Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Detecting semantic conflicts with unit tests☆

Léuson Da Silva [a,b,*], Paulo Borba [a], Toni Maciel [a], Wardah Mahmood [c], Thorsten Berger [c,d], João Moisakis [a], Aldiberg Gomes [a], Vinícius Leite [a]

[a] Centro de Informática, Universidade Federal de Pernambuco, Pernambuco, Brazil
[b] Polytechnique Montreal, Montreal, Canada
[c] Chalmers — University of Gothenburg, Gothenburg, Sweden
[d] Ruhr University Bochum, Bochum, Germany

ARTICLE INFO

Keywords:
Semantic conflicts
Differential testing
Behavior change

ABSTRACT

While modern merge techniques, such as 3-way and structured merge, can resolve textual conflicts automatically, they fail when the conflict arises not at the syntactic, but at the semantic level. Detecting such semantic conflicts requires understanding the behavior of the software, which is beyond the capabilities of most existing merge tools. Although semantic merge tools have been proposed, they are usually based on heavyweight static analyses, or need explicit specifications of program behavior. In this work, we take a different route and propose SAM (SemAntic Merge), a semantic merge tool based on the automated generation of unit tests that are used as partial specifications of the changes to be merged, and that drive the detection of unwanted behavior changes (conflicts) when merging software. To evaluate SAM's feasibility for detecting conflicts, we perform an empirical study relying on a dataset of more than 80 pairs of changes integrated to common class elements (constructors, methods, and fields) from 51 merge scenarios. We also assess how the four unit test generation tools used by SAM individually contribute to conflict identification. Our results show that SAM performs best when combining only the tests generated by Differential EvoSuite and EvoSuite, and using our proposed testability transformations (nine detected conflicts out of 29). These results reinforce previous findings about the potential of using test-case generation to detect conflicts as a method that is versatile and requires only limited deployment effort in practice.

1. Introduction

Branching and merging are common practices in collaborative software development. They facilitate effective teamwork, allowing developers to independently contribute to the same project. Still, branching and merging come with costs, including the need to resolve conflicts that are detected by merge tools when integrating code changes. Depending on project characteristics (Owhadi-Kareshk et al., 2019; Dias et al., 2020), such *merge* conflicts occur often (Perry et al., 2001; Mens, 2002; Zimmermann, 2007; Bird and Zimmermann, 2012; Kasi and Sarma, 2013; Brun et al., 2013; Mahmood et al., 2020), even when using more advanced merge tools (Apel et al., 2011, 2012; Cavalcanti et al., 2017; Accioly et al., 2018; Cavalcanti et al., 2019; Tavares et al., 2019; Shen et al., 2019) that explore language syntax and static semantics (Wąsowski and Berger, 2023) 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 (Sarma et al., 2012; Bird and Zimmermann, 2012; McKee et al., 2017). To avoid dealing with *merge* conflicts, developers sometimes even adopt risky practices, such as rushing to finish changes first (Grinter, 1996; Sarma et al., 2012) and partial check-ins (de Souza et al., 2003). Similarly, partially motivated by the need to reduce *merge* conflicts, development teams have been adopting techniques such as trunk-based development (Adams and McIntosh, 2016; Potvin and Levenberg, 2016; Henderson, 2017) and feature toggles (Bass et al., 2016; Adams and McIntosh, 2016; Fowler, 2017; Hodgson, 2017).

☆ Editor: Shane McIntosh.

* Corresponding author.
E-mail address: lmps2@cin.ufpe.br (L. Da Silva).

[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. Test and production conflicts (and undetected ones) are referred as behavioral semantic conflicts. For brevity, hereafter we omit the "behavioral" term in spite of focusing only on behavioral semantic conflicts in the paper.

https://doi.org/10.1016/j.jss.2024.112070
Received 1 October 2023; Received in revised form 12 March 2024; Accepted 21 April 2024

Although these practices might reduce the occurrence of *merge* conflicts, there is no evidence that they are effective in resolving or even detecting *test* (Brun et al., 2013) and *production* conflicts, which are only observed when running project tests and using the system. As such, they are more serious than *merge* conflicts, because they give rise to software failures. In fact, some of the practices mentioned before might even aggravate the costs of test and production conflicts, which are special kinds of what we hereafter 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 to detect and fix, especially when resolution occurs long after conflict introduction, because resolving them requires handling *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 is automatically performed generating a merged program, a build is created with success for this program, but its execution leads 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. For humans, it can be hard to detect semantic conflicts because this often involves analyzing the effect of the execution of (potentially long) chains of method calls. Horwitz et al. (1989) put this more formally: two contributions (sets of changes) to a base program *semantically conflict* — that is, interfere in an unplanned way — when the specifications they are individually supposed to satisfy are not jointly satisfied by the program that integrates them.

To help reduce the costs associated with semantic conflicts, we need merge tools that are able to detect them, going beyond textual line-based merge tools currently used in practice (Khanna et al., 2007). Previous work (Horwitz et al., 1989; Sousa et al., 2018) proposes semantic merge tools that rely on static analysis and model checking for detecting conflicts. Our previous study (Da Silva et al., 2020) proposes and assesses the use of *unit test generation* to reveal interference. The initial results bring evidence of the potential of using tests to detect semantic conflicts, but also show a number of limitations, including a significant false-negative rate.

To address these limitations, we extend our previous work by proposing and evaluating new techniques (testability and serialization transformations) and integrating them into SAM (SemAntic Merge), a semantic merge tool for Java that automatically generates unit tests and use them as partial specifications of the changes to be merged, with the aim of detecting semantic conflicts. SAM first applies a textual merge tool to integrate the changes. In case no textual conflicts are reported, SAM builds the four program versions associated with a merge scenario — a quadruple (*Base*, *Left*, *Right*, *Merge*) formed by a merge commit (*Merge*), its parents (*Left* and *Right*), and a *Base* commit; the four program versions are needed to check our new proposed conflict criteria. Optionally, SAM applies source code transformations that might increase program *testability* and feed the test generation tools with objects *serialized* during the execution of existing project tests. Then, SAM applies four test generation tools: EvoSuite and Differential Evosuite (Almasi et al., 2017; Fraser, 2018), Randoop (Pacheco et al., 2007), and Randoop Clean, an adapted version of Randoop we propose here. SAM then runs the generated tests against the four program builds, collects test failure information, interprets that with our interference criteria heuristics, and finally reports detected conflicts.

To evaluate our tool, we perform an empirical study with a dataset of 85 changes' pairs from 51 software merge scenarios that integrate changes to the same method, constructor, or field declaration. These scenarios come from open-source Java projects and are either mined by our scripts or used in previous studies (Cavalcanti et al., 2019; Sousa et al., 2018; Barros Filho, 2017; Da Silva et al., 2020). We investigate the following research questions:

- **RQ1:** *Can we adopt unit testing to detect semantic conflicts?* For each merge scenario, we invoke SAM (which uses unit test generation tools) and check its effectiveness in detecting interference following our *test-based* criteria;
- **RQ2:** *What are the causes of failures reported by SAM?* For the scenarios in which SAM fails to detect an existing interference, we manually analyze the causes of the failure. This sheds light on how SAM and the underlying unit test generation tools could be improved.
- **RQ3:** *What is the effect of adopting different unit test generation tools, and testability and serialization transformations?* Since SAM invokes different unit test generation tools on different versions of executables (original, transformed, and serialized), we can measure the effect of adopting each technique;
- **RQ4:** *How efficient are unit test generation tools in detecting behavior changes and related metrics?* Since some conflicts might be challenging to detect, we assess whether the generated test suites can detect general behavior changes. Furthermore, we also compare Randoop and *Randoop Clean*, our extended version.

Our results show that SAM performs best when combining only the tests generated by Differential EvoSuite and EvoSuite, and using our proposed testability transformations (nine detected conflicts out of 29). These results reinforce our previous findings of the potential of using test-case generation to detect semantic conflicts, as a method that is versatile and requires only limited deployment effort in practice, without needing explicit behavior specifications. Despite the low rate of true positives, the generated tests lead to only a few cases of false positives. This suggests that semantic merge tools based on unit test generation, as we propose here, can help developers detect semantic conflicts early, avoiding them to otherwise reach end users as failures. However, with the current capacity of the test generation tools, developers cannot rely solely on such semantic merge tools for detecting conflicts.

Our manual analysis of generated test suites lead to the identification of shortcomings of the existing tools. In line with those shortcomings, we suggest three potential improvements, that involve 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. Regarding the detection of behavior changes between commit pairs, although EvoSuite is the most successful tool detecting 53% of all reported changes, there is no combination of tools that detects all reported behavior changes. As a final contribution, we provide our study sample as a dataset of merge scenarios with source code, working executables (which are necessary for running tests), and interference ground truth. This can be used to run new studies with less effort and to replicate ours.

This study is an extension of our previous work (Da Silva et al., 2020). After our initial results on the detection of semantic conflicts using unit test generation tools, we focus on evaluating the effectiveness of our technique combined with different improvements, such as the generation of complex objects and the test generation process based on a target method. In summary, we can highlight the following contributions of our work:

- We propose and evaluate SAM, our semantic conflict detection tool;
- We propose and evaluate the use of serialization regarding the generation and use of required complex objects;
- New criteria for detecting semantic conflicts are introduced, comparing the unit test outputs of the four program versions in a merge scenario;
- We present and evaluate Randoop Clean, a modified version of Randoop;
- Finally, the dataset and scripts used to run this study (Online Appendix, 2024) are available online, supporting replications and new experiments.

```
1    class Text {
2      public String text;
3      void cleanText() {
4        this.removeDuplicatedWords();
5        this.removeComments();
6  –     this.normalizeWhitespace();
7      }
8
9      void removeDuplicatedWords() { ...
10 –     this.normalizeWhitespace();
11     }
12     void normalizeWhitespace() {...}
13   }
```

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

The rest of the paper is organized as follows: Section 2 motivates the problem under investigation here, while Section 3 presents SAM. In Section 4, we present the methodology. Section 5 presents the results, which are further discussed in Section 6. Threats to the validity and related work are discussed in Sections 7 and 8 respectively. Finally, in Section 9, we present our conclusions.

2. Motivating example and background

To illustrate the notion of behavioral semantic conflict we explore in this paper, consider the example in Fig. 1. Each change in this merge scenario independently aims to eliminate a redundancy in the `cleanText()` method, namely the two calls to `normalizeWhitespace()` whenever `cleanText()` is executed. The illustrated class `Text` results from a merge that integrates the deletion change in green (Line 6, say from a revision *Left*) with the deletion change in red (Line 10, say from a revision *Right*). This example is inspired by a *Merge* commit from the project Jsoup.[2] The other code lines originate from a *Base* revision, that is, the most recent common ancestor of *Left* and *Right*.[3] As the source code in the line 8 separates the two changes to be integrated, there is no textual 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 primary purpose of the `cleanText()` method is to apply some string cleaning. For that, it calls additional methods to remove duplicated words (Line 4), comments (Line 5), and normalize whitespace (Line 6). These calls were added in previous changes when developers independently added calls to `normalizeWhitespace()` causing the redundancy we just discussed. Someone may argue that these redundant calls could have been avoided by establishing a good communication channel between the involved developers, but that is often not in place.

Aiming to eliminate the redundancy, developers decide to eliminate one of the calls, but they unluckily do not pick the same call. While *Left* removes the method call in Line 6, *Right* removes the call in Line 10 (see Fig. 1). As a result, after integrating these revisions, there is no call left to `normalizeWhitespace()`, characterizing an undesired *interference* between the *Left* and *Right* revisions. This way, we might assume the occurrence of an infection, as the state of the target program is incorrect (Fraser and Ammann, 2008).

To detect the just discussed conflict, different approaches can be adopted, like careful code review practices and strong test suites. However, most semantic conflicts might escape to end users. In our example, we would have to investigate whether the defect is in the individual implementations of *Left* and *Right*, 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()`.

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

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

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 the use of *generated tests as partial specifications of the code revisions to be integrated* — tests then partially capture the effect of the changes in the revisions. This is the basis of SAM, the semantic merge tool that we propose.

In our motivating example, SAM could detect the interference with a test that explores the contents of the `text` field. For instance, suppose a regression test generation tool (such as Randoop (Pacheco et al., 2007) or EvoSuite (Almasi et al., 2017; Fraser, 2018), which are invoked by SAM) generates the test in Fig. 2 when given the revision *Left* as input. That test passes when executed against revision *Left*, which leads to a single call to `normalizeWhitespace()` when executing `removeDuplicatedWords()` (Line 4 in `Text` class). With the input illustrated in `test1`, when reaching Line 6, `t.text` stores a string similar to the test input string in Line 4 but not having the extra space character right before `dog`. Consequently, the assert successfully evaluates. Executing this test case against revision *Right*, the test also passes as there is a single call to `normalizeWhitespace()` (Line 6). Finally, the same test case also passes when executed against revision *Base* since it has two calls to `normalizeWhitespace()` (Lines 6 and 10). For this reason, passing in *Base* and in both parent revisions *Right* and *Left*, we say that test1 partially reveals a behavior that should be preserved by both revisions.

However note that `test1` fails when executed against revision *Merge* in Fig. 1. The `normalizeWhitespace()` method ends up not being called when executing `cleanText()`, as explained before. This way, `test1`, an original behavior expected to be preserved by both developers, is not satisfied in the merged version, revealing that the changes in *Right* and *Left* interfere (with respect to *Base*) (Binkley et al., 1995; Da Silva et al., 2020). This is essentially one of the criteria that SAM applies for automatically detecting *interference* by generating and executing tests, as detailed 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). However, for simplicity, hereafter we use both terms interchangeably and distinguish only where necessary for extra clarity.

2.1. Background

Unit test generation tools like Randoop (Pacheco and Ernst, 2007) and EvoSuite (Fraser and Arcuri, 2012) create tests following two steps: *test setup generation* and *assert generation*. While the first step is responsible for creating, initializing, and exercising objects required for the test, the second step creates assertions that check the test's expected results.

[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.

Regarding the first step, Randoop generates *sequences* of calls by randomly selecting the methods and constructors from the class under test. The arguments for such operations are also randomly selected from a pool of primitive type values and objects previously created through sequences. A sequence can group one or more *statements*: method calls or variable declarations. Each statement has two elements: a call for a method or constructor (including arguments that reference previously generated statements) and an assignment that stores the value returned by each call. EvoSuite, like Randoop, also starts from randomly generated test suites; however, it relies on genetic algorithms to evolve the test suites to optimize a specific goal, such as higher code coverage. This way, each test suite is evaluated based on a fitness criterion, while the fittest suites undergo mutations or crossovers, generating new tests. This process is repeated until EvoSuite finds a test suite that satisfies the target fitness criteria or the time budget is over.

Concerning the *assert generation* step, Randoop generates *Java asserts* which establish whether the test passes or fails. Such asserts check specific states held by an object, or whether exceptions are raised, for example. As a check is always associated with a specific sequence index, the check must be executed just after the related sequence. So, if there is a check at the index *i*, this check must be performed just after the end of the *i*-th sequence. EvoSuite adopts a similar approach by exploring the values returned from executing method sequences. EvoSuite can further calculate a reduced set of the generated assertions, whereas Randoop generates 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.

3. Detecting semantic conflicts

This section presents the solutions and techniques we propose to detect semantic conflicts. Initially, we motivate and introduce SAM, our semantic merge tool based on unit test generation tools. Next, we present different techniques, like the use of testability transformations and serialization, that we explore aiming to enhance the potential of detecting conflicts. Finally, we present Randoop Clean, our extended version of Randoop.

3.1. SAM: SemAntic merge tool based on unit test generation

Detecting semantic conflicts, as motivated in Section 2 is a complex task, not supported by current merge tools. Aiming to support developers in actively detecting these conflicts, we present SAM, our semantic merge tool based on unit test generation (Da Silva et al., 2020). The essence of SAM is to generate and execute tests for a given merge scenario (quadruple of *Base*, *Left*, *Right*, and *Merge* commits). These tests are executed over the different commits of a merge scenario, and after interpreting their results, the tool reports if a semantic conflict is detected.

SAM can be called right after a successful textual merge is performed. With the resulting merge scenario, SAM invokes unit test generation tools to generate test suites exploring the changes that have just been textually integrated. Next, SAM executes the generated suites in the different commits of a scenario, and analyzes the test results based on a number of heuristics, reporting conflicts accordingly. We explain in detail our heuristics later in Section 3.1.5. If a conflict is detected, the tool warns developers about its occurrence, informing the class and methods involved in the conflict. Next, we present in detail how the Java-Maven version of SAM works and its different steps (see the SAM's workflow area in Fig. 5).

3.1.1. Starting point

SAM is called when a merge is in progress in a local git client.[4] If no merge (textual) conflict occurs, SAM is invoked to verify the occurrence of semantic conflicts involving the parent commits' contributions. On the other hand, if merge conflicts are reported, SAM is not called since these conflicts would require manual fixes by the integrator, eventually leading to new changes not related to the original parent commits. For merge scenarios classified as *fast-forwards*, SAM does not take any action, leaving the default merge tool to lead the integration process. Once SAM is called, the first action is to collect the commits involved in the merge scenario. For that, the tool gets the *Merge commit hash* from the head of the current branch, while the *Left*, *Right* and *Base commit hashes* are taken by calling further git commands. After collecting these merge scenario information, the tool advances to the next step, when the parent commits' contributions are explored and mined.

3.1.2. Selecting mutual changes on same class elements

In this step, the tool explores the changes performed by the parent commits, aiming to collect mutually changed elements (methods, constructors, or field declarations). We assume that the chances one developer's contributions unexpectedly affect others might be higher under such situations, as these changes might modify, for example, the same variables, changing a method behavior in undesired ways. For that, using DiffJ,[5] SAM collects the set of Java class elements changed by each parent commit. If at least one element is changed in both parents, the tool moves to the next step. Otherwise, the textual merge operation continues without applying any further steps in its usual workflow.

3.1.3. Generating executable files

Since SAM invokes unit test generation tools that rely on test execution as part of the generation process, it must feed such tools with executable versions of the target code we want to assess.[6] For that, SAM generates one executable version of the system for each commit of the merge scenario (*Base*, *Left*, *Right*, and *Merge commits*). Executables of all versions of the system are required because SAM detects conflicts by comparing the results of the tests when executed against the different versions.

SAM performs a sequence of checkouts for each commit hash to generate the associated executables. This way, for each target commit, SAM calls the build manager to compile the code, resulting in a JAR file created and released on the *target* directory of the project. *pom.xml* file. Aiming to increase the testability of the target code under analysis, SAM adopts extra techniques like the adoption of testability and serialization transformations, which refine the just described build process. But we only explain these two techniques in Sections 3.2 and 3.3, respectively.

3.1.4. Generating and executing test suites

After collecting the mutually changed elements and the associated executables as explained in the two previous sections, SAM generates and executes test suites.

The generation is driven by the classes that contain the mutually changed elements. This way, SAM invokes each unit test generation tool once for each parent commit (*Left* and *Right*), as these commits are responsible for introducing the changes that could be conflicting in the *Merge* commit. If methods or constructors are mutually changed, SAM might further drive the generation of test suites based on these

[4] Currently, the tool is invoked when the `post-merge` *hook* is actived; that is a Git feature to execute custom scripts. This hook is responsible for performing additional and specific checks after a successful merge commit is created.

[5] https://github.com/jpace/diffj.

[6] A JAR file with all compiled classes of the system and the required external dependencies.

elements, aiming to directly explore the code where conflicts likely take place. SAM can be configured to work with different unit test generation tools, and invoke a number of them as needed. In our experiments, we use two versions of Evosuite and two versions of Randoop, as detailed later. One of the versions of Randoop, called *Randoop Clean*, we designed with the aim of generating tests more focused on our goal of detecting conflicts; Section 3.4 presents our tool, highlighting its changes compared to the standard version of Randoop.

Finally, after the test suites are generated, SAM executes each one against each of the four executables associated with a merge scenario, and collects the test results for further analysis.

3.1.5. Conflict detection based on test results heuristics

This step is responsible for reporting conflicts based on the results of executing the generated tests against the four executable versions of a merge scenario. SAM basically checks whether any test case satisfies one of our conflict criteria, which we present next, with their motivation. They all rely on the notion of partial specification, that is, a specification that constrains behavior only for a subset of the possible inputs. As such, each test case is seen as a partial behavior specification, and we can then refer to the definition of interference that relies on preserving parent specifications in the merged version of the code, as discussed in Section 1.

To detect conflicts, we rely on specific conflict criteria implemented by SAM. We believe these criteria cover common situations of semantic conflicts. The first two criteria (one for each parent, i.e., *Left* and *Right*) seek for test cases that present the same outputs in the *Base* and *Merge* executable versions, but a different one in the associated *parent* version. For example, consider a test case `test1` that passes when executed against the *Left* version, but fails against the *Base* version. So we might say that `test1` partially captures the intention of the *Left* change; we can then see `test1` as a partial specification of the changes of *Left*. Now if `test1` fails when executed against the *Merge* version, we conclude that `test1`, the partial specification of *Left*, is not satisfied in the merged version, revealing that the changes carried on by *Right* interfere with the changes of *Left* (with respect to the *Base* commit). So when SAM finds a test that satisfies the just mentioned criteria, it reports a *(test) conflict*.

Now, consider a different scenario (similar to the one in Section 2) and criteria, where a test case `test2` passes in the *Base*, *Left*, and *Right* versions. We can then consider that `test2` partially captures a behavior that is preserved by both *Left* and *Right* changes, and therefore we expect such behavior to be preserved in the merge too. So if `test2` fails in the *Merge* version, we conclude that a behavior that was expected to be preserved is actually not preserved, revealing that the changes in the parent commits interfere with each other (with respect to *Base*). So when SAM finds a test that passes in the *Base*, *Left*, and *Right* versions, but fails in *Merge*, it reports a *(test) conflict*.

Since our conflict criteria rely on the final statuses of test cases executed in different executable versions, we must further comment about test statuses different than *passed* and *fail*. For test cases that present *error* statuses, we opt to not consider them when reporting conflicts. We believe that considering these cases might introduce false positives in our results as we could not correctly verify the test case statuses. As a final remark, it is important to discuss that our conflict criteria are valid to detect conflicts if the associated test cases explore the conflicting changes integrated during a merge scenario. Otherwise, false positives might be reported as well.

3.1.6. Report of semantic conflict occurrence

Once a test case satisfies one of our conflict criteria, our tool warns the developer about a potential conflict occurrence by informing the element where the conflict takes place, as also the test that reveals the conflict. Then, the developer might evaluate whether the reported conflict represents an actual conflict or not. If so, she may apply changes in order to fix the conflict and change the current *Merge* commit; otherwise, she skips the warning leaving the *Merge* commit without applying any change.

3.2. Testability transformations

Previous studies (Silva et al., 2017) report that, due to a number of characteristics of the code under analysis, unit test generation tools might have a hard time generating tests that detect bugs. In our previous study (Da Silva et al., 2020), we observe similar limitations for detecting conflicts. For example, it might be harder to generate conflict revealing tests for classes with many private members, as these cannot be directly exercised by the tests. Nevertheless, directly invoking such members could reveal conflicts that would be hard to reveal by tests that only invoke public members (that indirectly invoke the private ones). To improve testability and increase the chances of detecting conflicts, we propose here three testability transformations that adapt the source code of the parent commits before creating the builds and feeding the generation tools with executables. These transformations are motivated by preliminary experiments we performed using the unit test generation tools with toy examples and a small subsample of the scenarios we consider in our evaluation.

As just motivated, the first proposed transformation replaces non-`public` access modifiers with `public` ones; it is applied to classes, methods, constructors, and fields declarations. By making all elements public, more elements can be called and accessed by the generated tests, possibly increasing the chances of detecting conflicts. This, however, brings the risk of reporting false positives, as could happen when a generated test accesses an originally private member in a way that is not equivalent to the indirect access from the available public members. There is also the risk of the tools using a significant part of the generation budget for directly calling elements not involved in the conflict, as we apply the transformation to all classes; the motivation is that calling non-related elements involved in the conflict might lead to indirect object state change that contribute to conflict detection. These aspects are evaluated in the experiments we describe later.

Our second transformation adds an empty constructor to classes lacking one, as this might help to generate tests that create and exercise objects of such classes. We observed that this could be especially useful for classes having only constructors that require complex object structures as arguments. Again, this transformation brings the risk of false positives, as reported conflicts might be revealed with object states that would not be reachable with the original class. For simplicity, in case a class does not directly extend `Object` no empty constructor is added, as this would potentially require adding a chain of constructors to the class hierarchy.

Finally, our third transformation handles scenarios in which the mutually changed declarations occur inside inner classes. As the generation tools cannot directly exercise inner classes, we extract them to the outer level. For simplicity, we manually apply this transformation, as it is not often required. The other two have been implemented and are automatically applied.

3.3. Serialization transformation

As previously discussed, unit test generation tools might not be able to generate tests that exercise complex object structures in useful ways (Da Silva et al., 2020). Such structures, however, might be required to reveal conflicts. Aiming to address this limitation, we feed unit test generation tools with concrete object graphs (Elbaum et al., 2006), which can then be used in the generated tests, increasing the chances of detecting conflicts. We collect and serialize these object graphs by monitoring the execution of existing, manually created, project tests. The effectiveness of this technique is then directly dependent on the availability of project tests that manipulate complex objects. For projects with no tests, we do not use this technique.

To serialize objects, we implement OSean.EX.[7] First, our tool instruments the target method — the method under analysis — by adding

[7] https://github.com/spgroup/OSean.EX.

a call for an auxiliary method that is responsible for receiving and serializing the object currently executing the target method, and the arguments passed to this method. OSean.EX also adds the auxiliary class to the original target project. With this first instrumented version of the project under analysis, the tool runs the manually created project test suite for a specific amount of time, creating new unique serialized objects each time the target method is reached; eventually, duplicated objects are discarded.

When the project test suite execution is finished, our tool discards the instrumented version of the project and creates a new class that declares a number of methods, one for each previously serialized object. Each method simply deserializes an associated object and returns it. OSean.EX then adds this class to the original version of the target project, creating a second instrumented version of the project, and building it. This version's executable can then be fed to the unit test generation tools, which are able to create tests that call the deserialization methods and use the returned complex objects.

Considering that a merge scenario has related commits, and that object serialization might be expensive, OSean.EX performs all the steps for a single commit, say *Left*, depending on how it is invoked. In this case, for the *Right* our tool would simply perform the last steps of adding the created class and building the extended version of *Right*. The cost of using this technique depends on the amount of time allocated to execute the project test suites.

3.4. Randoop clean

To increase the chances of detecting conflicts, we propose here Randoop Clean, which adapts Randoop with the aim of creating tests that more often invoke the method under analysis, and increasing the diversity of objects manipulated by the generated tests (see our online Appendix for more details (Online Appendix, 2024)). In our previous work, although Randoop did not detect as many conflicts as EvoSuite, Randoop generates more test cases reaching the target method holding the conflict (Da Silva et al., 2020). This way, we believe exploring multiple calls to the target method might increase the chances of detecting conflicts. As we preserve most behavior of original Randoop, we highlight here only the changes implemented by our tool. First, before Randoop starts to generate tests, it selects all public methods and constructors from a list of classes given as input and puts them in a *pool*. Next, Randoop creates test prefixes by randomly selecting elements from the pool and generating sequences of statements that invoke such elements. If a particular method is expected to be covered by the generated test cases, it can be given as input to the tool. However, in our context, we want to go beyond that and increase in these sequences the number of calls to the method under analysis, that is, the method changed by both parents commits (target method). In principle, this could increase the chances of conflict detection.

To address this idea of maximizing target method calls, Randoop Clean reduces randomness by optimizing the number of calls to a target method (see Fig. 3). The tool has access to a list of valid sequences generated during test suite creation (`errorSeqs`, line 1). Based on the number of classes given as input to the tool (`reqClasses`, line 3), Randoop Clean continuously checks the list length, verifying whether a specific number of new sequences were created (line 23). Each time this happens, the tool adds a new call to the target method (line 24). We adopt an interval among new target method calls so that other methods are called too. This way, objects required as parameters or holding the target method can be changed, allowing the target method to be executed against different objects.

To generate diverse objects required by the target method, Randoop Clean adopts a similar idea. Since the creation of objects occurs through classes' constructor or method calls, Randoop randomly selects calls from the pool, which holds the previously available methods and constructors. In our context, randomness might represent a weakness as a specific object could be generated and continuously used when

generateSequences(classes, contracts, filters, timeLimit, targetMethod)

```
1:  errorSeqs ← {}                                          ▷ Contract violations
2:  validSeqs ← {}                                          ▷ No contract violations
3:  reqClasses ← selectRequiredClasses(classes)
4:  steps = 0
5:  while timeLimit.isNotReached() do
6:                                                          ▷ Create new sequence
7:      steps++
8:      m(T1...Tk) ← selectMethod(classes, reqClasses, steps)
9:      ⟨hseqs, vals⟩ ← randomSeqsVals(validSeqs, T1...Tk)
10:     newSeq ← extend(m, seqs, vals)
11:     if newSeq ∈ validSeqs ∪ errorSeqs then
12:         continue
13:     end if
14:                                  ▷ Execute new sequence and check contracts.
15:     ⟨o⃗, violated⟩ ← execute(newSeq, contracts)
16:                                          ▷ Classify new sequence and outputs
17:     if violated = true then
18:         errorSeqs ← errorSeqs ∪ {newSeq}
19:     else
20:         validSeqs ← validSeqs ∪ {newSeq}
21:         setExtensibleFlags(newSeq, filters, o⃗)
22:     end if
23:     if isNumberOfValidSeqsEnough() then
24:         newSeq ← extend(targetMethod, seqs, vals)
25:     end if
26: end while
27: return ⟨validSeqs, errorSeqs⟩
```

Fig. 3. Randoop Clean process of generating sequences. Adaptations to the original Randoop appear in blue.

selectMethod(classes, requiredClasses, steps)

```
1: if steps%(10 × classes.size()) < 2 × classes.size() then return
   selectRandomMethod(requiredClasses)
2: else return selectRandomMethod(classes)
3: end if
```

Fig. 4. Method selection based on required objects by target method. Based on the number of generated steps, Randoop Clean randomly selects methods from the input classes.

required. A method could be called with the same objects, consequently producing the same results. Therefore, we opt to add new constructors or method calls that return objects of a specific type in order to increase the chances of generating diverse objects.

To generate objects that are used as arguments or targets of method calls, Randoop follows the process presented earlier. Hence, the number of calls to the target class constructor and of generated objects is random. However, this might lead to reduced diversity of the object pool, and consequently less chances of detecting conflicts. To address this Randoop Clean tries to *increase the number of calls to object creation operations (methods and constructors)*. After Randoop Clean generates a particular number of statements in a test prefix sequence (line 4 in Fig. 3), the tool adds calls to the object creation operations of the target class (Fig. 4). If the target method requires different object types, Randoop Clean selects one type each time and then randomly picks a method or constructor from the *pool* that returns this specific object type.

4. Evaluation method

Our evaluation method comprises five main steps to assess the potential of SAM and unit test generation tools to detect interference (Fig. 5). First, we extract and select merge scenarios from Java projects hosted on GitHub, including a number of scenarios that appear in previous code integration conflict studies (Sousa et al., 2018; Cavalcanti et al., 2019; Barros Filho, 2017). Second, we create executable JAR files of the program versions in each selected scenario. Initially, we create JAR files using the original source code of the four software

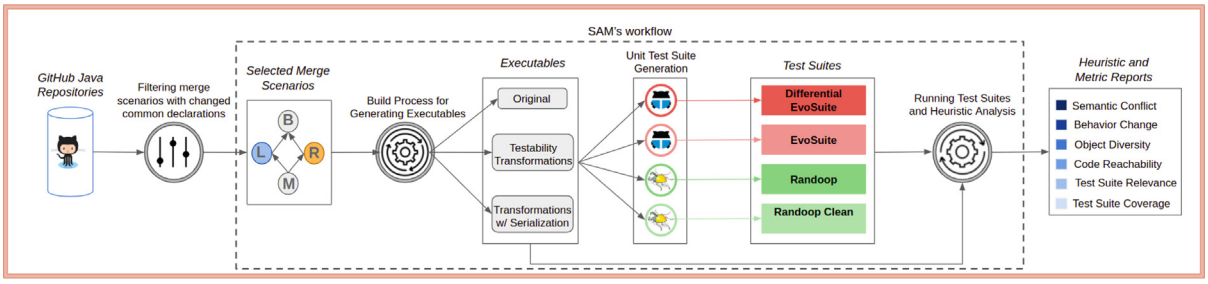

Fig. 5. Study setup. Starting with the selection of Java merge scenarios, we create our dataset, call the unit test generation tools, and execute the generated test suites to detect semantic conflicts. Besides that, we perform a manual analysis to explain the false positives and negatives in our sample. Inside the dashed area, we show the steps covered by SAM, our semantic merge tool.

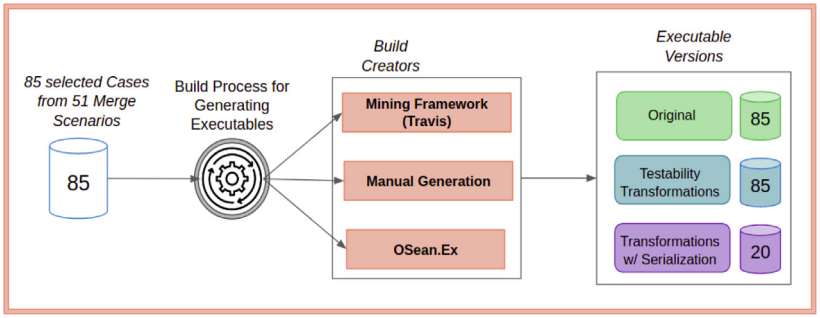

Fig. 6. The generation process of executables for merge scenario commits. For each merge scenario, we create a number of JAR files, which are given as inputs for the unit test generation tools. For the 85 cases of our sample we create JARs based on both the original code and the code resulting from applying the testability transformations. For a subsample of 20 cases, we create JARs based on the code resulting from applying the testability and serialization transformations.

versions corresponding to the *Base*, *Left*, *Right*, and *Merge commits* (see Fig. 6). Next, we generate additional four JAR files (one for each commit version) but this time applying our testability transformations (see Section 3.2). For some cases of our sample,[8] we generate a third set of JAR files, now applying in sequence the testability and serialization transformations (see Section 3.3). Third, we apply four test generation tools to create tests for the parent commits of a merge scenario based on each kind of JAR file available (*Left* and *Right* from original, transformed, and serialized JAR files). Next, we run our scripts to execute the tests and discard invalid tests avoiding flakiness issues (Luo et al., 2014).[9] Fourth, as a last automated step, we run our scripts to check the test-based interference criteria and additional related metrics regarding the quality of the generated tests. Fifth, we manually analyze each merge scenario and the obtained results to ensure that the reported interference is correct. Furthermore, we investigate the reasons behind the generated tests not detecting interference in some of the scenarios that suffer from interference.

4.1. Mining and selecting merge scenarios

Our dataset consists of 85 mutually integrated changes' pairs from 51 merge scenarios mined from 31 GitHub Java projects. Since we analyze class elements mutually changed by both parents in a merge scenario, one single merge scenario might hold more than one case of mutually changed element. As a result, for some merge scenarios, multiple cases of changed elements are considered in our evaluation. We focus only on Java projects because the unit test generation tools we use are language-dependent, and some of our scripts are also test 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.

Table 1
Distribution of mutual changed class elements.

Original sample	Selected changed mutual elements	
	With interference	Without interference
Da Silva et al. (2020)	4	2
Cavalcanti et al. (2019)	2	6
Sousa et al. (2018)	3	18
Barros Filho (2017)	13	15
Current study	7	15
Total	29	56

From the 85 cases we consider in our dataset, 63 first appeared in previous studies (Da Silva et al., 2020; Cavalcanti et al., 2019; Sousa et al., 2018; Barros Filho, 2017) that rely on datasets that share some scenarios and cases of mutually integrated changes' pairs. Six cases from five merge scenarios come originally from Da Silva et al. (2020); four cases with and two without interference. From Cavalcanti et al. (2019), we select eight original cases (from eight merge scenarios), two with and six without interference. From Sousa et al. (2018), we select 21 original cases (from 16 merge scenarios), three with and 18 without interference. Finally, from Barros Filho (2017), we include 28 original cases (from 22 merge scenarios), 13 with and 15 without interference (see Table 1).

The remaining 22 cases in our sample first appear in this paper; seven with interference and 15 without. All these cases come from seven scenarios that first appeared in previous work (Cavalcanti et al., 2019; Sousa et al., 2018; Barros Filho, 2017) that considered only a subset of the cases in these scenarios. With extra mining effort, we found out the remaining 22 cases we consider here. In our Online Appendix (2024), we provide further information and summarize all selected cases discussed here. Although we have not systematically

[8] Since we rely on the quality of original project test suites to generate serialized objects, we consider only a subsample.

[9] We consider tests invalid if they present different results on different executions.

targeted representativeness or even diversity (Nagappan et al., 2013), we believe that our sample has a considerable degree of diversity concerning different dimensions such as project domain, size, and number of collaborators. As future work, we plan to extend our sample by adding new merge scenarios based on our criteria.

4.2. Building the projects

As mentioned at the beginning of this section, for each case in our sample we must create JAR files that are used to generate and execute test suites. Considering we need to create build files with all project dependencies, for simplicity we initially try to use Travis (see Fig. 6) to create such executables. The main advantage of this approach is to reduce the chances of broken build processes due to local environment and configuration issues. As we use the Travis infrastructure, in case of merge scenarios requiring different environment options, we would not have to deal with each one directly. Instead, we just set up a Travis configuration file and reuse it when applicable. If Travis fails to create the builds due to no longer having access to old dependencies, no support for older Java versions, or by detecting problems when running additional analysis (like style checking) adopted by projects pipelines, we try to manually fix the problem on Travis by updating its configuration files; if that does not work, we locally create the builds.

The automated process involving Travis is only used for creating the builds for the original source code and the code resulting from the testability transformations. The builds for the code resulting from the serialization transformation are created manually, as our serialization tool requires extra configuration for each project. We have serialization builds only for a subsample of cases because that requires project test suites that exercise the method under analysis. We opt for running the project test suites for 60 s as most project test suites in our sample are finished by this time; so allocating more time would not significantly improve the pool of serialized objects. Once OSean.Ex accepts a list of commits, we invoke the tool giving as input the four commits in a merge scenario. This way, the serialized objects are generated based on the first commit of this list; in our study, the *Merge* commit is always first. The remaining commits of that list reuse the serialized objects by deserializing them based on their versions. So for *Base* and parent commits (*Left* and *Right*), OSean.Ex only performs the last step generating the executable files.

We also adopt the manual build creation process for Ant projects (one single case), as our automated infrastructure supports only Maven and Gradle projects. The process and infrastructure we use to create the builds appear in our Online Appendix (2024).

At this point, if we failed to create one of the builds for a case, we simply discarded the case in our experiment; five scenarios were discarded. At the end of this step, we have a sample composed of 51 *merge scenarios* and 85 *potential interference cases*, knowing that some merge scenarios contain more than one independent change on the same declaration. For all 85 cases, we have executables (eight, two for each commit version in a case) with the original source code and testability transformations. Finally, for 20 out of these 85 cases, we have additional executable files (four, one for each commit version in a case) with serialized objects.

4.3. Generating and executing tests

Each merge scenario and case resulting from the previous step has a number of proper executable files that tests can execute and exercise. These files are required by unit test generation tools that generate tests and run them against the system to be tested, discarding tests that fail or do not increase code coverage (see Fig. 7). This observation is valid here for the test generation tools we evaluate: EvoSuite (Almasi et al., 2017), Differential EvoSuite (Shamshiri, 2015), Randoop (Pacheco et al., 2007) and Randoop Clean, our extended version of Randoop. We chose the first three tools due to their robustness and popularity.

In this step, we readily apply the unit test generation tools to create tests for four of the executable versions (*Left*, *Right*, and their transformed versions, as explained above) associated with each merge scenario. For a subsample of the cases, we additionally invoke the unit test generation tools for two executable versions with serialization (serialized *Left* and *Right*). For each executable version, our scripts call EvoSuite, Randoop, and Randoop Clean passing the corresponding parent commit JAR file as input. For Differential EvoSuite, which tries to generate tests that reveal behavior differences between two program versions, we additionally give as input the JAR file of the *Base* commit, 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 each tool, we use a budget of 5 min and their default configuration.[10] We decide to adopt 5 min considering that related work opt for different budget configurations (1, 2, or even 10 min); in our previous study (Da Silva et al., 2020), we opt for 2 min. This time, we give more time for the tools and assess whether the budget affects the detection of conflicts.

Our scripts invoke each of the four tools for the two parents in a merge scenario, considering the three kinds of executables we create (original, testability and serialization), generating then 24 ($4 \times 2 \times 3$) test suites. For the scenarios with no serialization executable, we generate 16 test suites.[11] The number of tests in each suite varies a lot.

For each resulting test suite, our scripts execute each test case three times for each of the different versions: *Base*, both *parents*, and *Merge* (see Fig. 7), resulting in 12 executions. We execute the tests in both parents because two of our conflict criteria assess the test results against all executable files associated with the merge scenario commits. Finally, for each merge scenario without serialization executables, the 16 generated test suites are executed 12 times resulting in 192 executions; in the case of a scenario with serialization executables, the other eight test suites are also executed 12 times resulting in additional 96 executions.

We execute each test case three times aiming to detect test flakiness. If a test case does not yield the same result (pass or break) in the three repeated runs, we filter it out, as they would not help in conflict detection due to their flakinesses. The test suite execution results associated with each case of our sample are grouped 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 is generated for a given revision, say *Left*, but is executed in other revisions as well: *Base*, *Right*, 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* and *Right* version. In the same way, tests with error statuses are not considered as failed tests. An errored status signals an unexpected situation during test execution, which does not involve the program behavior under test. Such invalid tests are discarded as the last action in this step, and are not used for interference detection in the next step.

4.4. Detecting interference

We group test suite executions into sets for each case of our sample based on the executable versions used to generate the tests. Each set contains the executions associated with the *Base*, parents (*Left* and *Right*), and *Merge* commits for the original, transformed, and serialized

[10] The versions of the tools used by SAM and in our study are mentioned in our online appendix.

[11] In two cases of our sample, no test suites were generated due to environment issues (one case with and another without interference). Furthermore, for some specific versions, the tools presented errors and the generation process was interrupted, resulting in no test suite. In these cases, we do not discard the cases and simply consider that the tool does not report interference.

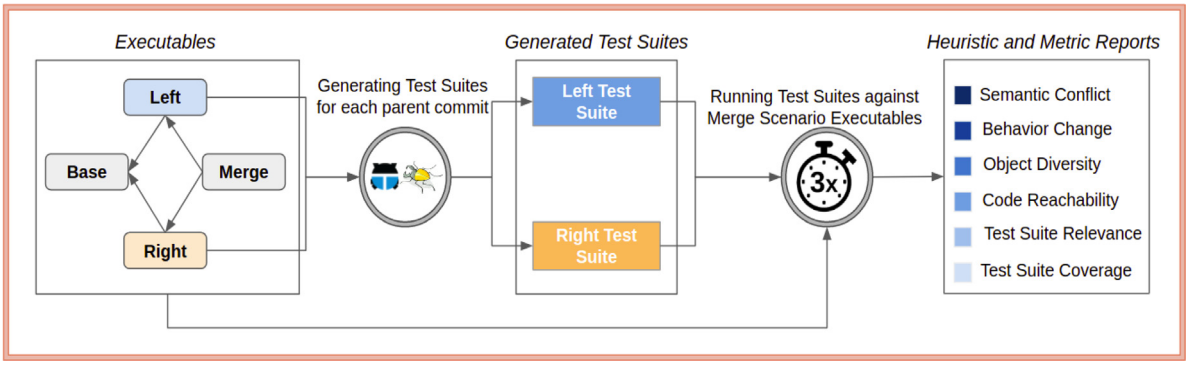

Fig. 7. Generation and execution of test suites. For each case of our sample, we generate test suites based on both merge scenario parent's commits. Next, we execute three times each generated test suite against all merge scenario commits in order to calculate our metrics.

versions. Next, for each execution result set, our scripts compute the test cases that satisfy one of our conflict criteria (see Section 3.1.5 and our Online Appendix (2024) for further information about our criteria). Finally, our scripts collect the results for further analysis and report interference if at least one test case satisfies at least one of our criteria.

4.5. Assessing other metrics

The steps so far are the essence for assessing the potential of SAM and unit test generation to detect interference. However, to better understand such potential and how it is limited by the unit test generation tools we use, we go further and assess other metrics. In particular, in the following, we consider metrics that may help us to better evaluate the effect and limitation of each technique we rely on: conflict detection criteria, testability and serialization transformations, and unit test generation tools.

4.5.1. Behavior change detection

Besides assessing whether the tests generated by the tools can detect interference, we assess whether they can establish a weaker property: behavior change between the commits in a merge scenario. Note that detecting interference requires detecting two behavior changes, for instance, one from *Base* to *Left* and another from *Left* to *Merge*. So when SAM fails to detect interference we want to assess whether the generated tests could detect one or none of the behavior changes as a measure of how far the tool was to detecting interference. To detect those behavior changes, we use the test suites previously generated and look for test cases that report different outcomes when running them against two commits. As we want to detect the behavior changes introduced by each parent in a merge scenario, we look for behavior involving the related parent and *Base* or *Merge* commits. So for each case in our sample, there are at most four possible behavior changes. We then compute and compare the number of behavior changes detected by each unit test generation tool.

4.5.2. Object diversity and target code reachability

To compare whether Randoop Clean is closer to detect interference than Randoop, we compute two metrics that are used to compare the test suites generated by both tools: the number of calls performed to a target method, and the number of different handled objects. To compute these metrics, we instrumented both tools to collect such information and report it after the generation of each test suite. So when we refer to Randoop in our study we actually mean a version of Randoop instrumented with this metric collection functionality. This way, our scripts have access to, for each tool, (i) the number of calls made for all possible methods of the target classes, and (ii) the number of different objects handled by the test suite.

4.5.3. Source code coverage

To further evaluate the improvements of Randoop Clean over Randoop, we compute the source code coverage (line, branch and instruction) of the test suites generated by each tool. This might help understanding which tool is closer to detecting interference. For that, we collect only the coverage achieved by each tool against the *Merge* commit. As the *Merge* commits contain the potentially interfering changes, if these are covered by a test suite on the *Merge* commit, they are likely covered on the *parents* and *Base* commits. As Randoop does not provide source code coverage information, we implement additional scripts to compute this using Jacoco (2022). For each generated test suite, our scripts call Jacoco to instrument the JAR file previously used to generate the test suite. Next, the test suite is executed against that new instrumented JAR file resulting in a file with the coverage results. Since we want to explore the coverage of a target method, our scripts compare the percentage achieved by each suite.

4.6. Manual analyses

In our study, we carry on two main manual analyses. First, we manually analyze each scenario in our dataset to establish interference ground truth. Second, after executing the experiment and detecting false positives and false negatives, we manually analyze each case and the generated tests and metrics to understand what caused the false result.

4.6.1. Ground truth analysis

Six researchers manually analyzed all cases of our sample; in pairs, the researchers individually analyzed each case to check for interference and later compared the analysis with his partner. If both researchers agree with the same decision, they present the scenario and its evaluation to the remaining four researchers. However, in case of a conflicting decision, the whole group discusses the case and reaches a verdict.

To reduce the chances of human error and misjudgment in this process, for each interference verdict, we manually designed a test case that reveals the interference. 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.

Our manual analysis is local, in the sense that it involves only the mutually changed program element and its dependencies (methods and fields it calls and accesses, for example). As we ignore the global context that depends on the analyzed program element, the changes could in fact globally interfere but we would not detect it. For example, say two developers, in the same method declaration, add assignments to disjoint fields of the same object. So, locally, the changes do not interfere with each other as they affect disjoint state elements that are unrelated in the local computations. But, if we consider the global context, say a method

`computeRate()` that calls the changed method and compares the two fields in specific ways, we could have interference. 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 method, as we do here. We opt for checking local interference only for two main reasons. First, it has the potential to detect a relevant part of interference cases. Second, the global context can be significantly large or hard to capture, especially when the changes occur in widely reusable classes that are either part of an API or are invoked from multiple program entry points (as in microservices systems); manual analysis and test execution in such cases could be challenging.

Instructions and guidelines used during this process are organized as a document, which is available in our Online Appendix (2024). For many cases (57), the ground truth is available in previous work, but we nevertheless follow the process above and compare verdicts. For all cases, we summarize the integrated changes to help reach verdicts and aid others interested in using our dataset for replications and further studies. We provide our dataset and its associated ground truth online, allowing its usage for new studies exploring semantic conflicts and general behavior changes.

4.6.2. False positives and false negatives analysis

Comparing the results of our experiment with our ground truth, we collect information on false positives (our interference criteria are satisfied, but there is actually no interference in the scenario) and false negatives (our interference criteria do not hold, but the scenario actually suffers from interference).

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 manually 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 (original, transformed, and serialized binary files), generated test suites, interference ground truth, and further information on the quality of each test generation tool.

For the merge scenarios reported with interference, we analyze the associated test suites to ensure that the tests explore the conflict that we find 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 explore the side effects of the elements involved in each conflict.

5. Results

We now present the results of our analysis of the 85 cases of changes' pairs in the 51 merge scenarios mined from 31 GitHub Java projects (see Section 4.1), including how semantic conflicts are detected by SAM's interference criteria and test generation process. We also discuss how the test generation tools used by SAM could be improved to increase conflict detection accuracy. Fig. 8 summarizes our findings. The right branch focuses on the changes with semantic conflicts, while the left branch focuses on the changes without conflicts, according to our ground truth.

Table 2
Comparison of SAM with related proposed tools.

Metrics	Tools		
	SAM	SAM	SafeMerge
Precision	0.75	1	0.28
Recall	0.31	0.33	0.66
F1 score	0.72	0.90	0.71
Accuracy	0.43	0.50	0.40
Cases	85	21	

5.1. RQ1: can we adopt unit testing to detect semantic conflicts?

To answer our first RQ, we compute the number of detected conflicts reported by SAM. Overall, SAM could automatically detect nine out of 29 conflicts (31%); these nine conflicts appear in five merge scenarios (the right branch in Fig. 8). Such a result reinforces our previous finding that unit test generation tools are useful to detect semantic conflicts (Da Silva et al., 2020). Although our tool misses a significant number of conflicts, it reports only three false positives, as depicted in the left branch. So, based on these results, a developer using our tool should plan to use additional techniques to detect the conflicts that SAM misses, but should not worry about wasting significant time investigating false positives.

Comparing our results with previous work, we observe that SAM outperforms them in some aspects. Table 2 presents a comparison with two previous work and their related datasets. Since these studies rely on a different dataset, we focus our comparison considering a common subsample used in their and our studies. Although SAM presents an overall precision of 0.75, outperforming the previous studies, when evaluating recall, we observe that SAM shows inferior performance (third and fourth rows in Table 2). Such a result brings to discussion the prominent and promising direction of combining the different approaches investigated by each study.

To illustrate one of the conflicts detected by our tool, consider the pair of changes integrated into the code of Fig. 10, which shows a *Merge* commit from the HikariCP project.[12] In this merge scenario, the *Left* commit adds a new condition to the if statement using the local variable `retries` (Line 6 in Fig. 10), restricting the call to `incrementAndGet`, which increments the number of total connections to a pool. Independently, the *Right* commit changes how `retries` is initialized (Line 3), which is referred by the changes performed by the *Left* commit. So the *Left* and *Right* commits individually change the program behavior (creating and adding single connections to a pool) based on their needs. These changes can be textually integrated with success. No *merge* conflict is reported since line 4 separates the changes made by the two developers.

The change from *Right*, however, interferes with the change from *Left*, breaking the intention of *Left*'s change. Fortunately, this is reported by our tool through the EvoSuite unit test case in Fig. 9. This test case was generated when our tool invoked Evosuite with the *Right* serialization executable. As highlighted in the test case (Line 5 in Fig. 9), the expected number of `totalConnections` is 10. Running this test case on the *Right* commit, the test passes. For the *Base* commit, the test fails as the returned number of connections is 12; as in this commit `retries` is initialized with `0`, some calls to `decrementAndGet` are bypassed (Line 11 in Fig. 10), not decrementing the total number of connections. Note that on the *Right* commit, the variable `retries` is initialized based on the number of acquired retries from the object `config`, which is later used in another `if` statement condition (Line 10 in Fig. 10). For the *Merge* commit, the test case also fails as the returned number of connections is −30; this time, no call to `incrementAndGet` is executed (Line 6) due

[12] HikariCP - merge commit: 1bca94a.

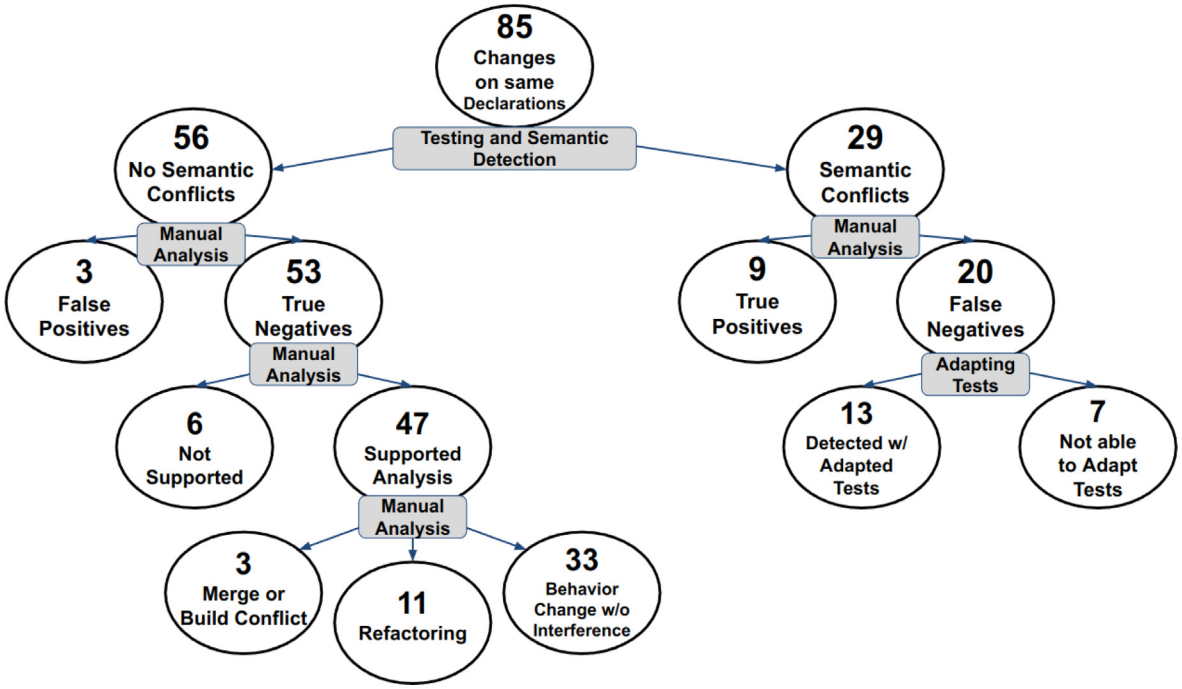

Fig. 8. Conflict detection results for our dataset of changes' pairs, using the four unit test generation tools with the three kinds of executables (original, testability and serialization). Distribution of changes (on same class element declaration), their classification, and whether a conflict is detected or not.

```
1  @Test
2  public void test1(){
3    HikariPool hikariPool = new HikariPool(hikariConfig0);
4    ...
5    assertEquals(10, hikariPool.getTotalConnections());
6  }
```

Fig. 9. Evosuite test case that detects semantic conflict.

```
1  private void addConnection(){
2  -  int retries = 0;
3  +  int retries = config.getAcquireRetries();
4     (...)
5     try{
6  +    if (retries == 0 && totalConnections.incrementAndGet()){
7        totalConnections.decrementAndGet();
8      }
9     } catch (Exception e){
10     if (retries++ > config.getAcquireRetries()){
11       totalConnections.decrementAndGet();
12     }
13    }
14 }
```

Fig. 10. Test conflict caused by changes that update the same variable.

to the interaction between *Left* and *Right* changes, which the `if` statement condition evaluate to `false`. On the other hand, multiple `decrementAndGet()` calls (line 11) are executed. The multiple calls for `decrementAndGet()` (line 11), in all evaluated commits, are motivated by an exception thrown due to mal-formed objects; however, the initialization of these objects is not impacted by the testability or serialization transformations presented in this study.

The other eight detected conflicts have common aspects with the previous example. First, the conflicts occur because the parent commits' changes impact the values stored in the same variables or object fields. So, to detect the conflict, the test revealing conflict might exercise a single specific object. Second, the test has to directly access the object involved in the conflict (side effects of the changes), having at least one assertion that explores the changed object fields.

Regarding the cases without conflicts, correctly not reported by SAM (true negatives), we observe 53 cases (left branch in Fig. 8). Six are classified as unsupported because the potentially conflicting changes occur in test cases and test classes, not in the code that implements system functionality. So even if the changes were conflicting, SAM would not be able to detect that as its current version does not support test classes; the unit test generation tools are not configured to generate test cases for test classes, as the associated testing framework and project environment are not fed to the tools, which would need to be adapted for such purpose.

For 33 of the 47 changes supported by SAM, the parent commits individually change program behavior, but when integrated the changes do not locally interfere with each other. Static analysis tools that rely on more conservative analysis could maybe err on that, but the chances of SAM erring on those cases are reduced. The same applies for the 11 cases in which one of the integrated branches only applies whitespace changes and structural refactorings such as renamings and extractions of variables and methods. In such circumstances, even if one parent commit changes program behavior, we have no interference and semantic conflict. This way, even if the generated tests present different results between the *Base* and one parent, let us say *Left*, the *Merge* commit behaves exactly like the *Left*, as *Right* does not change program behavior.

Finally, for the remaining three cases, the parent commit changes cause other kinds of conflicts during the integration (textual or build conflicts). When merge or build conflicts occur, human intervention is necessary to fix these conflicts, and their resolution often involves discarding some of the changes. This in turn likely reduces the chances of a residual semantic conflict. That is what we observed in these three cases: significant parts of the changes were discarded during the textual and build conflicts' resolutions, and so no conflict was wrongly detected by our tool.

5.2. RQ2: What are the causes of failures reported by SAM?

To answer RQ2, we explore the cases in which SAM erroneously reports or omits a conflict (false positives and negatives, respectively).

```
1 @Test
2 public void test1108(){
3     MongoClientOptions mongoClient = builder13.build();
4     try{
5         MongoClient mClient = mProp.createMongoClient();
6     } catch (IllegalStateException e) {...}
7 }
```

Fig. 11. Test case associated with false positive caused by unrelated parent conflicting contributions.

Regarding false positives, SAM might report some for some reasons. First, the generated test cases are not guaranteed to be free from flakiness. So, for instance, we can have a test case that fails in the *Base* in some executions and passes in others. SAM could then consider one of our interference criteria to hold when relying on an incorrect test result, and consequently wrongly report a conflict. To reduce this problem, we run each test suite three times in each commit, aiming to detect and discard flaky tests by comparing the results of the three test executions; if the three results are not identical, we discard the test case and do not consider it for detecting interference. However, this is not enough to eliminate flakiness in general. In fact, in our study, we still observe two (out of three) false positives due to flakiness.

The remaining case of false positives observed in this study corresponds to an interference, but not one caused by the changes in the analyzed method. As our manual analysis is local, focusing only on the analyzed method and its dependencies, we conservatively classify this case as a false positive. The parent commits change a common target method,[13] each setting the values of disjoint sets of object fields. Additionally, the *Left* commit updates the version of an external dependency. The generated test case based on the *Right* commit passes in that commit as expected (see Fig. 11), as its changes lead to an exception during execution (`IlegalStateException`). When running the same test on the *Base* commit, the expected exception is not thrown and the test fails. Finally, the test fails again when run against the *Merge* commit, but this time due to the new external dependency version integrated by *Right*, which leads to a different unexpected exception. This way, we have a test case that satisfies one of our conflict criteria, but only part of the failed states are caused by the parent changes on the common target method.

Another reason for false positives is that our interference criteria are simply approximations, as interference is not computable in general. SAM uses them regardless of the characteristics of the generated tests, but the criteria are guaranteed to be valid only if the test assertions solely explore the state elements affected by the changes of the parent commit that was used to generate the corresponding test. For example, consider a merge scenario with *Left* and *Right* commits that simply change how a disjoint set of state elements is updated. Say *Left* adds the assignment `left=1` and *Right* adds `right=1`, whereas both variables were initialized with 0 in *Base*. If SAM generates a test for *Left* asserting `left==1 & right==0`, this test breaks in *Base* (as `left` evaluates to 0), passes in *Left*, and breaks in *Merge* (as `right` evaluates to 1). As this test satisfies our criteria, SAM wrongly reports interference even though the integrated changes do not affect each other. The false positive is caused by the test assertion that unnecessarily constrains the value of the `right` variable, which is not affected by *Left*'s change. We, however, observed no such cases in our sample.

Regarding false negatives, we observe 20 cases, as reported in the right branch of Fig. 8. Next, we discuss the limitations behind these missed conflicts by manually analyzing the generated test suites and checking whether the conflicts could be detected after applying a few changes to the test cases. Our main goal is to evaluate how close the generated tests are to detecting conflicts. To guide us during this adaptation process, we consider the test descriptions that we created when establishing ground truth (see Section 4.6). The manually adapted test cases could detect conflicts in 13 out of 20 false negative cases in our sample. This suggests that improving SAM's test generation process, or allocating more resources for generation, could maybe significantly reduce the chances of false negatives. However, detecting the other six false negatives would likely require radical changes in how SAM works.

To understand how close SAM gets to detecting conflicts, we discuss now one of the 13 false negatives. Both parent commits add calls to methods that update the same object stored in the field `logger`, as they write different values on the log output.[14] While the *Left* commit updated the message using the method `info`, the *Right* commit updates using the method `debug`. Although the parents use different writing methods to update the object, they write on the same character stream. To detect such interference, SAM would have to generate at least one test case with an assertion that explores the fields of the object changed by the target method, the object stored in `logger` in this case. However, by manually inspecting the generated test cases, we only find one such assertion, which simply checks whether the object is null. Regardless of the particularities of this scenario, a conflict revealing assertion would have to compare the contents of the object stored in `logger`, not simply that it is different from null. This case shows that a generated test case reaches the interference location, the object state is infected, and the interference is propagated (Voas, 1992), but the test case assertions fail to explore that. As such, we consider that SAM is close to detecting interference in this case, but misses it.

In other cases, SAM is not even close to detecting a conflict. The methods holding the conflict are called by the generated test cases, but with arguments that are unable to lead test execution to reach the interference location; for example, a `null` argument makes the method raise an exception before even reaching the interference location. The same happens when infection depends on more complex object graphs that not easily created by the test generation tools. In these cases, with no infection and propagation, the assertions are often far from the ones that could detect the interference, making it harder to manually adapt the generated test cases.

Another major SAM limitation we observe is when infection or even reachability would only occur if the generated test cases were able to set dependencies to external resources such as database sessions. As this is highly project-dependent, SAM would need to have access to project-specific setup information and feed them to the unit test generation tools, in order to avoid missing conflicts. The current version of SAM has no such support, reducing the chances of detecting conflicts in projects that demand external resources. In one scenario, for example, the interfering changes are made inside a block that is only executed if a valid database session is available. As no such session is set up by the generated test cases, the interference location is never reached. By adapting the test cases with Junit annotations like `Before` or `BeforeEach`, or using *mocks*, could overcome this limitation.

5.3. RQ3: What is the effect of adopting different unit test generation tools, and testability and serialization transformations?

To answer our third research question, we initially assess how each unit test generation tool invoked by SAM contributes to the overall result. Table 3 illustrates the overall results for all tools, while each column shows the number of conflicts detected by each tool when invoked with a specific kind of executable.

We observe that EvoSuite and Differential EvoSuite are the most successful tools, detecting six conflicts each, while Randoop and Randoop Clean detect two conflicts each. Most tools perform better when applied to the testability executables; only Evosuite performs better when applied to the original executables. Differential EvoSuite and Randoop are the only tools not reporting false positives (presented in

[13] Spring Boot - merge commit: 958a0a4.

[14] Spring Boot - merge commit: fdd3f12.

Table 3
Distribution of detected conflicts by unit test generation tools and executables.

Unit test tools	Executable program versions		
	Testability	Original	Serialization
Differential EvoSuite	*6 (↓3, →3)* pr. 1 re. 0.21 ac. 0.74	*3 (→3)* pr. 1 re. 0.1 ac. 0.70	0
EvoSuite	*5 (↑2, ↓4) [1]* pr. 0.83 re. 0.17 ac. 0.71	*6 (↑3, ↓5, ←1, →6)* pr. 1 re. 0.21 ac. 0.74	*1 (↑1, ↓1)* pr. 1 re. 0.03 ac. 0.68
Randoop	*2 (↑1, →1)* pr. 1 re. 0.07 ac. 0.69	*1 (→1)* pr. 1 re. 0.03 ac. 0.68	0
Randoop Clean	*2 (→1) [1]* pr. 0.66 re. 0.07 ac. 0.68	*1 (→1) [1]* pr. 0.66 re. 0.03 ac. 0.67	0

Downward arrows (↓) stand for conflicts detected by the associated unit test generation tool, but not detected by the next tool below. Upward arrows (↑) stand for conflicts detected by the associated unit test generation tool, but not detected by the next tool above. Left arrows (←) stand for conflicts detected by the associated unit test generation tool, but not detected by the next left tool. Right arrows (→) stand for conflicts detected by the associated unit test generation tool, but not detected by the next right tool. Numbers between brackets represent false positives reported by the tools. *pr.*, *re.*, and *ac.* stands for *precision*, *recall*, and *accuracy*, respectively.

square brackets in the first line of some cells). EvoSuite and Randoop Clean present one and two false positives, respectively.

None of the tools is able to detect all conflicts. Although EvoSuite and Differential EvoSuite detect the same number of conflicts, together they detect all nine conflicts detected by SAM. All conflicts detected by Randoop and Randoop Clean are also detected by the other tools. So to catch all detected conflicts of our sample (31%), combining Differential EvoSuite and EvoSuite would be enough. A version of SAM that uses only these two tools would be computationally more efficient and detect the same conflicts as the full fledged version of SAM that invokes the four unit test generation tools.

The results mostly show that the testability transformations help to detect conflicts (first column in Table 3), but not when applied together with the serialization transformation (third column). For example, after applying testability transformations, the tools could directly access object fields and, consequently, explore them in assertions. So, the use of such transformations leads to the detection of three additional conflicts not detected by the same tools when applied to the original executables; while Differential EvoSuite detects all three new conflicts, the remaining tools detect one conflict each. This observation reinforces our previous results that testability transformations help detect more conflicts (Da Silva et al., 2020). Note, however, that one conflict is not detected by EvoSuite when applied to the testability executable. In this particular case, the target method is public, so the transformations would have no effect. Regarding the addition of empty constructors, detecting some conflicts was possible after applying this transformation; in these cases, the tools focus on calling the target element instead of dealing with problems when instantiating objects. We believe Evosuite misses this conflict in this case by chance due to its randomness, not because of the chosen executable.

Regarding the use of serialization (third column in Table 3), eight out of the 20 cases in our subsample have conflicts. Although the serialization numbers are quite low, remember that these numbers derive from a much smaller sample than the original and testability numbers. From the nine detected conflict cases with original and testability executables, only one case is in the serialization subsample. Moreover, this conflicting case is also detected with the serialization executable. Nevertheless, the detection is not due to the extra serialization information available in the executable, as the EvoSuite test case that detects the conflict (third column, second row in Table 3) does not explore serialized objects. So we have no evidence that the serialization technique can help improve conflict detection, but we also have no evidence that it can hinder conflict detection, as might be suggested by the illustrated number if one does not know that they are based on a subsample. We should, though, run new studies with a bigger sample in order to better assess this issue.

Comparing our results with previous work (Da Silva et al., 2020) shows that our initial results are replicable; all four conflicts previously detected are also reported in the study reported here. We adopt the same testability transformations in both works, but a larger budget (5 min) for the test generation process. Like our previous study, all nine conflicts detected here are detected through the same conflict criterion (see our Online Appendix (2024)); a test case that passes on the parent commit and fails on the *Base* and *Merge* commits. Although no conflict is detected using our new two criteria, they are still valid as they explore scenarios not supported by our previous criteria (see Section 2). We believe our approach for generating the test suites do not favor the new criteria. As we currently generate tests based on parent commits, these tests are expected to pass on these commits, partially limiting our new two conflict criteria. To better evaluate the new criteria, we should run a new study generating tests against the *Base* and *Merge* commits instead of only parent ones.

As a final remark, based on our sample, Differential EvoSuite, together with EvoSuite, is the best configuration of tools to be adopted by SAM. Similarly, we should configure SAM to use only testability executables. Finally, regarding our proposed conflict criteria, the first criterion is the best option for detecting all conflicts reported in this study.

5.4. RQ4: How efficient are unit test generation tools in detecting behavior changes and related metrics?

Although our main focus is the detection of semantic conflicts, RQ4 evaluates related metrics regarding the quality of the generated test suites. These metrics allow us to better understand the strengths and weaknesses of our proposed extended version of Randoop, *Randoop Clean*.

Regarding detecting Behavior Changes, we are looking for test cases that present different outputs when executed against a pair of commits. This way, we might evaluate how close the tools are of detecting conflicts, in case the reported behavior change is associated with the same changed class element.

Overall, 89 behavior changes are detected by the generated test cases when using all kinds of executables we consider in our experiment. These changes involve either a parent and the *Base*, or a parent and the *Merge* commit. EvoSuite is the most successful tool detecting 47 out of 89 behavior changes. Next, we have Randoop Clean, Randoop, and Differential EvoSuite with 38, 37, and 28 detections, respectively. Even combining the last three tools, they would not achieve the rate detection of EvoSuite, as their outputs overlap. Different from the results of conflict detection, Differential EvoSuite does not rank first this time. As each tool could detect at least one exclusive behavior change, no combination of tools could capture all reported behavior changes. However, the highest detection rate not including all tools could be obtained by combining EvoSuite, Randoop Clean, and Differential EvoSuite, as they report together 85 out of 89 behavior changes.

We observe that the adoption of testability transformations helps the tools to detect behavior changes. In the same direction as for semantic conflict detection, 20 additional changes are detected with testability executables, when compared to the original executables (only 69 changes detected). From the 20 false-negative cases observed in the experiment, we could detect behavior changes in 12 of them. However, the reported behavior changes are not caused by the changes involved in the semantic conflicts. So, we cannot say that the tools were close in these cases.

We also observe that the serialization executables help detect 15 changes not covered when using the original and testability executables. In these cases, as the tools have access to realistic objects, the generated test cases may further and deeply explore the instructions and branches of the target code under analysis.

To evaluate the proposed changes applied on our extended version of Randoop, *Randoop Clean*, we consider the following metrics: *Target Code Reachability*, *Object Diversity*, and *Source Code Coverage*. Compared to Randoop, *Randoop Clean* generates more tests that directly call target methods. Regarding the diversity of objects, no matter the kind of executable used in the experiment, *Randoop Clean* often generates more diverse objects than Randoop when the time budgets are not inferior to 5 min; the larger the budget, the greater the difference in favor of *Randoop Clean*. Finally, overall, the tools present similar coverage in most cases. Likely due to our sample size and the reduced effects of the *Randoop Clean* changes — its benefits cannot be observed in cases where the tools fail to reach the target method — we observed no statistically significant difference between the tools. Furthermore, for some cases, even when *Randoop Clean* generates tests reaching the target method, the diversity of objects generated was not good enough to properly reach the conflict and observe its behavior. Similar results are observed when using original and testability transformations executables. However, using executables with serialized objects leads to higher coverage. Thus, we may conclude that the quality of serialized objects allows both tools to explore instructions of the target code under analysis more deeply.

6. Discussion

Semantic merge tools based on regression testing, as we evaluate 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 semantic merge tool to detect semantic conflicts. They should still try to catch such conflicts by reviewing the code and executing project tests.

Although we evaluate the use of SAM with four unit test generation tools, our results show that combining EvoSuite and Differential EvoSuite would be the best option to detect all conflicts in our sample. Configuring SAM that way we might spend less time generating tests and detecting conflicts. Although we present four conflict criteria, not all of them detected conflicts in our study. However, we would not suggest a version of SAM that only applies a subset of the criteria, as checking them is not expensive.

Although the proposed testability transformations are not sound, in our sample we observe that the transformations contribute to increasing the testability of the code under analysis without drawbacks, allowing the tools to directly access and call all elements of a target class.

We believe the adoption of serialization may support the detection of conflicts, though we provide no such evidence in our study, likely due to the restricted subsample we consider for evaluating the serialization transformations. As the quality and coverage of project test suites play an essential role in providing diverse serialized objects, projects with weak test suites will not benefit from serialization. In our evaluation, only nine conflicting merge scenarios were associated with project test suites covering the target methods where the conflicts takes place. For the sample of nine scenarios with conflicts, we observe strong test suites associated with eight cases, covering the target method and providing more than 100 serialized objects for each case. However, these objects were not diverse enough to reach the infection state.

A special benefit of our regression testing approach to detect conflicts is that one ends up with a test case that reveals a conflict, when the tool reports one. This decreases 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, and can be used for debugging and understanding the mechanics of the conflict. Finally, the test case could also help making sure the conflict is fixed. This contrasts with static analysis approaches, which report a conflict and maybe the statements involved in the conflict mechanics, but provide no test case.

Although our study and results are restricted to Java, we could use a similar approach for other languages with support for unit test generation tools like those we use here.

As a final remark, although SAM focuses on detecting semantic conflicts, it does not support fixing the conflicts due to the particularities of this conflict type. Different from build conflicts (Da Silva et al., 2022), test conflicts are harder to fix as developers must take into account the semantics of the integrated changes that cause the conflict. This way, applying straightforward changes, like those adopted for build conflict fixes, is often not enough in this context. As a result, to fix test conflicts, developers must be aware of the program specifications or intents, and then, they might apply the required changes aiming to meet that.

6.1. Improving SAM

Our evaluation of SAM reveals improvements that might be implemented in future versions of our tool. We also discuss a number of usage scenarios for SAM, and different contexts in which our tool might be adopted.

Using SAM in a reactive way to detect conflicts

Knowing that SAM requires significant computing resources to generate the test suites and execute them on each commit of a merge scenario, SAM's usage by individual developers might require support from a server that runs the analysis without blocking local repository activities. So a developer would merge locally and move on while the analysis is performed on the server. The developer is later notified.

Alternatively, SAM can also be integrated to continuous integration pipelines, in such a way that contributions to be integrated into a main remote project repository are first analyzed by SAM before integration actually takes place. These are just two contrasting usage scenarios, but the tool could fit other scenarios as well.

Reusing original project test suites as input for SAM

For projects with solid and robust test suites, we could use a configurable version of SAM that extends the current version in such a way that the detection of conflicts relies not only on generated tests, but also on existing and manually created project test suites. This combination of generated and existing project tests could increase the potential to detect conflicts.

We also envision a version of SAM that generates test suites not only for the parent commits in a merge scenario, but also for the *Base* and *Merge* commits. If the interference criteria applies for these test suites, we could similarly report conflict. This could also increase the potential of detecting conflicts, but further studies are necessary to assess that.

6.2. Improving unit test generation

We observed a few limitations and weaknesses of generated unit tests in our specific context. For instance, in a few cases the generated tests are not able to create complex objects with internal or external dependencies. As presented in Section 4, we try to address some of these limitations by applying testability transformations in the code under analysis. Although this helps, a number of limitations persist. By manually analyzing generated test suites of false-negative cases, we were able to better understand the limitations. We discuss the main ones in the following.

Reaching interference location through relevant object creation

The unit test generation tools have a hard time creating tests that manipulate objects that need to be directly or indirectly exercised in order to detect a conflict. Many test cases prematurely finish their executions due to failed attempts to access fields or call methods on objects that are not properly initialized or configured.[15] The attempt to access fields through a variable `textNode0` throws a `NullPointerException`, since this variable is not properly instantiated with a valid object. Giving relevant objects for test cases like this is necessary to at least reach the interference location. This topic is also related to cases in which the *methods holding the conflict are not adequately called*. In these cases, besides the proper creation of relevant objects, a sequence of method calls must be invoked with the aim of properly instantiating an object in such a way that the interference location can be reached.

Relevant assertions exploring the propagated interference

In a few cases, the generated tests even reach the interference location, infection occurs, but the test assertions do not properly explore the *propagated* interference. For example, consider a conflict in the project CloudSlang,[16] where the parent commits change the same `array`. The test case generated for this scenario correctly creates the object, calls the method in which the conflict occurs, and saves the method return object into a local variable. However, the generated assertion checks whether the local variable is `null` instead of exploring the object size. So, depending on the type of method return object, the unit test generation tools could explore defined aspects that could detect the conflict. This way, tools could provide a list of handlers based on the types of objects under analysis. For example, for array objects, these handlers would force the assertions to explore their size and contents. For strings, assertions might explore comparisons between different strings, as well as whether substrings are part of others, and so on.

Relevant assertions relying on interference propagation

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

7. Threats to validity

Construct Validity As explained in Section 2, 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 the number of false negatives and false positives with respect to a global notion of interference could be different than what 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 apply testability transformations before performing our analysis. For example, we change access modifiers to `public`. Although this transformation breaks program encapsulation, it does not semantically change a program. 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, but we are aware of this and take it into consideration in our false positive analysis. Furthermore, the transformed program is only accessed by our semantic merge tool.

Internal Validity When creating the interference ground truth, we rely on manual analysis of unfamiliar source code. Although we cannot generalize our results due to our sample size, when compared with related work, overall, we analyze a similar or larger number of merge scenarios. Furthermore, we also provide a dataset that can be used for replications or new studies. To reduce this threat, we involve a group of six researchers, which are split in pairs during the analysis, and demand 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 are successful, as the threat can be minimized by analyzing the interference revealing test case, running it, and manually checking whether the test case assertions focus on the changed state elements.

External Validity Our results are limited to the context of open-source GitHub Java projects. In the same way, the diversity of real projects we analyze here might have an impact on the detection of potential conflicts by our semantic tool. The testability transformations, as we discuss, 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 testability transformations, if applicable.

8. Related work

In this section, we discuss the related work to our study. First, we focus on works targeting semantic conflict detection, as we do with SAM. Next, we present studies exploring regression testing.

8.1. Semantic conflict detection

Assistive tools

Researchers also present techniques to detect and prevent conflicts early. Palantir (Sarma et al., 2012) is a workspace awareness tool that notifies developers of parallel changes in the same artifact. Brun et al. (2013) propose incorporating speculative analysis for early detection and prevention of conflicts. This way, they present Crystal, an assistive tool that compares remote and local individual collaborators' repositories in order to warn about possible code integration conflicts. To detect test conflicts, they evaluate their technique by analyzing three Java projects and rely on project tests, which are often not enough for detecting interference as we explore here. The authors do not mitigate possible flaky tests in both studies, as we do in our study by executing the test suites multiple times. 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. These studies have also investigated ways in which conflicts can be prevented early, thereby minimizing their impact on productivity.

Cavalcanti et al. (2017), Tavares et al. (2019) and Cavalcanti et al. (2019) conduct empirical studies that analyze merge scenarios and compare the accuracy of different merge resolution techniques: unstructured, semi-structured, and structured merge. Contrasting with our investigation here, their proposed tools are not able to detect behavioral semantic conflicts, only syntactic and static semantics conflicts.

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

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

Conflict detection through static analysis

Wuensche et al. (2020) suggest an approach based on static analysis and a tool to detect and predict the occurrence of test conflicts, as they formally call *higher-order merge conflicts*. Based on the changes performed during a merge scenario, they (re)build a call graph and detect potential dependencies among merge scenario code fragments that lead to a conflict. To detect test conflicts, the authors manually analyze build records of merge scenarios and extract change patterns that lead to test conflicts based on the authors' observation. As a result, 22 potential conflicts out of 1489 merge scenarios are reported by the tool. To validate the potential conflicts, the authors search for bugs reported after each merge scenario occurrence. However, they do not confirm conflict occurrences as no bug is reported.

Sousa et al. (2018) 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 conclude 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. Due to the experimental design and dataset characteristics, the authors do not present false negative rates. However, concerning false positives, the authors disclose a rate of 3.8% (2 out of 52 cases), whereas our study demonstrates a rate of 3.5% (3 out of 85 cases).

de Jesus et al. (2023) investigate the detection of semantic conflicts using static analysis techniques. For that, the authors implement different algorithms based on the Soot framework and perform an empirical study evaluating 99 cases of candidate scenarios with conflicts. When compared to our results, the authors report that their approach shows better results regarding F1 score and recall. However, when we compare their results regarding precision and accuracy, SAM outperforms them. Combining these two approaches might represent a good way to overcome their individual limitations and achieving better results, based on the evaluated metrics discussed.

Conflict detection through dynamic analysis

Nguyen et al. (2015) present Semex, a tool for detecting which combination of merged changes causes a test conflict based on a technique called variability-aware execution (Nguyen et al., 2014). 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. 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. 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.

Tiwari et al. (2021) present PANKITI, a related approach regarding the use of serialization to support unit test generation tools. Unlike our approach to serializing objects based on a target method during merge scenarios, PANKITI monitors an application in production serializing objects when target methods are called. While we serialize the current objects holding the target method and its required parameters, the authors also serialize the returning target method objects. For our context, we are not interested in returning objects as we focus on objects that might let us reach infection states of conflicting contributions. Furthermore, infections are not always propagated through returning target method objects; in our sample, we observe infections being propagated through parameters, for example.

8.2. Regression testing

Regression testing has been used for detecting behavior changes in the past. Evans and Savoia (2007) combine regression and progressing testing (differential testing) to detect preserved, altered, and eliminated behavior of a program. Jin et al. (2010) present a test generator based on a list of changed classes between two versions of a program. Shamshiri et al. (2013) 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).

Campos et al. (2014) propose an approach (CTG) to more efficiently generate unit tests considering the whole project, instead of a single method or class as we do here. They focus on a *Continuous Integration* context, and their approach can help detect behavior changes and regressions, as tests generated in a previous commit might yield a different result when executed in the next commit. But this is not enough for detecting interference as we do here, as our interference heuristic goes beyond behavior change detection; a test that was generated and passes in a parent commit, and that breaks in the following (say merge) commit, might indicate interference only if it breaks in the corresponding base commit. It would, however, be important to use CTG, instead of raw Evosuite focused on a target method as we do here, to assess whether it could improve SAM. Hejderup and Gousios (2022) investigate the effectiveness of project test suites on detecting semantic changes motivated by updates to external dependencies instead of conflicting contributions applied to the project itself, as we do here.

Arcuri and Galeotti (2021) adopt a related approach by presenting a set of testability transformations; unlike our transformations, they do not focus on changing code element access modifiers or semantic changes but support and guide the search algorithm when generating tests. Their core idea is based on *Method Replacements*, which replace specific method calls at the bytecode level with their customized methods. To evaluate their technique, they implement it as an extension of the EvoMaster tool and perform an empirical study analyzing ten Rest web service projects (open-source and industrial ones). The results show that the techniques effectively improve code coverage and fault detection.

Regarding the issues we observe by the tools when generating test suites, previous studies also bring evidence about the hardness of dealing with complex objects (Fraser and Arcuri, 2015; Silva et al., 2017; Da Silva et al., 2020). These related studies discuss the difficulty of generating complex objects, which are required when calling specific methods under analysis. By complex objects, we consider objects with multiple other objects from internal as also external dependencies. In order to address these issues, we propose feeding the tools with serialized objects leading them to reuse previous objects originally created by the original project test suite.

9. Conclusion

In this work, we present and evaluate a semantic merge conflict detection technique using automated test-case generation. As opposed to prior attempts in the literature, our strategy does not require explicitly defined behavior specifications or substantial setup effort. We define interference criteria and systematically investigate their effectiveness by detecting conflicts upon a manually curated ground-truth dataset originating from 85 changes' pairs from 51 software merge scenarios that integrate changes to the same method, constructor, or field declaration mined from GitHub.

In order to detect conflicts, we combine unit test generation tools and adopt improvements, such as testability and serialization transformations. As a result, we show the feasibility of a semantic merge tool,

SAM (SemAntic Merge tool). While SAM is able to detect nine conflicts out of 29 conflicts from 85 changes' pairs, we report only three false positives according to our interference criteria. This suggests that semantic merge tools based on unit test generation would help developers detect semantic conflicts early, otherwise reaching end-users as failures. Our results show that SAM performs best when combining only the tests generated by Differential EvoSuite and EvoSuite. The testability transformations improve the testability of target code under analysis in three of the nine 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 a replication package (Online Appendix, 2024), also to help building future semantic merge tool.

We also explore and measure the impact of different improvements in our semantic merge-conflict detection technique due to the limitations of unit test generation tools and the complexity of the target code under analysis. First, we propose and evaluate the use of serialized objects as input for the tools during the generation process. Although we do not observe new semantic conflicts detected after applying this technique, new general behavior changes are detected involving pairs of commits. Second, we also extend Randoop aiming to maximize the number of tests exploring the target method when applicable. Although our tool Randoop Clean reports better results regarding test suites dealing with more diverse objects, we do not observe the detection of new conflicts. As future work, we plan to extend SAM to consider original project test suites to detect conflicts based on our conflict criteria, improve the use of serialization transformations, and extend unit test generation tools.

CRediT authorship contribution statement

Léuson Da Silva: Data curation, Conceptualization, Investigation, Methodology, Software, Writing – original draft, Writing – review & editing. **Paulo Borba:** Conceptualization, Funding acquisition, Methodology, Supervision, Validation, Writing – original draft, Writing – review & editing. **Toni Maciel:** Data curation, Investigation. **Wardah Mahmood:** Investigation, Writing – original draft, Writing – review & editing. **Thorsten Berger:** Funding acquisition, Supervision, Writing – original draft, Writing – review & editing. **João Moisakis:** Investigation. **Aldiberg Gomes:** Investigation. **Vinícius Leite:** Investigation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The dataset and scripts used to run our study are available in our Online Appendix (2024).

Acknowledgments

We thank Marcelo d'Amorim, Rohit Gheyi, Leonardo Fernandes, Breno Miranda, Leonardo Murta, and the anonymous reviewers for valuable comments to improve an earlier version of this paper. We thank Rafael Alves, Galileu Santos, Matheus Barbosa, and Thaís Burity for their 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) and the Wallenberg Academy.

References

Accioly, P., Borba, P., Cavalcanti, G., 2018. Understanding semi-structured merge conflict characteristics in open-source Java projects. Empir. Softw. Eng. 23 (4), 2051–2085.

Adams, B., McIntosh, S., 2016. Modern release engineering in a nutshell–why researchers should care. In: International Conference on Software Analysis, Evolution, and Reengineering. IEEE.

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

Apel, S., Leßenich, O., Lengauer, C., 2012. Structured merge with auto-tuning: balancing precision and performance. In: International Conference on Automated Software Engineering. ACM.

Apel, S., Liebig, J., Brandl, B., Lengauer, C., Kästner, C., 2011. Semistructured merge: rethinking merge in revision control systems. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM.

Arcuri, A., Galeotti, J.P., 2021. Enhancing search-based testing with testability transformations for existing apis. ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (1), 1–34.

Barros Filho, R.S., 2017. Using Information Flow to Estimate Interference Between Developers Same-Method Contributions (Master's thesis). Universidade Federal de Pernambuco.

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

Binkley, D., Horwitz, S., Reps, T., 1995. Program integration for languages with procedure calls. ACM Trans. Softw. Eng. Methodol. (TOSEM) 4 (1), 3–35.

Bird, C., Zimmermann, T., 2012. Assessing the value of branches with what-if analysis. In: Symposium on the Foundations of Software Engineering. ACM.

Brun, Y., Holmes, R., Ernst, M.D., Notkin, D., 2013. Early detection of collaboration conflicts and risks. IEEE Trans. Softw. Eng. 39 (10), 1358–1375.

Campos, J., Arcuri, A., Fraser, G., Abreu, R., 2014. Continuous test generation: Enhancing continuous integration with automated test generation. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 55–66.

Cavalcanti, G., Borba, P., Accioly, P., 2017. Evaluating and improving semistructured merge. ACM Trans. Program. Lang. Syst. 1 (OOPSLA), 59:1–59:27.

Cavalcanti, G., Borba, P., Seibt, G., Apel, S., 2019. The impact of structure on software merging: semistructured versus structured merge. In: International Conference on Automated Software Engineering. IEEE.

Da Silva, L., Borba, P., Mahmood, W., Berger, T., Moisakis, J., 2020. Detecting semantic conflicts via automated behavior change detection. In: International Conference on Software Maintenance and Evolution. IEEE, pp. 174–184. http://dx.doi.org/10.1109/ICSME46990.2020.00026.

Da Silva, L., Borba, P., Pires, A., 2022. Build conflicts in the wild. J. Softw.: Evol. Process 34 (4), e2441.

de Jesus, G.S., Borba, P.H.M., de Almeida, R.B., de Oliveira, M.B., 2023. Detecting semantic conflicts using static analysis. arXiv preprint arXiv:2310.04269.

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

Dias, K., Borba, P., Barreto, M., 2020. Understanding predictive factors for merge conflicts. Inf. Softw. Technol. 121, 106256.

Elbaum, S., Chin, H.N., Dwyer, M.B., Dokulil, J., 2006. Carving differential unit test cases from system test cases. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 253–264.

Evans, R.B., Savoia, A., 2007. Differential testing: a new approach to change detection. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM.

Fowler, M., 2017. Feature toggle. accessed: December 2017. URL https://goo.gl/QfJ6mM.

Fraser, G., 2018. A tutorial on using and extending the evosuite search-based test generator. In: Search-Based Software Engineering. Springer.

Fraser, G., Ammann, P., 2008. Reachability and propagation for ltl requirements testing. In: 2008 the Eighth International Conference on Quality Software. IEEE, pp. 189–198.

Fraser, G., Arcuri, A., 2012. Whole test suite generation. IEEE Trans. Softw. Eng. 39 (2), 276–291.

Fraser, G., Arcuri, A., 2015. 1600 Faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. Empir. Softw. Eng. 20 (3), 611–639.

Grinter, R.E., 1996. Supporting articulation work using software configuration management systems. Comput. Support. Coop. Work 5 (4), 447–465.

Hejderup, J., Gousios, G., 2022. Can we trust tests to automate dependency updates? a case study of java projects. J. Syst. Softw. 183, 111097.

Henderson, F., 2017. Software engineering at Google. accessed: December 2017. URL https://arxiv.org/abs/1702.01715.

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

Horwitz, S., Prins, J., Reps, T., 1989. Integrating noninterfering versions of programs. ACM Trans. Program. Lang. Syst. 11 (3), 345–387.

Jacoco, 2022. available at: https://www.eclemma.org/jacoco/.
Jin, W., Orso, A., Xie, T., 2010. Automated behavioral regression testing. In: International Conference on Software Testing, Verification and Validation. IEEE.
Kasi, B.K., Sarma, A., 2013. Cassandra: proactive conflict minimization through optimized task scheduling. In: International Conference on Software Engineering. IEEE.
Khanna, S., Kunal, K., Pierce, B.C., 2007. A formal investigation of diff3. In: International Conference on Foundations of Software Technology and Theoretical Computer Science. Springer-Verlag.
Luo, Q., Hariri, F., Eloussi, L., Marinov, D., 2014. An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 643–653.
Mahmood, W., Chagama, M., Berger, T., Hebig, R., 2020. Causes of merge conflicts: A case study of elasticsearch. In: International Working Conference on Variability Modelling of Software-Intensive Systems. ACM.
McKee, S., Nelson, N., Sarma, A., Dig, D., 2017. Software practitioner perspectives on merge conflicts and resolutions. In: International Conference on Software Maintenance and Evolution. IEEE.
Mens, T., 2002. A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng. 28 (5), 449–462.
Nagappan, M., Zimmermann, T., Bird, C., 2013. Diversity in software engineering research. In: Joint Meeting on Foundations of Software Engineering. ACM, pp. 466–476. http://dx.doi.org/10.1145/2491411.2491415.
Nguyen, H.V., Kästner, C., Nguyen, T.N., 2014. Exploring variability-aware execution for testing plugin-based web applications. In: International Conference on Software Engineering. IEEE.
Nguyen, H.V., Nguyen, M.H., Dang, S.C., Kästner, C., Nguyen, T.N., 2015. Detecting semantic merge conflicts with variability-aware execution. In: Symposium on the Foundations of Software Engineering. ACM.
Online Appendix, 2024. available at: https://spgroup.github.io/papers/sam-semantic-merge-tool.html.
Owhadi-Kareshk, M., Nadi, S., Rubin, J., 2019. Predicting merge conflicts in collaborative software development. In: International Symposium on Empirical Software Engineering and Measurement.
Pacheco, C., Ernst, M.D., 2007. Randoop: feedback-directed random testing for Java. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications. ACM.
Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T., 2007. Feedback-directed random test generation. In: International Conference on Software Engineering. IEEE.
Perry, D.E., Siy, H.P., Votta, L.G., 2001. Parallel changes in large-scale software development: an observational case stud. ACM Trans. Softw. Eng. Methodol. 10 (3), 308–337.
Potvin, R., Levenberg, J., 2016. Why Google stores billions of lines of code in a single repository. Commun. ACM 59 (7), 78–87.
Sarma, A., Redmiles, D.F., Van Der Hoek, A., 2012. Palantir: Early detection of development conflicts arising from parallel code changes. IEEE Trans. Softw. Eng. 38 (4), 889–908.
Shamshiri, S., 2015. Automated unit test generation for evolving software. In: Proceedings of Foundations of Software Engineering. pp. 1038–1041. http://dx.doi.org/10.1145/2786805.2803196.
Shamshiri, S., Fraser, G., Mcminn, P., Orso, A., 2013. Search-based propagation of regression faults in automated regression testing. In: International Conference on Software Testing, Verification and Validation. IEEE.
Shen, B., Zhang, W., Zhao, H., Liang, G., Jin, Z., Wang, Q., 2019. Intellimerge: A refactoring-aware software merging technique. ACM Trans. Program. Lang. Syst. 3 (OOPSLA).
Silva, I.P., Alves, E.L., Andrade, W.L., 2017. Analyzing automatic test generation tools for refactoring validation. In: 2017 IEEE/ACM 12th International Workshop on Automation of Software Testing. AST, IEEE, pp. 38–44.
Sousa, M., Dillig, I., Lahiri, S.K., 2018. Verified three-way program merge. ACM Trans. Program. Lang. Syst. 2 (OOPSLA), 1–29.
Tavares, A.T., Borba, P., Cavalcanti, G., Soares, S., 2019. Semistructured merge in JavaScript systems. In: International Conference on Automated Software Engineering. IEEE.
Tiwari, D., Zhang, L., Monperrus, M., Baudry, B., 2021. Production monitoring to improve test suites. IEEE Trans. Reliab..
Voas, J.M., 1992. Pie: A dynamic failure-based technique. IEEE Trans. Softw. Eng. 18 (8), 717.
Wąsowski, Andrzej, Berger, Thorsten, 2023. Domain-specific languages: effective modeling, automation, and reuse. Springer.
Wuensche, T., Andrzejak, A., Schwedes, S., 2020. Detecting higher-order merge conflicts in large software projects. In: International Conference on Software Testing, Validation and Verification. IEEE.
Zimmermann, T., 2007. Mining workspace updates in cvs. In: International Conference on Mining Software Repositories. IEEE.

Leuson Da Silva is a Postdoctoral Fellow at Polytechnique Montreal (Canada). He received his Ph.D. in Computer Science, focusing on Software Engineering, from the Federal University of Pernambuco, Brazil, in 2022. He holds a Bachelor's degree in Software Engineering and a Master's in Computer Science. He has published at prestigious international software engineering conferences and journals, including MSR, ICSME, and JSEP. He has reviewed papers at conferences as a sub-reviewer and as a main reviewer for journals (JSEP, JSS, TSE). His research interests are focused on the following areas: code integration conflicts, software modularity, software engineering for machine learning, and empirical software engineering.

Paulo Borba is a Professor of Software Engineering at the Informatics Center of the Federal University of Pernambuco, Brazil, where he leads the Software Productivity Group. He is a member of the ACM and of the Brazilian Computer Society. His main research interests are in the following topics and their integration: advanced (semi-structured, structured, semantic) code merging tools, code integration conflicts, continuous integration and deployment, software modularity, software product lines, and refactoring.

Toni Maciel has a B.Sc. in Computer Science from the Informatics Center of the Federal University of Pernambuco, Brazil, where he is a member of the Software Productivity Group. His main research interests are in the following topics and their integration: unit test generation tools, code integration conflicts, and software testing.

Wardah Mahmood is a Ph.D. Student in Computer Science from the University of Gothenburg, Sweden. She has published at prestigious international software engineering conferences and journals, including ICSE, ICSME, SPLC, and EMSE. Her research interests are focused on the following areas: software modularity, software product lines, and empirical software engineering.

Thorsten Berger is a Professor in Computer Science at Ruhr University Bochum in Germany. After receiving the Ph.D. degree from the University of Leipzig in Germany in 2013, he was a Postdoctoral Fellow at the University of Waterloo in Canada and the IT University of Copenhagen in Denmark, and then an Associate Professor jointly at Chalmers University of Technology and the University of Gothenburg in Sweden. He received competitive grants from the Swedish Research Council, the Wallenberg Autonomous Systems Program, Vinnova Sweden (EU ITEA), and the European Union. He is a fellow of the Wallenberg Academy — one of the highest recognitions for researchers in Sweden. He received two best-paper and two most influential-paper awards. His service was recognized with distinguished reviewer awards at the tier-one conferences ASE 2018 and ICSE 2020, and at SPLC 2022. His research focuses on software product line engineering, AI engineering, model-driven engineering, and software analytics. He is co-author of the textbook on Domain-Specific Languages: Effective Modeling, Automation, and Reuse.

João Moisakis has a B.Sc. in Computer Science from the Informatics Center of the Federal University of Pernambuco, Brazil, where he is a member of the Software Productivity Group. His main research interests are in the following topics and their integration: unit test generation tools, code integration conflicts, and software testing.

Aldiberg Gomes has a B.Sc. in Computer Science from the Informatics Center of the Federal University of Pernambuco, Brazil, where he was a member of the Software Productivity Group. Currently, he is a Full Stack Engineer at Bravi, Brazil.

Vinícius Leite has a B.Sc. in Computer Science from the Informatics Center of the Federal University of Pernambuco, Brazil, where he was a member of the Software Productivity Group. Currently, he is a Systems Analyst at Azul Seguros, Brazil.