Towards Developer Support for Merging Forked Test Cases

Sandro Schulze
Technical University Braunschweig
Braunschweig, Germany
sandro.schulze@tu-bs.de

Jacob Krüger
Ruhr-University Bochum
Bochum, Germany
jacob.krueger@rub.de

Johannes Wünsche
Otto-von-Guericke University
Magdeburg, Germany

ABSTRACT

Developers rely on branching and forking mechanisms of modern versioning systems to evolve and maintain their software systems. As a result, systems often exist in the form of various short-living or even long-living (i.e., clone & own development) variants. Such variants may have to be merged with the main system or other variants, for instance, to propagate features or bug fixes. Within such merging processes, test cases are highly interesting, since they allow to improve the test coverage and hopefully the reliability of the system (e.g., by merging missing tests and bug fixes in test code). However, as all source code, test cases may evolve independently between two or more variants, which makes it non-trivial to decide what changes of the test cases are relevant for the merging. For instance, some test cases in one variant may be irrelevant in another variant (e.g., because the feature shall not be propagated) or may subsume existing test cases. In this paper, we propose a technique that allows for a fine-grained comparison of test cases to support developers in deciding whether and how to merge these. Precisely, inspired by code-clone detection, we use abstract syntax trees to decide on the relations between test cases of different variants. We evaluate the applicability of our technique qualitatively on five open-source systems written in Java (e.g., JUnit 5, Guava). Our insights into the merge potential of 50 pull requests with test cases from these systems indicate that our technique can support the comprehension of differences in variants' test cases, and also highlight future research opportunities.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Software product lines**; **Software testing and debugging**.

KEYWORDS

feature forks, variant-rich systems, merging, test cases

ACM Reference Format:
Sandro Schulze, Jacob Krüger, and Johannes Wünsche. 2022. Towards Developer Support for Merging Forked Test Cases. In *26th ACM International Systems and Software Product Line Conference - Volume A (SPLC '22), September 12–16, 2022, Graz, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3546932.3547002

1 INTRODUCTION

Implementing and maintaining a large software system requires a team of software engineers to collaborate, for instance, within an organization or an open-source community that shares a common interest. Version-control systems (e.g., Git) [24, 33] and social-coding platforms that build upon these systems (e.g., GitHub) [12] have become de facto standards for managing the development of such systems and for facilitating collaboration. A particularly interesting practice are the branching and forking mechanisms (to which we refer to as *forking* for simplicity) that allow developers to create a copy of the system and modify that copy independently. Typically, this widely established practice is called *feature forking* and enables developers to implement and test new *features* (i.e., user-visible functionalities of the system [4]) without interfering with other developers [13, 14, 18–20, 34, 37]. When the new features are considered relevant and stable enough, the developers can merge the forks back into the main system (e.g., after issuing a pull request on GitHub).

Merging in modern version-control systems is somewhat automated, meaning that any line-based text changes can be merged automatically as long as there is no *textual merge conflict* (i.e., two different edits to the same line of text). Still, ensuring that the changes are also *semantically* correct (i.e., fulfill defined requirements, do not comprise bugs) is a different and far more complex problem [5, 25, 26, 32]. Aiming to check for semantic correctness, developers primarily rely on testing [14] and have adopted various concepts for automatically re-running defined test cases before actually merging forks into a system, for instance, continuous integration [7] and regression testing [23]. However, **what happens if the changes are not covered by existing test cases, require adaptations or bug fixes to existing test cases, or propose their own test cases?** We argue that automated analyses can help to identify and resolve such situations [17], and thus support developers ensure the semantic correctness of test-case merges.

In this paper, inspired by code-clone detection, we propose a technique for analyzing forked test cases regarding their merge potential and for guiding developers in ensuring the semantic correctness of the merged variants. Precisely, we build on abstract syntax trees (ASTs) to distinguish between different merge situations (e.g., new tests, changed tests, missing tests) and provide tool support to help developers decide what to do in such situations (e.g., use tests as they are, change tests). We implemented a prototype of our technique and evaluated it on five open-source systems. The results indicate that our technique can guide developers when merging test cases by highlighting which tests are still up-to-date or require adaptations.

In detail we contribute the following:

- We propose our technique for analyzing the merge potential of forked test cases.

- We evaluate our prototype implementation on 50 pull requests from five open-source systems to discuss our technique's feasibility.
- We publish an open-access repository with our prototype, evaluation dataset, and further documentation.[1]

Our contributions are a step towards supporting developers in identifying and handling problematic situations during test-case merging. Particularly, our technique can help practitioners save time during the merging of forks and improve their confidence that they merged forked test cases correctly. Consequently, we help limit the potential for bugs, violations of quality requirements, or reduced test coverage. For researchers, we highlight opportunities for future work, particularly exploring test-case merges in more detail and providing further support for developers.

2 FORK-BASED SOFTWARE DEVELOPMENT

In the following, we describe the workflows of version-control systems and fork-based software development, based on which we introduce the concept of code clones. We display a conceptual overview of fork-based software development in Figure 1. Typically, version-control systems enforce a pull-commit-push workflow (using different terminologies). For this purpose, a developer sets up a main repository on a server that comprises the system. Then, any collaborating developer can clone a local copy on their device to use or change the system. The usual workflow is to pull any new changes from the server into the local copy, implement the desired changes, commit them with a message, and push the commit back to the main repository. If something changed in the main repository in the meantime, an additional merge (potentially manually if a conflict occurs) and consequent commit are needed. For each commit, the version-control system creates a revision (black circles in Figure 1) that has a unique identifier (e.g., a hash) to allow for reverting to a previous revision.

More important for our idea is that modern social-coding platforms that build on version-control systems also allow to fork the main repository into another repository. In this case, the development of both repositories is independent of each other, if they are not explicitly synchronized. For instance, fork 1 in Figure 1 is created by forking the first revision of the main repository. After implementing a change, the fork is merged back into the main repository. Similarly, forks can be created from other forks (e.g., fork 3 stems from fork 2 in Figure 1). Since the forks are independent, they only comprise the system status of the revision they were forked from. To update a fork to a more recent revision, a developer may synchronize their fork with another fork by merging the other fork into their own (e.g., main to fork 3). For the other direction (e.g., fork 2 to main), the developers of a fork can ask that their changes are integrated into another fork or the main repository (e.g., via pull requests in GitHub). The other fork's developers then decide whether to merge the changes. Note that, in our examples, we refer to FORKED being merged into BASE, independently of whether BASE is the main repository or another fork.

Synchronizing between forks is not always planned, and sometimes they are kept independent to implement features for specific customers or users, resulting in a variant-rich system [2, 3, 18, 21, 34, 35]. However, even then some features implemented in one fork may be interesting for other forks, too. Again, the developers can synchronize between different forks by merging or by cherry-picking specific commits into their own fork.

[1]https://doi.org/10.5281/zenodo.6815052

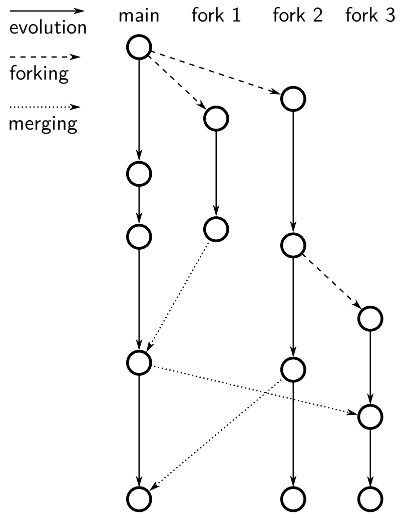


Figure 1: Conceptual overview of fork-based development.

Forks are essentially code clones on a system-level scale: They implement new features or change existing features, but most of their code is still a clone of another system. For our technique, we are concerned with differences of test cases (i.e., new tests, changed tests, removed tests) and the code they test. Consequently, we are interested in more fine-grained types of code clones that exist between forks, which we distinguish based on the definitions of code clones by Roy et al. [29]:

Type-1 clones implement the same functionality through identical pieces of code with potential changes in whitespaces, layout (e.g., indentation), or comments.

Type-2 clones implement the same functionality through syntactically identical code, but involve additional changes to Type-1 clones in identifiers, types, and literals.

Type-3 clones implement the same functionality through code that is even further modified than Type-2 clones, for instance, by changing, adding, or removing statements.

Type-4 clones implement the same functionality through syntactically completely different code.

Various techniques have been proposed to identify different types of code clones [1, 6, 29, 31]. Since they involve more and more complex changes, higher-level code clones are harder and harder to find. For our technique, we aim to compare different test cases against each other, which either existed already (i.e., have been forked as Type-1 clones), are new, or have been deleted. Consequently, we have to cope with code-clones up to Type-3.

3 AST-BASED TEST CASE SIMILARITY

In this section, we present our technique for determining test-case similarity using AST-based code-clone detection. Besides explaining why we cannot reuse existing code-clone detection techniques, we

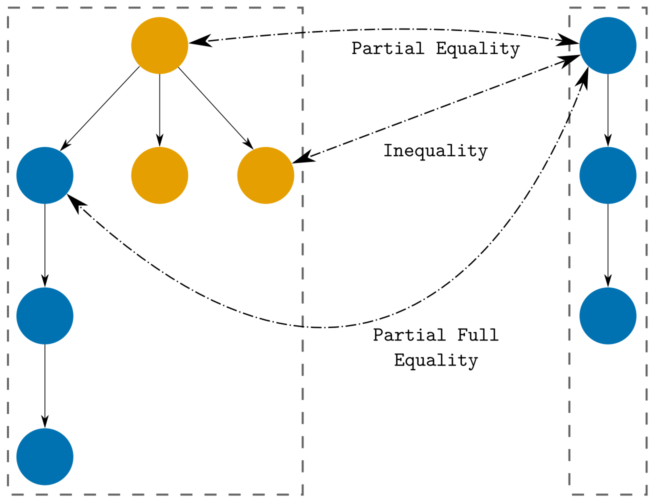


Figure 2: Two example ASTs illustrating the different types of similarity we use in our technique. Same colors indicate that the nodes are identical between the trees.

present an overview of the workflow of our technique and details of the different steps therein. Partly, we briefly elaborate on the implementation of our technique.

3.1 AST-Based Similarity

A crucial aspect of our technique, which also distinguishes its concept from typical code-clone detection, is the evaluation of the similarity between two test methods. More precisely, our main goal is to support the merging of test methods, and thus to provide sufficient information to facilitate developers' tasks when merging the test cases. For this purpose, in contrast to typical code-clone detection, we require more details from our similarity analysis to guide the merging of test methods. In detail, instead of only knowing what source code of two methods represents code clones, we have to understand whether and how the clones have been modified (e.g., added, modified, or moved lines of code). We use this information to guide developers in understanding modifications between cloned test cases, for instance, whether lines have been interchanged to fix a bug in the test or whether a new line corresponds to modifications in the method under test. Our argument is that this information is more helpful to decide how to merge test cases than the pure existence of code clones.

To compute the similarity between test cases, we implemented an AST-based technique to obtain a tree-based similarity score, which is similar to common tree-based code-clone detection techniques. However, to support developers during the actual merging, we propose more fine-grained definitions of equality in trees, which lead to different types of similarity according to the underlying tree-based structure. In particular, we propose the following four types of similarity (based on their tree representation), each of which results in different suggestions with respect to merging the corresponding test cases (cf. Figure 2 for examples):

- **Full equality** implies that both ASTs are identical, and thus the test cases can be merged without any modifications. So, a developer can simply keep either of the test cases, since neither has been changed in any of the forks.
- **Partial equality** implies that two ASTs share a common subtree. We show an example for a common subtree in Figure 2 (top arrow), where the blue nodes indicate the common subtree in both ASTs. Note that both ASTs comprise these nodes, but at different positions. Moreover, we show a corresponding code example in Listing 1. This listing contains two test cases, where the code in Lines 3–6 and Lines 13–17 constitutes a common subtree on code level. Using this similarity, we are able to identify that test cases have been modified, but kept parts of their original implementation (e.g., added lines of code).
- **Partial full equality** implies that one AST is completely contained in the other AST. As an example, consider the two ASTs in Figure 2: The right AST is completely contained as a subtree in the left AST, and thus completely subsumed. Depending on which of these trees represents a test method in Base and in Forked, respectively, this has crucial implications on possible merge scenarios. Precisely, it is likely that the subsumed test case can be replaced by the other test case if the corresponding changes in the methods under test are also merged. In contrast to partial equality that only identifies commonalities and requires detailed manual inspection, partial full equality means that one test can simply be replaced.
- **Inequality** implies that two ASTs do not share any common subtrees, or share only a few nodes (i.e., below a specified threshold). Consequently, we consider these test cases to be disjunct, and thus constitute independent test cases that should not be merged at all. We display an example of two nodes in two ASTs that are unequal in Figure 2, indicated by the second arrow in the middle of the other two. These nodes are unequal, because they represent completely different source code in the test cases.

Note that these similarities are important for comparing cloned test cases and their modifications. Handling new or removed test cases only does not require such similarities, since there exist no counterparts to which we could compare the test cases.

Listing 1: A collection of example test cases.

```
1  @Test
2  public void testList() {
3      List<Integer> bar = new ArrayList<>();
4      bar.add(21);
5      bar.add(42);
6      assert bar.size() == 2;
7  }
8
9  @Test
10 public void testArrayList() {
11     TestObject foo = new TestObject() {
12         public test() {
13             List<Integer> foo = new ArrayList<>();
14             foo.add(21);
15             foo.add(42);
16             assert foo.size() == 2;
17         }
18     };
19     foo.test();
20 }
```

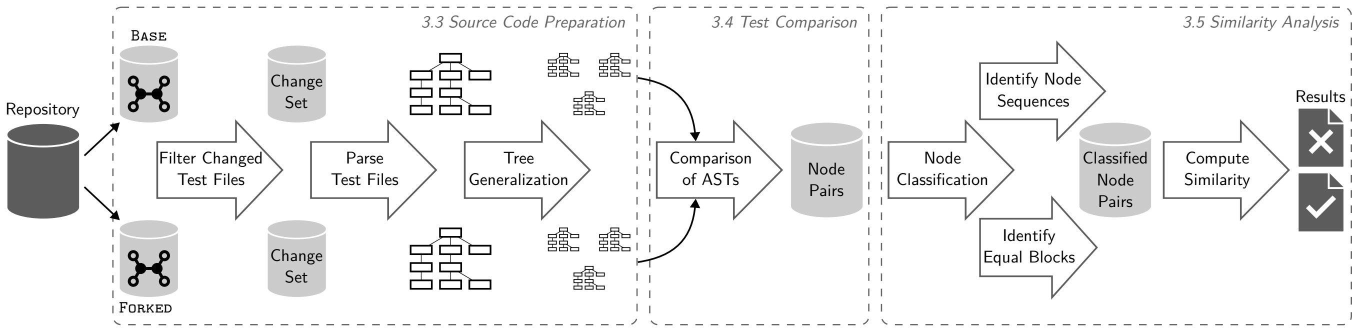


Figure 3: Workflow of our AST-based technique for analyzing test case similarity.

3.2 Overview

In Figure 3, we display the overall workflow of our proposed technique. Basically, our technique involves three phases: *source code preparation*, *test case comparison*, and *similarity analysis*. Initially, a developer needs to specify the two variants that they want to compare (i.e., merge). We remark again that we refer to these variants as BASE and FORKED, where the latter is assumed to be a forked variant of BASE and shall be merged into BASE. Note that our assumption is that both, BASE and FORKED, can be subject to changes, and thus must be analyzed with our AST-based similarity technique.

Source Code Preparation. In the first phase, our technique filters the source code of both variants to identify changed test files. Subsequently, it parses only those files into an AST-based representation, since it is not necessary to merge identical test cases (i.e., those representing full equality). Moreover, we employ some additional filtering and normalization (generalization) to the ASTs to reduce the amount of code our technique has to compare in later phases, and to improve the recall when comparing methods. As the result of this phase, we obtain an AST-based representation of all test cases that have been changed between BASE and FORKED.

Test Case Comparison. In the second phase, our technique compares all test cases (i.e., methods) of the two variants with each other; precisely, their AST-based representations created in the previous phase. For this purpose, we employ a hash-based comparison, which reduces the effort of comparing the ASTs of all test cases in BASE with those in FORKED. As the result of this phase, we obtain a similarity score for each pair of test cases that has been subject to this comparison (i.e., all that have not been removed due to full equality before).

Similarity Analysis. In the last phase, our technique analyzes the similarity scores of the test-case pairs from the previous phase. First, we search for best matches, that is, the pairs with the highest similarity. Afterwards, we classify these best matches according to the different types of tree similarity we introduced in Section 3.1 to derive guidance for the developer on how to best merge the test-case pair.

Note that our technique is intended to support the merging of larger changes, for instance, in large software ecosystems with many forks (e.g., the Linux Kernel). So, we implemented automation to allow for a fast analysis of many forked test cases. An ad hoc use to support a single test-case merge can be easily implemented, but the usefulness of our technique in this scenario depends on the complexity of the forked test cases and their modifications. In the remainder of this section, we explain each of the three phases in more detail. We remark that our technique is designed for Java source code and we focus on unit testing (i.e., JUnit[2]) in this paper, and thus certain steps of our technique are language-specific (e.g., how we assume test code to be structured).

3.3 Source Code Preparation

We use the source code preparation to identify relevant test code that we have to compare, and to transform these test cases into AST-based representations. Initially, we have to specify the two variants that our technique shall compare. Since our focus is on fork-based development, we consider the main branch of a certain project (specified via its GitHub repository) as BASE and a user-specified fork of that main branch as second variant, called FORKED. Consequently, our perspective is that changed test cases in FORKED are subject to be merged into BASE, for example, by means of a pull request. Based on this perspective, our technique prepares source code as follows.

Filter Changed Test Files. First, our technique identifies relevant files that are subject to our analysis in a pre-filtering step. More precisely, we aim to reduce the number of test cases in FORKED that we have to consider during the comparison by identifying those cases that have changed compared to BASE, as only such test cases are of interest during a merge. Since we assume a Git-based repository, we can use `git -diff` to obtain the modified files from FORKED. In contrast, we do not know which of the files may be relevant in BASE, and thus we take all files from BASE into account. However, we apply one more step of optimization to both variants: Since we are only interested in test cases, which usually reside in a dedicated test folder, we can omit any source code files in other folders. Thus, after this pre-filtering step, we end up only with test cases that have actually been modified in FORKED.

Parse Test Files. In the next step, our technique *parses* the previously filtered files (in our prototype, we use *JavaParser*[3]). Since test cases are usually implemented in corresponding test methods (i.e., JUnit tests), parsing takes place on method level. So, for each method that our technique parses, we keep its qualified name, the

[2]https://junit.org/junit5/
[3]https://javaparser.org/

whole method itself as an AST, any annotations attached to the method (e.g., `@Test, @Before`, or any other commonly used annotations), and a link to the file containing the method. Along with this parsing step, we also perform another optimization: For each test case that occurs in the same class in Base and Forked, we perform a simple comparison on the corresponding ASTs to check for full equality (cf. Section 3.1). Our rationale is that, while a file may have changed in Forked, certain test cases in that file are potentially still identical with the original version in Base. Consequently, if we identify such test cases, we can discard them for the remaining steps, because they are not relevant for the merging (since there are no differences between the test cases of the two variants). Note that we currently identify corresponding classes between Base and Forked by comparing the absolute class paths. So, if a class has been moved or renamed in one variant, we would miss potential full equality of some test cases in this class at this point. However, we consider it more important for our technique to achieve a high precision (i.e., no classes or test cases should be wrongly considered as corresponding, and thus be wrongly discarded).

Test File Assignment. As an intermediate step, we assign the ASTs from the parsing step to the variant they originated from, that is, either Base or Forked. This way, we ensure that, in the comparison phase (cf. Section 3.4), we only compare test cases between the two variants, but not within the same variant. For merging, only such comparisons between variants provide relevant information.

Tree Generalization. As last step of the source code preparation, we perform *tree generalizations* on the obtained ASTs by employing different normalizations to detect test cases that still exhibit a high structural similarity (as done during code clone detection as well). In particular, we create two additional versions of each AST: One version in which we remove all comments, and thus can perform a more relaxed detection of Type-1 clones. Second, we normalize all identifiers and literals, which allows us to detect structurally similar test cases, even though they differ in variable names or constants (which is comparable to Type-2 clones). As a result, we keep three versions of each AST for the subsequent test case comparison:

1. the original AST,
2. AST without comments ("Type-1 AST"), and
3. AST with normalized identifiers and literals ("Type-2 AST").

These ASTs are the input for the test case comparison, which we describe next. Note that we also consider changes in comments (original AST), since they may not pose challenges in the code merge but can indicate improved or updated documentation.

3.4 Test Comparison

In this phase, we perform the actual comparison of test cases that we obtained through the previous phase, that is, we compare all remaining test cases from Base with their counterparts from Forked. Note that the comparison takes place on all three kinds of ASTs, individually: the original ASTs as created by the parser, the Type-1 ASTs, and the Type-2 ASTs. Since this can become quite exhaustive, we initially apply a pre-hashing step. Namely, we compute the hash value for each node of an AST and store these hashes in a list, associated with the corresponding AST. To this end, we use the built-in hash function of *JavaParser*. For each node, the hash function not only considers the node for which the hash value is computed itself, it also takes any child nodes into account. Thus, if two nodes are identical and also have identical child nodes, they will exhibit the same hash value. As a result, we obtain a list of hash values for every AST with each value maintaining a link to its corresponding node in the AST.

Using the resulting hash lists, we now perform the comparison for all pairs of test cases between the two variants by comparing the hash values of their AST nodes. If two hash values between two ASTs are equal, we keep the corresponding nodes as identical node pair for further comparison (i.e., there is similarity between the test cases). Otherwise, we discard this pair of test cases as not being cloned from each other (i.e., since there is not a single equal node, we assume that the test cases are independent). We remark that the meaning of two nodes being identical depends on the type of AST we are considering. For instance, nodes being identical in the original AST means that the corresponding source code is syntactically identical, since we did not perform any normalization. In contrast, identical nodes between Type-2 ASTs only indicate that the normalized code is identical. This implies that the source code in one variant differs from the other regarding literals or identifiers (since we normalized those), and thus the corresponding source code is syntactically similar to a certain extent. Our technique interprets such a comparison result as the two test cases being related (i.e., cloned), but modified—which are the test cases developers have to pay particular attention to during a merge.

As a result of this comparison, we obtain a (probably large) list of node pairs, each of which represents identical nodes. However, this list contains redundant information, because of the way we compute the hash values: If a pair of nodes represents identical subtrees, we also have pairs of nodes in our list that represent the nodes of these subtrees. Since we are interested in sequences of a certain length (i.e., whole subtrees rather than sub-sequences or single nodes therein), we can remove any node pair that is subsumed by another one. As an example, consider the blue nodes in the two trees we display in Figure 2. Our comparison would yield three node pairs for this subtree. One for the leave node, one for the middle node, and one for the root node of this subtree. However, the result we are interested in is the pair of root nodes, since these nodes subsume the other nodes as part of their subtrees.

We clean up the list that we obtained after the hash comparison as follows. For each node pair, we check whether there is a node pair of their parent nodes in the respective ASTs. If this is the case, we can safely discard the pair of child nodes from our result list. For each pair of test cases, we proceed iteratively in a bottom-up fashion until we found the topmost nodes that are identical. These nodes represent the node pair (with the respective subtrees) we keep for our remaining analysis. Then, our technique uses the resulting list to continue with the last phase, which we explain next.

3.5 Similarity Analysis

In this last phase, our technique analyzes the result list from the previous step in greater detail to determine the type of equality between test cases to provide information useful for developers in possible merging scenarios. As a preliminary step, we determine for each pair of nodes which kind of syntactical elements they represent, for instance, whether it is a block statement, a method declaration, and so on. Depending on the type of nodes, we analyze the pair of nodes in different ways, as explained in the following.

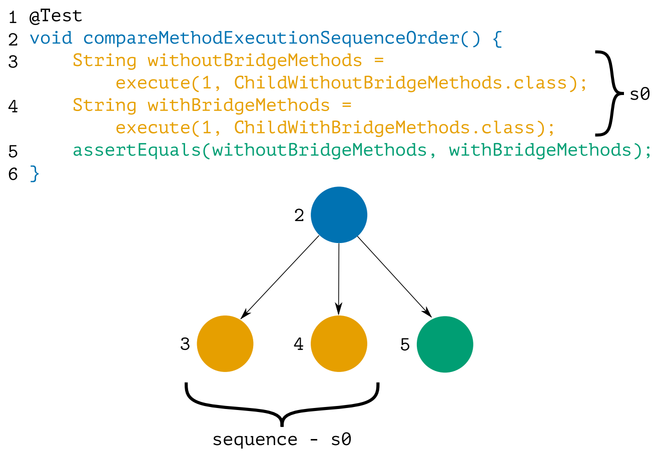
Search Longest Node Sequence. In case that the two nodes represent leave nodes in the ASTs we compare (e.g., assignments, variable declarations), we perform a search for the longest sequence these nodes can build with other nodes. The reason is that, while we already found the topmost nodes representing a subtree in the previous phase, we may still have multiple sibling nodes within an AST (and thus between the compared ASTs) that are equal, even though their parents are not.

As an example, consider the code and AST we display in Figure 4. In this example, we assume that the two yellow sibling nodes represent identical statements (i.e., Type-2 code clones). However, in our results list, they occur in at least two independent pairs of nodes (due to the comparison with the other AST), since there is no parent-child relationship between them. So, based on the structure of the AST, we identify which node pairs constitute sibling nodes and join them into one common sequence as long as there are identical sibling nodes available. As a result, we obtain sequences of nodes that represent an identical sequence of statements in the corresponding test cases.

Search Equal Blocks. If the nodes of a node pair constitute a block statement (e.g., `if` blocks, `for` loops, or even whole methods), we now check in which type of AST we found them. In case that they are part of the original AST (i.e., the one we did not normalize), we know that also the source code between the compared variants is identical. This means that, if the nodes represent methods (i.e., the block is a method declaration), we can safely remove them, since they are not subject to merging. If they are of any other kind of block statement, we propose possible refactorings as part of the merging process, for example, to extract the block in a dedicated method and merging the remaining (different) parts of the test case. In case that we found the identical blocks in any of the other two types of ASTs (i.e., those we normalized), we cannot automatically reason on the differences. Then, the test cases are subject to merging without any refactoring recommendations—meaning that a developer has to inspect and comprehend the differences between the test cases.

Compute Similarity. For all node pairs remaining after the previous steps, we now compute their similarity (as percentage of matching). More precisely, we determine the number of nodes from Forked that are identical with the AST of Base, where they are located in the ASTs, and whether the lengths of both ASTs differ. Furthermore, we categorize each pair of ASTs regarding whether they exhibit *partial equality* or *partial full equality* (cf. Section 3.1). We build on this information to support developers during the merging process. For instance, by considering the categorization of *partial full equality*, a developer knows about cases where a test case from Base is completely contained in a test case of Forked. As a result, we can suggest to merge the latter test case into Base, since it is an extended version of the original method in Base.

All results from this analysis phase together with the information about where identical nodes are located in the source code are consolidated into a report. Currently, this is a JSON[4] file. This file can then be further processed into a format that is understandable for the developer (e.g., static HTML pages, visualizations in an IDE or Git client).



```
1 @Test
2 void compareMethodExecutionSequenceOrder() {
3     String withoutBridgeMethods =
          execute(1, ChildWithoutBridgeMethods.class);
4     String withBridgeMethods =
          execute(1, ChildWithBridgeMethods.class);
5     assertEquals(withoutBridgeMethods, withBridgeMethods);
6 }
```

Figure 4: An example for sibling nodes that are identical between two ASTs. The colors indicate which node is associated with which expression.

4 EVALUATION

In this section, we first describe the design of our evaluation. Then, we report and discuss the results we obtained as well as potential threats to validity.

4.1 Design

To evaluate our technique, we decided to perform a qualitative feasibility study. For this purpose, we employed the following design.

Objectives. Our goal for our evaluation was to *understand to what extent our technique can identify similarities between forked test cases*, and thus can suggest proper actions to developers. For this purpose, we decided to perform a simulation by using existing, merged pull requests from real-world software projects. Precisely, we used the two commits (i.e., main repository, fork) that have been merged as input for our technique and compare its results to the actual merge. By analyzing which code structures are detected dependably and which are falsely reported during this simulation, we evaluate the performance of our technique.

Dataset. We created an own dataset of pull requests that we picked from GitHub projects, each of which had to fulfill the following project-level selection criteria (SC_P):

$SC_{P\text{-}1}$ The pull requests of the projects are publicly available.
$SC_{P\text{-}2}$ New features of the project are accompanied by tests.

With $SC_{P\text{-}1}$, we ensured that we could inspect the pull requests and use them for our simulation; which is typically fulfilled by all public repositories. We defined $SC_{P\text{-}2}$ to ensure that the projects we consider use test cases actively (e.g., not only at random or solely legacy ones), and that we could identify pull requests on which we can evaluate our technique regarding the different situations we specified. To identify projects, we inspected the most popular ones according to mvnrepository.[5] We decided to limit our simulation to

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

[5]https://mvnrepository.com/popular

Table 1: Overview of the projects and pull requests we selected for our evaluation. The pull requests are hyperlinked.

project	domain	# contributors	# forks	SLOC	# test cases	pull requests
Guava https://github.com/google/guava	programming libraries	267	≈ 9,500	756,069	1,071	5347, 5321, 5307, 5280, 5252, 4035, 4029, 4025, 4020, 3998
Jackson https://github.com/FasterXML/jackson-databind	file parser	181	≈ 1,200	198,650	2,667	2952, 2948, 2942, 2938, 2931, 2927, 2925, 2922, 2921, 2915
JavaParser https://github.com/javaparser/javaparser	code transformation	160	≈910	272,627	2,371	2974, 2972, 2971, 2969, 2966, 2961, 2956, 2955, 2954, 2952
JUnit 5 https://github.com/junit-team/junit5	unit testing	173	≈ 1,100	134,727	3,653	2442, 2383, 2382, 2371, 2368, 2342, 2336, 2328, 2387, 2380
mockito https://github.com/mockito/mockito	mocking	230	≈ 2,200	88,851	2,018	2135, 2071, 2051, 2042, 2034, 2023, 2013, 1996, 1989, 1971

five projects that fulfilled our project-level selection criteria: Guava, Jackson-Databind, Javaparser, JUnit5, and Mockito.

In Table 1, we display an overview of these five projects. As we can see, the projects cover different domains and involve 160 to 267 contributors, indicating larger communities of interest. All projects are established and widely-used open-source solutions, for example, the Guava libraries developed by Google or the well-known JUnit framework. Moreover, we can see that the projects span from 910 to around 9,500 forks, with 88,851 to 756,069 SLOC, and 1,071 to 3,653 test cases. Consequently, we argue that our sample comprises a diverse set of medium- to large-scale projects that rely on fork-based software development and extensive testing. These are ideal subjects, since we aim to support such projects with our technique.

In the next step, we identified pull requests we could use for our simulation. To this end, we defined the following pull-request-level selection criteria (SC_{PR}):

$SC_{PR\text{-}1}$ The pull request must involve at least one change to an existing test case.

$SC_{PR\text{-}2}$ The pull request has been merged into the main repository.

We defined $SC_{PR\text{-}1}$ to ensure that we do not only cover additions of new or deletions of old test cases, but force our technique to compare modified test cases. Precisely, handling added or deleted test cases is straight forwardly supported through our technique (i.e., the developer has to decide whether to add or delete the test case during merging), but forked test cases comprising modifications (i.e., changing the functionality of the test case) are those actually challenging program comprehension and merges. Using $SC_{PR\text{-}2}$, we ensured that we have a baseline (i.e., the test cases merged by the developers) to which we can compare the results of our technique. We selected the 10 most recent (as of December 11, 2020) pull requests from each of the five projects, resulting in a dataset of the 50 pull requests we display in the last column of Table 1.

Implementation. We implemented two prototypes to conduct our evaluation: The first implements our technique as described in Section 3, while the second provides a framework for combining our technique with Git. For both, we used Kotlin and Gradle to be able to use existing Java libraries. We relied on JavaParser for parsing and analyzing ASTs as well as JGit[6] for communicating with the Git repositories. As output, our prototype generates JSON files that specify changed files and lines of code relating to test cases, suggested changes to the tests, and potential warnings (i.e., code structures our prototype identified but did not analyze, e.g., out of scope, errors). For our evaluation, we used a Debian Buster operating system with a Linux 4.19 LTS Kernel, a 2.5 GHz eight-core AMP EPYC processor, and 16 GB RAM.

[6]https://www.eclipse.org/jgit/

Analysis. To evaluate our technique, we relied on a qualitative analysis of all 50 pull requests. We decided to perform a qualitative analysis because our technique does not automatically refactor test cases, but suggests which test cases may require manual changes by the developers. So, we decided to inspect the JSON files our prototype generated, and in which it stores information regarding suggested changes. In detail, the third author of this paper compared the properties of the suggested (i.e., our prototype) to the real-world changes (i.e., the merge commit of the pull request) . To this end, he considered the location, similarity, and resolutions (e.g., removal, addition, modification) of changes.

4.2 Results

In the following, we first report descriptive statistics of the test cases in each project. Then, we describe more in-depth insights into our technique that we derived from qualitatively inspecting the analysis results of our technique for each pull request we investigated.

Descriptive Results. In Table 2, we provide a descriptive overview of the results our prototype generated. Note that we analyzed all test cases within the projects compared to the forks involved in the pull requests we inspected. We can see that our technique identified between 10 and 69 similar sequences within the test cases of each project, with an average length of three SLOC for each sequence. Moreover, we found a highly varying number of highly-similar test cases (i.e., more than 95 % similarity), ranging from nine to 2,412 with an average number of SLOC between 3.06 and 11.71. Note that our prototype correctly identified all sequences, and thus works as intended. Moreover, we found that almost all identified sequences are relevant for developers when merging forks (i.e., the corresponding test cases have been modified and exhibit similarity). During the following qualitative analysis, we aimed to understand the reasons for the numbers, particularly for

Table 2: Overview of the descriptive results of our evaluation, separated by individual sequences and highly-similar test cases (>95 % similarity).

project	sequences		test cases	
	#	avg. SLOC	#	avg. SLOC
Guava	18	3.00	154	3.06
Jackson	11	3.00	559	4.43
JavaParser	69	3.00	9	11.34
JUnit 5	10	3.39	2,412	3.11
mockito	30	3.00	55	11.71

the large variations regarding highly-similar test cases. Moreover, we aimed to understand to what extent corner cases occur in our results that may require adaptations to our technique (e.g., wrong matches between test cases, test case designs interfering with the design of our technique).

Guava. Overall, we identified 154 highly-similar test cases in the Guava project. However, this is primarily due to the design of the test suite itself, with our prototype revealing empty test cases used throughout the project. Reporting such empty test cases as similar during a merge could arguably cause confusion for the developers. Moreover, our prototype revealed several test cases that are identical; typically overwritten wrapper methods of classes (e.g., `hashCode()`). During our manual review, we further found that some added test code (i.e., from a fork) was completely new, so that no changes were required to existing test cases.

Jackson. Analyzing the Jackson project, we found 559 highly-similar test cases. During our manual review, we found that this is mainly due to the structure of the Jackson tests, which use inherited classes in separate test cases. The highly-similar test cases are often duplicated class methods, such as `getter` and `setter`. Arguably, our prototype can help developers identify and resolve such redundancies by introducing a better abstraction in some cases. Furthermore, we found some error-prone matches in Jackson. For instance, we display a high-similarity match in Listing 2. Our prototype identified this match, because many small sub-trees in the ASTs of the test cases are similar (e.g., the order of assertions from Lines 16–18 and Lines 25–27 is represented identically in our Type-2 ASTs). The issue here is that a large AST in the main repository means that only a small set of its sub-trees have to match the AST of a forked test case to yield a high similarity. However, such cases seem rare and, in general, our manual review of Jackson also indicates that our technique is helpful. In the future, we have to update our technique to handle such specific structures of test cases and to combine the results of the different ASTs more.

JavaParser. The selected pull requests we inspected for the JavaParser project yield similar results as the ones of the other projects. Most matches our prototype reports are concerning similar sequences between test cases, found in several positions of the source code. For example, our prototype often detects builder sequences, which we would recommend to extract into distinct methods. Notably, we identified one false match by our prototype. Namely, it mismatched two forked test cases, because one involved only a few lines of code that matched code in another test case with far more

Listing 2: Test case example from Jackson.

```
1  // base
2  public void testCustomBeanDeserializer() throws
       Exception {
3    // [...]
4    assertNotNull(beans);
5    results = beans.beans;
6    assertNotNull(results);
7    assertEquals(2, results.size());
8    bean = results.get(0);
9    assertEquals("", bean.d);
10   c = bean.c; assertNotNull(c);
11   assertEquals(-4, c.a);
12   assertEquals(3, c.b);
13   bean = results.get(1);
14   assertEquals("abc", bean.d);
15   c = bean.c; assertNotNull(c);
16   assertEquals(0, c.a);
17   assertEquals(15, c.b);
18 }
19
20 // fork
21 public void testSingleElementWithStringFactoryRead()
       throws Exception {
22   String json = aposToQuotes("{ 'values': '333' }");
23   WrapperWithStringFactoryInList response = MAPPER.
       readValue(json, WrapperWithStringFactoryInList.
       class);
24   assertNotNull(response.values);
25   assertEquals(1, response.values.size());
26   assertEquals("333", response.values.get(0).role.Name)
       ;
27 }
```

lines (similar to the Jackson example). Consequently, our prototype assumed that the test cases are forked from each other (i.e., it identified identical nodes), but this was not the case. In the future, we aim to handle such cases either by discarding them earlier or by providing specific refactoring recommendations to the developer (e.g., to extract the common code if feasible).

JUnit 5. As in the other projects, our prototype reveals similar sequences in JUnit 5 in object builders and initializations in forked test cases. Analogous to the other examples, an extraction may be done, but many short sequence extractions could lead to a reduction in code readability. So, we would highlight such cases to the developer and ask them to decide whether to merge and extract which of the common sequences. However, our prototype also reveals almost identical test cases except for literals passed to builder functions and checks, which indicates a high merge potential.

One interesting observation we found only for JUnit 5 is the existence of exactly equal test cases with intentionally different names. For example, we found two test cases in the test suite with exactly the same content, which call a method with a shared object from the global scope. The actual test is whether all methods are called correctly and in the order of the test criteria (i.e., checking for side effects on the shared object caused during the execution of the methods). While our prototype detected the similarity in the forked test cases correctly, the implication of this result violates the actual intent of the test cases. In fact, merging the test cases would modify their outcome and mean that they do not fulfill their purpose anymore. To avoid such false matches, we would need exceptions to the similarity detection of our technique. However, a general implementation of such exceptions is hardly possible, since they may vary between different projects and the duplication may also be an error.

Listing 3: Test case example from Mockito Base.

```java
1  @Test
2  public void can_define_class_in_closed_module() throws Exception {
3    assumeThat(Plugins.getMockMaker() instanceof InlineByteBuddyMockMaker, is(false));
4    Path jar = modularJar(true, true, false);
5    ModuleLayer layer = layer(jar, false);
6    ClassLoader loader = layer.findLoader("mockito.test");
7    Class<?> type = loader.loadClass("sample.MyCallable");
8    ClassLoader contextLoader = Thread.currentThread().getContextClassLoader();
9    Thread.currentThread().setContextClassLoader(loader);
10   try {
11     Class<?> mockito = loader.loadClass(Mockito.class.getName());
12     @SuppressWarnings("unchecked")
13     Callable<String> mock = (Callable<String>) mockito.getMethod("mock", Class.class).invoke(null, type);
14     Object stubbing = mockito.getMethod("when", Object.class).invoke(null, mock.call());
15     loader.loadClass(OngoingStubbing.class.getName()).getMethod("thenCallRealMethod").invoke(stubbing);
16     boolean relocated = !Boolean.getBoolean("org.mockito.internal.noUnsafeInjection") && ClassInjector.UsingReflection.isAvailable();
17     String prefix = relocated ? "sample.MyCallable$MockitoMock$" : "org.mockito.codegen.MyCallable$MockitoMock$";
18     assertThat(mock.getClass().getName()).startsWith(prefix);
19     assertThat(mock.call()).isEqualTo("foo");
20   } finally {
21     Thread.currentThread().setContextClassLoader(contextLoader);
22   }
23 }
```

Listing 4: Test case example from Mockito Forked.

```java
1  @Test
2  public void can_define_class_in_open_java_util_module() throws Exception {
3    assumeThat(Plugins.getMockMaker() instanceof InlineByteBuddyMockMaker, is(false) );
4    Path jar = modularJar(true, true, true);
5    ModuleLayer layer = layer(jar, true, namedModules);
6    ClassLoader loader = layer.findLoader("mockito.test");
7    Class<?> type = loader.loadClass("java.util.concurrent.locks.Lock");
8    ClassLoader contextLoader = Thread.currentThread().getContextClassLoader();
9    Thread.currentThread().setContextClassLoader(loader);
10   try {
11     Class<?> mockito = loader.loadClass(Mockito.class.getName());
12     @SuppressWarnings("unchecked")
13     Lock mock = (Lock) mockito.getMethod("mock", Class.class).invoke(null, type);
14     Object stubbing = mockito.getMethod("when", Object.class).invoke(null, mock.tryLock());
15     loader.loadClass(OngoingStubbing.class.getName()).getMethod("thenReturn", Object.class).invoke(stubbing, true);
16     boolean relocated = !Boolean.getBoolean("org.mockito.internal.noUnsafeInjection") && ClassInjector.UsingReflection.isAvailable();
17     String prefix = relocated ? "org.mockito.codegen.Lock$MockitoMock$" : "java.util.concurrent.locks.Lock$MockitoMock$";
18     assertThat(mock.getClass().getName()).startsWith(prefix);
19     assertThat(mock.tryLock()).isEqualTo(true);
20   } finally {
21     Thread.currentThread().setContextClassLoader(contextLoader);
22   }
23 }
```

Some other tests in JUnit 5 are empty (i.e., they test whether tests can be named in a certain way). While these cases were not relevant during the pull requests we analyzed with our prototype, this would cause a similar problem. A possible solution to such problems would be to add configuration options to our technique or annotations to the test cases that specify their purpose. In fact, incorporating a detailed analysis of test annotations (which are already available in our prototype) could help us improve its performance.

During our continued analysis of JUnit 5, we noticed that one commit did not pass the parsing and detection of commonalities of ASTs in our technique. While we did not find specific properties in this commit that would cause this problem, some part of our analysis exceeded the memory reserved for our prototype. We intend to conduct further tests to isolate and understand this problem, but this is out of scope for this paper. Generally, our prototype correctly identifies relevant test case changes in JUnit 5, which are typically highly relevant for developers during merges. However, we also found a bug and potentially misleading recommendations, which highlight the limitations of our technique and prototype.

Mockito. In Mockito, we found large overlaps between many test cases, with some different calls in between, indicating a high similarity with smaller changes. Due to the high similarities and identical literals, we consider such test cases to have a high potential for merging—and for introducing errors (e.g., because it can be unclear why a test case has been changed in what way). We display a prime example for such regular occurrences across our whole dataset in Listing 3 (Base) and Listing 4 (Forked). The test case in Listing 4 has been modified in eight of 23 lines (i.e., Lines 4, 5, 7, 13, 14, 15, 17, 19). However, we can see that the basic structure of the test cases is still identical, and most of the modifications are small (e.g., a Boolean value in Line 4, a String in Line 7). So, a developer may want to merge the test cases if the underlying changes have also been merged, or refactor the test cases to provide a common template that can be adapted more easily. Our technique can help developers identify such cases, understand to what extent the test cases have been modified, and abstract their implementation to mitigate bugs. In general, we found that the reports of our technique for Mockito are a good showcase for using it.

Moreover, our technique identified further duplicated sequences in added and changed tests. Mainly, these are already existing builder sequences to prepare objects for testing. The longer of these sequences are good candidates for simplifications by merging common code into helper methods. Interestingly, we found one mismatch between an existing test case and a method that was introduced in a fork, but that is not a test. Instead, the new method only instantiates objects for other tests (which is why it is part of the test suite). While this is somewhat of a mismatch, we consider such cases also valuable, since they may provide an indication for developers how to merge test cases in their project.

4.3 Discussion

Our feasibility study indicates that our technique produces good results for most of the pull requests we inspected, but requires a few adjustments in the future. Some situations we analyzed have indicated matches of empty methods or side-effect driven test cases, which are correct matches but do not serve the purpose of detecting code duplication or helping developers during fork merges—they are often purposefully designed and created to behave in this manner. Additionally, in case of short methods (one to five SLOC) false matches occurred if other methods in comparison were considerably larger, for instance, in Jackson. Generally, our technique does not miss major modifications within the test cases of the selected pull requests, more often the matching was too inclusive. This indicates that we should concentrate on understanding the context in which tests are used to be able to provide more automated suggestions and better warnings.

One concept we implemented in our technique did not occur: the unification of exactly equal code blocks. We did not observe such situations, since its restrictions are rather strict compared to the others, though this stands in relation to our dataset. This dataset contains only mature projects and recent pull requests at the time of our analysis. Due to the projects' nature (i.e., extensive testing), this maturity comes with an established review process of code changes before they are merged. So, it is not likely that our technique unifies complete code blocks.

Overall, we argue that our technique provides a foundation for supporting developers in merging and evolving the test cases in their systems. In particular, our results indicate that our technique reliably identifies similarities and differences between test cases, guiding developers in which test cases require more effort and which can simply be merged. To improve our technique in the future, we have to fix small bugs, consider how test cases are used by developers and when these should not be merged, as well as how to present the results to a developer.

4.4 Threats to Validity

The validity of our evaluation is compromised by the dataset we created, which is relatively small since we performed a manual feasibility study. We assessed only 50 recent pull requests of five open-source projects, but more projects and pull requests can be added. Historic data in the form of older pull requests can offer valuable insights regarding how our technique performs in an evolving system. Moreover, the project range we considered is rather narrow, since we chose to pick popular Java projects. Our technique may perform differently on less popular projects, since these are not as stable. The generalizability of our technique could be improved by including projects in other programming languages, such as C, Rust, or Kotlin. However, we argue that the selection of projects from different communities that employ testing extensively is a reasonable method for evaluating the feasibility of our technique; and the focus on Java is caused by technical necessities. Finally, we identified positive results that improve our confidence in our technique, while also identifying some mismatches or unintended matches that we want to tackle in future work. So, we argue that our evaluation results are reasonable and provide a good indication that our technique can be helpful for developers when merging test cases.

5 RELATED WORK

Test-Case Similarity. Test-case (dis-)similarity is of particular interest for test-suite reduction, for instance, during the execution (i.e., sampling) or during the development (i.e., merging) [11, 16, 22, 28, 30] of a software system. Particularly merging similar test cases is closely related to our work. In principle, any code-clone detection technique [6, 29] can be used to identify similar test cases on code level. However, researchers have proposed other techniques to measure test-case similarity. For example, Cartaxo et al. [8] rely on analyzing stack traces and Cichos and Heinze [10] compare state machines instead of source code. While related, we have a different goal than such techniques, since we aim to merge tests cases of different forks instead of only reducing a test suite. So, we have to handle a number of scenarios (e.g., for new and modified test cases) that differ from these for test-suite reduction. As a consequence, we also had to adopt AST-based code-clone detection to consider the specific cases we described in Section 3.1.

Automated Test-Case Refactoring. Different researchers have studied test-case refactoring and proposed techniques for automating the process [9, 15, 17, 36]. For instance, Passier et al. [27] have proposed a technique for tracing changes to the source code that impact Unit tests, and for advising developers how to update the tests. In contrast to such techniques, we do not focus on the evolution of a single system, but the merging of forked test cases. Combining the different techniques for covering all aspects of software evolution is a direction for future work.

Semantic Merging and Regression Testing. Semantic merging aims to elevate merges from a purely textual level towards a semantic level, which is why they often need test cases that cover corresponding requirements [5, 26, 32]. Similarly, regression testing [23] aims to identify changed behavior in a system by executing a (sub-)set of tests that cover the corresponding requirements. Both types of techniques require a well-defined test suite to cover the requirements of the system's behavior, which is typically based on the test suite of the main repository. However, this does not cover the problem we aim to solve with our technique: deciding which forked test cases to keep, add, or modify. Consequently, our technique is a complement to enable more reliable semantic merging and regression testing by updating the tests used by such techniques.

6 CONCLUSION

In this paper, we have introduced a technique for comparing forked test cases to provide developers with more support for comprehending the differences in these test cases during merges. We evaluated our technique in a qualitative analysis of 50 pull requests from five established open-source systems. The results indicate that our technique faces some limitations, particularly due to the test structures employed in some projects. Still, we could also show that our technique is a helpful means to support developers by highlighting what test cases are similar to what extent, which we plan to improve in the future. In this regard, we plan to tackle the technical limitations of our technique and conduct an actual user study with open-source developers to see whether our technique can help them during real-world merges.

REFERENCES

[1] Qurat U. Ain, Wasi H. Butt, Muhammad W. Anwar, Farooque Azam, and Bilal Maqbool. 2019. A Systematic Review on Code Clone Detection. *IEEE Access* 7 (2019), 86121–86144. https://doi.org/10.1109/access.2019.2918202

[2] Sofia Ananieva, Sandra Greiner, Timo Kehrer, Jacob Krüger, Thomas Kühn, Lukas Linsbauer, Sten Grüner, Anne Koziolek, Henrik Lönn, S. Ramesh, and Ralf Reussner. 2022. A Conceptual Model for Unifying Variability in Space and Time: Rationale, Validation, and Illustrative Applications. *Empirical Software Engineering* 27, 101 (2022), 1–53. https://doi.org/10.1007/s10664-021-10097-z

[3] Sofia Ananieva, Sandra Greiner, Thomas Kühn, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, Sebastian Krieter, Christoph Seidl, S. Ramesh, Ralf Reussner, and Bernhard Westfechtel. 2020. A Conceptual Model for Unifying Variability in Space and Time. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 15:1–12. https://doi.org/10.1145/3382025.3414955

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer. https://doi.org/10.1007/978-3-642-37521-7

[5] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 190–200. https://doi.org/10.1145/2025113.2025141

[6] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591. https://doi.org/10.1109/tse.2007.70725

[7] Jan Bosch (Ed.). 2014. *Continuous Software Engineering*. Springer. https://doi.org/10.1007/978-3-319-11283-1

[8] Emanuela G. Cartaxo, Patrícia D. L. Machado, and Francisco G. O. Neto. 2009. On the Use of a Similarity Function for Test Case Selection in the Context of Model-Based Testing. *Software Testing, Verification and Reliability* 21, 2 (2009), 75–100. https://doi.org/10.1002/stvr.413

[9] Peng-Hua Chu, Nien-Lin Hsueh, Hong-Hsiang Chen, and Chien-Hung Liu. 2011. A Test Case Refactoring Approach for Pattern-Based Software Development. *Software Quality Journal* 20, 1 (2011), 43–75. https://doi.org/10.1007/s11219-011-9143-x

[10] Harald Cichos and Thomas S. Heinze. 2011. Efficient Test Suite Reduction by Merging Pairs of Suitable Test Cases. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. Springer, 244–258. https://doi.org/10.1007/978-3-642-21210-9_24

[11] Emilio Cruciani, Breno Miranda, Roberto Verdecchia, and Antonia Bertolino. 2019. Scalable Approaches for Test Suite Reduction. In *International Conference on Software Engineering (ICSE)*. IEEE, 419–429. https://doi.org/10.1109/icse.2019.00055

[12] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Conference on Computer Supported Cooperative Work (CSCW)*. ACM, 1277–1286. https://doi.org/10.1145/2145204.2145396

[13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34. https://doi.org/10.1109/csmr.2013.13

[14] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. 2015. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *International Conference on Software Engineering (ICSE)*. IEEE, 358–368. https://doi.org/10.1109/icse.2015.55

[15] Eduardo M. Guerra and Clovis T. Fernandes. 2007. Refactoring Test Code Safely. In *International Conference on Software Engineering Advances (ICSEA)*. IEEE, 44:1–6. https://doi.org/10.1109/icsea.2007.57

[16] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. 2012. On-Demand Test Suite Reduction. In *International Conference on Software Engineering (ICSE)*. IEEE, 738–748. https://doi.org/10.1109/icse.2012.6227144

[17] Jacob Krüger, Mustafa Al-Hajjaji, Sandro Schulze, Gunter Saake, and Thomas Leich. 2018. Towards Automated Test Refactoring for Software Product Lines. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 143–148. https://doi.org/10.1145/3233027.3233040

[18] Jacob Krüger and Thorsten Berger. 2020. 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, 432–444. https://doi.org/10.1145/3368089.3409684

[19] Jacob Krüger, Wardah Mahmood, and Thorsten Berger. 2020. Promote-pl: A Round-Trip Engineering Process Model for Adopting and Evolving Product Lines. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 2:1–12. https://doi.org/10.1145/3382025.3414970

[20] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is My Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* 152 (2019), 239–253. https://doi.org/10.1016/j.jss.2019.01.057

[21] Elias Kuiter, Jacob Krüger, Sebastian Krieter, Thomas Leich, and Gunter Saake. 2018. Getting Rid of Clone-And-Own: Moving to a Software Product Line for Temperature Monitoring. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 189–189. https://doi.org/10.1145/3233027.3233050

[22] Remo Lachmann, Sascha Lity, Mustafa Al-Hajjaji, Franz Fürchtegott, and Ina Schaefer. 2016. Fine-Grained Test Case Prioritization for Integration Testing of Delta-Oriented Software Product Lines. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 1–10. https://doi.org/10.1145/3001867.3001868

[23] Yuejian Li and Nancy J. Wahl. 1999. An Overview of Regression Testing. *ACM SIGSOFT Software Engineering Notes* 24, 1 (1999), 69–73. https://doi.org/10.1145/308769.308790

[24] Panagiotis Louridas. 2006. Version Control. *IEEE Software* 23, 1 (2006), 104–107. https://doi.org/10.1109/ms.2006.32

[25] Wardah Mahmood, Moses Chagama, Thorsten Berger, and Regina Hebig. 2020. Causes of Merge Conflicts: A Case Study of ElasticSearch. In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 9:1–9. https://doi.org/10.1145/3377024.3377047

[26] Hung V. Nguyen, My H. Nguyen, Son C. Dang, Christian Kästner, and Tien N. Nguyen. 2015. Detecting Semantic Merge Conflicts with Variability-Aware Execution. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 926–929. https://doi.org/10.1145/2786805.2803208

[27] Harrie Passier, Lex Bijlsma, and Christoph Bockisch. 2016. Maintaining Unit Tests During Refactoring. In *International Conference on Principles and Practices of Programming on the Java Platform (PPPJ)*. ACM, 1–6. https://doi.org/10.1145/2972206.2972223

[28] Gregg Rothermel, Mary J. Harrold, Jeffery von Ronne, and Christie Hong. 2002. Empirical Studies of Test-Suite Reduction. *Software Testing, Verification and Reliability* 12, 4 (2002), 219–249. https://doi.org/10.1002/stvr.256

[29] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming* 74, 7 (2009), 470–495. https://doi.org/10.1016/j.scico.2009.02.007

[30] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating Test-Suite Reduction in Real Software Evolution. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 84–94. https://doi.org/10.1145/3213846.3213875

[31] G. Shobha, Ajay Rana, Vineet Kansal, and Sarvesh Tanwar. 2021. Code Clone Detection—A Systematic Review. In *International Conference on Emerging Technologies in Data Mining and Information Security (IEMIS)*. Springer, 645–655. https://doi.org/10.1007/978-981-33-4367-2_61

[32] Leuson D. Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and João Moisakis. 2020. Detecting Semantic Conflicts via Automated Behavior Change Detection. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 174–184. https://doi.org/10.1109/icsme46990.2020.00026

[33] Diomidis Spinellis. 2005. Version Control Systems. *IEEE Software* 22, 5 (2005), 108–109. https://doi.org/10.1109/ms.2005.140

[34] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 151–160. https://doi.org/10.1109/icsm.2015.7332461

[35] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 177–188. https://doi.org/10.1145/3336294.3336302

[36] Arie van Deursen, Leon M. F. Moonen, Alex van den Bergh, and Gerard Kok. 2001. *Refactoring Test Code*. Technical Report SEN-R0119. CWI.

[37] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying Features in Forks. In *International Conference on Software Engineering (ICSE)*. ACM, 106–116. https://doi.org/10.1145/3180155.3180205