of test cases generated automatically using current techniques. In doing so, we first manually reconstruct the scenario exploited by Bacchelli et al. [5], by digging into the source code repository of FREENET, and make the resulting dataset publicly available for further studies. Then, we compare the tests that were manually created in 2007 against those automatically generated by top three testing tools, namely EVOSUITE [10], RANDOOP [7], and JTEXPERT [13]. On top of the replication study, we also compare manually generated tests to automatically generated ones with respect to their abilities in finding regression faults. The key findings of our study confirm that the automatic tools can achieve very high line coverage and mutation score; however most of the actual defects cannot be identified.

II. Related Work

This work presented in this paper is a partial replication of the study of Bacchelli et al. [5], with the goal of evaluating the results of tools that are available nowadays, i.e., ten years after the initial study. The reference paper [5] aimed at comparing the manually and automatically generated tests considering code coverage, mutation score, and fault detection ability. Other authors have proposed a similar comparison. Fraser et al. [4] conducted a human study aimed at understanding the practical value of automated testing tools. The experiment was organized

in two studies: In the first one, they asked the participants to manually write tests for 4 different Java classes; in the second one, they asked them to generate test suites using EVOSUITE for the same subjects. The authors showed that test case generation tools can achieve higher code coverage compared to manually created tests, while automatically generated tests resulted as less effective in detecting faults. Similarly, Shamshiri et al. [14] investigated the fault detection ability of automatically generated tests compared to the manually written ones. The study involved 3 testing tools ran over the Defects4j [15] dataset. In particular, they exercised the tools in a regression testing scenario, where the test suites were generated on the fixed versions of the code and then executed against the buggy versions. Their findings were in line with the ones of Fraser et al. [4]: The tools were only able to find about half of the bugs. Grano et al. [16] compared the readability of manually and generated tests, finding the latter as less readable. Finally, a comparison between manually and automatically generated tests has also been proposed in the context of the Java Unit Testing Tool Competition of the IEEE/ACM International Workshop on Search-Based Software Testing. The closest work has been proposed in the round four of the competition [17]: in particular, researchers compared four automatic tools against human made tests in terms of statement, branch, mutation coverage, and real-fault detection. Our study shares a similar experimental setup, but has a different goal: We strive to measure the improvement achieved by automated test generation techniques over the course of the last ten years. For this reason, differently from the competition [17], we consider another set of data (the same classes, tests, and defects of the reference paper, which we reconstruct and make publicly available) and add another metric (how able are the techniques to find defects not through regression).

III. METHODOLOGY

The *goal* of the study is to understand how automatically generated test cases compare to manually written ones, with the *purpose* of assessing whether and to what extent automatically generated tests can actually complement manual ones and increase the overall effectiveness of test suites. The *perspective* is of both researchers and practitioners: the former are interested in understanding how automatic test case generation can be improved, while the latter aims at evaluating the feasibility of using automatic testing tools in practice. To reach our goal, we set up a replication of the work done by Bacchelli et al. [5], performed with the new tools resulting from a decade of research. More specifically, our study is driven by the following research questions:

- **RQ$_1$ - Coverage**: *What is the code coverage of manual tests versus automatically generated ones?*
- **RQ$_2$ - Mutation**: *What is the mutation score of manual tests versus automatically generated ones?*
- **RQ$_3$ - Fault Detection**: *What is the fault detection ability of manual tests versus automatically generated ones?*

By addressing the aforementioned questions, we want to provide an updated view of the capabilities of automatically generated test suites when compared to manually created ones. We following detail the *context* of the study, as well as the methodological steps we perform in our empirical investigation.

A. Dataset Construction

To replicate the original study [5], we first re-construct the dataset that was originally exploited, which was unfortunately not publicly available. More specifically, such a dataset comes from the FREENET project [6] where Bacchelli et al. [5] spent about dozens of hours to create a set of manual regression tests for 15 classes that also detected 14 different bugs. To do that, we proceeded as follows.

Detecting manually-written regression tests. For each class tested in the original paper, we detect the commit introducing the test suite for such a class (i.e., the one created by Bacchelli et al. [5]). The process is done by scanning each commit of the version history of FREENET and looking for $\mathcal{T}_i^c$, namely the commit i that firstly introduced the test suite $\mathcal{T}$ for the class c. Since the test suite followed the naming convention that dictates that each test has the suffix `Test` after the name of the tested production class, we could do an accurate mapping between tests and the corresponding production classes. The tests identified in this step represent the *manually-written regression tests*. At the same time, the corresponding production classes available in the commit $\mathcal{T}_i^c$ represent the versions where the observed defective behavior is *fixed*: In fact, the tests were only committed *after* the corresponding production class was fixed of any bug found during the testing.

Detecting defective classes. In the original dataset, one manually tested class could have zero or more defects. Thus, we need to identify, for each defect, the exact version of the production class with the defective behavior. In so doing, we execute a test suite introduced at $\mathcal{T}_i^c$ over the previous commits of c: if $\mathcal{T}$ fails on the production class in a certain commit c_i, we assume that one defect is found. The procedure is repeated until all the defects declared in the reference paper are found. At the end of this process, for each defect we have a triple composed of (1) the *defective version* of the production class, (2) the corresponding *manually written regression test*, and (3) the *fixed version* of the class.

Generating automatic regression tests. Once identified the set of defective classes and their fixes, we can generate regression tests automatically. To have an overview of the performance of different existing tools, we consider three state-of-the-art testing tools, namely EVOSUITE [10], RANDOOP [7], and JTEXPERT [13]. We run these tools on both defective and fixed versions of such classes, setting 180 seconds as search budget and leaving the remaining parameters to their default values, following the same methodology of previous work [12], [18], [19]. According to the experimentation done by Arcuri and Fraser [20], parameter tuning represents an expensive task that does not automatically imply better performance; as such, the choice of using the default settings is reasonable and often does not influence the overall results [20].

RANDOOP generates two kind of suites: a *regression* suite that record the current behavior and an *error-revealing* suite,

that checks for specific specifications or contract violations. In the fault-detection analysis we treated those two suites separately. Finally, to cope with the randomness of automatic test case generation due to the search-based techniques exploited, we repeat the generation 10 times for each production class. We run the process on a Linux server running Ubuntu 18.04, having 16 cores and 64GB of RAM.

Dataset validation. While the re-construction of the dataset is done following the description provided by Bacchelli et al. [5], we conduct an additional validation to ensure its reliability. To this aim, we ask the first author of the reference paper (who was developer at FREENET) to validate the re-constructed dataset. We provide him with a directory containing, for each defect, a separate sub-directory containing the triples previously generated: or each defect the author could peruse (i) the defective version of the production class, (ii) the corresponding regression tests (manually written), and (iii) the fixed version of the class. Furthermore, we provide the author with an additional file reporting the meta-information (e.g., message and author) of the commits referring to the files he can analyze. As a result of this validation, the author confirms that the operations done in the re-construction phase are correct: this makes us confident of the reliability of the dataset.

Dataset availability. The re-constructed dataset is publicly available [21].

B. Data Analysis

To address our research questions, we first compute line coverage ($\mathbf{RQ}_1$) and mutation score ($\mathbf{RQ}_2$) for both manually and automatically written test suites. To compute the mutation score we rely on PIT [22], which is currently the most mature tool available for mutation testing [23]. Since we perform multiple test generation runs for the automatic testing tools, in Section IV we report and discuss the mean values of coverage and mutation score achieved by the tests generated with the three experimented tools. To answer $\mathbf{RQ}_3$, we compute the number of times manually and automatically generated tests are able to correctly identify a failure: for the automated tests, we consider a failure to be detected if the tools identify it in at least one of the runs. It is important to note that the three employed automatic testing tools share the common goal of generating regression tests, while RANDOOP can be also employed to generate error-revealing suites. For this reason, in $\mathbf{RQ}_3$ we perform two complementary analyses. First, we generate tests for a *fixed* version and we check whether the generated tests can detect a failure in the defective versions. Second, we investigate the ability of the exploited tools to *directly detect faults* by generating tests for the defective versions of the classes.

IV. ANALYSIS OF THE RESULTS

In the following, we discuss the results of each RQ, summarized in Table I for space reasons.

A. $\boldsymbol{RQ}_1$ — On Code Coverage of Manual vs Automatic Tests

Looking at the results achieved when considering this perspective, the first observation is related to the coverage achieved by the automatic testing tools: All of them have a line coverage comparable or even higher than the one achieved by the manual tests. This result likely reflects the way in which the generation process actually works. In fact, the primary goal of all the tools is optimizing the coverage of the test on the production code; as such, we confirm that they seem to reach their target of overcoming manual tests under this aspect.

Among the tools, EVOSUITE reaches the highest coverage (88% on average); this is generally true also looking at the individual classes, with a very few exceptions. On the contrary, RANDOOP reaches the lowest coverage, thus indicating that a random-based approach can, at times, be less performing than others–a finding that corroborates previous work in the field [24]. Compared to the results achieved in the study by Bacchelli et al. (Table 2[1] in [5]), we see that the results achieved by RANDOOP ten years later are 3 points percentage higher on average, but no tool is able to reach the coverage achieved by JUNIT FACTORY ten years ago, thus indicating that this industrial tool is still the best performer along this dimension. Our results indicate that automatic test case generation tools can support developers in ensuring a high code coverage (compared to manual tests), yet the improvement with respect to the original study is limited.

B. $\boldsymbol{RQ}_2$ — On Mutation Score of Manual vs Automatic Tests

Mutation testing allows to measure the ability of test cases to cover the so-called *mutants*, i.e., variation of the production code that make it defective and that the test is supposed to identify [25]. The discussion of our results is similar to what reported for code coverage. Indeed, the automated tools reach mutation score values superior to the one of manually written tests. Also in this case, EVOSUITE is the best tool among the experimented ones, while RANDOOP and JTEXPERT achieve a similar score. Interestingly, we observe cases where it seems to exist some complementarity between manual and automatic tests. This is, for instance, the case for the class `HTMLEncoder`: while the mutation score of the manual test in this case is fairly low (28%), the automatic ones are much higher. In other cases, like with the class `URLDecoder`, not all the automated tools reach a high mutation score while the manual one can capture more generated mutants. This seems to highlight that manual and automatic tests can sometimes work in a complementary manner and interchangeably: we see this finding as a possible input for further research on the topic. Compared to the results achieved in the study by Bacchelli et al. (Table 3 in [5]), we still see that JUNIT FACTORY remains the best performer after ten years (77% on average), yet the difference with EVOSUITE is small (five point percentage less) and the improvement achieved by RANDOOP is very significant (18 points percentage higher now). Overall, our results indicate that automatic test case generation tools can support developers in ensuring a high mutation score (even though manual tests are sometimes better) and there has been a significant improvement in the last years along this dimension of effectiveness.

[1]Given the limited space, we refer to the tables in the original paper [5].

Table I: Performance of manually versus automatically generated test suites. The column 'Direct' is the fault detection ability of the tests when trying to directly identify faults in production code, while the column 'Regr.' is the ability to detect regressions.

Class Name	Line Coverage (**RQ$_1$**)				Mutation Score (**RQ$_2$**)				Fault Detection (**RQ$_3$**)									
	Manual	Evosuite	Randoop	JTExpert	Manual	Evosuite	Randoop	JTExpert	Manual	Evosuite		Randoop Reg.		Randoop E.R.		JTExpert	
										Direct	Regr.	Direct	Regr.	Direct	Regr.	Direct	Regr.
Base64	80	93	78	84	73	64	81	71	0	0	0	0	0	0	0	0	0
BitArray	71	100	95	94	46	69	84	77	0	0	0	0	0	1	0	0	0
HTMLDecoder	56	99	84	87	37	52	17	45	1	0	0	0	0	0	0	1	1
HTMLEncoder	71	100	83	100	28	85	88	91	0	0	0	0	0	0	0	0	0
HTMLNode	97	75	91	99	94	79	96	92	4	1	2 (1)	0	0	0	0	3	4 (2)
HexUtil	74	73	83	82	59	58	68	66	2	0	1 (1)	0	1 (1)	0	0	0	1 (1)
LRUHashtable	83	100	88	31	91	88	88	35	1	0	0	0	0	0	0	1	0
LRUQueue	83	100	92	60	58	61	96	46	0	0	0	0	0	0	0	1	0
MultiValueTable	85	92	85	70	75	93	76	70	0	0	0	0	0	0	0	2	0
SimpleFieldSet	54	51	85	5	49	35	69	4	2	2 (1)	0	0	1 (1)	2 (1)	2 (1)	2	2
SizeUtil	83	95	0	100	57	87	0	87	0	0	0	1	0	0	0	1	0
TimeUtil	95	96	56	100	93	93	41	88	3	0	0	0	1 (1)	0	0	0	3 (3)
URIPreEncoder	79	79	42	84	19	75	50	66	0	0	0	0	0	0	0	0	0
URLDecoder	68	80	36	86	71	65	38	75	1	0	0	0	0	0	0	0	1 (1)
URLEncoder	86	80	85	83	67	83	79	72	0	0	0	0	0	0	0	0	0
Overall	**78**	**88**	**72**	**78**	**61**	**72**	**65**	**66**	**14**	**3 (1)**	**3 (2)**	**1**	**3 (3)**	**3 (1)**	**2 (1)**	**11**	**12 (7)**

C. *RQ$_3$ — On Fault Detection of Manual vs Automatic Tests*

As for the fault detection ability, Table I reports the number of defects identified by manual tests and by the three exploited tools. For RANDOOP, we report the results for the *regression* and for the *error revealing* configurations separately (indicated by the columns "Randoop Reg" and "Randoop E.R."). The number in parenthesis represents the defects that were also found by the manual tests (e.g., EVOSUITE found two defects in `SimpleFieldSet`, one of which was also discovered with the manual testing). We observe that manually written tests have much high fault detection capabilities, while the automatic ones create unexpected scenarios that lead to a defect identification in a small number of real defects. Surprisingly, EVOSUITE and RANDOOP capture only a few of the real faults, when introduced as regressions. All in all, our findings suggest that all the automated tools can support developers with keeping coverage and mutation score high, but their ability in detecting real defects is not as good. Interestingly, we observe that the defects identified by automated tools are generally different from those captured by manually written tests, thus bringing evidence that developers can employ automatically and manually written in a complementary fashion to find more defects in production code.

V. THREATS TO VALIDITY

To conduct the study, we re-constructed the original dataset used by Bacchelli et al. [5], following the exact procedure of the reference paper to identify manually written test cases, defective, and fixed versions of the production classes involved in each defect. As a further validation of our activities, we involved the first author of the reference paper, who confirmed the correctness of the re-constructed dataset.

To investigate the capabilities of automatically generated tests, we ran the considered tools using a search budget of 180 seconds, which allowed the tools to extensively optimize the generated test suites [26]–[28]. At the same time, we left the other parameters to their default configuration: according to the findings of Arcuri and Fraser, it is not easy to find settings that significantly outperform the default suggested values [20].

The selected tools explicitly aim at finding regression faults rather than defects already present in production code. However, as shown by both Bacchelli et al. [5] and Pacheco et al. [7], such tools can also be configured to generate unexpected scenarios that may lead to discovering existing defects. Therefore, our study aimed at assessing to what extent newer testing tools can (i) help developers in finding existing defects, and (ii) actually identify regressions. This preliminary investigation took into account real defects from the FREENET project: Our findings are likely to differ when considering other environments and projects.

VI. CONCLUSION

We empirically compared the performance of manually versus automatically created test suites, considering three perspectives: code coverage, mutation score, and fault detection ability. To this aim, we first re-constructed and made publicly available a dataset originally used by Bacchelli et al. [5]. Our findings revealed that current automatic test case generation tools are able to optimize coverage and mutation score more than manually written tests; nevertheless, the improvement in the last ten years has not been dramatic. In terms of defect finding, while there has been little improvement over the last years, it has not reached the same level of quality as the other dimensions. This is expected, as the goal of the current tools is creating regression tests that are able to capture the current behavior, rather than exposing problems. Nevertheless, in some cases, the automatically generated tests were indeed able to expose unexpected scenarios that were defective, thus indicating that this could be a potentially viable path for our future research focus.

Our future research agenda includes the comparison of manually and automatically written test code in the context of a larger set of projects and defects (e.g., by exploiting the DEFECT4J dataset [15]) as well as the definition of strategies that can make automatic tests more effective. By publishing this work in the main mining software repositories venue, it is our hope that it can contribute to bring the mining community closer to that of automated software testing.

ACKNOWLEDGMENTS

A. Bacchelli and F. Palomba gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[2] P. Runeson, "A survey of unit testing practices," *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

[3] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[4] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? a controlled empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, p. 23, 2015.

[5] A. Bacchelli, P. Ciancarini, and D. Rossi, "On the effectiveness of manual and automatic unit test generation," in *The Third International Conference on Software Engineering Advances*. IEEE, 2008, pp. 252–257.

[6] The Freenet Project: a peer-to-peer software platform for censorship-resitant communication. [Online]. Available: https://freenetproject.org

[7] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 75–84.

[8] JUnitFactory webiste. [Online]. Available: http://www.junitfactory.com

[9] C. Csallner and Y. Smaragdakis, "Jcrasher: an automatic robustness tester for java," *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[10] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 416–419.

[11] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: high coverage, no false alarms," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 67–77.

[12] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 130–141.

[13] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc, "Instance generator and problem representation to improve object oriented code coverage," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 294–313, 2015.

[14] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

[15] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[16] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An empirical investigation on the readability of manual and generated test cases," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 348–351. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196363

[17] U. Rueda, R. Just, J. P. Galeotti, and T. E. Vos, "Unit testing tool competition—round four," in *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2016, pp. 19–28.

[18] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 8, 2014.

[19] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2018.

[20] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.

[21] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. Gall, and A. Bacchelli, "Replication Package - On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later," 3 2019. [Online]. Available: https://figshare.com/articles/On_the_Effectiveness_of_Manual_and_Automatic_Unit_Test_Generation_Ten_Years_Later/7628642

[22] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 449–452.

[23] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. Le Traon, "How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2426–2463, 2018.

[24] S. Bauersfeld, T. E. Vos, and K. Lakhotia, "Unit testing tool competitions–lessons learned," in *International Workshop on Future Internet Testing*. Springer, 2013, pp. 75–94.

[25] 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.

[26] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *International Symposium on Search Based Software Engineering*. Springer, 2011, pp. 33–47.

[27] G. Fraser and A. Arcuri, "The seed is strong: Seeding strategies in search-based software testing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 121–130.

[28] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn, "Random or genetic algorithm search for object-oriented test suite generation?" in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2015, pp. 1367–1374.