To Share, or Not to Share: Exploring Test-Case Reusability in Fork Ecosystems

Mukelabai Mukelabai,*§ Christoph Derks,† Jacob Krüger,‡ and Thorsten Berger†§
*The University of Zambia, Zambia
†Ruhr University Bochum, Germany
‡Eindhoven University of Technology, The Netherlands
§Chalmers | University of Gothenburg, Sweden

Abstract—**Code is often reused to facilitate collaborative development, to create software variants, to experiment with new ideas, or to develop new features in isolation. Social-coding platforms, such as GitHub, enable enhanced code reuse with forking, pull requests, and cross-project traceability. With these concepts, forking has become a common strategy to reuse code by creating clones (i.e., forks) of projects. Thereby, forking establishes fork ecosystems of co-existing projects that are similar, but developed in parallel, often with rather sporadic code propagation and synchronization. Consequently, forked projects vary in quality and often involve redundant development efforts. Unfortunately, as we will show, many projects do not benefit from test cases created in other forks, even though those test cases could actually be reused to enhance the quality of other projects. We believe that reusing test cases—in addition to the implementation code—can improve software quality, software maintainability, and coding efficiency in fork ecosystems. While researchers have worked on test-case-reuse techniques, their potential to improve the quality of real fork ecosystems is unknown. To shed light on test-case reusability, we study to what extent test cases can be reused across forked projects. We mined a dataset of test cases from 305 fork ecosystems on GitHub—totaling 1,089 projects—and assessed the potential for reusing these test cases among the forked projects. By performing a manual inspection of the test cases' applicability, by transplanting the test cases, and by analyzing the causes of non-applicability, we contribute an understanding of the benefits (e.g., uncovering bugs) and of the challenges (e.g., automated code transplantation, deciding about applicability) of reusing test cases in fork ecosystems.**

Index Terms—**test cases, reuse, test propagation, code transplantation, forking, ecosystems**

I. Introduction

Code is often reused in software development. For instance, developers copy code excerpts from question-answering platforms and exemplars [5], [44]; they clone repositories from version-control systems to enable collaboration or isolated development (e.g., feature forks) [35], [37], [70]; or they use forking mechanisms to create independent software variants (a.k.a. clone & own) [14], [63], [79]. Especially modern social-coding platforms, such as GitHub, actively enhance and promote the ability to reuse complete systems via forking and the pull-based development paradigm [8], [9], [23]. Forks typically involve changes that do not alter the general purpose of the system. Among others, they implement new or tailor existing features, or port or improve the system [2], [11], [54], [70].

Together, forks establish *fork ecosystems*, which are sets of related projects in a fork relationship. In such an ecosystem, each fork represents a different, yet similar variant of the mainline repository. While many forks are short-lived—created to apply changes, to submit a pull request, and then to abandon the fork after the pull request was accepted [28], [37]—others are long-lived, being evolved and maintained independently of the mainline project. While such code reuse in fork-based development promises many benefits, changes are typically only sporadically propagated or synchronized between forks [25], [35], [70], [71]. Consequently, the different variants (forked projects) sometimes co-evolve and then diverge in functionality as well as in quality [73], since bug fixes and test cases are not shared between them, among other reasons. Of course, sharing quality improvements is only relevant for actively maintained forks rather than for those that are abandoned after a pull request to the mainline project is accepted. To improve the quality of the projects within a fork ecosystem, we advocate the propagation of quality improvements, such as bug fixes and newly created or evolved test cases, among forks [34], [52], [64]. While change-propagation and program repair techniques have been proposed [19], there is no specific support to reuse test cases within fork ecosystems, not even studies of the *potential of test-case reuse*—our focus in this paper.

Reusing test cases has been recognized as a viable avenue to improve the quality of software. Various studies and techniques have been presented that focus on specific kinds of applications and application parts, such as on UI testing in Android apps and in web applications [26], [42], [57], [84], or on highly configurable systems [16], [18], [34], [64]. However, none of these works on test-case reuse considers *ecosystems of forked projects*, where the positive effect may be highest. A notable exception is the test-case reuse and adaptation technique by Zhang and Kim [83], which analyzes variations among clones and adapts existing test cases based on different criteria. Our long-term goal is to establish techniques, like the one we have proposed before [52], for automatically reusing test cases among the forked projects in a fork ecosystem. Still, it is an open question whether reusing (a.k.a. propagating or transplanting [6]) test cases with a fork ecosystem promises substantial benefits, or whether it is just a rare scenario. A particular problem is to understand and decide whether a test case is actually applicable at another, forked (a.k.a. cloned or copied) code location, which may have been modified by the fork developers.

Consider, for instance, a forked project with test cases for checking a security requirement (e.g., preventing SQL injections) that are missing in other forks of the ecosystem. If the tested functionality also exists in other forks, it may be highly important for the developers to propagate the test case to assess whether their fork is threatened (i.e., vulnerable to SQL injections). Beyond cases of simple equivalence, it is challenging to assess whether a test case is applicable [64]. Among others, the tested functionality may have been modified, or the test may be concerned with a feature interaction that does not exist in the other fork. To leverage existing and design new techniques for test-case reuse in fork ecosystems, we *need to determine whether and to what extent test cases in a fork ecosystem may be relevant for other forks, what adaptations may be required for propagating them, and whether we can identify test-case applicability criteria for typical changes in fork ecosystems*. This requires to systematically explore the potential of reusing test cases in fork ecosystems and to assess their applicability.

In this paper, we present a study in which we used code-clone detection to identify cloned methods within 305 ecosystems mined from GitHub, comprising 1,089 projects with substantial test case evolution (i.e., on average, 62 new test cases added by each fork). We identified methods with test cases (i.e., units under test (UUTs) [77]) for which cloned methods without the same test cases exist within the ecosystems. Then, we investigated the reuse potential of 4,347 test cases and manually inspected the applicability of a sample of 230 candidates for test-case reuse, of which we manually propagated 23 (10 %) to deepen our understanding of the necessary adaptations or limitations to test-case applicability.

Our research questions are:

RQ$_1$ *To what extent can test cases be propagated within fork ecosystems?* Using our quantitative analysis, we explored to what extent cloned methods in an ecosystem miss test cases that exist for their clones. So, we provide an understanding to what extent test cases are missing in forks and could be reused to improve their quality.

RQ$_2$ *To what extent are non-reused test cases applicable to other projects in fork ecosystems?* By manually inspecting 230 and actually reusing 23 of the non-reused test cases, we explored under what circumstances test cases are (not) applicable at the cloned locations. This analysis is far from trivial, especially for clones adding, altering, or removing entire statements. We focused on determining under what circumstances test cases are applicable even when the cloned code has been severely modified and applicability cannot be determined automatically.

Our results motivate the need for test-case propagation techniques. While we found evidence for active test-case sharing among projects in the ecosystems, with 6 % (i.e., 1,300 commits) of all merged pull-request commits adding or modifying test cases, we also found that 48 % (i.e., 18,058 test cases) of all newly added test cases were missing in at least one project of their respective ecosystem. Of these missing test cases, 24 % (i.e., 4,347 test cases) had matching UUTs to which they could potentially be applied, thus necessitating an investigation into the extent to which the test cases could be reused and what could limit their applicability. We contribute our replication package as an online appendix [1].

II. Background and Related Work

Fork Ecosystems. A fork ecosystem consists of one mainline project and the forks that have been derived from it. The notion of fork ecosystems stems from the forking mechanism of the social-coding platform GitHub, but essentially describes that a number of systems are cloned from one another (a.k.a. clone & own). Typically, forking is used in the pull-based development workflow to implement new features, fix bugs, or integrate other improvements [23], [28], [35], [37]. However, fork ecosystems often also involve forks that represent customized variants of the mainline (e.g., for different customers or varying hardware [7]–[9], [70]).

Researchers have studied how developers use forking mechanisms and have proposed techniques to facilitate code reuse in fork ecosystems. For instance, Kawamitsu et al. [31] and Gharehyazie et al. [20] present empirical studies to understand the extent of cross-project source-code reuse on GitHub, indicating that developers clone particularly within their project and fork ecosystem. Ray et al. [58] use the BSD ecosystem to detect and characterize semantic inconsistencies in code that is ported between individual forks. To improve code reuse, Ren [59] proposes a technique for automatically propagating bug fixes from one fork to another, which is inspired by the idea of automatic code transplantation [6], [12], [67], [68]. Lillack et al. [41] propose a technique to integrate forked projects by specifying the high-level intentions for individual parts that have diverged. Such works are known under the term clone management [7], [46], [62], [72]. Other studies focus on properties and uses of forks, such as developers' practices and challenges when forking, or they focus on the evolution and purposes of forks in practice [8], [9], [23]–[25], [28], [33], [36], [70], [85]. In particular, Ma et al. [45] study how developers fix cross-project correlated bugs (i.e., bugs of a system caused by another system). They find it a challenging activity that can be eased by the bug reporter sending a feasible test case. Moreover, Mondal et al. [49], [50] find that even within the same system 18.42 % of the (copy & paste) clones they investigated involved propagated bugs. Such works indicate the potential benefits test-case reuse may yield (e.g., increasing test coverage, providing test cases for bugs in another fork). We complement this research on fork ecosystems by providing a better understanding of the potential for reusing test cases across forks.

Test-Case Generation and Program Repair. Testing helps improve the quality of software by unveiling behavior that conflicts the intended one (specified by test cases). So, test cases can also serve as a design mechanism for new software [80]. Various techniques have been proposed to improve the quality of software by generating test cases [13], [55], [66], [81], typically building on the competent-programmer hypothesis [21]. This hypothesis states that catching smaller mutations will also unveil bigger problems, building on the intuition that good programmers who are less likely to introduce accidental mutations are also less likely to introduce bigger ones. Generated test cases aim to catch unwanted code mutations developers may implement. Consequently, test-case generators exploit ideas from regression testing [40], mutation testing [27], and search-based test creation. Closely related is automatic program repair, which aims to derive patterns of bug fixes to repair similar bugs in other projects (i.e., bugs identified in similar code locations) [10], [32], [38], [43], [51]. Typically, automated program repair builds on bug-fixing changes to derive fixing patterns. Unfortunately, comparative benchmarks and empirical studies in practice indicate that test generators and automatic program repair are not perfect for generating meaningful test cases or propagating bug fixes [4], [15], [65]. To improve this situation, we have suggested in recent works to involve developers more directly into the processes of deriving test cases [52], [53], [64]. Unfortunately, existing techniques are limited to specific testing techniques (e.g., performance testing), and only recommend modifications to developers [3]. For designing better automation and recommenders, further empirical studies on bugs, their causes, and feasible test cases in real-world systems are needed.

Reusing test cases is a means to improve software quality. Studies on the potential of test-case reuse exist for (i) highly configurable systems [16], [18] (applicability of test cases for other system configurations) and (ii) UI testing of Android apps or web applications [26], [42], [57], [84] (using similarities between UI elements and test cases). Most relevant for us is a technique by Zhang and Kim [83] that reuses test cases within a project to test code clones. To this end, the authors measure variations in referenced variables, data types, and method calls between methods to adapt the test case of one clone to be applicable for the other. While such works indicate criteria for assessing the applicability of test cases to similar systems, none studies the potential of test-case reuse within fork ecosystems, provides detailed empirical data on test-case applicability, identifies what kinds of changes break test cases, or suggests what test cases could be accepted and (automatically) modified to still reuse them. Our study provides more in-depth insights that are needed to understand when and how test-case reuse can be implemented in fork ecosystems—guiding researchers in advancing and improving the reliability of automated techniques and unleashing the potential for improving the quality of huge fork ecosystems.

Code-Test Co-Evolution. Co-evolution in fork ecosystems is a well-known problem, and, identically, source code and the test cases covering it may also co-evolve [34], [64], [73], [82]. To resolve co-evolution problems, several techniques and refactorings have been proposed that aim to identify whether a source-code change also requires updates to test cases—and to ideally update the test cases automatically [39], [56], [76]. Most interestingly, Mirzaaghaei et al. [48] propose a technique for repairing and generating test cases during the evolution of a software system. To adapt test cases for the evolved code, the authors propose four algorithms for identifying and adapting differences: signature changes (e.g., input and output), test-class hierarchies, interface implementations, and new overloaded methods. While software evolution faces similar problems as test-case reuse, none of these works focuses on the propagation of test cases in forked ecosystems. So, we complement this research direction by investigating test-case reuse and applicability for different forks.

III. Methodology

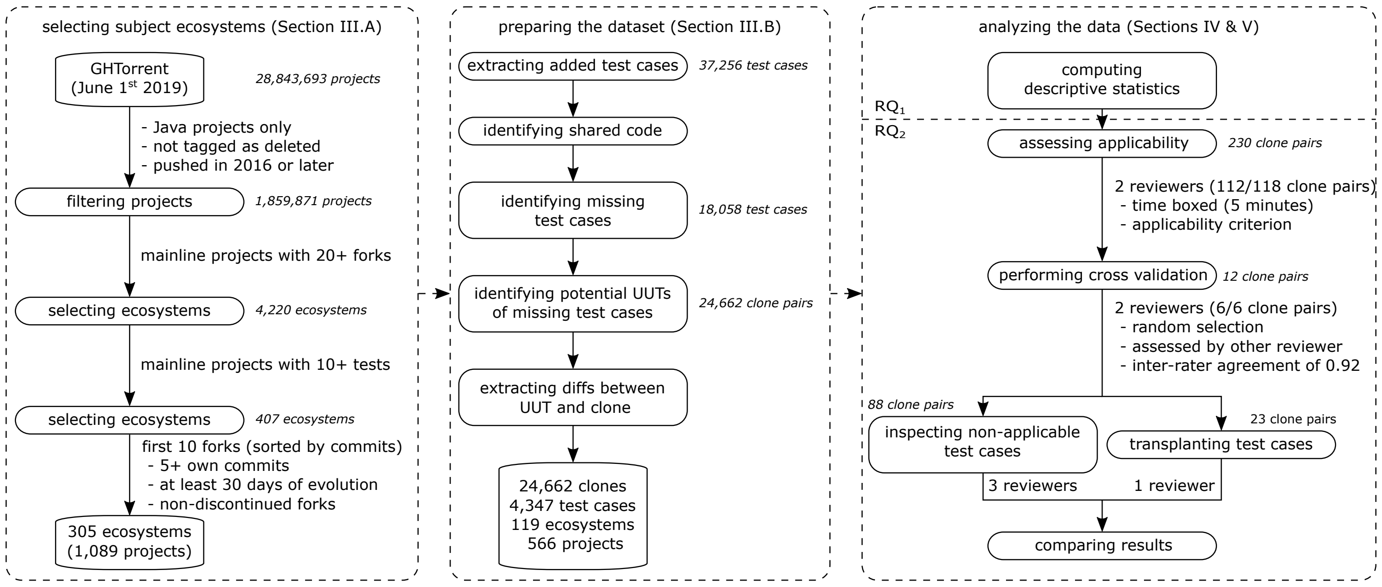
Our methodology involved three phases (cf. Fig. 1): First, we selected subject ecosystems from GHTorrent [22] by considering mainline projects (not forks) with substantial evolution (at least 20 forks and 10 test cases). Second, we extracted test cases from this dataset and identified potential code locations in other projects at which these could be reused. For this purpose, we constructed a list of test cases that were added in one project, but were missing in other projects of a fork ecosystem. We identified code clones for the code that is covered by the test case (UUTs). Finally, we analyzed the resulting dataset of 24,662 clone pairs. We derived descriptive statistics to assess the overall potential for test-case reuse (**RQ$_1$**). Thereafter, we manually inspected a sample of the test cases to understand why they are non-applicable or how they could still benefit the code missing the test case, and propagated a sample of them to validate our results (**RQ$_2$**).

A. Selecting Subject Ecosystems

On the left of Fig. 1, we show our steps for creating our dataset of projects to analyze.

Filtering Projects. Our subject ecosystems are open-source projects hosted on GitHub. We used the June 1st 2019 MySQL dump of the GHTorrent database [22] to mine our list of projects, thus circumventing the limitations of the GitHub API, which allows only a maximum of 1,000 records in a query session. The MySQL dump comprises different datasets, such as project meta-data, commits, and developers. We iterated through 28,843,693 records of the project meta-data dataset to select *Java* projects that were *last pushed to in 2016 or later* and were *not flagged as deleted*. Our focus on Java projects allows us to use existing tools that leverage JUnit annotations (e.g., `@Test`) to identify test cases (though this could be extended to any other language with explicit support for test case identification). Also, Java is one of the most popular languages. We wanted to capture recent development practices, thus limiting our analysis to projects still updated within the last six years. This resulted in 1,859,871 projects.

Selecting Ecosystems. Next, we selected ecosystems by considering all projects that are not forks themselves (i.e., mainline projects), but have forks. A mainline project together with its forks forms a fork-ecosystem. Since we are interested in projects that have wider community engagement (more diverging forks), we selected mainline projects with *at least 20 forks*. This resulted in 4,220 ecosystems. To be able to investigate test-case sharing, we filtered the mainline projects to only include those with *at least 10 test cases*. This number may seem small, but we use it as our minimum because we assume that any test case that can be reused improves the quality of a project. This left us with 407 mainline projects and, respectively,



selecting subject ecosystems (Section III.A)
GHTorrent (June 1st 2019)
28,843,693 projects
- Java projects only
- not tagged as deleted
- pushed in 2016 or later
filtering projects
1,859,871 projects
mainline projects with 20+ forks
selecting ecosystems
4,220 ecosystems
mainline projects with 10+ tests
selecting ecosystems
407 ecosystems
first 10 forks (sorted by commits)
- 5+ own commits
- at least 30 days of evolution
- non-discontinued forks
305 ecosystems (1,089 projects)
preparing the dataset (Section III.B)
extracting added test cases
37,256 test cases
identifying shared code
identifying missing test cases
18,058 test cases
identifying potential UUTs of missing test cases
24,662 clone pairs
extracting diffs between UUT and clone
24,662 clones
4,347 test cases
119 ecosystems
566 projects
analyzing the data (Sections IV & V)
computing descriptive statistics
RQ1
RQ2
assessing applicability
230 clone pairs
2 reviewers (112/118 clone pairs)
- time boxed (5 minutes)
- applicability criterion
performing cross validation
12 clone pairs
2 reviewers (6/6 clone pairs)
- random selection
- assessed by other reviewer
- inter-rater agreement of 0.92
88 clone pairs
inspecting non-applicable test cases
23 clone pairs
transplanting test cases
3 reviewers
1 reviewer
comparing results

Figure 1: Overview of our methodology

ecosystems. Note that we scope our analysis to mainline projects and their direct forks; we do not consider forks of forks.

Selecting Forks. The mainline projects we identified in our previous step had 274 forks on average, with a maximum of 3,719. However, we are interested in long-living forks that co-exist with the mainline projects and add test cases as well. Therefore, taking inspiration from previous studies [8], [52] that analyze the evolution and management of forked projects, we include only forks that (i) have *at least five fork-specific commits*, (ii) comprise *at least 30 days of evolution*, and (iii) were *not discontinued after a merge commit*. We applied these criteria and, due to space and computation limitations, cloned only the *top ten forks with the most fork-specific commits*. Note that we excluded mainline projects comprising no forks matching our criteria.

Final Dataset. We elicited a final dataset of 305 ecosystems, totaling 1,089 projects—305 mainline and 784 direct forks, with a minimum of 2, average of 4, and maximum of 10 projects per ecosystem. Note that these are only the projects that met our criteria, whereas some of the ecosystems have hundreds of forks. In Table I, we summarize the ecosystems. For most (196), we analyzed 3 or more projects per ecosystem, whereas 109 ecosystems involved 2 projects each. The ecosystems are from various domains, such as software libraries, frameworks, desktop applications, web applications, and mobile apps. The list of all projects is in our online appendix [1].

B. Preparing the Dataset

In the middle part of Fig. 1, we show how we prepared the dataset for our analysis. This comprised extracting test cases and their UUTs, identifying which test cases were missing in projects of their respective ecosystems, identifying potential UUTs to which the test cases could be applied, and extracting code differences between the clone pairs of the UUTs, which we used to analyze test-case applicability.

Extracting Test Cases. We extracted test cases in two steps. First, from each project, we extracted all test cases and corresponding UUTs. We used the test-case extraction tool we have developed in previous work [52] to identify test cases and related UUTs. The C#-based tool relies on N-unit annotations, such as `@Test`, and srcML [47] to extract a project's test cases and corresponding UUTs. We considered all project-specific methods referenced within a test case as its UUTs, while excluding methods from external libraries and those that are part of the test suite. For each test case, we recorded details, such as the project name, file path, as well as start and end line numbers of the test case and its UUTs.

Many test cases may be shared between forks by default, since forking also clones test cases that already exist in the mainline project. In contrast, we are interested in test cases that are not shared by default. Therefore, we scoped our analysis to test cases that were added by each project during its *fork-evolution period*. In Fig. 2, we display the evolution periods we analyzed: For mainline projects, we considered test cases added from the date of the first fork (e.g., F1). For forks, we considered those added from the date the fork was created. In the second step, we used pydriller [69] to extract all files and methods that were added during each project's fork evolution period. Then, we mapped the extracted methods and filenames to the test-case methods and files we identified in the first step. This resulted in 37,256 test cases that were added during the fork evolution periods.

Identifying Shared Code. To identify shared code—for test cases and UUTs—we used the code-clone detection tool Simian, a commercial code-similarity analyzer that is freely available for research. Simian can detect Type 1, 2, 3, and (to some extent) 4 clones [60], [61], [74], with an accuracy comparable to Deckard [29] and CCFinderX [30], which are two commonly used code-clone detection tools in research. We

Table I: Overview of the selected ecosystems. Size refers to the number of projects we analyzed. Forks refer to the overall size, including forks we did not analyze.

size of ecosystem (# of projects)	2	3	4	5	6	7	8	10	$\sum$
# of ecosystems	109	65	54	36	18	14	8	1	305
average # of forks	77	67	66	43	53	24	75	84	60
$\sum$ forks	16,927	13,193	14,476	7,909	5,787	2,391	4,836	844	66,363

selected Simian over these C-based tools, because it was easier to integrate into the Java and C# tools we used and extended for extracting test cases and analyzing variations between UUTs. We used Simian's default settings and set the minimum number of lines of code for a clone to three. To identify cross-project code clones, we put all projects of an ecosystem into one parent folder and supplied that folder as a parameter to Simian. Simian then outputted the clones identified in the ecosystem by mapping what line-ranges between what files are similar—with similar code blocks grouped and assigned a unique identifier.

Identifying Missing Test Cases. We considered a test case to be missing within an ecosystem if it is absent in at least one project of that ecosystem. Using our list of the 37,256 *added* test cases, we used Simian to identity which test cases had clones in other projects of the respective ecosystem to which the test case belonged. More specifically, for each test case in the ecosystem, we checked if its code line-ranges occur in any of the code-clone groups Simian found for the ecosystem. We identified 18,058 (48%) missing test cases in at least one project of the respective ecosystems (i.e., 272 ecosystems).

Identifying Potential UUTs for Missing Test Cases. Next, we identified missing test cases that could be reused in the projects they were missing from. Therefore, we extracted clones of their UUTs: For each missing test case's UUT whose code location (line ranges) appeared in the clone detection results, we collected all matched code clone locations from other projects in the ecosystem (we excluded within-project clones). Since each code clone-location may have multiple methods with similar or different signatures to the missing test case's UUT, we applied AST differencing using GumTreeDiff [17] to map

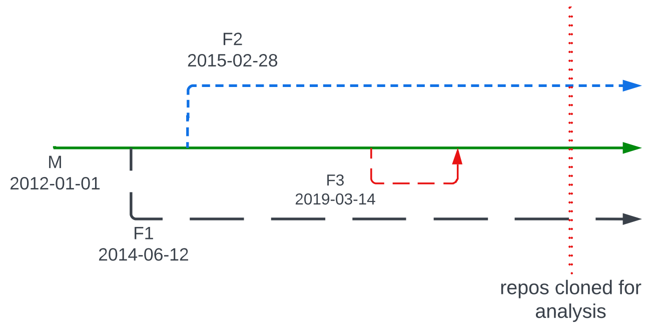


Figure 2: Illustration of the fork evolution periods we analyzed. For a mainline project (M), we consider only test cases added from the date of the first fork (F1). For forks (F1, F2), we consider test cases added from the date of forking. We exclude all forks (F3) discontinued after a merged pull request.

Table II: Descriptive statistics of the fork ecosystems

	type	min.	1st qu.	median	mean	3rd qu.	max.
main	commits[1]	3	115	250	414	472	3,691
	developers[1]	1	12	21	29	37	220
	stargazers	2	87	263	812	778	15,616
forks	commits	2	34	92	234	234	3,621
	developers	1	5	10	17	22	359
	stargazers	0	0	0	2	0	378

[1] Our analysis focuses only on (i) commits affecting Java files and (ii) commits made from the forking date for forks, or from the date of the first fork for mainline projects.

the missing test case's UUT to the correct method in the project where the test case was missing from. This phase produced our final set of 4,347 (24 %) missing test cases from 119 ecosystems for which we identified 24,662 UUT clone pairs. For each clone pair, we indicated the percentage of mapped AST tokens in both directions: source to target and target to source—which helped us understand the extent to which the clone pairs varied.

IV. Potential of Test-Case Reuse (RQ$_1$)

With our first research question, we aimed to understand and quantify the *potential* of reusing test cases in fork ecosystems.

A. Methodology

We derived descriptive statistics and visualizations to assess the potential of reusing the 37,256 added test cases we identified within their respective ecosystem. So, we provide a quantitative overview of how many test cases are missing in the ecosystems that can potentially be propagated to improve other forks.

B. Results

We first present an overview of the development activity in the ecosystems (e.g., number of commits and developers, added test cases). Then, we analyze the test-case sharing and conclude with a discussion of our results.

Development and Test-Case Evolution. Our dataset comprises 305 ecosystems totaling 1,089 software repositories (305 mainline projects, 784 forks). We analyzed commits affecting Java files and made from the date a fork was created (if the project is a fork) or from the date of the first fork (for mainline projects). In Table II, we can see that the mainline projects have on average 812 stargazers (an indication of how engaged a repository is), and, for the fork-evolution periods analyzed, each mainline has on average 414 commits made by 29 developers. This gives us confidence that our projects are non-trivial. Furthermore, we can see in Table IV that more test cases were modified than added; mainline projects added 72 and modified 204 test cases on average, while forks added 62 and modified 155 test cases on average. Interestingly, some projects added thousands of test cases, up to 2,219 test cases in a mainline project and up to 8,932 test cases in a fork. Our interest was in understanding the extent to which the added test cases are shared, and whether they can be reused in projects they were missing from.

Test-Case Sharing in Fork Ecosystems. We identified a total of 37,256 test cases that were added during the fork-evolution periods of our ecosystems. Of these, we found that at the time of our analysis, 19,198 (52 %) were present in all projects of their respective ecosystems while 18,058 (48 %) were missing in at least one project. In Fig. 3, we display the distribution of test-case presence per ecosystem size. We can see that the fewer projects in an ecosystem, the more test cases are shared between the projects: ecosystems with two projects share almost all the test cases added while those with ten projects have almost all test cases shared in less than 25 % of the projects of the ecosystem.

This result is interesting from two perspectives. First, the high number of shared test cases overall (i.e., 52 %), which were previously added by individual projects within an ecosystem, demonstrates the need for techniques that can recommend and propagate such test cases within an ecosystem. Indeed, for the fork evolution periods we analyzed, we found a total of 37,118 pull requests, of which 23,558 (63 %) were merged. Of the merged pull requests, most (i.e., 11,893) were from forks merging changes into the mainline projects, while 2,246 merged changes from mainline projects into forks. We counted the distinct number of pull-request merge commits that affected test cases and found that 300 commits added test cases, 1,000 commits modified test cases, 23 commits renamed test case files, and 3 commits deleted test cases. So, it is evident that developers actively share test cases across forked projects and could benefit from techniques that proactively recommend and propagate test cases in addition to feature improvements. Second, the high number of non-shared test cases (i.e., 48 %) also warrants an investigation into whether they can be reused or not, and to what extent.

Missing Test Cases with Potential Target UUTs. For each of the 18,058 test cases missing in at least one project of their ecosystem, we extracted clones of its UUTs in the projects the test case was missing from. After cleansing the dataset (cf. Sec. III-B), we found a final set of 24,662 clone pairs for 4,347 test cases from 119 ecosystems (i.e., 39 % of ecosystems), comprising 566 projects (i.e., 52 % of projects).

Analyzing the mapped AST tokens of the clone pairs revealed four kinds of variations between source and target UUT:

- Var_1 Source and target UUTs had 100 % mapped tokens in both directions, representing Type 1 or 2 code clones;
- Var_2 Source tokens were mapped 100 % to target tokens, but not vice versa, indicating that the target UUT grew more than the source UUT;
- Var_3 Target tokens were mapped 100 % to source tokens, but not vice versa, indicating that the source UUT grew more than the target UUT; and
- Var_4 Source and target UUTs both comprised tokens that could not be mapped to the other, indicating that the source and target UUTs co-evolved.

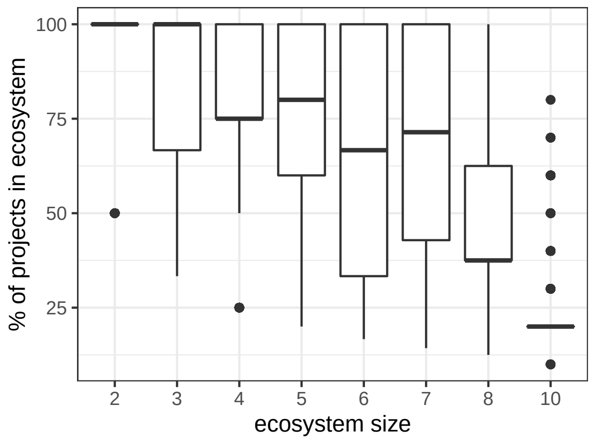


Figure 3: Distribution of test-case sharing per ecosystem size for all 37,256 test cases added during the fork-evolution periods of the 305 ecosystems (1,089 projects).

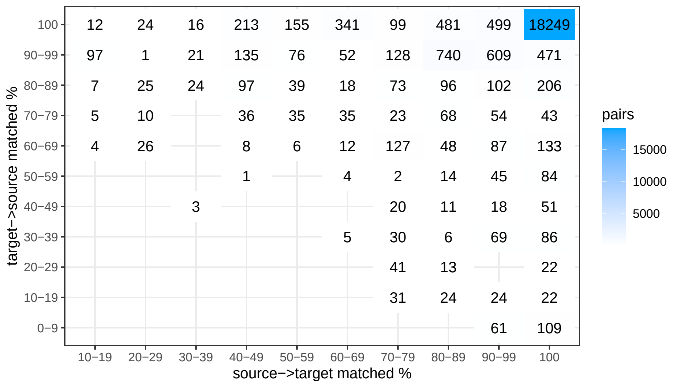


Figure 4: Variations between the source and target UUTs of all 24,662 clone pairs. The percentages of mapped AST tokens are grouped into 11 bins (from 0–9 % to 100 %).

Figure 4 summarizes these variations between all 24,662 clone pairs, expressed by the percentage of mapped AST tokens between the source and target UUT. We grouped the percentages into 11 bins from 0–9 % to 100 %. The x-axis shows bins for the percentage of tokens mapped from the source UUT to the target UUT, while the y-axis represents the mapping from target UUT to source UUT. We can see that 74% (18,249) of the clone pairs are Type 1 or 2 code clones, since tokens from the source and target UUTs are mapped to 100 % in both directions (Var_1, top-right corner of Fig. 4). The other clone pairs detected are of Type 3, and very few also of Type 4. In the top row (except the top-right corner), we can see that for 7 % (1,840) of the clone pairs 100 % of the tokens of the target UUT map to the source tokens, but not vice-versa (Var_2). Identically, the last column (except the top-right corner) represents that for approximately 5 % (1,227) of the clone pairs 100 % of the source tokens map to target tokens, but not vice-versa (Var$_3$). Finally, 14 % (3,346) of the clone pairs comprise tokens that could not be mapped in either direction and, thus, did evolve differently (i.e., all other entries in Fig. 4 represent Var$_4$).

C. Discussion

Missing Test Cases. Almost half of the test cases from all our considered fork ecosystems are missing in one of the respective projects. Particularly, larger ecosystems are more likely to involve projects with missing test cases. These results indicate a huge potential for supporting developers in identifying opportunities for test-case reuse. Considering that we only took into account a maximum of 10 projects per ecosystem, the actual potential in practice may be even higher. Providing tools that allow to identify missing test cases and indicate whether as well as how developers could reuse them would help share test cases across forked projects.

Automated Test Propagation. Since 74 % of UUT clone pairs for our missing test cases target Type 1 or Type 2 code clones, it should be simple to propagate these test cases (e.g., by potentially refactoring Type 2 clones or fixing class hierarchies [48]). Techniques to automatically transplant and refactor such test cases would greatly benefit developers. Still, to further improve test-case reuse and provide a better understanding of challenges and potential improvements, it is particularly interesting to study the remaining 26 % that are not trivially applicable—which is the scope of our manual analysis we report in Sec. V.

RQ$_1$: Potential of Test-Case Reuse

We explored to what extent 37,256 new test cases from 305 ecosystems with 1,089 projects are shared or missing in projects of their ecosystem. Our analysis indicates that:

- *many test cases are not shared among the projects of a fork ecosystem (e.g., 48 % of test cases are missing in at least one project), which is particularly apparent for larger ecosystems.*
- *many of 4,347 test cases involved in 24,662 UUT clone pairs could likely be reused automatically (74 % of target UUTs are Type 1 or 2 code clones of their source UUT).*

So, a very high potential to (automatically) reuse test cases in fork ecosystems exists.

V. Test-Case Applicability (RQ$_2$)

With our second research question, we aimed to understand the extent to which missing test cases are *applicable* to the projects they are missing from in an ecosystem.

A. Methodology

Assessing Applicability. We aimed to understand what makes a test case applicable to a cloned code location, or what prevents its reuse. While actual transplantation of test cases and related code is the ideal way of determining applicability, it may not always be possible, especially when we want to recommend test cases to developers for propagation. So, it

Table III: Overview of applicability (appl=applicable, n/appl=non-applicable) assessment of reviewers (rev) on whole sample of clone pairs (left), and their cross validation (right)

	rev					**rev 1**		
decision	1	2	total			appl	n/appl	total
appl	79	63	142	**rev 2**	appl	10	0	10
n/appl	33	55	88		n/appl	1	1	2
total	112	118	230		total	11	1	12
				91.67 % agreement, Cohen's κ = 0.62				

would be helpful to rely on other techniques like static data flow analysis before proceeding to a transplantation. Therefore, we assessed the applicability of missing test cases in two parts: (1) static data flow analysis through manual code inspection of variations between the source and target UUTs and (2) manual transplantation of test cases.

Part 1: Manual Code Inspection. We assume that the setup part (e.g., creating objects, preparing data) of a test case can always be reused, and thus we assess a test case's applicability based only on variations in the source and target UUTs. Using our list of the 24,662 clone pairs, we grouped each pair into one of 11 bins according to the percentage of mapped AST tokens from the source to the target UUTs (i.e., similarity of the UUTs), namely into: 0–9 %, 10–19 %, 20–29 %, 30–39 %, 40–49 %, 50–59 %, 60–69 %, 70–79 %, 80–89 %, 90–99 %, and 100 %. Then, for each ecosystem, we randomly sampled (up to) two pairs from the bin of pairs with 100 % mapped tokens and one pair from each of the other bins, giving us a total of 230 pairs. We sampled two pairs from the bin of 100 %, because it was the largest group—comprising 74 % of all pairs. Note that, not surprisingly, most ecosystems did not involve clone pairs for every bin, which is why we sampled 230 clone pairs (instead of 12 random samples for each of the 119 ecosystems).

Next, for each of the 230 pairs in the sample, we assessed whether or not the test case in question can be applied to the target UUT considering the code modifications in the UUTs. We did this in two steps: First, similar to previous research [48], [83], we marked all test cases as non applicable if the UUTs varied in the input (method parameters) or output (returned types or values), otherwise we marked them as applicable. Second, for all test cases marked as non applicable in the first step, we assessed if and to what extent the test cases were still applicable.

Step 1: Applicability Based on Non-Modification to Input and Output. In the first step, we marked a test case as applicable if its source and target UUTs do not vary in their input and output. This strict criterion allowed us to understand potential for automated transplantation of test cases, especially those involving Type 1 and 2 clones. For instance, consider the excerpt we display in Listing 1, in which we focus on lines added and modified in the target UUT. We see that the target UUT adds exception handling at lines 3, 4, and 18, and modifies the output string by URL-encoding it in lines 6, 7, and 9, making this a Type 3 clone [61]. Though the input (no parameters) and output types (String) are identical, the output

Listing 1: Simplified and unified example of a clone pair with indications for code **add**ed and **mod**ified in the target UUT

```
public String toQueryString()
{
 try                                                add
 {                                                  add
  // ...
  key = URLEncoder.encode(key, "UTF-8");            add
  result.append(key);
  // ...
  value = URLEncoder.encode(value, "UTF-8");        add
  result.append("=" + value);
  // ...
  if (i < (values.size() -1))                       mod
  {
   result.append("&" + key);
  }
  // ...
  return result.toString();
 }                                                  add
}
```

value is modified differently. So, we would assess the test case for this UUT clone pair as non applicable. After training on five ecosystems together, we assigned two reviewers to assess the applicability of test cases in the 230 clone pairs, each independently inspecting approximately half (112 and 118) of the pairs (see left side of Table III). We time-boxed the assessment of each pair to five minutes.

Performing Cross Validation. To ascertain the level of agreement between the two reviewers, we randomly sampled 5 % of the 230 pairs they reviewed; six pairs from each reviewer's half, totaling 12 pairs. Then, we let each reviewer assess the applicability of the six pairs from the other reviewer's half, without knowing the original reviewer's assessment. We investigated all disagreements and summarize the results of our cross-validation on the right side of Table III. We can see that the two reviewers disagreed only in one case of the 12 validated clone pairs. So, the inter-rater agreement is very high (91.67 %) with Cohen's κ indicating substantial agreement (0.62). This improves our confidence that our assessment builds on a common understanding between the reviewers and provides a reliable dataset.

Step 2: Applicability of Test Cases With UUTs Varying in Input and Output. Since our first assessment criterion was conservative (i.e., scoped for reliable automation of test-case transplantation), many of the test cases we marked as not applicable may still be applicable by modifying the target UUT or test case. For instance, consider Listing 1 again. The changed encoding introduced in the target UUT prevents that the provided String may cause bugs (e.g., preventing code injection). Consequently, the developers of the fork may want to reuse the test case we identified, even though it would fail before adapting it to the modified implementation. Even more interesting, the developers of the target UUT could likely benefit from knowing this case to improve the quality of their fork by propagating the String encoding. So, the test would become applicable after fixing the code.

To understand such cases and guide advanced test-case reuse (i.e., cases challenging simple transplantation), we performed a manual inspection of all 88 test cases (in the 88 clone pairs) we assessed as non-applicable (left side of Table III). Precisely, after a training on seven examples among three reviewers (including the previous two reviewers), these three re-iterated through test cases marked as non-applicable to assess under what circumstances these could still be reused; inspecting code, comments, and the clone pair to comprehend how a test case could be made applicable. Then, they derived a decision based on the modifications between source and target UUTs, providing a short description for their reasoning, which the other reviewers re-checked and discussed to achieve agreement. For example, the test case in Listing 1 would now be assessed as applicable, since the test case would only require that the String in its assertion is also encoded in UTF-8. In contrast, some target UUTs involved context changes, with context referring to the class the source UUT stems from and a context change referring to the target UUT being part of the same class (same context), a similarly named or sibling class (similar context), or a completely different class (major context change). For instance, a test case for an overwritten `equals` method could be applicable. However, the target `equals` method may be implemented in a completely different class, overwrites a different interface, and has several further modifications in itself—basically requiring developers to completely rewrite the test case, which is why we would assess such a case as still not applicable. We used a card-sorting-like method to cluster the re-assessed test cases into thematic topics based on the documented reasonings, discussed these topics to achieve agreement, and structured the topics based on how the respective test cases could be made applicable.

Part 2: Transplanting Test Cases. To deepen our understanding about the applicability of the missing test cases, and about possible challenges that may arise when propagating such test cases to target projects, we manually transplanted 23 (10 %) test cases randomly sampled from the 230 UUT clones pairs that we previously reviewed. Then, we compared the classification of these transplanted test cases to their classification when we judged their applicability solely based on manually reviewing modifications to their UUT.

B. Results

From the sample of 230 clone pairs with a missing test case, we assessed 142 to be applicable based on our data-flow criterion (cf. Table III). In contrast, the remaining 88 cases differed in input and output or involved modifications to the target UUT that impacted the data-flow compared to the source UUT. While one cannot simply see it, these test cases may still be reusable for the target UUT. We re-assessed all of these cases by performing a more in-depth manual inspection and transplanting a sample of 23 test cases.

Manual Analysis. Table V summarizes our reasoning behind each test case's final classification after our in-depth analysis. We now assessed 32 (36 %) test cases as applicable and 9 (10 %) as not applicable. However, we still could not derive a clear decision for 44 (50 %) cases, and we marked three as special cases in which an assessment was not useful because

Table IV: Descriptive statistics of test case evolution during the fork evolution periods of the fork ecosystems

	type	min.	1st qu.	median	mean	3rd qu.	max.
main	ADD	1	12	27	72	60	2,219
	MODIFY	1	29	86	204	220	3,691
	DELETE	1	2	5	14	14	111
	RENAME	1	2	4	24	13	639
forks	ADD	1	5	16	62	46	3,816
	MODIFY	1	11	37	155	106	8,932
	DELETE	1	3	7	20	13	624
	RENAME	1	2	6	18	19	408

the UUT mappings were test-specific UUTs, such as the test setup or helper methods.

Non-Applicable. We identified five main issues that led us to classify nine of the 88 test cases as still not applicable to the target UUT. Foremost, the target UUT was implemented in an unrelated context. This was followed by: cases were the dataflow differed considerably, accessing different methods or attributes that made a relation to the source UUT very difficult; cases where the perceived purpose of the methods varied to an extent that the test case did not fit the target UUT; cases were the differences in the input and output of the UUTs were non-trivial to convert or did not make sense in the remainder of the test case; and, lastly, cases that would require the whole test case to be rewritten to make it applicable to the target because the test case has a very specific execution environment.

Applicable. Even though all UUTs involved in the 88 pairs differed either in terms of input and output or dataflow, we assessed 32 cases to still be applicable. In fact, in 22 cases, the UUTs were implemented in the same or a closely related context while in 23 cases, the UUTs had an identical or at least similar purpose. In general, we viewed test cases as applicable if we identified a clear way to substitute the source UUT with the target UUT within the test case—allowing for modifications to the setup or parametrization of the UUT or the test case. We found eight cases in which input and output were convertible, since inputs that the target UUT needed existed in the source UUT as attributes, or the target UUT could be called through an existing gateway method providing default values for missing parameters. Similarly, differences in output types could be converted using an additional method call or simply did not matter as the test case did not use the output, but just the method's side effects. In 14 cases, differences in the data flow were irrelevant, since they were guarded by conditional statements that were not met by the test case's input to the UUT or based on different, yet name-wise similar, code elements that seemed to require little modification to make the test case applicable. Finally, we identified nine cases in which the test case was actually applicable because a perfectly matching UUT existed in the target project instead of the one in the clone pair that only had few similarities.

Unclear. In 44 cases, we could not arrive at a clear decision. In some cases, we needed detailed domain knowledge to understand whether a given long setup method for a test case could sensibly be reused or had to be rewritten from scratch. Especially in cases with context switches, it was hard to infer whether a test case would actually make sense in the target as it may test context-specific functionality. We found a set of target UUTs that could not easily be executed as they were implemented in abstract classes or private methods, where a developer would need to identify suitable ways of making them testable. Most interestingly, we discovered a set of pairs that tested functionality present in the source project but missing in the target project. These were test cases that check for specific edge cases, which may also be new functionality. Reusing such a test case likely requires modifying the target UUT, which may, in turn, fix existing bugs or enrich the target's functionality. Finally, we could not classify cases where we had difficulties comprehending the test case's purpose or the UUT implementation.

Manual Transplantation. Following our manual inspection and classification of the 230 UUT pairs, we randomly sampled 23 (10 %) of them and attempted to transplant them to determine their applicability. We sampled nine applicable, seven not applicable, and seven undecidable test cases—indicating that of the 16 decidable test cases, 56 % were actually useful for target UUTs. Of the nine applicable ones, three test cases were applicable without requiring any changes to the target project, four required minor modifications to the test case (i.e., calling a different method or declaring and instantiating a different type), and two required some modification to the project of the target UUT because the test cases targeted specific functionality missing from the target, which we copied from the source and integrated into the target classes.

Of the seven non-applicable test cases, four had target projects missing the tested functionality: either the UUTs were missing in the target and we could not transplant them or the tested functionality was deprecated in the target project. In the remaining three cases, the test cases were transplantable and reusable. However, they were irrelevant or redundant because they were already present in the target UUT's project, even though implemented differently to match the target's functionality. This highlights some of the limitations of using clone detection tools to assess test case applicability.

Note that most of these 16 test cases that we transplanted required us to update testing library dependencies, such as Mockito, since their versions varied between the source and target projects. Lastly, we were not able to decide a test case's applicability in seven of the 23 pairs, mainly because we could not setup or build the projects: either we could not load the project's dependencies (via Maven or Gradle), the project had several sub-projects, some of which could not be retrieved by the IDE, or adding missing UUTs in the target project caused ripple effects with compilation errors.

Comparing the Results.. We compare our classification of the 23 manually transplanted test cases to how we classified these during our manual code inspection. Note that we exclude all cases that we marked as undecidable in either classification (nine cases in total: two in the manual inspection, five in transplantation, and two in both). This leaves us with 14 cases with a clear decision. In seven of these cases, we have agreement

Table V: Overview of the reasonings we elicited through our re-assessment of 88 test cases assessed as non-applicable.

applicability	#	reasoning	#
non-applicable	9	unrelated context	6
		complete rewrite	4
		different dataflow	4
		different purpose	4
		different I/O	3
applicable	32	same/similar context	22
		same/similar purpose	23
		irrelevant differences	14
		same I/O	15
		convertible/irrelevant I/O	8
		mismatches	9
		different context	2
unclear	44	comprehension of UUTs	17
		reuse of setup	16
		additional setup	13
		comprehension of test case	11
		keeping test purpose	16
		modifications to UUT	12
		applicability hindrance	7
special cases	3	—	—

whereas we disagreed in the other seven. However, three of the cases we disagreed on are those we marked as not applicable during the transplantation because they were redundant for the target project, but we marked them as applicable during the manual inspection. So, our actual agreement is 10 out of 14 cases (i.e., 71 %). In three of the four cases we disagreed on, we marked test cases as applicable during the manual code inspection, but we marked them not applicable during the transplantation. For the last case, we marked the test case as not applicable during the manual inspection, while we could actually transplant it with modifications to the test case (i.e., applicable). These disagreements occurred due to well-known limitations of our manual code inspection, namely analyzing a single UUT pair for each test case and limiting the comparison to the files containing these specific UUTs, as well as lacking details on the impact of applying certain code changes. Therefore, while our limited set of actually transplanted test cases may not allow for statistical conclusions, our result of matching 71 % cases between manual code inspection and transplantation indicate that an analysis similar to ours may be useful as a heuristic to assess test-case applicability.

C. Discussion

Test-Case Applicability. Deciding whether a test case is applicable to a target UUT is challenging and requires detailed comprehension of the UUTs, test case, and domain, as we experienced first hand. It is not surprising that we could often not decide whether a test case is applicable because of missing domain knowledge. In fact, program comprehension is one of the most challenging activities for developers [75], [78], and domain knowledge can benefit it. To solve this problem, automated analyses and propagation techniques are of limited use at best. Instead, it would be necessary to involve experts of the domain and document knowledge, as proposed in similar research areas recently [52], [53], [64], also to decide whether some missing test cases and even functionalities could be useful to a project. While the feasibility of full automation is questionable, we believe that there is great potential for guiding developers through recommendation systems. After deciding whether a test case is applicable, the next challenge is to understand how to make it applicable to the target UUT. We found several types of smaller modifications that could be exploited for automated techniques, such as utilizing inheritance structures, identifying behavior-keeping modifications (e.g., refactorings), or identifying the test case to reveal bug fixes.

Automated Test Propagation. Our insights highlight that techniques for automatically propagating test cases in fork ecosystems can be highly beneficial, but require more advanced tooling than currently available. For instance, the algorithms proposed by Mirzaaghaei et al. [48] for evolving test cases in a single system can be adapted to support the identification and resolution of inheritance modifications. Also, it is rather simple to identify some cases of whether a test case can be executed on a method (i.e., target UUT is in an abstract class) or where the modifications are not within the test coverage. However, there are various research opportunities to tackle the area between existing or straight-forward automation and the need for extensive domain knowledge. Specifically, the most interesting test cases to propagate are those that could unveil bugs in the target UUT, which requires more advanced analyses.

Benefits of Reusable Test Cases. While testing improves the software quality, it is expensive to create test cases. So, test-case reuse is a viable avenue to improve the software quality at lower costs. Our investigation shows that most of the missing test cases from our subject ecosystems are reusable. However, to fully appreciate the benefits of these reusable test cases, we still need to address some open questions as future work.

First, an open question is *to what extent do reusable test cases capture bugs introduced in the projects they are missing from?* While we found a few cases during our manual transplantation where a reusable test case would fail in the target project (cf. Listing 1), we still need to quantify why the test case fails and whether this failure may actually reveal a bug. Particularly the latter case is interesting to improve the quality of forked projects. However, determining whether test cases actually fix bug is a general and non-trivial problem in software engineering, and understanding what constitutes a bug may in some cases require extensive domain knowledge of the target system.

Second, we found that some test cases were not useful to propagate because a similar test case existed in the target project. Consequently, it is an open question *to what extent do reusable test cases improve the test coverage?* Answering this question should consider the fact that we found several cases where a target UUT already had other test cases exercising it. Deciding whether a test case is useful is a complex task that may benefit from heuristics, but most likely will require developer involvement.

Finally, an open question is: *what would make a developer not reuse a reusable test case?* We know from experience that assessing test-case applicability and even propagating test cases requires cognitive load and effort that developers may not be willing to spend due to other priorities. In contrast, as we explained in Sec. IV-B, we have evidence that developers do propagate test cases between forked projects (as seen from pull requests). We also know that it is unlikely that a developer will reuse a test case they are not aware of, which is why we need techniques that can identify, recommend, and propagate such test cases in the future. However, we cannot speculate why a developer would choose not to reuse an applicable test case that they are aware of, which is a topic that requires a future study.

RQ$_2$: Test-Case Applicability

We determined when and how test cases are applicable despite changes above Type 2 code clones between source and target UUT. Our results indicate that:

- *neither a simple data-flow analysis nor a manual inspection of a single UUT can reliably identify whether a test case is (non-)applicable, since the source and target projects may differ heavily, while the purpose of the test case is still fulfilled.*
- *reusing test cases for clone pairs with more differences is an important research problem, helping to unveil unknown bugs and improving the test coverage of forks.*

So, assessing the applicability of test cases is challenging, but promises to improve the quality of fork ecosystems.

VI. Threats to Validity

Internal Validity. We relied on various tools and libraries to elicit our dataset, such as Simian for code-clone detection, GumTreeDiff [17] for AST diffing, and our own tool [52] for identifying test cases through srcML [47]. Moreover, we implemented further tooling to combine and extend such tools. To mitigate that bugs may have biased our dataset (e.g., not identifying all clone pairs or missing test cases), we cross-checked results, tested our setup, relied on existing tools, created a large dataset, and conducted quantitative as well as qualitative analyses—with our analysis showing that we have a reliable dataset for our study.

We manually assessed the applicability of a sample of 230 clone pairs, mimicking an independent developer inspecting test cases for reuse. So, the validity of our sample may be threatened by our understanding of the source code and test cases (e.g., disagreements or errors), the time-boxing we employed to keep the effort manageable, as well as the sample itself. To mitigate inconsistent assessments, we performed training among the reviewers, defined a strict applicability criterion upon related work [48], [83], and cross-validated a random sample of 12 clone pairs. Thereafter, we performed a detailed manual inspection of 88 test cases we assessed as non-applicable to obtain and provide an in-depth understanding of why these are non-applicable—or could still be applicable. Since all of these manual analyses rely on our own understanding and may require domain knowledge, we sometimes disagreed about reasonings and individual corner cases (e.g., regarding side effects of other methods in the target UUT when testing constructors). We discussed the disagreements, derived a solution, re-checked the test case (and similar ones) based on the solution, and performed a card-sorting-like method to abstract the many individual cases into comprehensible topics (each with multiple examples)—limiting the threat that misinterpretations bias the outcome. Additionally, we transplanted 23 test cases and compared the outcomes against our manual assessment.

External Validity. We only considered Java ecosystems, partially due to tooling limitations. So, our results may not apply to other languages. However, Java is among the most popular languages, and our insights about changes and kinds of similarities are not specific to Java. They involve language elements found in typical object-oriented languages. Similarly, we considered only ecosystems from GitHub, which may yield different results compared to other platforms like BitBucket or GitLab. Still, GitHub is the largest and most widely used platform, involving open-source and industrial projects, various domains, different development practices, as well as expert and novice developers. As such, GitHub involves a representative and large portion of fork ecosystems.

We focused on test-case reuse in fork ecosystems. This allowed us to easily identify cloned code with missing tests that would be very likely to benefit from propagating a missing test case. So, as intended, our results are primarily relevant for fork ecosystems and may not be transferable to test-case reuse in more independent projects. Still, the more high-level insights we obtained can also be helpful in such contexts (e.g., program repair and code transplantation in general).

VII. Conclusion

We presented a study on the potential of reusing test cases within fork ecosystems. Our results indicate that forked projects often do not benefit from test cases created in other forks (48 % of test cases are missing in at least one fork), even though they could be propagated (74 % of UUTs are Type 1 and 2 clones). We quantified the benefit of reusing test cases among forks, investigated what changes between projects do not hinder test-case propagation, and inspected changes that hinder the propagation. Importantly, we found that 62 % of the test case's applicability can be decided based on data-flow analysis. Our results show that there is huge potential to enhance the software quality of fork-ecosystems by propagating test cases, and that we need automated techniques to support this propagation. We hope to pave the way to enhanced and fork-ecosystem-specific test-case reuse techniques [26], [34], [42], [57], [64], [83], [84]. It is also a call to arms for research on reusing test cases for fork or clone pairs with more intricate differences.

Acknowledgment

Supported by a fellowship of the Royal Swedish Academy of Sciences and of the Wallenberg Foundation, and by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] "Appendix." [Online]. Available: https://bitbucket.org/easelab/testcasepropagationappendix

[2] J. Åkesson, S. Nilsson, J. Krüger, and T. Berger, "Migrating the Android Apo-Games into an Annotation-Based Software Product Line," in *International Systems and Software Product Line Conference (SPLC)*. ACM, 2019, pp. 103–107.

[3] H. M. AlGhmadi, M. D. Syer, W. Shang, and A. E. Hassan, "An Automated Approach for Recommending When to Stop Performance Tests," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 279–289.

[4] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application," in *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 263–272.

[5] S. Baltes and S. Diehl, "Usage and Attribution of Stack Overflow Code Snippets in GitHub Projects," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1259–1295, 2019.

[6] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated Software Transplantation," in *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2015, pp. 257–269.

[7] T. Berger, J.-P. Steghöfer, T. Ziadi, J. Robin, and J. Martinez, "The State of Adoption and the Challenges of Systematic Variability Management in Industry," *Empirical Software Engineering*, vol. 25, pp. 1755–1797, 2020.

[8] J. Businge, O. Moses, S. Nadi, E. Bainomugisha, and T. Berger, "Clone-Based Variability Management in the Android Ecosystem," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 625–634.

[9] J. Businge, O. Moses, S. Nadi, and T. Berger, "Reuse and Maintenance Practices among Divergent Forks in Three Software Ecosystems," *Empirical Software Engineering*, vol. 27, no. 2, pp. 54:1–47, 2022.

[10] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "ReAssert: Suggesting Repairs for Broken Unit Tests," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2009, pp. 433–444.

[11] J. Debbiche, O. Lignell, J. Krüger, and T. Berger, "Migrating Java-Based Apo-Games into a Composition-Based Software Product Line," in *International Systems and Software Product Line Conference (SPLC)*. ACM, 2019, pp. 98–102.

[12] C. Derks, D. Strüber, and T. Berger, "A benchmark generator framework for evolving variant-rich software," *Journal of Systems and Software*, vol. 203, p. 111736, 2023.

[13] X. Devroey, S. Panichella, and A. Gambi, "Java Unit Testing Tool Competition: Eighth Round," in *International Conference on Software Engineering Workshops (ICSEW)*. ACM, 2020, pp. 545–548.

[14] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *CSMR*, 2013.

[15] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 302–313.

[16] 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.

[17] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-Grained and Accurate Source Code Differencing," in *International Conference on Automated Software Engineering (ASE)*, 2014, pp. 313–324.

[18] S. Fischer, R. Ramler, L. Linsbauer, and A. Egyed, "Automating Test Reuse for Highly Configurable Software," in *International Systems and Software Product Line Conference (SPLC)*. ACM, 2019, pp. 1–11.

[19] L. Gazzola, D. Micucci, and L. Mariani, "Automatic Software Repair: A Survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[20] M. Gharehyazie, B. Ray, and V. Filkov, "Some from Here, Some from There: Cross-Project Code Reuse in Github," in *International Coference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 291–301.

[21] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How Close Are They to Real Faults?" in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2014, pp. 189–200.

[22] G. Gousios, "The GHTorent Dataset and Tool Suite," in *International Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 233–236.

[23] G. Gousios, M. Pinzger, and A. van Deursen, "An Exploratory Study of the Pull-Based Software Development Model," in *International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 345–355.

[24] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work Practices and Challenges in Pull-Based Development: The Contributor's Perspective," in *International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 285–296.

[25] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective," in *International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 358–368.

[26] G. Hu, L. Zhu, and J. Yang, "AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 269–282.

[27] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[28] J. Jiang, D. Lo, J. He, X. Xia, P. S. Kochhar, and L. Zhang, "Why and How Developers Fork What from Whom in GitHub," *Empirical Software Engineering*, vol. 22, no. 1, pp. 547–578, 2017.

[29] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and Accurate Tree-Based Detection of Code Clones," in *International Conference on Software Engineering (ICSE)*. IEEE, 2007, pp. 96–105.

[30] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[31] 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 *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2014, pp. 305–314.

[32] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "FixMiner: Mining Relevant Fix Patterns for Automated Program Repair," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1980–2024, 2020.

[33] S. Krieter, J. Krüger, T. Leich, and G. Saake, "VariantInc: Automatically Pruning and Integrating Versioned Software Variants," in *International Systems and Software Product Line Conference (SPLC)*. ACM, 2023.

[34] J. Krüger, M. Al-Hajjaji, S. Schulze, G. Saake, and T. Leich, "Towards Automated Test Refactoring for Software Product Lines," in *International Systems and Software Product Line Conference (SPLC)*. ACM, 2018, pp. 143–148.

[35] J. Krüger and T. Berger, "An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020, pp. 432–444.

[36] J. Krüger, A. Mikulinski, S. Schulze, T. Leich, and G. Saake, "DSDGen: Extracting Documentation to Comprehend Fork Merges," in *International Systems and Software Product Line Conference (SPLC)*. ACM, 2016.

[37] J. Krüger, M. Mukelabai, W. Gu, H. Shen, R. Hebig, and T. Berger, "Where is My Feature and What is it About? A Case Study on Recovering Feature Facets," *Journal of Systems and Software*, vol. 152, pp. 239–253, 2019.

[38] X.-B. D. Le, D. Lo, and C. Le Goues, "History Driven Program Repair," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2016, pp. 213–224.

[39] X. Li, M. d'Amorim, and A. Orso, "Intent-Preserving Test Repair," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2019, pp. 217–227.

[40] Y. Li and N. J. Wahl, "An Overview of Regression Testing," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 1, pp. 69–73, 1999.

[41] M. Lillack, Ş. Stănciulescu, W. Hedman, T. Berger, and A. Wąsowski, "Intention-Based Integration of Software Variants," in *International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 831–842.

[42] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test Transfer Across Mobile Apps Through Semantic Mapping," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 42–53.

[43] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting Template-Based Automated Program Repair," in *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2019, pp. 31–42.

[44] X. Liu and H. Zhong, "Mining StackOverflow for Program Repair," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 118–129.

[45] W. Ma, L. Chen, X. Zhang, Y. Zhou, and B. Xu, "How Do Developers Fix Cross-Project Correlated Bugs? A Case Study on the GitHub Scientific Python Ecosystem," in *International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 381–392.

[46] W. Mahmood, D. Strueber, T. Berger, R. Laemmel, and M. Mukelabai, "Seamless variability management with the virtual platform," in *43rd International Conference on Software Engineering (ICSE)*, 2021.

[47] J. I. Maletic, M. L. Collard, and A. Marcus, "Source Code Files as Structured Documents," in *International Workshop on Program Comprehension (IWPC)*. IEEE, 2002, pp. 289–292.

[48] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Supporting Test Suite Evolution through Test Case Adaptation," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2012, pp. 231–240.

[49] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider, "An Empirical Study on Bug Propagation Through Code Cloning," *Journal of Systems and Software*, vol. 158, pp. 110 407:1–18, 2019.

[50] M. Mondal, C. K. Roy, and K. A. Schneider, "Bug Propagation Through Code Cloning: An Empirical Study," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 227–237.

[51] M. Monperrus, "Automatic Software Repair: A Bibliography," *ACM Computing Surveys*, vol. 51, no. 1, pp. 17:1–24, 2018.

[52] M. Mukelabai, T. Berger, and P. Borba, "Semi-Automated Test-Case Propagation in Fork Ecosystems," in *International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2021, pp. 46–50.

[53] S. Nielebock, R. Heumüller, J. Krüger, and F. Ortmeier, "Cooperative API Misuse Detection Using Correction Rules," in *International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. ACM, 2020, pp. 73–76.

[54] L. Nyman and T. Mikkonen, "To Fork or Not to Fork: Fork Motivations in SourceForge Projects," *International Journal of Open Source Software and Processes*, vol. 3, no. 3, pp. 1–9, 2011.

[55] C. Pacheco and M. D. Ernst, "Randoop: Feedback-Directed Random Testing for Java," in *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 2007, pp. 815–816.

[56] H. Passier, L. Bijlsma, and C. Bockisch, "Maintaining Unit Tests During Refactoring," in *International Conference on Principles and Practices of Programming on the Java Platform (PPPJ)*. ACM, 2016, pp. 1–6.

[57] A. Rau, J. Hotzkow, and A. Zeller, "Transferring Tests Across Web Applications," in *International Conference on Web Engineering (ICWE)*. Springer, 2018, pp. 50–64.

[58] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and Characterizing Semantic Inconsistencies in Ported Code," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 367–377.

[59] L. Ren, "Automated Patch Porting Across Forked Projects," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 1199–1201.

[60] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," Queen's University at Kingston, Tech. Rep. 2007-541, 2007.

[61] 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.

[62] J. Rubin, K. Czarnecki, and M. Chechik, "Managing Cloned Variants: A Framework and Experience," in *International Software Product Line Conference (SPLC)*. ACM, 2013, pp. 101–110.

[63] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik, "Managing Forked Product Variants," in *International Software Product Line Conference (SPLC)*. ACM, 2012, pp. 156–160.

[64] S. Schulze, J. Krüger, and J. Wünsche, "Towards Developer Support for Merging Forked Test Cases," in *International Systems and Software Product Line Conference (SPLC)*. ACM, 2022, pp. 131–141.

[65] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. C. Gall, and A. Bacchelli, "On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later," in *International Coference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 121–125.

[66] S. Shamshiri, "Automated Unit Test Generation for Evolving Software," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 1038–1041.

[67] R. S. Shariffdeen, S. H. Tan, M. Gao, and A. Roychoudhury, "Automated Patch Transplantation," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 1, pp. 6:1–36, 2020.

[68] G. S. Sodhi and D. Rattan, "An Insight on Software Features Supporting Software Transplantation: A Systematic Review," *Archives of Computational Methods in Engineering*, pp. 1–38, 2021.

[69] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python Framework for Mining Software Repositories," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 908–911.

[70] Ș. Stănciulescu, S. Schulze, and A. Wąsowski, "Forked and Integrated Variants in an Open-Source Firmware Project," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 151–160.

[71] T. A. Standish, "An Essay on Software Reuse," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 494–497, 1984.

[72] M. Staples and D. Hill, "Experiences Adopting Software Product Line Development without a Product Line Architecture," in *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2004, pp. 176–183.

[73] D. Strüber, M. Mukelabai, J. Krüger, S. Fischer, L. Linsbauer, J. Martinez, and T. Berger, "Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems," in *International Systems and Software Product Line Conference (SPLC)*. ACM, 2019, pp. 177–188.

[74] J. Svajlenko and C. K. Roy, "Evaluating Clone Detection Tools With Bigclonebench," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 131–140.

[75] R. Tiarks, "What Maintenance Programmers Really Do: An Observational Study," in *Workshop on Software Reengineering (WSR)*. University of Siegen, 2011, pp. 36–37.

[76] A. van Deursen, L. M. F. Moonen, A. van den Bergh, and G. Kok, "Refactoring Test Code," CWI, Tech. Rep. SEN-R0119, 2001.

[77] B. Van Rompaey and S. Demeyer, "Establishing Traceability Links Between Unit Test Cases and Units Under Test," in *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2009, pp. 209–218.

[78] A. von Mayrhauser and A. M. Vans, "Program Comprehension During Software Maintenance and Evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[79] A. Wąsowski and T. Berger, *Software Product Lines*. Springer International Publishing, 2023, pp. 395–435. [Online]. Available: https://doi.org/10.1007/978-3-031-23669-3_11

[80] T. Winters, T. Manshreck, and H. Wright, *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly, 2020.

[81] Z. Xu and G. Rothermel, "Directed Test Suite Augmentation," in *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2009, pp. 406–413.

[82] 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.

[83] T. Zhang and M. Kim, "Automated Transplantation and Differential Testing for Clones," in *International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 665–676.

[84] 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 *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020, pp. 1190–1201.

[85] S. Zhou, B. Vasilescu, and C. Kästner, "How Has Forking Changed in the Last 20 Years? A Study of Hard Forks on GitHub," in *International Conference on Software Engineering (ICSE)*. ACM, 2020, pp. 445–456.