Detecting Semantic Conflicts via Automated Behavior Change Detection

Leuson Da Silva[*], Paulo Borba[*], Wardah Mahmood[+], Thorsten Berger[+], and João Moisakis[*]

[*]Federal University of Pernambuco, Recife, Brazil
[+]Chalmers | University of Gothenburg, Gothenburg, Sweden

***Abstract*—Branching and merging are common practices in collaborative software development. They increase developer productivity by fostering teamwork, allowing developers to independently contribute to a software project. Despite such benefits, branching and merging comes at a cost—the need to merge software and to resolve merge conflicts, which often occur in practice. While modern merge techniques, such as 3-way or structured merge, can resolve many such conflicts automatically, they fail when the conflict arises not at the syntactic, but the semantic level. Detecting such conflicts requires understanding the behavior of the software, which is beyond the capabilities of most existing merge tools. As such, semantic conflicts can only be identified and fixed with significant effort and knowledge of the changes to be merged. While semantic merge tools have been proposed, they are usually heavyweight, based on static analysis, and need explicit specifications of program behavior. In this work, we take a different route and explore the automated creation of unit tests as partial specifications to detect unwanted behavior changes (conflicts) when merging software.**

We systematically explore the detection of semantic conflicts through unit-test generation. Relying on a ground-truth dataset of 38 software merge scenarios, which we extracted from GitHub, we manually analyzed them and investigated whether semantic conflicts exist. Next, we apply test-generation tools to study their detection rates. We propose improvements (code transformations) and study their effectiveness, as well as we qualitatively analyze the detection results and propose future improvements. For example, we analyze the generated test suites for false-negative cases to understand why the conflict was not detected. Our results evidence the feasibility of using test-case generation to detect semantic conflicts as a method that is versatile and requires only limited deployment effort in practice, as well as it does not require explicit behavior specifications.

***Index Terms*—Differential Testing, Semantic Conflicts, Behavior Change Detection**

I. Introduction

Branching and merging is a common practice in collaborative software development. It facilitates effective teamwork, allowing developers to independently contribute to the same project. Still, branching and merging comes with costs, including the need of resolving conflicts that are detected by merge tools when integrating code changes. Depending on project characteristics [1], [2], such *merge* conflicts might often occur [3], [4], [5], [6], [7], [8], [9], even when using more advanced merge tools [10], [11], [12], [13], [14], [15], [16] that explore language syntax and static semantics to avoid spurious conflicts.

While many merge conflicts are easy to fix, some of them can only be fixed with significant effort and knowledge of the code changes to be merged. This can negatively affect development productivity, and even compromise software quality in case developers incorrectly fix conflicts [17], [6], [18]. To avoid dealing with merge conflicts, developers sometimes even adopt risky practices, such as rushing to finish changes first [19], [17] and partial check-ins [20]. Similarly, partially motivated by the need to reduce merge conflicts, development teams have been adopting techniques such as trunk-based development [21], [22], [23] and feature toggles [24], [21], [25], [26].

Although these practices might reduce the occurrence of merge conflicts, there is no evidence that they are effective for resolving or even detecting so-called *test* [8] and *production* conflicts, which are only observed when running project tests and using the system in production. As such, they are more serious, because they reveal software failures. In fact, some of the practices mentioned above might even aggravate the costs of test and production conflicts, which are special kinds of what we here call *semantic* conflicts.[1] To make matters worse, we expect semantic conflicts to cost more than merge conflicts, as they are often harder to detect and resolve, and might end up negatively affecting users.

Resolving merge conflicts is often simpler, because it mostly involves reconciling incompatible independent *textual* changes in the same area of a file. Semantic conflicts are harder, especially when resolution occurs long after conflict introduction, because they involve reconciling *behavioral semantic* incompatibilities—as when the changes made by one developer affect a state element that is accessed by code contributed by another developer, who assumed a state invariant that no longer holds after merging. In such cases, textual integration was automatically performed generating a merged program, a build was created with success for this program, but its execution lead to unexpected behavior caused by unplanned *interference* between the developers changes—the behavior of the integrated changes does not preserve the intended behavior of the individual changes. Horwitz *et al* [27] have put this more formally: two contributions (sets of changes) to a base program *semantically conflict*—that is, interfere in an unplanned way—

[1]This relates two conflict terminologies; one based on the development phase in which a conflict is detected, and the other based on the language aspect that causes a conflict. We use *merge* conflict and *textual* conflict as synonyms. *Build* conflict refers to *syntactic* and *static semantic* conflicts. *Behavioral semantic* conflict refers to *test* and *production* conflicts (and undetected ones). For brevity, hereafter we omit the "behavioral" term in spite of focusing only on *behavioral semantic* conflicts in the paper.

when the specifications they are individually supposed to satisfy are not jointly satisfied by the program that integrates them.

To help reduce the branching and merging costs associated with semantic conflicts, we need merge tools that are able to detect such conflicts, going beyond textual line-based merge tools currently used in practice [28]. Previous work [27], [29] proposes semantic merge tools that rely on static analysis and model checking for detecting conflicts.

In this paper, we assess to what extent a semantic merge tool could *rely on unit test generation to reveal interference* between developers' changes (say commits in branches) that should be merged. The core idea we propose and assess is to use generated tests as partial specifications of the code revisions (say *L* and *R*, for left and right) resulting from the changes. The tests will then partially capture the effect of the changes on the behavior of *L* and *R*. Roughly, a test that passes on *L* and breaks on the base version *B* (the most recent common ancestor to *L* and *R*), partially reveals the intention of the change that lead to *L*. If that same test breaks in the merged version *M*, assuming the integration was carried on with a textual merge tool, we know that the changes that lead to *R* likely interfere with the changes that lead to *L*, making the test fail. This summarizes the criteria we use to detect that *R* interferes with *L*; we use similar criteria for detecting interference in the opposite direction. In fact, behavior that could be observed by running *L* can no longer be observed in *M*, which simply integrates to *L* the changes made by another developer.

To evaluate the potential of unit test generation to reveal interference, we apply widely recognized test generation tools (EvoSuite [30], [31] and Randoop [32]) to a sample of 38 merge scenarios—quadruples (B, L, R, M) formed by a merge commit, its parents, and a base—in which integrate changes to the same method or field declarations, as when two developers independently change the same methods and later integrate the changes. These scenarios come from open-source Java projects, and were either mined by our scripts or used in previous studies [14], [29], [33]. For each merge scenario, we invoke EvoSuite (the standard and the differential version) and Randoop, and check their effectiveness in detecting interference following our test based criteria; strictly checking semantic conflict would require access to the specifications of the changes or knowledge about the developers intentions. We also invoke EvoSuite and Randoop on slightly transformed versions of the original merge scenarios (see Sec. III-B1), to check whether the transformations we apply improve testability and the potential of unit tests in our context. This way, we are able to measure the effect of the code transformations we apply, and the contribution of each unit test generation tool. For the scenarios where the test generation tools fail to detect an existing interference, we analyze the causes of the failure. This sheds light on how the underlying unit test generation tools could be improved.

Our results show that, by first applying the transformations, the test generation tools can detect interference in only four out of 15 changes on same declarations(in three merge scenarios) that in fact suffer from interference between the integrated changes. Although this results in a small true positive rate, the generated tests lead to no false positives according to our interference criteria—the tools generated no test that satisfies our criteria for the merge scenarios in our sample that do not suffer from interference. This suggests that semantic merge tools based on unit test generation, as we propose, can help developers to detect semantic conflicts early, that would otherwise reach end users as failures. The associated benefits are likely achieved with small false-positive costs. However, with the current capacity of the test generation tools, developers cannot solely rely on such semantic merge tools for detecting semantic conflicts.

The transformations improved testability in two of the four detected interference cases, suggesting that they might be useful for interference detection. Differential EvoSuite detected interference in the four scenarios, while the standard EvoSuite succeeded in two scenarios, and Randoop in only one scenario. The identified improvements reflect three main problems highlighting the need for creating relevant objects required for the declarations holding the conflict, and relevant assertions exploring the *propagated* interference. For some false-negative cases, we identify and categorize improvements that could benefit unit test generation tools. Additionally, we provide our study sample as a dataset of merge scenarios with source code, working builds (which are necessary for running tests), and interference ground truth. This can be used to run new studies with less effort, and replicate ours.

II. Motivating Example and Background

To illustrate the notion of behavioral semantic conflict we explore in this paper, consider the example in Fig. 1. The illustrated class `Text` results from a merge that integrates the change in green (Line 6 was added, say from a revision *L*) with the change in red (Line 8 was added, say from a revision *R*). This example is inspired by a real merge commit from the project Jsoup.[2] The other code lines originate from a base revision *B*, that is, the most recent common ancestor of *L* and *R*.[3] As the method call in Line 7 separates the two changes to be integrated, there is no syntactic merge conflict in this case, and we cleanly obtain the syntactically valid class in the figure. We can then compile, build, and execute it.

The intention of the developer who created the revision *L*, say developer Green, was to extend the method `cleanText()`, which does some string cleaning through side effects, to also remove duplicated whitespace in the text by adding the method call `normalizeWhitespace()`, since the goal of her development task was to make sure that the resulting text had no such duplications. The intention of developer Red, who created revision *R*, was to clean the text by removing consecutive duplicated words as in the string "the␣the␣dog." Red, however, was not aware of the task allocated to Green and implemented the removal of duplicated words without eliminating whitespace between them, resulting in "the␣ ␣dog" for our string example.

This shows that, although Green's implementation is correct (it conforms to the implicit specification it is supposed to

[2]https://github.com/jhy/jsoup/commit/a44e18a

[3]For simplicity, we assume a single most recent common ancestor. With so-called criss-cross merge situations in git, there could be more than one.

```
  class Text {

    public String text;

    void cleanText() {
+     this.normalizeWhitespace();
      this.removeComments();
+     this.removeDuplicatedWords();
    }
  }
```

Fig. 1. A merge of two changes (each parent added one of the highlighted lines) that are semantically conflicting

```
1 class TextTestSuite {
2
3   public void test1() throws Throwable {
4     Text t = new Text();
5     t.text = "the␣the␣ ␣dog";
6     t.cleanText();
7     assertTrue(t.noDuplicateWhiteSpace());
8   }
9 }
```

Fig. 2. A test case that reveals the interference in Fig. 1

satisfy), the resulting method `cleanText()` we obtain after the merge does not fully eliminate duplicated whitespace from the text, which will certainly surprise Green. In fact, the implicit specification "resulting text has no duplicated whitespace" that is individually satisfied by revision *L* is not satisfied by revision *M* that we illustrate in Fig. 1. So, we say that *L's* and *R's* changes *semantically conflict*—or *interfere* in an unintended way—with respect to the base revision *B*. Red's implementation is also correct, assuming he was not required to eliminate whitespace between words, but ends up unexpectedly interfering with Green's implementation. Notably, we do not observe *interference* in the opposite direction, as the resulting merged code fully eliminates duplicated words—the implicit specification implemented by Red holds in revisions *R* and *M*.

As current merge tools are not able to detect such *semantic conflicts*, it is often difficult and expensive to detect and resolve them. In fact, unless a project adopts careful code review practices and has strong test suites, most semantic conflicts are expected to escape to users. Even with such careful and expensive practices for detection, semantic conflicts are expected to escape. If detection is not immediate after integration, it might be even harder to fix semantic conflicts, as resolution involves reconciling behavioral semantic incompatibilities. In our example, we would have to investigate whether the defect is in the individual implementations of Green and Red, or in how one of them interferes with the other. This would require a non-superficial investigation that breaks the abstraction boundaries established by the declarations of the methods called in `cleanText()`. It would not be enough to check the specification of `removeDuplicatedWords()`, but we would have to recognize that its implementation does not eliminate extra space after the deletion of a duplicated word.

To reduce this discussed difficulty and the costs associated with semantic conflict detection and resolution, it is important to investigate to what extent unit test generation tools could help to reveal the kind of interference we illustrate here. The core idea we propose and assess in this paper is to use *generated tests as partial specifications of the code revisions to be integrated*—tests then partially capture the effect of the changes in the revisions.

For instance, suppose a regression test generation tool (such as Randoop [32] or EvoSuite [30], [31] as we use in the remainder) generates the test in Fig. 2 when given the revision *L* as input. That test passes when executed against revision *L*, which leads to a call to the `normalizeWhitespace()` method when executing `cleanText()` in Line 6 of the test. With the input illustrated in `test1`, when reaching Line 6, `t.text` stores the test input string in Line 5, except for the extra space character right before `dog`. Consequently, the assert successfully evaluates. The same test breaks when executed against revision *B*, since this revision involves no call to `normalizeWhitespace()`, and so the assert throws an exception. For this reason, breaking in *B* and passing in *L*, we say that `test1` partially reveals the intention of the change that leads from the first to the second. We can see `test1` as a partial specification of the changes in *L*.

Now note that `test1` fails when executed against revision *M* in Fig. 1. The `normalizeWhitespace()` ends up being called when executing `cleanText()`, but the call to `removeDuplicatedWords()` leads to a new duplicated whitespace in the text, as explained before. This way, `test1`, the partial specification of *L*, is not satisfied in the merged version, revealing that the changes in *R* interfere with *L* (with respect to *B*). If we could find a test that fails in *B*, passes in *R*, and fails again in *M*, we would similarly say that *L* interferes with *R*. This is essentially the criteria we apply for automatically detecting *interference* by generating and executing tests in the rest of the paper. Making sure that *interference* actually leads to a *semantic conflict* cannot be automatically checked in general because it involves understanding developers' intentions or proving that implementations satisfy specifications (in this case, specifications of the changes, which are hardly available in public repositories).

A. Background

Unit testing generation tools such as Randoop [34] and EvoSuite [35] generate tests that consist of a sequence of method and constructor calls followed by assert statements. The calls create, initialize, and exercise objects, playing the role of test setup and test actions. The asserts verify the test's expected results. Randoop generates such a sequence of calls by randomly selecting the methods and constructors in the class under test. The arguments for such operations are also randomly selected from a pool of values of primitive types, and of objects previously created in the sequence. For optimizing the process, it incrementally executes the sequences being created to make sure that it is worth further extending them. EvoSuite starts from randomly generated sequences as in Randoop, but relies on genetic algorithms to evolve the sequences with the aim of optimizing a specific goal, such as higher code coverage. With respect to the generation of assertions, both tools adopt similar approaches that explore the values returned by executing the method sequences. EvoSuite can further calculate a reduced set of the generated assertions, whereas Randoop can also generate assertions that check basic and general contracts. A contract expresses invariant properties that hold both at entry and exit from a call; it checks whether the resulting call values conform with its specification.

III. Methodology

To evaluate the potential of unit test generation to reveal interference, our methodology comprises five main steps. First, we extract and select merge scenarios from Java projects hosted on GitHub, including a number of scenarios that appear in previous integration conflict studies. Second, for each selected scenario, we create builds for eight software versions: base, left, right, merge, and four more derived from these by applying testability transformations we conceived (explained shortly in Sec. III-B1). Third, we apply three test generation tools to create tests for four of these releases (left, right, and their testability-transformed versions) and run our scripts for executing the tests, discarding invalid tests, and avoiding flakiness issues. Fourth, as a last automated step, we run our scripts for checking the test-based interference criteria and detecting interference. Fifth, we manually analyze each merge scenario and the generated results, further investigating why the generated tests are not able to detect interference in some of the scenarios that suffer from interference.

A. Mining and Selecting Merge Scenarios

Our merge scenario sample consists of two parts. The first part contains merge scenarios mined from a number of Java projects hosted on GitHub. We opt for Java projects only because the unit test generation tools are language-dependent, and some of our scripts are tool-dependent; the tools we use in our study primarily generate test cases for Java. Most related studies also focus on Java projects. We also limit our study to GitHub projects as it is one of the most popular sources of open-source projects, and most related studies also use GitHub.

For this first part, we start with a list of projects from a previous study [36], [37] that focuses on Maven projects, which could help in the build creation process of our second step. We then arbitrarily selected a subsample of projects, available in our online Appendix, and used our merge mining framework [38] to select merge scenarios that integrate changes in the same method body, constructor or field initialization, as we thought this would increase the chances of collecting interference situations. We mined only recent project histories, for reducing build creation effort in the next step.[4]

The second part of our merge scenario sample contains Java merge scenarios from previous related studies [29], [14], [33]. From the first, we selected eight scenarios that integrate independent changes to the same declaration, six with interference and two without. One of the last two scenarios integrates independent changes to two declarations, so we count them as two potential interference cases in one merge scenario. We tried to add five more scenarios with interference (they have 11 in total) to our sample, but we were not able to build them due to missing old dependencies or unavailable required resources. From the second study, we selected 11 Java merge scenarios: three with interference (all they had) and eight without. We also tried to add more cases without interference but due to missing old dependencies, we could not build them. In a few scenarios we did not observe changes to the same declaration, so we discarded them. Finally, from the last work, we included 15 scenarios that suffer from interference, five that do not. At this point, we had a sample of 43 merge scenarios.

[4]We adopted a limit date for merge scenarios not older than five years.

B. Applying Code Transformations and Building the Projects

For each scenario selected in the previous step, we apply the following code transformations to each version in a merge scenario (base, left, right, merge). Thereafter, we build the project of each scenario to run the test-generation tools.

1) Testability Transformations: The application of the testability transformations was motivated by preliminary experiments we performed by using the unit test generation tools with toy examples and a small subsample of the scenarios we consider here. As guidelines, we used four scenarios of our sample. In these experiments, we analyzed the method declarations independently changed by two developers, and observed, in a few cases, that the interference was *propagated* through a class *field*. However, the interference could not be detected by the generated tests because they would not have direct access to the involved private fields. The tests also did not invoke additional methods that had access to such attributes. So interference could be *reached*, *infection* could actually occur, but *propagation* could be hardly observed. For being able to more easily detect interference in such situations, we applied source code transformations to increase testability, with possible impact on correctness, as assessed in the last step of our study. In this case, the transformations replace non public access modifiers with public access ones, for the class that contains the method or field declaration that was independently changed by two developers. We apply this transformation for class fields, methods, and constructors.

During our initial experiments, we also realized that the unit test generation tools could not even *reach* the interference locations because they were not able to create objects of the class that should be exercised to reveal the interference, or of classes that appeared as parameters of methods in the generated test. Aiming to address this issue, we applied a transformation to add an empty constructor to the class under analysis, in case a non-empty constructor was not available. An empty constructor is relevant when the tools are unable to create complex objects; objects having internal and external dependencies. Furthermore, for scenarios where the independently changed declarations occur inside inner classes, we extracted them to the outer level, as the test generation tools were not able to exercise inner classes.

Such transformations could be transparently applied by a semantic merge tool based on the ideas we describe here. We expect no major negative implications for users of the tool.

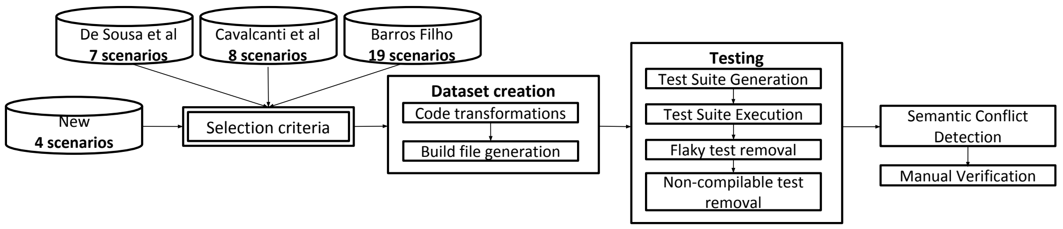

Fig. 3. Study setup. Starting with the selection of Java project merge scenarios, we generate our sample dataset, call the unit test generation tools, and execute the generated test suites to detect semantic conflicts. Besides that, we also perform a manual analysis to verify false positives and negatives in our sample.

Applying the transformations is computationally not expensive compared to the costs of generating and executing tests.

2) Building the Projects: Considering we need to execute build files with all dependencies defined for a project, we decide to create these builds on Travis. So our infrastructure requests Travis to create each build for the eight software versions. When Travis fails to create the builds for lack of dependencies, support for older Java versions, or additional analysis adopted by projects, like style checking, we try to manually fix the problem on Travis updating the configuration files; otherwise we locally create the builds. We also adopt the manual build creation process for Ant projects, as our infrastructure supports only Maven and Gradle projects. At this point, if we fail to create one of the eight builds for a scenario, we simply discard the scenario—in our experiment, we discarded five scenarios. The process and infrastructure we use to create the builds appear in our online Appendix [38].

At the end of this step, we were left with a sample composed of 38 *merge scenarios* and 40 *potential interference cases*, as two merge scenarios contain more than one independent changes on same declaration.

C. Generating and Executing Tests

Each merge scenario resulting from the previous step has proper builds that can be executed and exercised by tests. Such builds are required by unit test generation tools that work by generating tests and running them against the system to be tested, discarding tests that fail or do not increase code coverage. This is the case for the test generation tools we evaluate: EvoSuite, Differential EvoSuite, and Randoop [30], [31], [32]. We also chose these tools for their robustness and popularity.

In this step, we readily apply the three unit test generation tools to create tests for four of the versions (left, right, and their transformed versions, as explained above) associated with each merge scenario. This way, the tools try to generate tests that pass in these versions, which well fits with our interference criteria: test passes in one of the merge commit parents and breaks in the base and merge versions. For each of the four versions, our scripts simply call EvoSuite and Randoop with the version as input. For Differential EvoSuite, which tries to generate tests that reveal behavior differences between two versions, we additionally give the base commit as input, which is used as the regression version. So, the tool will try to generate a test that passes in the parent commit and fails in the base commit. For controlling the randomness of the test generation process,[5] our scripts repeat this three times, obtaining 12 test suites from each tool, three for each version in a merge scenario.So for each merge scenario, 36 test suites are generated.

For each resulting test suite, our scripts execute the contained test cases three times, each time against one of the different versions: base, associated parent,[6] and merge, resulting in nine executions. Finally, for each merge scenario, the generated 36 test suites are executed nine times resulting in 324 executions. These executions are repeated with the aim of detecting test flakiness. However, we did not observe any such case in our experiment. If a test case does not yield the same result (pass or break) in the three repeated runs, we simply filter it out, as not doing that could compromise the accuracy of our interference detection criteria. For each of the 324 test suite executions, we group the test results into three sets: tests with failed status, tests with passed status, and tests that could not be executed because they do not even compile with the version under test. Such validity issues with tests might occur because the test was generated for a given revision, say left, but is executed in other revisions as well: base and merge. If the left revision, for example, adds a method declaration that is called in the generated test, this test will not even compile with the base version. Such invalid tests are also discarded as a last action in this step.

D. Detecting Interference

For each scenario, we group the 324 test suite executions from the previous step into four sets of 81 executions. Each set contains the executions associated with the base, parent (left or right), and merge commits for the original and transformed versions. Next, for each such set, our scripts compute the test cases that present the same result when executed on the base and merge commits, but different result on the parent commit (left or right).

Finally, our scripts collect the results for further analysis, and report interference if our criteria are satisfied. In this case, for a set of executions, a test must fail on base and merge, but pass on a parent commit. The same criteria are applied for cases in which a test case passes on base and merge commits, but fails on the parent commit.

[5]The tools can behave non-deterministically, even when called with fixed seeds, as the tests that they generate and execute might suffer from flakiness.

[6]Remember, each generated suite is associated with a merge parent.

E. Analyzing the Scenarios and Establishing the Ground Truth

With the results reported by our scripts, we manually analyzed each merge scenario to establish a ground truth of actual interferences in order to contrast it with the obtained results. In particular, we collected information on false positives (our interference criteria hold but there is actually no interference in the scenario) and false negatives (our interference criteria does not hold but the scenario actually suffers from interference). This also helps to understand the potential of unit test generation and of our criteria to detect interference.

Two authors manually analyze each merge scenario to check for interference. One of them performed an initial analysis, presented to the second, jointly discussed the case, and reached a verdict. To reduce the chances of human error and misjudgement in this process, for each interference verdict, we manually design a test that could reveal the problem, especially when the tools are not able to find one. Similarly, each non-interference verdict has an explanation of why we could not design such a test case; for example, one of the changes is a structural refactoring, not affecting the behavior of the other integrated changes.

For many of the cases (33), the ground truth is available in previous work, but we nevertheless follow the process above and compared verdicts. For all cases, we summarize the integrated changes to help reach verdicts and using our dataset for replications and further studies. For the merge scenarios reported with interference, we analyze the associated test suites to ensure that the tests explore the conflict that we found during our manual analysis. This analysis is essential, since the testability transformations could introduce false positives to our results, as some semantically change the program behavior. For that, we check whether the failed test case assertions are exploring the side effects of the elements involved in each conflict.

Aiming to understand the limitations that unit test generation tools face—in our context of exploiting the generated test cases to detect interference—we analyze the test suites of the identified false negatives. Based on the test descriptions we wrote during our ground truth analysis, we try to change the unsuccessful generated test cases and check if they could then detect interference. As a result, we identify improvements that could be applied to the tools, as well as to better understand and help assess how close the tools are to generating a test case that would reveal interference.

At the end of this step, we obtain a dataset composed of merge scenarios associated with their build files, both with and without testability transformations, generated test suites, interference ground truth, and further information on the performance of each test generation tool.

IV. Results

We now present the results of our analysis of the 40 parallel changes to the same method and field declarations in the 38 merge scenarios mined from GitHub Java projects (see Sec. III-A), including how semantic conflicts were detected through our criteria and automatically generated tests. We also discuss how the test generation tools could be improved to increase conflict detection accuracy. Our focus is first on the cases with conflicts. Later we discuss the cases with no conflicts, concluding with suggestions for improvement.

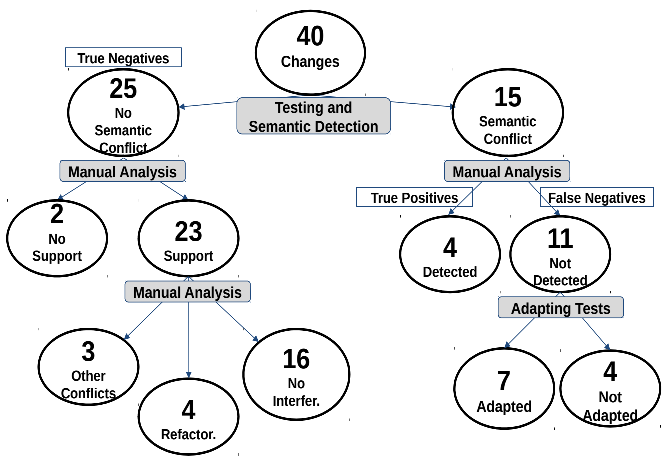

Fig. 4. Results Overview: distribution of changes (on same source code declaration), their classification, and whether conflict is detected or not. Note: *Refactor.* abbreviates *Refactoring*, *No Interfer.* abbreviates *No Interference*.

A. Cases With Conflicts

Recall that, to verify the potential of regression testing for detecting semantic conflicts, we select 40 changes with changes on the same declarations in the source code from 38 merge scenarios from GitHub Java projects, apply code transformations to increase their testability, call the test-generation tools to generate test suites, and finally, execute, filter, and remove invalid test cases from our analysis.

Figure 4 illustrates our results; the right-hand branch gathers the changes with semantic conflicts, while the left branch represents the changes without conflicts. In total, we could automatically detect four conflicts (in three merge scenarios) in the 40 changes (10%).

Analyzing the right branch in Fig. 4 shows that in fact 15 changes have semantic conflicts based on our manual analysis. Note that, even when the scenarios stemmed from another study, we double-checked it, since other works might have a slightly different understanding of semantic conflicts.

Figure 5 shows a change extracted from a merge scenario of the project Storm with a semantic conflict detected by Differential EvoSuite.[7] In this merge scenario, the *L* commit removed the reference for `maxRetries` of the local variable returned by the method `toString()` (Line 4 in Fig. 5), while the *R* commit added a new reference for `subscription`

[7]Merge commit from project Storm: ad2be67

```
1   public String toString() {
2     return "pollTimeoutMs=" + pollMS +
3       ",offsetCommitPeriodMs=" + offsetMS +
4 -     ",maxRetries=" + maxRetries +
5       ",maxUncommittedOffsets=" + maxSet +
6       ",firstPollStrategy=" + firstPool +
7 +     ",subscription=" + subscription +
8       ",retryService=" + retryService;
9   }
```

Fig. 5. Semantic conflict caused by changes in the same variable.

```
 1  @Test
 2  public void test1() throws Throwable {
 3    KafkaSpoutConfig kafka0 = new KafkaSpoutConfig();
 4    String string0 = kafka0.toString();
 5    assertEquals(
 6    "pollTimeoutMs=1,"+"offsetCommitPeriodMs=0,"+
 7    "maxRetries=0,"+"maxUncommittedOffsets=1,"+
 8    "firstPollStrategy=true,"+"subscription=2,"+
 9    "retryService=false", string0);
10  }
```

Fig. 6. Test case generated by Differential EvoSuite detecting semantic conflict.

(Line 7 in Fig. 5). So, *L* and *R* individually change the program behavior (string construction) based on their own needs. These changes can be integrated without reporting *merge* or *build* conflicts (Lines 5 and 6 separate the changes in Fig. 5). Unless there is a test case available to verify this method's behavior, this semantic conflict would not be identified. In Fig. 6, we show the test case generated by Differential EvoSuite using the *R* commit. As highlighted in the test case (Lines 7 and 8 in Fig. 6), the expected object should contain references for `maxRetries` and `subscription`. The test case passes on the *R* commit as expected. For the *B* (base) commit, the test fails as the received object does not have a reference for `subscription`. In the same way, in the *M* (merge) commit, the test case also fails as the received object does not contain a reference for `maxRetries`.

The other three detected conflicts have some common aspects with the example we discussed. First, the conflicts occur because the parent commits change the same object. So, to detect the conflict, the tools should focus on one specific object. Second, the tools could directly access the object involved in the conflict (side effects). Thus, the test cases should have at least one assertion exploring this object's contents.

Differential EvoSuite was the most efficient tool in our study, detecting four conflicts, while EvoSuite and Randoop detects two and one conflicts, respectively. Although EvoSuite did not detect the four conflicts, this tool analyzed and generated tests for all four cases; Randoop generated test suites for three of them. Analyzing the generated test suites, we verified that the test cases adequately create objects and call the elements that hold the side effects leading to the conflict. Nonetheless, their assertions were irrelevant to detect the conflict.

For these conflicts, applying code transformations was a good starting point to the conflict detection, as these transformations increase the testability of the source code under analysis. For example, after applying testability transformations, the tools could directly access the objects and, consequently, explore these objects in their assertions. This led to the detection of two additional conflicts not detected by the same tools applied to the same sample without the transformations.

Moving to the remaining 11 cases that represent false negatives, we focused on understanding the limitations behind the missed detection. To this end, we analyzed the generated test suites and verified if after applying a few changes to the test cases, the conflicts could be detected. To guide us during this adaptation process, we considered the test descriptions that we previously created during our initial manual analysis (see Sec. III). For seven out of 11 changes on the same declarations, the changed test cases could detect the conflict. To illustrate it better, we must have a look at the changes in this case. Both parent commits change the same `array` that is returned by the changed method; while the *L* commit adds a new entry to the array, *R* removes an old one. In these circumstances, the original test case assertions are expected to explore the contents of the object returned by the method. However, only one assertion checked it, verifying whether the object was null. Adding nothing more than an assertion exploring the object size was enough to detect the conflict. It shows that even for false negatives of unit test generation tools, the generated test suites could *reach* the location of the *infection*, but failed to explore the *propagated* interference. As such, the tools were close to detect interference in this case. In other cases, the tools were not even close. The methods holding the conflict were called, but with arguments that prevent the test from *reaching* the interference location; for example, with `null` argument values that lead to exceptions before reaching the location. As a consequence, irrelevant assertions were generated, which could not detect the interference even if it was *propagated*. However, in some cases, slightly adapting the assertion would be enough to detect the interference. Below in this section, we discuss improvements for unit test generation tools that could improve efficiency.

B. *Cases Without Conflicts*

Focusing now on the left branch in Fig. 4, we discuss the 25 changes without conflicts. They are all true negatives; in our experiment, using the tools to detect interference leads to no false positives. Initially, we classified these scenarios based on their support for generating and executing tests. Two cases were classified as *not supported*, since the changes to be merged occur in test classes. The unit test generation tools could not generate test cases for them, as the associated test framework and project environment were not provided to run the analysis. Even if we could provide the requested environment, based on our manual analysis and interference criteria, we would still classify the changes *without semantic conflict*. The changes involve a refactoring and semantic changes, but are not interfering with each other.

Moving to the 23 changes (to the same declarations in source code) that are supported by the unit test generation tools, we expected no test case to detect a semantic conflict. So even if the test suites exercised the parent commit changes, no conflict should be reported in the local context. For 16 changes, the parent commits individually change the program behavior, but when integrated, they do not interfere with each other at least in the local context. It might be possible that these changes globally interfere. For example, two developers may, in the same method declaration, add assignments to different fields of the same object. So, locally, the changes do not interfere with each other. But, if we consider the surrounding program context, say a method `computeRate()` that calls the changed method and uses the two fields as part of an `if` condition, we could have interference and conflict. Unit tests that exercise the context classes could still detect this interference by invoking `computeRate()`, but not by focusing only on the class that declares the changed declaration, as we do in our experiment.

Next, we observed four changes resulting from structural refactorings. As we explained above, in such circumstances, even if one parent commit changes the program behavior, we expected no semantic conflict. Finally, for the remaining three changes (to the same declarations in the source code), the parent commit changes cause other kinds of conflicts during the integration (textual or build conflicts). Our goal in this study is to analyze the interference among contributions, so it is essential to ensure the class files that hold the semantic conflicts have only the changes performed by the parent commits. When merge or build conflicts occur, human intervention would be necessary to fix these conflicts, and then it would be more difficult to isolate the parent commit changes and perform our analysis. Even with this risk, the tools generated test suites but no conflict was detected.

C. Improvements for Unit Test Generation

We observed the following limitations and weaknesses of the generated unit-tests in our specific context. For instance, we observed a limitation to create complex objects with internal or external dependencies. As presented in Sec. III, we tried to address some of these limitations applying testability transformations in the code under analysis. However, this rather increased code testability, not the quality of the generated test suites. Based on this perspective, we manually analyzed some generated test suites of false-negative cases to understand the limitations of these suites better.

Recall that, during our manual analysis, we documented the conflicts in our scenarios in detail. Based on these descriptions, we tried to apply changes to the generated test suites of false negatives and verify whether the test suites could detect the conflicts with the minimum possible number of changes. Interestingly, the improvements we propose here are not fully exclusive to detect semantic conflicts, but can help to improve unit test generation tools in general.

Relevant object creation is required to reach the interference location. The unit test generation tools face some problems regarding the creation of objects that should be used directly or indirectly by the test cases. Many test cases finish their executions due to failing attempts to access fields or calling a method from a received parameter, which was not well-created. As an example, consider the case of project Jsoup.[8] The attempt to access fields of the object `textNode0` throws a `NullPointerException`, since the object was not well-created. Giving relevant objects for test cases like this is necessary to the test case reaches at least the *interference* location. Based on this, we extended this test case and the conflict was detected. This topic is also related to cases that *methods holding the conflict are not adequately called.* In these cases, besides the creation of relevant objects, a sequence of method calls must be done aiming to assign values to objects, that are required to reach the *interference* location.

[8]This case refers to merge commit a44e18a in project Jsoup.

Relevant assertions must explore the propagated interference. Considering the hard work required to reach the location where the conflict occurs, we expected that the assertions could better explore the *propagated* interference. For example, as we mentioned above, one conflict of project CloudSlang,[9] the parent commits change the same `array`. The test case generated for this scenario correctly creates the object, calls the method, where the conflict occurs, and saves the method return object into a local variable. However, the generated assertion checked whether the local variable was `null` instead of exploring the object size. So, depending on the type of a method return object, the unit test generation tools could explore defined aspects that could detect the conflict. For example, for array objects, assertions should explore their size and contents.

Relevant assertions rely on the way an interference is propagated. Test case assertions often use the object returned by a method, but not objects that are passed as parameters. For example, again analyzing the changes performed of previous mentioned merge scenario of project Jsoup, the method `outerHtmlHead` requires three parameters as input, not returning any object (void method). However, a semantic conflict occurs and the first parameter holds the *propagated* interference. The test case generated by EvoSuite focusses on verifying whether an exception is thrown during its execution. The assertions should not be restricted to explore objects returned by a method, but also other objects that were used by a method or any other way of communication.

On a final note, the weaknesses of the tools we observed may be motivated by the diverse sample of real projects we adopt here. Previous work assessing Randoop [39], for example, focuses mostly on generating tests for APIs. For EvoSuite [40], previous work considers other kinds of projects, but many of them come from the same owner. In other work [41], the evaluation did not focus on detecting behavior changes.

V. Discussion

The occurrence of conflicts negatively impacts team productivity and software quality. Depending on the kind of conflicts, like merge conflicts, there are novel techniques to support the resolution of them or even avoid that developers spend time fixing them. These new tools, even with new ways to handle merge conflicts, do not support behavioral semantic conflict detection or resolution. While no approach is available to detect these conflicts, they will continue to occur and, at some point, show up to the team or even the final user. The later these conflicts are observed, the harder it will be to fix them.

First, our results, in this study sample, do not report any false positives. So if a semantic merge tool based on regression testing generation is available for a development team, we expect developers would most of the time be warned about real conflicts—the team productivity would not be affected for no reason. In the worst scenario, the tool would never warn about any conflict, which is the same scenario of no tool supporting semantic conflict detection.

[9]This case refers to merge commit 20bac30 in project CloudSlang.

One may argue the testability transformations could introduce false positives in our results—a concern we are aware of, but analyzing the reported detected conflicts, we observe they are true positives. The transformations contribute to increasing the testability of the code under analysis without major drawbacks.

Another novelty about using regression testing is on how the semantic conflict is detected. Other approaches, as static analysis, could inform that a prominent conflict was caused by the data flow involving two variables. Despite the chances this merge scenario could represent a false positive, the developer should stop her/his work and spend some time analyzing it until a decision could be made. In case a conflict is detected, it would also be necessary to define how the conflict could externally be observed. Adopting regression testing may decrease the effort to understand how a conflict occurs. The test case limits the amount of source code that should be analyzed and changed to fix the conflict. Second, the test cases can be used to observe the external final state of a program in all commits of the merge scenario, so the behavior changes individually for each commit. Regression testing could also be applied during the conflict fix process. The developer could use these test cases to verify whether the semantic conflict is observed while changes are applied to fix the conflict.

In practice, semantic merge tools based on regression testing, as we propose here, can help developers detect semantic conflicts. Due to the observed low number of false positives, the benefits can be obtained by avoiding major costs on wasted developer effort. However, due to the significant number of observed false negatives, developers should not exclusively rely on our proposal of semantic merge tool to detect semantic conflicts. They should still try to catch such conflicts by reviewing the code review and executing project tests.

VI. Threats to Validity

Construct Validity. As explained above, we cannot assess semantic conflict occurrence without having access to the developers intentions or specification of the changes they make. So our study focuses on interference occurrence. As manually assessing global interference, and generating and running tests for the whole system, would demand considerable effort, our study is restricted to local interference occurrence. So it is possible that our sample has merge scenarios that parent commit changes do not interfere with each other locally, but interfere globally. The opposite can also occur. So the number of false negatives and false positives with respect to a global notion of interference could be different than our results report. Nonetheless, regression tests could detect global interference if the interference is propagated, and if we generate tests for other classes in addition to the one that integrates the parallel changes made by two developers.

Aiming to increase the testability of the source code under analysis for the unit test generation tools, we decided to apply testability transformations before performing our analysis. For example, we change access modifiers to `public`. This transformation does not semantically change a program; it only makes some elements reachable for the test cases. If a semantic conflict can be observed accessing a class field, but this field is `private`, the unit test generation tools would face many problems trying to indirectly access this attribute without the transformation. Some may argue that, without this transformation, such conflict could never be observed. That might be true if indirect access, for instance with accessor methods, is not available. However, this is not a problem here since our ground truth and analysis focuses on locally observable interference, not globally observable interference.

Internal Validity. When creating the interference ground truth, knowing the results of the test generation tools before the manual analysis could have influenced the verdict. For instance, knowing that the tools were not successful for a given scenario brings the risk of precluding a more in-depth analysis from our side. To reduce this threat, we involved two authors in the analysis, and demanded they provide an explanation of why there is no interference; this often requires understanding the changes in detail to detect refactorings, changed state elements, and how they impact each other. The risk is significantly reduced for the cases in which the tools were successful, as the threat can be minimized by analyzing the interference revealing threat, running it, and manually checking whether the test case assertions focus on the changed state elements.

As we discuss when presenting our results, we remove from our analysis test cases that are not compilable on at least one commit of the triple of commits in the merge scenario.

External Validity. Our results are specific to the context of open-source GitHub Java projects. The code transformations, as we discussed, positively impact our results and contribute to increasing the source code testability; in some cases, also detecting the conflict. Applying our proposal of semantic merge tool to other programming languages would require test generation tools for the desired language and also the code transformations, if applicable.

VII. Related work

Regression testing has been used for detecting behavior changes in the past. Evans and Savoia [42] combine regression testing with progression testing to detect preserved, altered and eliminated behavior of a program. They evaluate a parser written in Java showing significantly better detection rates than regression testing alone. Jin *et al.* [43] leverage change analyzers to generate test cases for changed parts of a software program. Shamshiri *et al.* [44] present EvosuiteR, a test generation tool for differential testing that uses search-based algorithms to find regression faults on different versions of a program. While the previous studies evaluate the detection of regression faults between two different versions of a program using regression tests, in this work, we evaluate the potential of regression tests to detect semantic conflicts on merge scenarios (three different versions of a program). We also consider in our evaluation a sample of diverse real Java projects, while the previous studies only consider small or toy projects.

Researchers have also investigated ways in which conflicts can be detected and prevented early, thereby minimizing their impact on productivity. Sarma *et al.* [17] present Palantir, a workspace awareness tool, which aims to minimize the occurrence of conflicts by notifying the developers of parallel changes in the same artifact. Brun *et al.* [8] propose incorporating speculative analysis for early detection and prevention of conflicts. To detect test conflicts, they analyze three Java projects and rely on project tests, which are often not enough for detecting interference as we explore here. They locally build the merge commits, and if the build process fails because of failed tests, they consider the merge scenario having a semantic conflict. The failed tests are not executed on the parent and base commits of the merge scenario, as we do here, which may result in false positives, as the failed test may occur due to the changes exclusively performed by one parent.

Cavalcanti *et al.* [12], [15], [14] conduct empirical studies where they analyze merge scenarios and compare the accuracy of different merge resolution techniques: unstructured, semistructured, and structured merge. They also propose a new, semistructured tool with significant advantages over unstructured merge tools by reducing the false-positive and false-negative rates of earlier semi-structured tools. When comparing with structured merge, the authors verify semistructured merge reports more false positives, but presents less false negatives. Overall, they find that exploring more structure does not necessarily improve merge accuracy. Contrasting with our investigation here, their proposed tools are not able to detect behavioral semantic conflicts, only syntactic and static semantics conflicts.

Nguyen *et al* [45] present Semex, a tool for detecting which combination of merged changes causes a test conflict based on a technique called variability-aware execution [46]. First, the tool separates the changes done by each parent commit in the merge scenario and encodes each one using conditionals around them (`if` statements) to integrate all these changes in a single program. Semex then uses variability-aware execution to detect semantic conflicts by running existing project tests, if available, on this single program, exploring all possible combinations of the encoded changes. The tool then knows which combinations of commits lead to test failure and reports the set of commits that, if integrated, would cause a test conflict. Reporting a conflict exclusively based on the failure of a test in the merged code does not always imply a conflict or interference. If the test fails in one of the parent commits too, failure in the merge might simply indicate inheritance of a defect. If the test passes in both base and a parent commit, failing in merged code might simply be caused by a non-interfering behavior change from another parent commit. That is why we propose different criteria, based on the idea of tests as partial specifications of the changes to be integrated. We also rely on and assess the use of test generation tools to detect conflicts, instead of relying on existing project tests, which are often missing or have limitations, as described above. Finally, Semex is preliminarily evaluated using PHP toy projects, with manually created test cases especially aimed at revealing conflicts, as the main focus is on assessing the potential of variability-aware execution for identifying conflicting branch combinations, not really the potential for conflict detection, as we do here. Moreover, in our study we use non trivial Java open-source projects.

Sousa *et al.* [29] propose *SafeMerge*, a tool that leverages compositional verification to check *semantic conflict freedom* in merge scenarios. In principle, this kind of static analysis should lead to more false positives and fewer false negatives, when compared to the use of tests as we propose here. An evaluation with 52 merge scenarios indicates that *SafeMerge* reports 75% of the scenarios without conflicts, with a false positive rate of 15%. However, analyzing the merge scenarios reported with conflicts, we concluded that some of them do not represent conflicts according to our criteria. In these cases, the changes involved do not interfere with each other or are only refactorings, leading to no behavior change and consequently no interference.

VIII. Conclusion

Branching and merging are common practices in collaborative software development. Despite increasing the productivity of development teams, they come with substantial costs when developers integrate their changes and conflicts arise. Modern merge tools alleviate many of the conflicts occurring in practice, they are largely limited to syntactic conflicts, while semantic conflicts are much harder to detect, with potentially much more serious consequence on the software behavior at runtime.

We studied the detection of semantic conflicts using automated test-case generation techniques. As opposed to prior attempts, this strategy requires not much setup effort and does not need explicitly defined behavior specifications. We systematically investigate our interference criteria upon a manually curated ground-truth dataset originating from real merge scenarios mined from GitHub. We combine unit test generation tools with testability transformation in the source code to be analyzed.

We show the feasibility of a semantic merge tool based on regression test generation. While it was only able to detect four conflicts out of 40 changes on same declarations, we did not face any false positives according to our interference criteria. This suggests that semantic merge tools based on regression test generation would help developers detect semantic conflicts early, otherwise reaching end-users as failures. The transformations improved testability in two of the four detected interference cases, suggesting that they might be useful for interference detection. We discuss necessary improvements to test generation and make our manually curated dataset available in an online appendix [38], for replication and future technique building upon our proposal of semantic merge tool.

Acknowledgements

We thank Marcelo d'Amorim, Rohit Gheyi, Leonardo Fernandes and the anonymous reviewers for valuable comments to improve an earlier version of this paper. We thank Rafael Alves, Galileu Santos, Matheus Barbosa and Vinícius dos Santos for the support when creating our dataset. We also thank INES (National Software Engineering Institute), the Brazilian research funding agencies CNPq (309741/2013-0), FACEPE (IBPG-0692-1.03/17 and APQ/0388-1.03/14), and CAPES, as well as the Swedish Research Council (257822902), Vinnova Sweden (2016-02804) and the Wallenberg Academy.

REFERENCES

[1] M. Owhadi-Kareshk, S. Nadi, and J. Rubin, "Predicting merge conflicts in collaborative software development," *International Symposium on Empirical Software Engineering and Measurement*, 2019.

[2] K. Dias, P. Borba, and M. Barreto, "Understanding predictive factors for merge conflicts," *Information and Software Technology*, vol. 121, p. 106256, 2020.

[3] D. E. Perry, H. P. Siy, and L. G. Votta, "Parallel changes in large-scale software development: an observational case stud," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 3, pp. 308–337, 2001.

[4] T. Mens, "A state-of-the-art survey on software merging," *IEEE transactions on software engineering*, vol. 28, no. 5, pp. 449–462, 2002.

[5] T. Zimmermann, "Mining workspace updates in cvs," in *International Conference on Mining Software Repositories*. IEEE, 2007.

[6] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Symposium on the Foundations of Software Engineering*. ACM, 2012.

[7] B. K. Kasi and A. Sarma, "Cassandra: proactive conflict minimization through optimized task scheduling," in *International Conference on Software Engineering*. IEEE, 2013.

[8] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early detection of collaboration conflicts and risks," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1358–1375, 2013.

[9] W. Mahmood, M. Chagama, T. Berger, and R. Hebig, "Causes of merge conflicts: A case study of elasticsearch," in *International Working Conference on Variability Modelling of Software-intensive Systems*. ACM, 2020.

[10] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: rethinking merge in revision control systems," in *European software engineering conference and Symposium on the Foundations of Software Engineering*. ACM, 2011.

[11] S. Apel, O. Leßenich, and C. Lengauer, "Structured merge with auto-tuning: balancing precision and performance," in *International Conference on Automated Software Engineering*. ACM, 2012.

[12] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. OOPSLA, pp. 59:1–59:27, 2017.

[13] P. Accioly, P. Borba, and G. Cavalcanti, "Understanding semi-structured merge conflict characteristics in open-source Java projects," *Empirical Software Engineering*, vol. 23, no. 4, p. 2051–2085, 2018.

[14] G. Cavalcanti, P. Borba, G. Seibt, and S. Apel, "The impact of structure on software merging: semistructured versus structured merge," in *International Conference on Automated Software Engineering*. IEEE, 2019.

[15] A. T. Tavares, P. Borba, G. Cavalcanti, and S. Soares, "Semistructured merge in JavaScript systems," in *International Conference on Automated Software Engineering*. IEEE, 2019.

[16] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang, "Intellimerge: A refactoring-aware software merging technique," *ACM Transactions on Programming Languages and Systems*, vol. 3, no. OOPSLA, 2019.

[17] A. Sarma, D. F. Redmiles, and A. Van Der Hoek, "Palantir: Early detection of development conflicts arising from parallel code changes," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 889–908, 2012.

[18] S. McKee, N. Nelson, A. Sarma, and D. Dig, "Software practitioner perspectives on merge conflicts and resolutions," in *International Conference on Software Maintenance and Evolution*. IEEE, 2017.

[19] R. E. Grinter, "Supporting articulation work using software configuration management systems," *Computer Supported Cooperative Work*, vol. 5, no. 4, pp. 447–465, 1996.

[20] C. R. B. de Souza, D. Redmiles, and P. Dourish, "Breaking the code, moving between private and public work in collaborative software development," in *International ACM SIGGROUP Conference on Supporting Group Work*. ACM, 2003.

[21] B. Adams and S. McIntosh, "Modern release engineering in a nutshell–why researchers should care," in *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016.

[22] R. Potvin and J. Levenberg, "Why Google stores billions of lines of code in a single repository," *Communications of ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[23] F. Henderson, "Software engineering at Google," accessed: December 2017. [Online]. Available: https://arxiv.org/abs/1702.01715

[24] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2016.

[25] M. Fowler, "Feature toggle," accessed: December 2017. [Online]. Available: https://goo.gl/QfJ6mM

[26] P. Hodgson, "Feature branching vs. feature flags: What's the right tool for the job?" accessed: December 2017. [Online]. Available: https://goo.gl/4D2AMv

[27] S. Horwitz, J. Prins, and T. Reps, "Integrating noninterfering versions of programs," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 3, pp. 345–387, 1989.

[28] S. Khanna, K. Kunal, and B. C. Pierce, "A formal investigation of diff3," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, 2007.

[29] M. Sousa, I. Dillig, and S. K. Lahiri, "Verified three-way program merge," *ACM Transactions on Programming Languages and Systems*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[30] M. M. Almasi, H. Hemmati, G. Fraser, and A. Arcuri, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *International Conference on Software Engineering*. IEEE, 2017.

[31] G. Fraser, "A tutorial on using and extending the evosuite search-based test generator," in *Search-Based Software Engineering*. Springer, 2018.

[32] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*. IEEE, 2007.

[33] R. S. Barros Filho, "Using information flow to estimate interference between developers same-method contributions," Master's thesis, Universidade Federal de Pernambuco, 2017.

[34] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, 2007.

[35] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[36] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *International Conference on Mining Software Repositories*. IEEE, 2017.

[37] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[38] Online Appendix, 2020, available at: https://spgroup.github.io/papers/semantic-conflicts-testing.html.

[39] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*. IEEE, 2007.

[40] A. Salahirad, H. Almulla, and G. Gay, "Choosing the fitness function for the job: Automated generation of test suites that detect real faults," *Software Testing, Verification and Reliability*, vol. 29, no. 4-5, p. e1701, 2019.

[41] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, pp. 1–42, 2014.

[42] R. B. Evans and A. Savoia, "Differential testing: a new approach to change detection," in *European software engineering conference and Symposium on the Foundations of Software Engineering*. ACM, 2007.

[43] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *International Conference on Software Testing, Verification and Validation*. IEEE, 2010.

[44] S. Shamshiri, G. Fraser, P. Mcminn, and A. Orso, "Search-based propagation of regression faults in automated regression testing," in *International Conference on Software Testing, Verification and Validation*. IEEE, 2013.

[45] H. V. Nguyen, M. H. Nguyen, S. C. Dang, C. Kästner, and T. N. Nguyen, "Detecting semantic merge conflicts with variability-aware execution," in *Symposium on the Foundations of Software Engineering*. ACM, 2015.

[46] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Exploring variability-aware execution for testing plugin-based web applications," in *International Conference on Software Engineering*. IEEE, 2014.