Software-Testing Contests: Observations and Lessons Learned

Xingya Wang and Weisong Sun, Nanjing University
Linghuan Hu, University of Texas at Dallas
Yuan Zhao, Nanjing University
W. Eric Wong, University of Texas at Dallas
Zhenyu Chen, Nanjing University

While a significant amount of resources can be spent on software testing, the software produced may still suffer from low quality. The authors describe their experience of hosting industry-sponsored software-testing contests to help undergraduate and graduate students, as well as practitioners, improve their testing skills.

Software has played vital roles in many services and mission-critical systems and even replaced humans in some functions to improve efficiency, reliability, and safety. However, severe consequences, including property loss and life-threatening accidents, can be caused by compromised systems and defective software.[1] In the industry, testing remains the most commonly used technique to help engineers ensure the quality of software and prevent accidents. A project can invest significant resources in testing, but the software produced may still suffer from low quality. The key factor is not how much has been spent on testing but how the testing was conducted and who conducted it. Today, unit testing can be used to effectively test an individual software module.[2, 3]

To introduce unit testing to more college students and practitioners in the software industry and improve their unit-testing skills, we organized two software-testing contests [(STCs); STC 2016 and STC 2017], sponsored by the IEEE Reliability Society[4] and Mooctest LLC,[5] as part of the National Student Contest of Software Testing in China.[6] In addition to the educational objective, we collected the testing data, such as manually generated test cases, for research purposes. The data (after removal of private information) were saved in a repository that is open access. We conducted two empirical studies using data from this repository. Our findings are presented in the following sections.

TESTING CONTESTS

Both STC 2016 and STC 2017 had preliminary and final stages. STC 2017 also had an additional semifinal stage. STC 2016 included 521 undergraduates from 131 affiliations in the preliminary stage; the top 30 ranked contestants participated in the final stage. For STC 2017, the number of participants nearly doubled. Hence, we added a semifinal stage to select 45 contestants from

Digital Object Identifier 10.1109/MC.2019.2905533
Date of publication: 24 September 2019

21 affiliations to attend the final. The top-ranked contestants in the final stages each year received cash prizes. Both STC 2016 and STC 2017 used Mooctest,[5] an online software-testing platform that can automatically deploy the testing environment and provide several software-testing-related measurements to help practitioners improve their testing efficiency and effectiveness. Mooctest is straightforward to use; each contestant needs only to download and install the Mooctest plug-in for the Eclipse integrated development environment (IDE). To help contestants become familiar with the Mooctest platform and the contest environment, we provided a tutorial covering the contest, Mooctest platform, JUnit, and ranking criteria, as well as many hands-on exercises, on the Mooctest website several weeks before each contest.

Each contestant used an Eclipse IDE with a Mooctest plug-in installed to connect to the Mooctest server. When the contest started, the Mooctest server deployed several subject programs to each contestant's Eclipse IDE. Once the deployment was completed, the contestant started to test, and an activated timer was shown on the screen. During the test, the contestants needed to read and understand the source code of the subject programs. Then, they wrote JUnit test cases and submitted them to the Mooctest server. The server compiled the submitted test cases, executed them against the subject programs, measured the branch coverage, and computed the mutation score. The achieved branch coverage of each submission was sent back to the contestants to help them generate additional test cases, although the achieved mutation score was hidden from them. To ensure a fair contest environment for all, no other tools or coverage plug-ins were allowed. In both contests, multiple submissions were allowed, and the final score of each contestant was determined by his or her last submission. When the timer ran out, the contest ended, and the Mooctest server would no longer accept new submissions.

For the contest organizer, Mooctest offered a web interface to manage and monitor the contest that showed all of the contestants' real-time scores and their statistics, such as average, median, and standard deviation.

SUBJECT PROGRAMS

Before generating test cases for a subject program, a contestant must understand its specifications and the source code. However, this can be difficult for those under a time constraint. Ideally, the subject programs should be reasonably small (but not trivial) so that the contestants will face some challenges but should be able to achieve reasonable scores. Because of this, we did not select subject programs from the well-maintained, open online benchmarks SF100[7] and Defects4J,[8] since the programs on these benchmarks, in general, are too large and depend on many third-party libraries, which make them harder to understand.

To select appropriate subject programs, we measured the software complexity of the programs used in the examinations for the software-testing classes at Nanjing University using four complexity metrics: lines of code (LOC), number of branches, average method complexity (AMC), and average block depth (ABD).[9,10] We then selected open source subject programs for the STC from GitHub and SourceForge. The complexities of these programs in terms of LOC, number of branches, AMC, and ABD are similar to those of the programs used in our software-testing classes at Nanjing University. Our experiences suggest that students (at least those in our classes) could generate test cases achieving reasonable branch coverage and mutation scores. Stated differently, programs with such complexities are neither too difficult nor too easy and are good candidates for testing contests, as they can help us rank contestants. Table 1 shows the values of four complexity metrics of the subject programs used in STC 2016 and STC 2017. For comparison purposes, complexities of programs used in our classes are also included. For each subject program, we used PITest,[11] a state-of-the-art mutation testing tool, to generate mutants using its default mutation operators.[12]

ASSESSMENT CRITERIA

In STC 2016 and STC 2017, we used branch coverage commonly mandated in the industry to evaluate the quality of the generated test cases.[13] We also used mutation score, which has been widely used in academia,[14] to evaluate and rank contestants. In mutation testing, a mutant is a faulty program derived from the original program. It is killed when the original program and the mutant are executed by the same test case but behave differently. For example, given a C program with a statement "return $x/6$," a mutant can be generated by changing the statement to "return $x/5$." This mutant will be killed by a test case $\{x = 5\}$ as the original program returns 0, while the mutant returns 1. However, another test case $\{x = 2\}$ cannot kill the mutant, as both the original program and the mutant return the same value (that is, zero). Once the test execution against the original program and mutants is completed, the mutation score is computed as the total number of mutants killed divided by the number of all generated mutants.

Inspired by a previous unit-testing competition,[10] we combined branch coverage and mutation score together to develop the following ranking equation:

$$\text{Score} = \sum_{P}[\alpha \cdot \text{coverage}_{\text{branch}}(P) + \beta \cdot \text{score}_{\text{mutation}}(P)] \,/\, \text{num}_{P},$$

where P represents a subject program, $\text{coverage}_{\text{branch}}$ and $\text{score}_{\text{mutation}}$ refer to the archived branch coverage and mutant score, respectively, and num_p refers to the total number of subject programs in each contest, where α and β are the weights.

Except for the preliminary stage of STC 2016, where mutation score was not included in the ranking equation, both branch coverage and mutant score were applied in our ranking system with α and β of 0.5. We weighted both factors equally because, to the best of our knowledge, there are no discussions or studies on which ratio of branch coverage to mutation score should be used to measure testing effectiveness. To ensure that contestants were familiar with mutation testing, we gave lectures on mutation testing at several universities. A mutation testing tutorial was available on the STC website.

TABLE 1. Software complexity of the subject programs used in STCs 2016 and 2017.

Contest	Subject program	LOC	Number of branches	AMC	ABD
2016 preliminary	Calculator	352	62	3.23	2.46
	Command	691	142	2.22	1.79
2016 final	Elevator	434	77	2.25	2.20
	MoreTriangle	131	11	1.80	1.77
	NBC	174	27	1.97	1.78
2017 preliminary	Datalog	288	56	2.00	1.95
	QuadTree	298	45	2.10	2.10
	JMerkle	367	85	3.44	2.57
2017 semifinal	CMD	227	31	1.52	2.84
	ITClocks	496	83	1.52	2.84
2017 final	BPlusTree	384	69	2.37	2.20
	JCLO	194	57	4.56	2.36
	SuffixArray	157	46	4.26	2.21
For comparison purposes, complexities of programs used in our classes at Nanjing University are listed below.					
Testing classes at Nanjing University	NodeFlatIndex	405	91	2.64	2.23
	GeneticAlgorithm	229	40	2.44	2.1
	FPTree	260	49	2.87	2.54
	AbstractMatrix	282	55	2.05	1.8

EMPIRICAL STUDIES USING DATA COLLECTED FROM CONTESTS

As the demands for reliable and trustworthy software increase dramatically, improving the software-testing skills of undergraduates, graduate students, and practitioners is important. Software-testing contests can provide significant contributions to our community, as we can make software testing and its techniques more visible. Our contests are more valuable because we have created a data repository, the Software Testing Contest Data Repository (STCDR) at http://www.iselab

.cn/contest/data/. It includes data collected from STC 2016 and STC 2017 that can be accessed by the public. Statistics are also included, such as coverage achievement and mutation scores of test cases generated by each contestant and the average over all the contestants.

Using the data from STCDR, we conducted empirical studies to answer two research questions.

- Does branch coverage have a strong correlation with mutation score in unit testing?
- Does test order at class level have an impact on the effectiveness of unit testing?

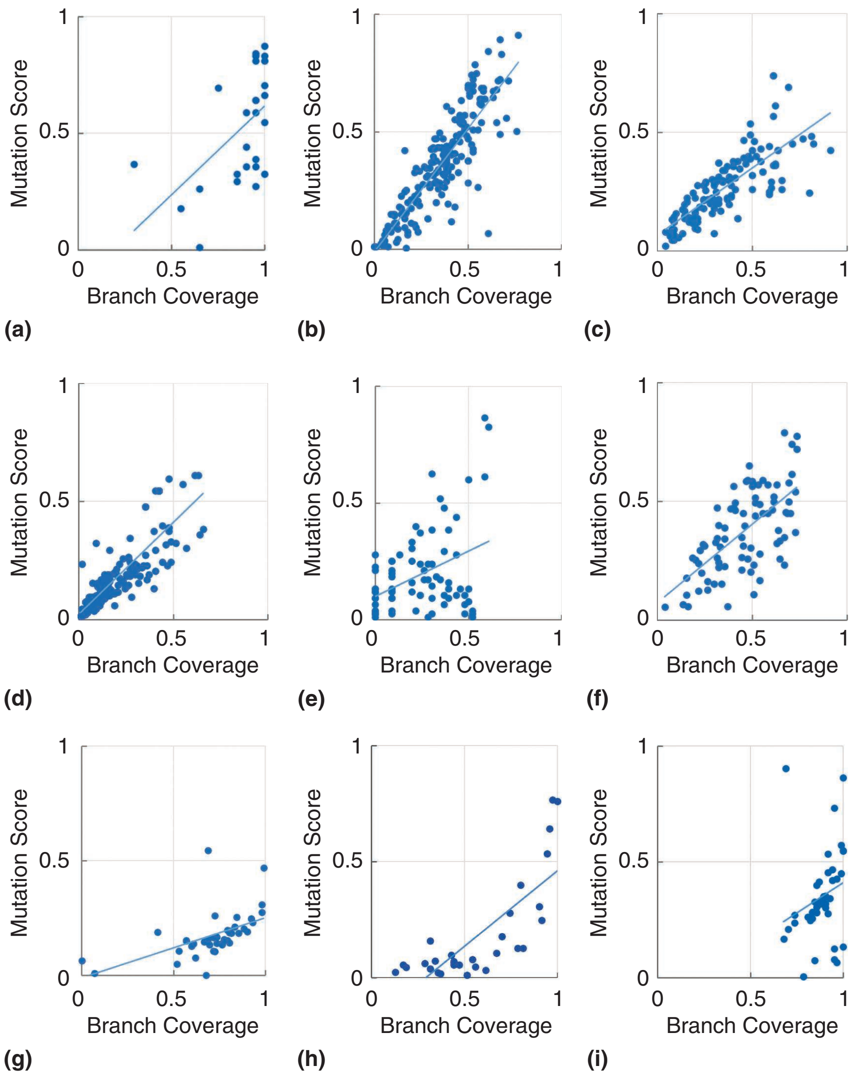


FIGURE 1. A set of scatterplots of mutation scores (vertical axis) against branch coverage (horizontal axis) for nine subject programs. (a) MoreTriangle, (b) Datalog, (c) QuadTree, (d) JMerkle, (e) CMD, (f) ITClocks, (g) BPlusTree, (h) JCLO, and (i) SuffixArray.

Does branch coverage have a strong correlation with mutation score in unit testing?

Studies[15–17] analyzed the correlation between branch coverage and mutation score. However, the test cases in those studies were all randomly generated. This can be a threat to validity, as not all test cases used in the industry are randomly created. Since our test cases were manually created, this provides an opportunity to analyze the correlation in a more realistic setting. In our experiments, some test suites generated by the contestants cannot be used in our analysis due to the limitations of the mutation testing tool—PITest. First, a test suite is incomplete if it contains no assertion, which is required by PITest to kill mutants. Those for the preliminary stage of STC 2016 are in this category. Second, if the correct program does not pass all of the assertions of a test suite, PITest stops the mutation testing without measuring its mutation score. In addition, two programs, Elevator and NBC, were not included because they did not have at least 25 test suites to satisfy a suggested threshold for conducting the Pearson's correlation (PC) analysis.[18] As a result, we conducted the analysis using 846 test suites of nine subject programs.

Figure 1 shows the scatterplots of mutation scores and branch coverage for each subject program, with a straight line representing the trend. We observed a positive correlation between branch coverage and mutation score in all subject programs. We computed the PC coefficient using SPSS software to measure the linear correlation of branch coverage and mutation score for each subject program. The details are shown in Table 2.

The first two columns present the subject programs and their corresponding PC. In seven subject programs, we observed moderate or strong positive correlation (PC > 0.40).[19] In two subject programs (CMD and SuffixArray), the correlations are positive but weak, different from those of the other subject programs. We manually inspected these two subject programs along with the test suites generated by contestants and compared them with other subject programs. We found that all branches of CMD are in seven out of 52 methods. In other words, the remaining 45 methods of CMD do not have any branches. As a result, the correlation between mutation score and branch coverage is weakened, since either of them can be achieved independently. This matches our results as shown in Figure 1(e). Regarding SuffixArray, we found that some mutations were rarely killed by the contestants, which also weakened the correlation. Since we used the same mutation operators for all subject programs, this suggests that SuffixArray was more difficult for the contestants. To determine whether the observed correlation is statistically significant or not, we applied the paired Wilcoxon test and carried out the two-tailed alternative hypothesis. The third column presents the significant values. The *p* values of all tests range from 0.141 to 0.001. Therefore, we can accept the alternative hypothesis that branch coverage has a significant positive correlation with mutation score, with a confidence level of at least 0.859.

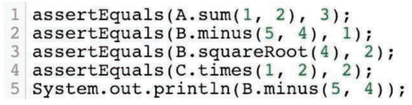
Does test order at class level have an impact on the effectiveness of unit testing?

A good test order at class level can reduce the number of test stubs and increase testing efficiency.[20] This raises an interesting question of whether test order has an impact on testing effectiveness. One challenge of using data from open source repositories such as GitHub is that the time spent on test generation can have an impact on testing effectiveness. For example, given two test suites generated by two testers who have similar testing skills and domain knowledge, a reasonable assumption is that the test suite that took more time to generate may achieve better effectiveness. Since we do not know how much time was spent on the test generation, the evaluation can be biased. Using the data from our contests can diminish this threat because the time spent on test generation by each contestant is almost the same.

In both contests, we observed that contestants tested their subject programs differently. We first identified eight test orders based on feedback from practitioners in the industry, researchers in academia, and some contestants. The first two test orders are in alphabetically ascending and descending orders (ALPHA ASC/DESC) based on the names of the Java classes. In the contest, the Eclipse IDE used by each contestant listed the classes of subject programs in alphabetical order. This could have led the contestants to test the classes in an alphabetically ascending or descending order. The third and fourth test orders are ascending and descending orders based on the size of the subject program (LOC ASC/DESC), which is measured by LOC. These two test orders are from the examination strategies—working on the easiest or hardest first. The other four test orders are based on the forward and backward directions (the direction or reverse direction of

TABLE 2. Correlations between branch coverage and mutation score.

Subject program	PC	*p* Value	Description
MoreTriangle	0.538	0.005	Positive, moderate, with a significance level of 0.01
Datalog	0.861	<0.001	Positive, very strong, with a significance level of 0.01
QuadTree	0.781	<0.001	Positive, strong, with a significance level of 0.01
JMerkle	0.831	<0.001	Positive, very strong, with a significance level 0.01
CMD	0.397	<0.001	Positive, weak, with a significance level of 0.01
ITClocks	0.650	<0.001	Positive, strong, with a significance level of 0.01
BPlusTree	0.533	<0.001	Positive, moderate, with a significance level of 0.01
JCLO	0.778	<0.001	Positive, strong, with a significance level of 0.01
SuffixArray	0.234	0.141	Positive, weak



```
1 assertEquals(A.sum(1, 2), 3);
2 assertEquals(B.minus(5, 4), 1);
3 assertEquals(B.squareRoot(4), 2);
4 assertEquals(C.times(1, 2), 2);
5 System.out.println(B.minus(5, 4));
```

FIGURE 2. A sample test suite.

a directed relationship) of the dependency (DEPEND) and association (ASSN) relationship of Unified Modeling Language (UML) class diagrams.

We determine the test order favored by a contestant in four steps. First, for a subject program, we identify a set of pilot test sequences (PTSs) with respect to each test order. Second, for each contestant, we identify his or her TS on each subject program, based on the sequence of classes invoked by his or her methods where each is a parameter of an assertion. Third, we compare the identified TS with each of the eight PTSs. Regarding ALPHA (ASC/DESC) and LOC (ASC/DESC), if the TS is the longest common subsequence between the TS and the only element of the PTS, we determine that the corresponding test order of the PTS is used once. Regarding DEPEND (FWD/BWD) and ASSN (FWD/BWD), we obtain the reverse PTS (RPTS), which contains the reverse sequence of each element in the PTS. If any subsequence of length two of the TS is not in the RPTS and there is at least one in the PTS, we determine that the corresponding test order is used once. Fourth, if multiple test orders are equally used by a contestant, we randomly select one as his or her preferred test order.

Take a program with four classes, named *A*, *B*, *C*, and *D*, and the test suite as shown in Figure 2 as an example. In this case, $\text{PTS}_{\text{ALPHA(ASC)}}$ is {(*A*, *B*, *C*, *D*)}, while the TS of the test suite is (*A*, *B*, *C*). Note that *B* is not considered to be tested at line 5 because it is not a parameter of an assertion. As a result, we determine that alphabetically ascending order is used once, as the TS is the longest common subsequence between the TS and the only element of $\text{PTS}_{\text{ALPHA(ASC)}}$. Regarding forward dependency test order, assume *A* and *B* depend on *C*, while *D* has no dependencies. In this case,

- $\text{PTS}_{\text{DEPEND(FWD)}}$ is [(*A*, *C*), (*B*, *C*)].
- $\text{RPTS}_{\text{DEPEND(FWD)}}$ is [(*C*, *A*), (*C*, *B*)].
- TS is (*A*, *B*, *C*). All subsequences of length two of the TS are [(*A*, *B*), (*A*, *C*), (*B*, *C*)].

As a result, forward dependency test order is determined to be used once because (*A*, *C*) and (*B*, *C*) are in $\text{PTS}_{\text{DEPEND(FWD)}}$, and no subsequences of length two of the TS are in $\text{RPTS}_{\text{DEPEND(FWD)}}$.

To measure the test order accurately, we used the data from the contestants from the preliminary stage of STC 2017 who wrote only one JUnit class with one test function for each subject program. We ran the analysis 10 times to obtain the results on average to reduce bias introduced by randomization. The results are shown in Table 3.

The results show that, although the alphabetical order by class names (given by Eclipse IDE) is the first order presented to contestants, it is not the favorite test order. Nearly 30% of the contestants chose the famous "answering the easiest or hardest question first" examination strategy in the contest. Approximately 25% of the contestants tested the subject programs based on their dependency relationships, with the forward dependency test order being more popular. This indicates that, if contestants discovered a referenced class (e.g., class B) when they tested a class (e.g., class A), it was likely that they would test class B after class A. In addition, there was a small portion of the contestants who favored the association-based order.

The average score of the contestants who used ALPHA (ASC) is only slightly lower than that of the contestants who used ALPHA (DESC) by 2.25. This met our expectation, as we believed that testing using ascending versus descending alphabetical order by class name should not cause a significant impact. The average score of LOC (ASC) is higher than that of LOC (DESC) by 7.92. This suggests that the strategy of answering the easiest question first

TABLE 3. Test orders favored by contestants and their average scores in the preliminary stage of STC 2017.

	ALPHA (ASC)	ALPHA (DESC)	LOC (ASC)	LOC (DESC)	DEPEND (FWD)	DEPEND (BWD)	ASSN (FWD)	ASSN (BWD)	OTHER
Percentage	13.72%	6.35%	21.01%	7.91%	20.47%	4.32%	11.15%	2.43%	12.64%
Average score	17.91	20.16	26.95	19.03	27.20	20.07	27.14	9.14	37.94

is more effective than the strategy of answering the hardest question first. Regarding the test orders based on UML relationships, both forward test orders—DEPEND (FWD) and ASSN (FWD)—have almost the same average score. The average scores of these forward test orders are higher than those of the backward test orders—DEPEND (BWD) and ASSN (BWD)—by 7.13 and 18.0, respectively. This suggests that both test orders based on the UML relationship—dependency and association—can be equally effective, and the forward test order is more effective than the backward test order. Contestants who used other test orders achieved the highest average score of 37.94.

OBSERVATIONS AND LESSONS LEARNED

By hosting software-testing contests, we can improve students' software-testing skills and also collect real testing data for research on software testing and engineering.

Mutation testing has been proposed to measure the fault detection strength of test cases based on the mutation score. However, mutation testing might not be feasible due to its high execution cost. Our analysis using 846 manually created test suites shows that there is a significant and moderate to strongly positive correlation between branch coverage and mutation score. This suggests that branch coverage can still be used as an alternative when mutation testing is not feasible.

In addition to the correlation analysis between branch coverage and mutation score, we also analyzed the testing effectiveness of different test orders and their popularity. Three interesting observations were drawn from the analysis results. First, the answering the easiest question first strategy performed better than the answering the most difficult question first strategy. Second, forward UML-based test orders performed better than backward UML-based test orders. Third, the test order "Other" achieved the highest average score of 37.94, and it is noticeably higher than the second highest average score of 27.20. This observation raises the question of whether there were specific test orders we did not identify. In our experimental design, we analyzed eight test orders based on feedback from practitioners in the industry, researchers in academia, and some contestants. We are confident that we did not miss any specific test order in our analysis. If this is true, this could suggest that flexible test order could achieve good effectiveness. Nevertheless, we will conduct more experiments to further investigate this observation.

THREATS TO VALIDITY

A potential threat to the first experiment is that the difficulty, in general, of killing mutants can impact the analysis results. We mitigated this threat by applying PITest[11] with its default mutation operators.[12] This experiment setting has also been applied in other empirical studies and tool contests.[2,14] In addition, our first empirical study used nine subject programs with similar values of four software complexity metrics. If the complexity can cause a significant impact, our empirical analysis results may not be generalized to all programs. For the second experiment, we used randomization as a tie breaker to determine the favorite test order for each contestant. We reduced the bias introduced by the randomization by conducting the analysis 10 times.

Although we asked contestants to use the concept of unit testing and the JUnit library to perform unit testing, integration testing might be performed on some classes of the subject programs. This is caused by class dependencies, as some classes depend on other classes during execution. In practice, it is recommended to exclude these dependencies from the testing environment as much as possible to prevent unexpected failures caused by other modules instead of the module being tested. In STC 2016 and STC 2017, the contestants were not asked to deal with the dependency issue during the testing. As a result, our observations and findings might not conform to the strict unit-testing settings. In the future, we will alleviate this by asking contestants to deal with dependencies in unit testing.

We are expanding the contest worldwide to reach more students and affiliations. We are carefully designing a controlled experiment to quantitatively analyze the impact of the contest. We will also further revise our contest ranking equation by including more factors. In addition to the regular sections, a new section, Testathon, will be added to our contest. In this section, contestants will be given two days to test a much larger software system. The data from Testathon will be used for cross comparison with the data from the regular sections. More state-of-the-art automated testing tools (e.g., EvoSuite[3]) will be integrated into the Mooctest platform to conduct human versus machine analysis. ■

ABOUT THE AUTHORS

XINGYA WANG is with the State Key Laboratory for Novel Software Technology, Nanjing University, China. His research interests include software testing and blockchain-based systems. Wang received a Ph.D. in computer science from China University of Mining and Technology. Contact him at xingyawang@smail.nju.edu.cn.

WEISONG SUN is a graduate student in the Software Institute at Nanjing University, China. His research interests include software testing and blockchain-based systems. Sun received a B.E. in software engineering from Yangzhou University, China. Contact him at weisongsun@smail.nju.edu.cn.

LINGHUAN HU is a Ph.D. student in software engineering at the University of Texas at Dallas. His research interests include combinatorial testing and test generation. Hu received a M.S. in software engineering from the University of Texas at Dallas. Contact him at linghuan.hu@utdallas.edu.

YUAN ZHAO is a software engineer at Mooctest. His research interests include test generation and program synthesis. Zhao received an M.S. from the Software Institute at Nanjing University, China. Contact him at allenzcrazy@gmail.com.

W. ERIC WONG is a professor and the director of the Advanced Research Center for Software Testing and Quality Assurance in Computer Science at the University of Texas at Dallas. His research focuses on helping practitioners improve the quality of software while reducing the cost of production. Wong received a Ph.D. in computer science from Purdue University, West Lafayette, Indiana. He is a corresponding author for this article. Contact him at ewong@utdallas.edu.

ZHENYU CHEN is the founder of Mooctest and a professor at the Software Institute, Nanjing University, China. His research interests focus on software analysis and testing. Chen received a Ph.D. in mathematics from Nanjing University, China. He is a corresponding author for this article. Contact him at zychen@nju.edu.cn.

REFERENCES

1. W. Eric Wong, X. Li, and P. A. Laplante, “Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures,” *J. Syst. Softw.*, vol. 133, pp. 68–94, Nov. 2017.
2. A. Panichella and U. R. Molina, “Java unit testing tool competition: Fifth round,” in *Proc. IEEE/ACM 10th Int. Workshop on Search-Based Software Testing*, 2017, pp. 32–38.
3. G. Fraser and A. Arcuri, “EvoSuite: Automatic test suite generation for object-oriented software,” in *Proc. SIGSOFT/FSE’11 ACM SIGSOFT Symp. Foundations of Software Engineering*, 2011, pp. 416–419.
4. “IEEE Reliability Society.” Accessed on: 2018. [Online]. Available: http://rs.ieee.org/
5. MoocTest. Accessed on: 2018. [Online]. Available: http://www.mooctest.net/login2
6. “National student contest of software testing,” 2018. [Online]. Available: https://swtesting.techconf.org/
7. G. Fraser and A. Arcuri, “Sound empirical evidence in software testing,” in *Proc. Int. Conf. Software Engineering*, 2012, pp. 178–188.
8. D. Jalali and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for Java programs,” in *Proc. Int. Symp. Software Testing and Analysis*, 2014, pp. 437–440.
9. T. J. Mccabe, “A complexity measure,” *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, 1976.
10. S. Bauersfeld, T. E. Vos, K. Lakhotia, S. Poulding, and N. Condori, “Unit testing tool competition,” in *Proc. IEEE 6th Int. Conf. Software Testing, Verification and Validation Workshop*, 2013, pp. 414–420.
11. H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, “PIT: A practical mutation testing tool for Java (demo),” in *Proc. Int. Symp. Software Testing and Analysis*, 2016, pp. 449–452.
12. H. Coles, “Mutation operators of PITest,” 2018. [Online]. Available: http://pitest.org/quickstart/mutators/

13. T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proc. Int. Conf. Software Engineering*, 2017, pp. 597–608.
14. D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proc. ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, 2014, pp. 654–665.
15. J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, 2006.
16. P. G. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness," *ACM SIGSOFT Softw. Eng. Notes*, vol. 23, no. 6, pp. 153–162, 1998.
17. L. Inozemtseva and R. Holmes. "Coverage is not strongly correlated with test suite effectiveness," in *Proc. 36th Int. Conf. Software Engineering*, ACM, 2014, pp. 435–445.
18. F. N. David, *Tables of the Ordinates and Probability Integral of the Distribution of the Correlation Coefficient in Small Samples*, Cambridge, UK: Cambridge University Press, 1954.
19. J. D. Evans, *Straightforward Statistics for the Behavioral Sciences*. Pacific Grove, CA: Brooks/Cole, 1996.
20. L. C. Briand, Y. Labiche, and Y. Wang, "An investigation of graph-based class integration test order strategies," *IEEE Trans. Softw. Eng.*, vol. 29, no. 7, pp. 594–607, 2003.

Digital Object Identifier 10.1109/MC.2019.2937702