JTESTMIGBENCH and JTESTMIGTAX: A benchmark and taxonomy for unit test migration

Ajay Kumar Jha
North Dakota State University
Fargo, USA
ajay.jha.1@ndsu.edu

Mohayeminul Islam, Sarah Nadi
University of Alberta
Edmonton, Canada
mohayemin@ualberta.ca, nadi@ualberta.ca

***Abstract*—Unit tests play a critical role in improving software quality. However, writing effective unit tests from scratch is difficult and tedious. One way to reduce this difficulty is to recommend existing tests of semantically similar functions. However, modifying the recommended tests manually might still be difficult and tedious. For example, developers have to understand various code elements in the recommended tests to accurately replace them with semantically similar code elements from the target application. One way to mitigate the issue is by developing a test migration or reuse technique that could automatically transform the code elements in the recommended tests and migrate them to the target application. However, to develop such a technique, we first need to identify what types of code transformations are required to successfully migrate the recommended tests. Therefore, in this paper, we first recruit two external participants to create JTESTMIGBENCH, a benchmark of 510 manually migrated JUnit tests for 186 methods from five popular libraries. We then analyze the code changes in the migrated tests to create JTESTMIGTAX, a taxonomy of test code transformation patterns. Our contributions provide the necessary foundations to develop automated unit test migration or reuse techniques.**

***Index Terms*—unit test migration, reuse, code transformation, transplantation, benchmark, taxonomy**

I. INTRODUCTION

Unit testing is a popular form of testing, where developers test the behavior of individual components of a software system. Unit testing plays a vital role in improving the quality of software by detecting bugs earlier. However, writing effective unit tests manually from scratch is difficult and time-consuming. To alleviate the difficulties, researchers have proposed various unit test recommendation techniques [1]–[4], which recommend unit tests based on the semantic similarity of the method to be tested and another method for which unit tests exist. Developers can then manually modify the recommended tests to test the target method. Although these techniques reduce the difficulty of writing tests by suggesting potential test oracles [2], developers still have to put substantial effort into modifying the recommended tests. Even semantically similar methods have some syntactical and behavioral differences [5], which developers need to adjust in the recommended tests. They also have to identify the equivalent counterparts of other code elements in the recommended tests (e.g., types or constructor calls) from the target application.

One way to further reduce or eliminate the manual effort required in modifying recommended tests is by developing test migration or reuse techniques that automatically transform the recommended tests. However, to develop such techniques, we have to first identify what types of code transformations or transplantations are required to successfully transform and migrate the recommended tests, which in turn requires a benchmark of migrated tests. Current test reuse and adaptation techniques [6], [7] do not provide an accessible benchmark of reused tests, or their data contains a limited number of reused JUnit tests. Moreover, these techniques do not support much code transformation, because they reuse tests in the modified version of the same application or adapt limited code elements (i.e., test oracles) to insert them into manually created test templates. Therefore, even the available benchmarks are not suitable to identify general code transformation patterns.

To address these gaps and create foundations for developing test migration techniques, in this paper, we first recruit two external participants to create JTESTMIGBENCH, a benchmark of 510 manually migrated JUnit tests for 186 methods from five popular libraries. We then analyze the code changes in JTESTMIGBENCH to create JTESTMIGTAX, a taxonomy of test code transformation patterns. JTESTMIGTAX can help researchers in identifying different types of code transformation that test migration techniques need to support to successfully migrate unit tests. Researchers can also use JTESTMIGTAX to compare test migration techniques based on what types of code transformation they support. Finally, researchers can use JTESTMIGBENCH to evaluate test migration techniques. Overall, our contributions are first steps to facilitate the development, comparison, and evaluation of test migration techniques. All our data is available on our artifact page[1].

II. CONSTRUCTING JTESTMIGBENCH AND JTESTMIGTAX

To create JTESTMIGBENCH and JTESTMIGTAX, we use a combination of automated and manual approaches. We first select applications that might have common functionalities. We then use an automated approach to find semantically similar method pairs in the selected applications. We manually validate similar method pairs and check for the existence of unit tests. We then recruit two external participants to manually migrate tests to create JTESTMIGBENCH. Finally, we analyze the code changes in the migrated tests to create JTESTMIGTAX. We now describe each step.

A. Select Applications

To migrate unit tests, we first need a collection of applications or libraries that might have semantically similar

[1]https://www.github.com/STAM-NDSU/JTestMigBench

methods across them. To identify such applications, we leverage existing studies that have investigated similar software components across libraries [2], [7], [8]. From these studies, we select libraries from the JSON domain, because several popular JSON libraries are available. Among the 24 Java JSON libraries listed on the official JSON website[2], we select the top three libraries, JSON-java, google-gson, and fastjson, based on the stargazer and fork counts of their GitHub repositories. Additionally, we select commons-lang and guava libraries from the common utilities domain, because they are commonly investigated libraries for test recommendation [2], [7]. Overall, this results in four unique library pairs, three from the JSON domain and one from the utilities domain.

B. Find Semantically Similar Candidate Method Pairs

To find semantically similar candidate method pairs in the selected libraries, we build a tool named SMFINDER. It primarily leverages Word2Vec [9] to find semantically similar methods. We generate our Word2Vec model using method signatures present in the Java classes of 29,271 Android applications from AndroZooOpen [10]. For each method signature, we consider the method name, parameter types, and return type and split them by Camel case. The model produces a feature vector for each word, and the similarity between two words is determined by the cosine similarity between their vectors.

Given method names N_s and N_t, SMFINDER first splits N_s and N_t by Camel case and stores the words into two separate lists L_s and L_t, respectively. For each word in L_s, SMFINDER then computes the cosine similarity between the word and each word in L_t and identifies the best matching word from L_t based on the highest cosine similarity score. SMFINDER then computes the similarity score S_m by taking the average of the highest cosine similarity scores obtained for each word in L_s. For example, S_m for `loadExceptionCount` and `loadFailureCount` is calculated as follows:

$$\begin{array}{l} \\ load \\ exception \\ count \end{array}\begin{array}{c} \begin{array}{ccc} load & failure & count \end{array} \\ \begin{bmatrix} \mathbf{1.0} & 0.2 & 0.0 \\ 0.2 & \mathbf{0.6} & 0.1 \\ 0.0 & 0.0 & \mathbf{1.0} \end{bmatrix} \end{array} = (1.0 + 0.6 + 1.0)/3 = \mathbf{0.9}$$

SMFINDER considers a method pair as semantically similar if the value of S_m is greater than a threshold value. If SMFINDER finds more than one similar method in the target application, it selects the method with the highest similarity score as the similar method. If multiple methods have the same highest similarity score, SMFINDER uses Levenshtein [11] to resolve the conflict and select a similar method.

We set the similarity threshold value to 0.75 and run SMFINDER on the selected libraries. SMFINDER finds 365 and 1,571 candidate similar method pairs in the three JSON libraries and two common utilities libraries, respectively.

C. Validate Candidate Method Pairs and Check for Tests

We manually validate the candidate similar method pairs found by SMFINDER and check for existing unit tests. We assign each method pair to two reviewers. Specifically, two authors validate method pairs from two JSON libraries, and one of the same authors and a senior undergraduate student[3] with one year of industrial experience validate pairs from the remaining two library pairs. We discuss any disagreements that arise. We measure inter-rater agreement using Cohen's kappa.

We validate all the candidate similar method pairs in the JSON libraries and confirm a total of 127 similar method pairs, where also at least one method in the pair has tests. We find that the kappa score is 0.74, which is a substantial agreement. However, given the large number of candidate pairs in the utilities libraries, we use a different strategy to select similar method pairs to review. We target including 20 method pairs from the utilities libraries. To select these 20 pairs, we sort the 1,571 candidate similar method pairs based on their similarity score. We manually go through the pairs in this sorted list (from most similar to least similar) to verify their similarity and to identify if any method in the pair has a test. We stop when we reach our goal of 20 similar method pairs with tests. We manually validated 32 candidate method pairs in the sorted list to get the 20 similar method pairs.

Overall, we collect 147 similar method pairs from the selected libraries, where at least one method in the pair has tests. Note that if both methods in a pair have tests, then we can migrate tests in both directions. For example, given a method pair `max` and `maxNum` where both methods have tests, we can migrate the tests of `max` in the first library to `maxNum` in the second library and vice-versa. Among the 147 similar method pairs, we find 52 method pairs that have tests in both methods. Taking that into account, overall, we consider test migration for 199 methods from five different libraries. Table I shows the number of similar method pairs in each library pair.

TABLE I: Number of Confirmed Similar Method Pairs

Source Library	**Target Library**	**# Similar Method Pairs**
JSON-java	google-gson	27
google-gson	JSON-java	11
JSON-java	fastjson	31
fastjson	JSON-java	37
google-gson	fastjson	42
fastjson	google-gson	31
commons-lang	guava	10
guava	commons-lang	10
Total	**-**	**199**

D. Migrate Tests Manually

To manually migrate tests for the 199 methods, we ask two members of our research group, a graduate student[4] with five years of experience in Java including two years of industrial experience, and the member who validated candidate similar method pairs. Since we migrate tests in both directions, such as JSON-Java to google-gson and vice-versa, we assign the methods in one direction to one test migrator and the methods in another direction to the other test migrator. We use this technique for JSON libraries, whereas we assign all the methods from the utilities libraries to the same test

[2]https://www.json.org/json-en.html

[3]Xichen Pan, University of Alberta

[4]Mansur Gulami, University of Alberta

migrator to manage our human resources. Overall, we assign 89 methods to one test migrator and 110 methods to the other test migrator. We then provide each test migrator with a sheet that contains the name of the source library, source class, source method, source test class, target library, target class, and target method for each validated method pair. We ask them to manually migrate tests for each method pair with the following instructions: (1) create a separate test class for each target method, (2) identify test methods in the source test class that test the source method (i.e. identify the tests that need to be migrated), (3) copy the identified test methods along with any helper method you want to use into the newly created test class (target test class), and (4) modify the copied test methods and their helper methods in the target test class so that the modified tests pass. We ask them to provide us with their final migrated tests along with the following information: (1) whether the migrated tests passed or failed, (2) if failed, the reasons for failure, and (3) any open comments.

Overall, the test migrators identify and attempt the migration of 571 unit tests for the 199 methods. Among the attempted migration, they successfully migrate 510 unit tests for 186 methods, which we include in JTESTMIGBENCH. We analyze the migrators' comments to understand the reasons for migration failures. We find that migrations failed mainly due to implementation differences or the unavailability of similar software components across libraries.

To give us confidence in the benchmark, we assess the quality of the migrated tests by measuring line and branch coverage at the method level using JaCoCo[5]. We consider two different scenarios: (1) coverage for only migrated tests without considering any existing tests for the target method (*do migrated tests test the method's functionality?*), and (2) increased coverage of migrated tests w.r.t any existing tests for the target method (*do migrated tests augment existing tests?*). To measure increased coverage, we first measure coverage of the existing tests C_{old}. If tests do not already exist for the target methods, we record C_{old} as 0%. We then measure coverage of the migrated and existing tests combined C_{new}. We calculate the increased coverage as C_{in} = C_{new} - C_{old}.

E. Identify Code Transformation Patterns

For each successfully migrated test in JTESTMIGBENCH, we compare the migrated test with the corresponding original or source test. We then analyze code changes in the migrated test to identify code transformation patterns to create JTESTMIGTAX. We use a combination of open and closed coding approaches to create JTESTMIGTAX. The first author of this paper first manually reviews code changes in 50 randomly selected migrated tests and writes notes, such as `replace a constructor`, to describe the code changes. Based on these notes, the first author creates labels for the code changes. The second author then uses a closed coding approach to separately label the code changes in the 50 migrated tests that the first author labeled, which means the second author only uses the labels already created by the first author. The first and second authors then discuss the results to resolve any disagreements. We find that the kappa score is 0.94. Given an almost perfect agreement, the first author then labels the code changes in the remaining migrated tests and creates JTESTMIGTAX.

[5]https://www.eclemma.org/jacoco/

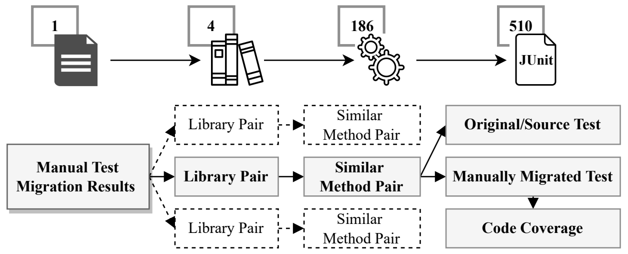

Fig. 1: Data included in JTESTMIGBENCH.

III. JTESTMIGBENCH

JTESTMIGBENCH as shown in Figure 1 contains the 510 successfully migrated JUnit tests with their corresponding original tests and the code coverage data. JTESTMIGBENCH also contains all the supporting data, which includes a manual test migration results sheet containing all the information we provided to the manual test migrators for test migration and the responses we received from them. It also includes the library pairs we selected for test migration, the candidate similar method pairs found by SMFINDER in the library pairs, and the 186 manually validated similar method pairs.

We find that the median value for both line and branch coverage of the migrated tests for the 186 methods is 100%. The line and branch coverage of the migrated tests for only 3 out of 186 methods is 0%, because these methods throw exceptions without completing the execution. The results show that the migrated tests generally cover a large percentage of the code of the 183 methods, which indicates that the migrated tests are useful for testing the methods' functionalities. We also find that line or branch coverage increased for 35 of the 186 methods. Among these 35 methods, 11 methods already have tests, while the remaining 24 methods do not. The median increase in line and branch coverage values for the 35 methods are 67% and 50%, respectively. Note that the libraries used in our study are highly popular with many unit tests. Despite that, we find the migrated tests for the 35 methods provide some practical values in terms of increased coverage. This motivates the benefits of the concept of test migration, even in the presence of existing tests.

IV. JTESTMIGTAX

Figure 2 shows JTESTMIGTAX, a taxonomy of migration-related code changes. JTESTMIGTAX has three different hierarchy levels. The first level shows the types of operations we observed in the migrations, such as *replace* and *change*. The second level shows the program elements on which the operations are performed such as *type* and *method call*. The third level shows more specific information about the operations or program elements. For example, *one-to-many* under *method call* specifies one method call replaced with

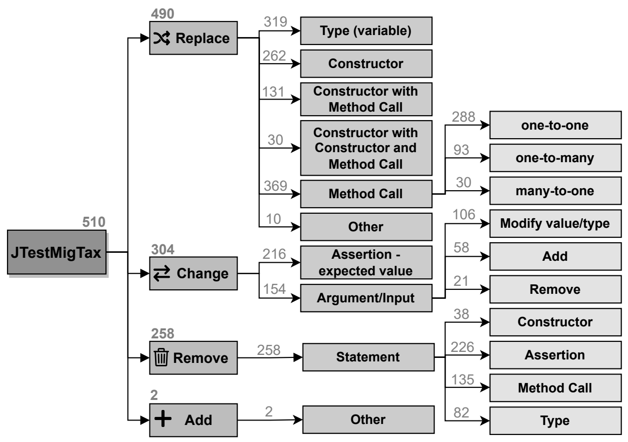

Fig. 2: JTESTMIGTAX: Taxonomy of migrated-related code changes

multiple method calls. JTESTMIGTAX also shows the number of migrated tests in JTESTMIGBENCH with each type of code change. For example, among the 510 successfully migrated tests, 490 tests required the replace operation, among which 319 required type replacement. We now briefly describe the code changes in JTESTMIGTAX.

Unit tests are mainly composed of method/constructor calls and types. Therefore, most of the successfully migrated tests (96%) required the replacement of types or method/constructor calls. Only 20 (4%) migrated tests did not require any replacement of program elements, because these program elements have the same name in the source and target libraries. Although method and constructor calls are syntactically different program elements, we find that 160 (31%) of the migrated tests required the replacement of constructor calls with method calls or a combination of method and constructor calls. We also find that the library pairs in our benchmark use different numbers of methods to implement the same functional behavior. Therefore, 93 (18%) migrated tests required the replacement of one method call with multiple method calls, whereas 30 (5%) migrated tests required the replacement of multiple method calls with one method call.

Test oracles are other key elements of unit tests. A test oracle compares the actual output of a method with the expected output to determine whether the method works as specified, usually through an assert statement. The actual output depends on the test inputs passed to the method as arguments. We find that 304 (60%) migrated tests required changes in test oracles, among which 216 (42%) tests required changes in expected outputs and 154 (30%) tests required changes in inputs/arguments. Note that a large number of these changes are only type conversions. For example, among the 106 migrated tests that required an argument modification, 66 tests required only argument type conversion, such as changing `add("Hello", new JsonPrimitive(1))` to `put("Hello", 1)`.

Finally, 258 (51%) tests required the removal of statements including assertions, types, constructors, and method calls, mainly because they are irrelevant to test the target methods.

V. DISCUSSION

JTESTMIGBENCH and JTESTMIGTAX have various implications for advancing unit-test migration techniques.

A. Implications

JTESTMIGTAX shows that automated test migration techniques need to handle mainly three types of code transformation to successfully migrate unit tests.

(1) Finding and replacing semantically similar code elements. The results show that test migration techniques need to find and replace semantically similar code elements, specifically types, constructors, and method calls, which is a challenging task. Most code clone detection techniques still struggle to find semantically similar code accurately [12]. In this work, we used Word2Vec to find similar methods, which is also not 100% accurate. The results also show that test migration techniques also need to consider code replacement between different program element types (i.e., constructors and method calls) and different cardinalities (e.g., one method call with multiple method calls). These types of code transformation further increase the complexity of finding and replacing similar code elements. Overall, to successfully find and replace semantically similar code elements, we need to improve the accuracy of existing code clone detection techniques and develop new techniques that consider the similarity between different types of code elements and different cardinalities.

(2) Modifying test oracles. The majority of the migrated tests (60%) required changes in test oracles. These changes are syntactical (i.e., change in type) and behavioral (i.e., change in value). Test migration techniques need capabilities to handle both syntactical and behavioral changes. One way to handle the syntactical changes is by manually creating a database of convertible types for commonly used data types as done by Sondhi et al. [7], but this is limited. Ideally, we need to develop techniques, perhaps using machine learning, to automatically identify convertible data types. To handle behavioral changes, specifically expected outputs, we could execute the target method and use the actual output as the expected output. However, if the target method has bugs, the expected output would be incorrect. A human-in-the-loop approach could be used to handle both types of changes [13].

(3) Removing irrelevant code elements. A large number of migrated tests (51%) required the removal of code elements that are irrelevant to test the target methods. Therefore, test migration techniques need to identify and remove irrelevant code elements. Otherwise, they might cause compilation errors if the target library does not have their equivalent counterparts. While removing any elements that the migration technique could not transform is one possible solution, this may lead to also removing relevant code. A more sound approach is to perform control/data flow analysis in the original test, remove the code elements that are irrelevant to test the source method, and then perform the other types of code transformation.

In addition to identifying different types of code transformation that test migration techniques need to support, JTESTMIGTAX along with JTESTMIGBENCH can also help in comparing and evaluating test migration techniques. For example, we can compare test migration techniques based on what types of code transformation in JTESTMIGTAX they support. Also, the unit tests in JTESTMIGBENCH can be used to evaluate test migration techniques.

B. Threats to Validity

Internal Validity. We use only one test migrator per test to create JTESTMIGBENCH. However, we measure the coverage of the migrated tests to show the quality of JTESTMIGBENCH. We rely on manual analysis for several steps, which is generally subject to mistakes and biases. To mitigate the threats, two validators independently validated similar method pairs, checked for the presence of tests, and identified code changes.

External Validity. JTESTMIGTAX is based on the code changes in 510 migrated tests for 186 methods. Although JTESTMIGTAX is based on a small number of migrated tests, these tests have diverse types of code changes, mainly due to the highly diverse functional behavior of JSON libraries [8].

Construct Validity. Due to the absence of a test migration benchmark, we first create JTESTMIGBENCH and then identify code changes. However, there is not only one way to modify and migrate tests. To mitigate the threat, we exclude the code changes related to style improvement in JTESTMIGTAX.

VI. Related Work

Test tracing and recommendation are preliminary and necessary steps for test migration. White et al. [14] use various criteria, such as the similarity of method and test names, to automatically establish test-to-code links. Since the automated approach is not 100% accurate, our test migrators use a manual approach to identify the relevant unit tests to migrate. Built on test tracing techniques, test recommendation techniques recommend tests based on the similarity of the methods [1]–[3]. Similar to some of these techniques, SMFINDER also uses word2vec and Levenshtein to find similar methods.

Makady and Walker [6] proposed SKIPPER to reuse unit tests in the modified version of the same application. They removed some code (fields, methods, or classes) from the original application to create a modified version of the application. Therefore, SKIPPER does not need to perform various types of code transformation, which makes it unsuitable to identify general code transformation patterns. Moreover, it does not provide an accessible benchmark of reused tests. White et al. [2] adapted some of the recommended tests manually to show the real-world applicability of test recommendation techniques, but they adapted tests for only 12 methods. Sondhi et al. [5] manually reused unit tests for some of the similar functions they identified across libraries. They later extended this study and proposed METALLICUS to reuse oracles of the recommended tests automatically [7]. However, instead of directly transforming the code of the recommended tests, METALLICUS extracts oracles from the recommended tests and inserts them in the test templates manually created for the target applications. Therefore, METALLICUS only identifies code transformation related to test oracles. Moreover, the dataset includes only 17 reused JUnit tests (i.e., test templates) for 17 methods. Overall, these studies are limited in identifying code transformation patterns or providing reused JUnit tests.

Similar to Zhao et al. [15]'s work that provides a benchmark for evaluating UI test reuse techniques, our work provides a benchmark and taxonomy to facilitate the evaluation and advancement of unit test reuse techniques.

VII. Conclusion

In this paper, we created JTESTMIGBENCH, a benchmark of 510 manually migrated tests for 186 methods from five popular libraries, and JTESTMIGTAX, a taxonomy of migration-related code changes. We also discussed various challenges in transforming and migrating recommended tests to the target applications. Our contributions can facilitate the development, comparison, and evaluation of unit test migration techniques.

References

[1] W. Janjic and C. Atkinson, "Utilizing software reuse experience for automated test recommendation," in *2013 8th International Workshop on Automation of Software Test (AST)*. IEEE, 2013, pp. 100–106.

[2] R. White, J. Krinke, E. T. Barr, F. Sarro, and C. Ragkhitwetsagul, "Artefact relation graphs for unit test reuse recommendation," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 137–147.

[3] C. Zhu, W. Sun, Q. Liu, Y. Yuan, C. Fang, and Y. Huang, "Homotr: online test recommendation system based on homologous code matching," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1302–1306.

[4] R. Qian, Y. Zhao, D. Men, Y. Feng, Q. Shi, Y. Huang, and Z. Chen, "Test recommendation system based on slicing coverage filtering," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 573–576.

[5] D. Sondhi, D. Rani, and R. Purandare, "Similarities across libraries: Making a case for leveraging test suites," in *2019 12th IEEE Conference on Software Testing, Validation and Verification*, 2019, pp. 79–89.

[6] S. Makady and R. J. Walker, "Validating pragmatic reuse tasks by leveraging existing test suites," *Software: Practice and Experience*, vol. 43, no. 9, pp. 1039–1070, 2013.

[7] D. Sondhi, M. Jobanputra, D. Rani, S. Purandare, S. Sharma, and R. Purandare, "Mining similar methods for test adaptation," *IEEE Transactions on Software Engineering*, 2021.

[8] N. Harrand, T. Durieux, D. Broman, and B. Baudry, "The behavioral diversity of java json libraries," *arXiv preprint arXiv:2104.14323*, 2021.

[9] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *arXiv preprint arXiv:1310.4546*, 2013.

[10] P. Liu, L. Li, Y. Zhao, X. Sun, and J. Grundy, "Androzooopen: Collecting large-scale open source android apps for the research community," in *2020 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 2020.

[11] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[12] A. Walker, T. Cerny, and E. Song, "Open-source tools and benchmarks for code-clone detection: past, present, and future trends," *ACM SIGAPP Applied Computing Review*, vol. 19, no. 4, pp. 28–39, 2020.

[13] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "An empirical validation of oracle improvement," *IEEE Transactions on Software Engineering*, 2019.

[14] R. White, J. Krinke, and R. Tan, "Establishing multilevel test-to-code traceability links," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 861–872.

[15] Y. Zhao, J. Chen, A. Sejfia, M. Schmitt Laser, J. Zhang, F. Sarro, M. Harman, and N. Medvidovic, "Fruiter: a framework for evaluating ui test reuse," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1190–1201.