Semi-Automated Test-Case Propagation in Fork Ecosystems

Mukelabai Mukelabai*, Thorsten Berger*†, and Paulo Borba‡
*Chalmers | University of Gothenburg, Sweden
†Ruhr University Bochum, Germany
‡Federal University of Pernambuco, Brazil

***Abstract*—Forking provides a flexible and low-cost strategy for developers to adapt an existing project to new requirements, for instance, when addressing different market segments, hardware constraints, or runtime environments. Then, small ecosystems of forked projects are formed, with each project in the ecosystem maintained by a separate team or organization. The software quality of projects in fork ecosystems varies with the resources available as well as team experience, and expertise, especially when the forked projects are maintained independently by teams that are unaware of the evolution of other's forks. Consequently, the quality of forked projects could be improved by reusing test cases as well as code, thereby leveraging community expertise and experience, and commonalities between the projects. We propose a novel technique for recommending and propagating test cases across forked projects. We motivate our idea with a pre-study we conducted to investigate the extent to which test cases are shared or can potentially be reused in a fork ecosystem. We also present the theoretical and practical implications underpinning the proposed idea, together with a research agenda.**

***Index Terms*—test propagation, code transplantation, forking**

I. Introduction

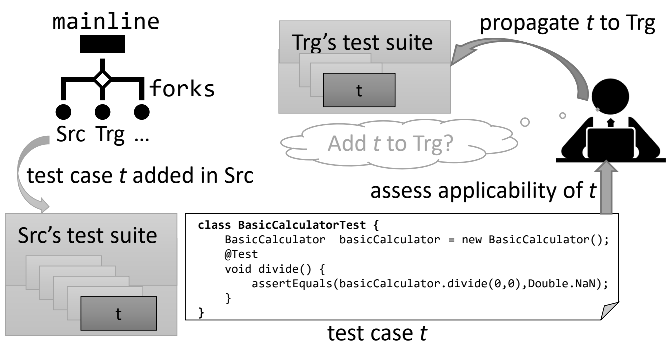
Developers often create forks[1] of a given project to adapt it to new requirements, for instance, different market segments, runtime environments or hardware constraints. Consequently, several forked projects may be created and maintained in parallel, often independently with little-to-no integration of changes between the projects [1]. Such forks can easily form small ecosystems [2] whose projects vary in quality based on the level of quality assurance performed in each project. However, the overlap between the projects provides potential for test-case reuse.

Testing (e.g., unit and regression testing) is widely used [3] to assure software quality. Yet, the level of quality assurance in a project varies with experience, expertise, and resources available [3], [4]. As a result, a fork ecosystem may have projects that have *weak test suites—insufficient test cases*, or *duplicated effort* [5] to improve their test suites, or *too many bugs that have already been resolved* in the mainline repository or other forks [6], [7]. Developers of one project might fix bugs and introduce test cases. Depending on the applicability of such test cases to other projects in the fork ecosystem, which depends on the modifications done in the projects—especially the units under test (UUT)—ideally, other developers could be notified and supported with a semi-automated technique to propagate the test cases.

[1]*Forking* refers to copying a project to create a different product from it

We propose a technique for improving the quality of fork ecosystems by supporting the reuse/sharing of test cases. Since many forked projects exist that are maintained by different organizations or teams, with varying expertise and experience, their quality can be significantly improved by reusing test cases in addition to code.

Figure 1 illustrates our idea for recommending and propagating test cases within fork ecosystems. First, when a developer adds a test case to any project in the ecosystem (whether mainline or fork—here, the test case is added to *Src*), we assess which other projects in the ecosystem can reuse the test case. We refer to this as the *test-case applicability problem*. Here, we check that the behavior of the UUTs, including their dependencies, is the same in both the source and target projects. Given that fork ecosystems may be large, it is infeasible to compare all the context code and follow all dependencies. Thus, the challenge lies in finding a compromise between precision (how much of the context to analyze, since behavior can be changed through adjacent code or even project dependencies) and performance (how fast we perform the analysis given the size of each project and number of forks). Second, the technique recommends the test case to the identified target projects (*Trg* in this case) and if accepted by the project maintainers, propagates the test case. We refer to this as the *test-case propagation problem*. The UUTs may have been modified in both *Src* and *Trg*, in which case we try to automatically adapt t to *Trg* as much as possible, using techniques such as code-clone detection [8], [9] or automated software transplantation [10]. For situations where the adaptation cannot be fully



```
class BasicCalculatorTest {
    BasicCalculator  basicCalculator = new BasicCalculator();
    @Test
    void divide() {
        assertEquals(basicCalculator.divide(0,0),Double.NaN);
    }
}
```

Figure 1: General test-case recommendation and propagation scenario

automated, we make suggestions to the developer regarding which parts need to be manually adapted. The challenge here lies in correctly mapping t's context to the target project and integrating related code (e.g., bug fixes) given different code modification scenarios that may occur in both *Src* and *Trg* (see Sec. II). We discuss these two research problems in detail in sections III and IV.

Our proposed technique allows *(i)* notifying developers about applicable test cases in other forked projects; *(ii)* detecting bugs early when still unknown to other projects; and *(iii)* improved software quality assessment as a result of test cases written by teams with more expertise and experience, thereby also improving the overall software quality in fork ecosystems.

II. Motivation and Background

While bug-fix propagation between forked projects is an active research area [6], [7], [11]–[19], we, in contrast, argue for improved software quality assurance and early detection of bugs in fork ecosystems by propagating test cases instead of bug fixes only. To motivate our proposed technique, we conducted a pre-study to investigate the extent to which test cases are shared within fork ecosystems.

A. Pre-Study Design

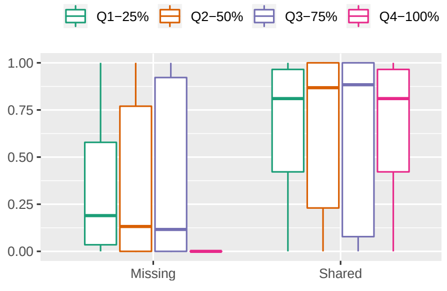
We drew our sample of fork ecosystems from the Android ecosystem [2], [20], which is one of the largest and fastest-growing due to its large set of reused apps. Using the GitHub API, we mined our set of subject projects as follows.

First, we collected mainline (i.e., not forked) Android projects that had 91 or more forks each. This gave us 809 projects. We then filtered this list to obtain only Java projects that had file contents with the word *Test* in it (potentially signifying the presence of test cases); this left us with 58 projects. Each of the 58 projects had between 91 and 9,275 forks. Next, for each mainline project, we collected its forks that had 5 or more commits and had a minimum of 30 days between the first and last commit. We then selected the top 5 forks for each project (sorted in descending order of commits). The final set comprised 26 mainline projects (whose forks met our criteria above) and 64 forks, totaling 90 projects. The 26 mainline projects and their forks constitute 26 ecosystems, which we analyzed. The majority (54 %) had 3 to 6 projects, with 23 % having 6 projects (i.e., 1 mainline and 5 forks), while 46 % had the smallest ecosystem size of 2 projects (1 mainline and 1 fork).

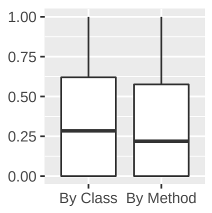
Next, we cloned all projects in each ecosystem and generated *(*i) a list of all classes and methods in each project, and *(*ii) a list of test cases and corresponding UUTs (classes and methods). We identified test cases in files using the test-case annotation keywords *Test, TestCase, SmallTest, MediumTest,* and *TestMethod*.

Subject systems. From the 26 ecosystems, we excluded 10 whose projects had no test-case annotations (e.g., @Test) and 4 more that had fewer than 10 test cases in total. The final set comprised 12 ecosystems totaling 43 projects—12 mainline and 31 forks, with a total of 6,569 test cases. We analyzed this dataset [21] to answer:

- *RQ1: What is the proportion of shared test cases within each fork ecosystem?*
- *RQ2: What proportion of non-shared test cases target similar UUTs in other projects of the ecosystem?*



(a) Shared and missing test cases



(b) Non-shared test cases with matching UUTs

Figure 2: (a) Proportion of missing and shared test cases in 25 %, 50 %, 75 %, and 100 % of the projects in each ecosystem; (b) Proportion of non-shared test cases with matching UUTs in other projects of the ecosystem.

B. Pre-Study Results

RQ1: Proportion of Shared and Missing Test Cases. We consider a test case shared if it exists in two or more projects of a fork ecosystem, and missing if it is absent from at least one project. We calculated the proportions of shared and missing test cases over their sum, i.e., *proportion of shared = shared/(shared+missing)*. We calculated the proportions in four quartiles of projects within an ecosystem; e.g., proportion of test cases missing in 25 %, 50 %, 75 %, and 100 % of projects. To match test cases, we used the fully qualified names of the test methods (i.e., packageName.className.methodName).

Figure 2a indicates that more than half of our ecosystems have over 75 % of test cases present in all their projects. However, the left-hand side of Fig. 2a also indicates that few projects have very high proportions of missing test cases. For instance, 25 % of the ecosystems have more than **60 %** of test cases missing in 25 % of their projects; **75 %** of test cases missing in 50 % of their projects, and more than **87 %** of test cases missing in 75 % of their projects.

RQ2: Proportion of non-shared test cases with similar UUT in projects they are missing from. For each ecosystem, we collected the set of all test cases marked missing in one or more projects. For each missing test case, we collected its set of UUTs: one set (TC) of fully qualified class names under test, and another (TM) of method names under test. We also collected a set (PC) of fully qualified class names present in all projects from which the test case is missing, and another set (PM) of method names. We consider the test case to have matching classes under test if $|TC \cap PC| \geq 1$, and matching methods under test if $|TM \cap PM| \geq 1$.

Figure 2b indicates that 75 % of the ecosystems have non-shared test cases with matching UUTs in projects where they are missing; with more matches by class name than method names (as expected). For 25 % of the ecosystems, **60 %** to **100 %** of the non-shared test cases had matching UUTs.

Our results suggest potential for test case propagation in fork ecosystems. However, even though we were able to match class names and methods in target projects for missing test cases, a detailed analysis and design is required based on how the target UUT may be modified in practice.

C. *Code-change Scenarios Affecting Propagation*

Changes to UUTs may be due to refactoring, bug fixes or addition of features. Table I presents our five main change scenarios. The first four relate to modifications of UUT, while the fifth relates to modifications of the test case. The added or modified test case may apply to code that has changed in both *Src* and *Trg* (S1), or only *Src* (S2), or only *Trg* (S3), or old code that has not been changed in either project (S4). The concrete example sub-scenarios indicate what recommendation actions are taken based on what changes occurred and where.

III. Stage 1: Assessing Test-Case Applicability

Our proposed technique comprises two main stages: assessing test-case applicability and test-case-recommendation and propagation (Sec. IV). We now describe how we envision the technique to work when assessing test case applicability.

Definition 1. (Test-Case Applicability Level) A test case t from a project *Src* is applicable to a target project *Trg* if its UUTs and their dependencies are similar in both *Src* and *Trg* based on some similarity criteria C.

A. *Applicability Technique*

Detecting Test Cases in Changed Code. We only detect test cases from changes introduced after fork creation. For each test case added or modified, we collect its UUTs and establish when they were last changed. Our technique's preliminary design and implementation relies on Ghafari et al.'s [22] technique to trace test cases to method-level UUTs. Similar to Li et al.'s [4] implementation of the Ghafari technique, we convert a given project's source code into srcML and compute dataflow paths that influence each variable in an assertion statement of a test case—a.k.a., backward slicing [23]. We use xUnit-based test case annotations, e.g., JUnit's @Test, to identify test cases in code.

Assessing Test-Case Existence. We check for the existence of a test case in other projects of the fork ecosystem by comparing its fully qualified class and method name to test-classes and methods in the target projects. If only the test-class is matched, we propose to compare statements in the *setup*, *execution*, and *oracle* parts of the test case to those of the target test methods in the matched class. For this, we shall rely on techniques such as AST differencing [24]. If we establish that the test case does not exist, we recommend it, else we ignore it. Our current implementation used in the pre-study only compares class or method names without comparing inner statements.

Assessing Applicability. If the test case does not exist in the target project, we assess whether it is applicable to the project. To that end, we check that its UUT (the code exercised by the test case), including its context (class or project library dependencies referenced by the UUT) exist in the target. Since some units may have been changed in the target, e.g., classes or functions renamed, we apply code-change detection techniques such as code-clone detection [8] to find such units whose behavior is still preserved. By assessing the context of the test case, we can estimate whether or not the behavior of its UUT is preserved in the target project. Our technique checks, with a certain precision, that the behavior of UUTs has not changed; for that, it will look into the code of UUTs and their context, and explore code similarities following control-flow or data-flow paths. We recommend the test case if it has matching UUTs in the target.

B. *Research Agenda*

Explore Design Decisions for Test-Case Recommendation. To assess applicability, our technique checks, with a certain precision, that the behavior of UUTs has not changed, by exploring control- and data-flow paths of the test case's UUTs and context. This exploration is at different depths/levels that need to be empirically calibrated to obtain a good trade-off between precision and performance, since some projects may be too large to analyze all dependencies. Hence, we shall investigate *(i)* how much of the UUT and context we analyze given the trade-off between precision and performance, and *(ii)* how we perform the analysis. We shall rely on code-change detection techniques, such as code-clone detection and refactoring tools, to analyze changes to the UUT and context in the target project. Our goal here is to maximize the precision of the test-case applicability technique while ensuring its practicality, considering the sizes of projects and number of forks. We shall investigate the extent to which existing techniques for code change-detection can be used (or improved upon) to assess the applicability of test cases in fork ecosystems. Therefore, we shall define criteria for assessing when a test case can be adapted to changes introduced in the target.

IV. Stage 2: Recommendation and Propagation

In the second stage we recommend applicable test cases to maintainers of target projects and, if accepted, integrate the test cases (and related code) into the target projects.

Definition 2. (Test-Case Propagation) A test case t from a source project *Src* is successfully propagated to target *Trg*, if t has equivalent output when executed on both *Src* and *Trg*, given the same input (if t was not adapted) or similar input (if t was adapted to *Trg*).

A. *Propagation Technique*

We perform the following steps to propagate the test case t:

Copy the Test Case and Attempt to Build the Target. If the test case requires no adaptation, we copy it to the target and attempt to build the project (Note: depending on where the target project is hosted, we may need CI infrastructure setup to automatically build). If the build is successful, we execute the test case to assess equality of output when executed on both the source and target; otherwise, we adapt the test case to the target's context.

Table I: Scenarios for propagating test case t from *Src* to *Trg*

code change scenario with example sub-scenarios	propagation action
S1: t applies to code changed in both *Src* and *Trg*	
S1a: Test case for a function existing in both *Src* and *Trg* but modified in both projects	assess applicability and recommend accordingly
S2: t applies to old code (in *Src*) that has been changed in *Trg*	
S2a: Test case for a function that exists in both *Src* and *Trg* but is modified in *Trg*	assess applicability and recommend accordingly
S2b: Test case for a function in *Src* that no longer exists in *Trg*	ignore
S3: t applies to code that has changed in *Src* but not in *Trg*	
S3a: Test case applies to a new function in *Src* that does not exist in *Trg*	ignore
S3b: Test case applies to an old function that exists in *Trg* but has been modified in *Src*	assess applicability and recommend accordingly
S4: t applies to old code that is common to both *Src* and *Trg*	
Test-case for function that is the same in both *Src* and *Trg*	recommend and propagate as is
S5: t added or modified	
S5a: Test case exists only in D	consider S1-S4 and assess applicability
S5b: Test case exists in both D and H but is modified in D only	assess test-case similarity and applicability (S1-S4)
S5c: Test case exists in both D and H and is modified in both projects	assess test-case similarity and applicability (S1-S4)

Adapt the Test Case. By analyzing the test case's data- and control-flow dependencies (based on variables referenced in assertion statements), we explore possible mappings between the test case's UUTs and the context, and those in the target. While exploring the search space of the host's variables to adapt the test case, we build the target project to ensure that it is compilable. When the project builds successfully, we execute the test case on the target to compare with output from the source project. We automatically adapt the test case as much as possible. However, if this is not possible, we suggest manual adaptation to the developer with an indication of parts that need to be adapted, e.g., indicating variables or statements in the test case that could not be mapped to the target's context.

B. Research Agenda

Conceive Technique for Automatic Test-Case Adaptation. Since code changes can occur to both test cases (see scenario S5 in Table I) and UUT (see scenario S1-S4 in Table I), we shall explore different techniques, such as AST differencing [24], techniques applied in automated test repair [25], or code transplantation techniques [10], to formalize an algorithm that automatically adapts the test case and related code to the target project. Through an iterative design and evaluation of the algorithm, we aim to understand *(i)* the kinds of changes to UUT under which a test case can be automatically adapted, and *(ii)* the kind of manual adaptations required.

V. Related Work

Test-Case Reuse and Evolution. Several studies address redundancy in test cases to promote reuse. Most focus on vertical reuse [26], that is, reuse of test cases over different integration levels of a system, or within software product lines and highly configurable systems [27], [28], or UI testing related apps [29]–[32]. Other studies [25], [33] have focused on supporting the co-evolution of test-cases with code artifacts, for instance, by automatically repairing test-cases based on code changes. However, we address test-case reuse across forked projects that may evolve in different ways, thereby alleviating the problem of redundant development. We also foster community development to leverage expertise for better quality assurance, that may otherwise not be easily realized.

Test-Case-to-Code Traceability. Several techniques have been proposed to trace test cases to code. The majority, such as *naming convention* and *last call before assert*, only trace classes under test [34], [35]. Ghafari et al. [22] proposed a test-code traceability technique that lowers the granularity to method-level traces; to which Li et al. [4] added multi-language support. We rely on this technique to identify UUTs of test-cases.

Code Porting and Transplantation. Several techniques [6], [7], [11], [12], [14]–[19] have been proposed for porting code changes across forks. However, these techniques focus on porting feature implementation and bug-fixes. In contrast, we aim to be proactive and improve software quality by reusing test-cases, thereby also saving development cost spent in fixing bugs already resolved. Furthermore, our problem scenario calls for investigation of several aspects outlined in our research agenda that are not addressed by previous studies.

Code Recommendation. Recommendation systems are widely used in software engineering [36], [37], for tasks such as suggesting bug-fixes, code snippets, and associated requirements. Github offers recommendations for projects that may have dependencies with known security vulnerabilities [38]. Other works exist, such as using unit tests to offer code learning examples [39], recommending test suites based on historical data [40], and recommending when to stop performance tests [26]. However, none of these works addresses our use case.

VI. Conclusion

Many software projects exist as fork ecosystems. In many cases, developers of individual forks are unaware of the evolution of other forks. In this paper, we presented a novel approach to recommending and propagating test cases across forked projects. We motivated our idea with a pre-study in which we found that 75 % of the ecosystems in our dataset had test cases that could potentially be propagated to projects where they were missing; and that for 25 % of the ecosystems, such non-shared test cases were between 60 % to 100 % of all non-shared test cases in the ecosystem. We complemented our discussion with a research agenda.

Acknowledgment. This work has been supported by the Swedish Research Council (Vetenskapsrådet).

References

[1] S. Zhou, B. Vasilescu, and C. Kästner, "How has forking changed in the last 20 years? a study of hard forks on github," in *ICSE*, 2020.

[2] J. Businge, M. Openja, S. Nadi, E. Bainomugisha, and T. Berger, "Clone-based variability management in the android ecosystem," in *ICSME*, 2018.

[3] M. Mukelabai, D. Nešić, S. Maro, T. Berger, and J.-P. Steghöfer, "Tackling combinatorial explosion: a study of industrial needs and practices for analyzing highly configurable systems," in *ASE*, 2018.

[4] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft, "Automatically documenting unit test cases," in *ICST*, 2016.

[5] S. Zhou, S. Stanciulescu, O. Leßenich, Y. Xiong, A. Wasowski, and C. Kästner, "Identifying features in forks," in *ICSE*, 2018.

[6] L. Ren, "Automated patch porting across forked projects," in *ESEC/FSE*, 2019.

[7] B. Ray and M. Kim, "A case study of cross-system porting in forked projects," in *FSE*, 2012.

[8] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[9] Y. Lin, X. Peng, Z. Xing, D. Zheng, and W. Zhao, "Clone-based and interactive recommendation for modifying pasted code," in *FSE*, 2015.

[10] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *ISSTA*, 2015.

[11] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, "Evolution of software in automated production systems: Challenges and research directions," *Journal of Systems and Software*, vol. 110, pp. 54–84, 2015.

[12] N. Meng, M. Kim, and K. S. McKinley, "Lase: locating and applying systematic edits by learning from examples," in *ICSE*, 2013.

[13] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *FSE*, 2014.

[14] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: a framework and experience," in *SPLC*, 2013.

[15] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *ASE*, 2013.

[16] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and characterizing semantic inconsistencies in ported code," in *ASE*, 2013.

[17] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann, "The uniqueness of changes: Characteristics and applications," in *MSR*, 2015.

[18] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github," in *MSR*, 2017.

[19] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, and K. Inoue, "Identifying source code reuse across repositories using lcs-based source code similarity," in *SCAM*, 2014.

[20] T. Berger, R.-H. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wąsowski, and S. She, "Variability mechanisms in software ecosystems," *Information and Software Technology*, vol. 56, no. 11, pp. 1520–1535, 2014.

[21] The Authors, "Online Appendix," https://bitbucket.org/easelab/testpropagataion-prestudy/, 2020.

[22] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *SCAM*, 2015.

[23] R. Jhala and R. Majumdar, "Path slicing," in *PLDI*, 2005.

[24] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ASE*, 2014.

[25] X. Li, M. d'Amorim, and A. Orso, "Intent-preserving test repair," in *ICST*, 2019.

[26] H. M. AlGhmadi, M. D. Syer, W. Shang, and A. E. Hassan, "An automated approach for recommending when to stop performance tests," in *ICSME*, 2016.

[27] S. Fischer, R. Ramler, L. Linsbauer, and A. Egyed, "Automating test reuse for highly configurable software," in *SPLC*, 2019.

[28] E. Engström and P. Runeson, "Software product line testing—a systematic mapping study," *Information and Software Technology*, vol. 53, no. 1, pp. 2–13, 2011.

[29] G. Hu, L. Zhu, and J. Yang, "Appflow: using machine learning to synthesize robust, reusable ui tests," in *FSE*, 2018.

[30] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *ASE*, 2019.

[31] A. Rau, J. Hotzkow, and A. Zeller, "Transferring tests across web applications," in *ICWE*, 2018.

[32] 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 *FSE*, 2020.

[33] V. G. Yusifoğlu, Y. Amannejad, and A. B. Can, "Software test-code engineering: A systematic mapping," *Information and Software Technology*, vol. 58, pp. 123–147, 2015.

[34] B. Van Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *CSMR*, 2009.

[35] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Recovering test-to-code traceability using slicing and textual analysis," *Journal of Systems and Software*, vol. 88, pp. 147–168, 2014.

[36] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE software*, vol. 27, no. 4, pp. 80–86, 2009.

[37] U. Pakdeetrakulwong, P. Wongthongtham, and W. V. Siricharoen, "Recommendation systems for software engineering: A survey from software development life cycle phase perspective," in *ICITST*, 2014.

[38] Github, "Managing Security Vulnerabilities," https://help.github.com/en/github/managing-security-vulnerabilities, 2020.

[39] S. M. Nasehi and F. Maurer, "Unit tests as api usage examples," in *ICSME*, 2010.

[40] D. S. Prasad, S. Chacko, S. Ramaraju, G. K. Durbhaka *et al.*, "Automatically recommending test suite from historical data based on randomized evolutionary techniques," Oct. 18 2016, uS Patent 9,471,470.