Automated Software Engineering (2026) 33:40
https://doi.org/10.1007/s10515-025-00580-y

Comparing static analyses for improved semantic conflict detection

Galileu Santos de Jesus[1] · **Paulo Borba**[1] · **Rodrigo Bonifácio**[2] · **Matheus Barbosa de Oliveira**[1]

Received: 28 May 2025 / Accepted: 18 November 2025 / Published online: 23 December 2025

Abstract
Version control systems are essential in software development, allowing teams to collaborate simultaneously without interfering with each other's work. Tools like Git facilitate code integration through merge operations, which automatically detect textual conflicts. However, these systems focus solely on source code differences, overlooking more complex semantic conflicts that can lead to failures or unexpected behavior after integration. To address this challenge, static analysis emerges as an effective solution, detecting semantic conflicts that traditional merge tools might miss, providing an additional layer of security and quality to the code integration process. In this study, we explore combinations of static analysis techniques to improve the detection of semantic conflicts. Our approach was evaluated on a dataset from 32 real-world GitHub projects, all manually labeled to include ground truth information. These outcomes highlight the adaptability of our approach: in contexts where minimizing false positives is essential, high-precision techniques can be prioritized; in contrast, recall-focused techniques are preferable for broader conflict coverage. The results show that combining static analysis strategies delivers superior performance in terms of precision, recall, F1 score, and accuracy compared to previous methods, and is a more lightweight and flexible approach to adapt to the application context.

Keywords Software merging · Interference · Configuration management · Software evolution · Static analysis

1 Introduction

As software projects evolve and become increasingly complex and collaborative, effectively managing code integration issues becomes a crucial task. While textual conflicts are well-known and widely addressed by existing tools, they are not the only problems that may arise during the integration process.

Extended author information available on the last page of the article

In collaborative environments where multiple developers work on different parts of a codebase, inconsistencies can emerge that go undetected by traditional line-based merge tools. These issues, known as semantic conflicts, occur when changes that are syntactically correct nonetheless introduce unintended behaviors or inconsistencies. Semantic conflicts can be further classified into static and dynamic types, each requiring more sophisticated analysis techniques to be identified (Sarma et al. 2011; Brun et al. 2013; Towqir et al. 2022; Zhang et al. 2022; Sung et al. 2020).

Textual merge tools, by design, operate on a line-by-line basis and cannot recognize logically incompatible changes that are not contiguous in the source code. For instance, if one developer modifies a method's signature and another, several lines apart, adds a call to the original version of that method, the merged code will compile with no warnings - even though it results in a broken build. This represents a static semantic conflict (Sarma et al. 2011; Brun et al. 2013; Towqir et al. 2022; Zhang et al. 2022; Sung et al. 2020).

A more challenging category involves dynamic semantic conflicts, which are not detected by either textual or syntactic tools. These arise when one developer's change modifies a state element.[1] In such cases, although the code merges and builds successfully, it may behave incorrectly at runtime (Horwitz et al. 1989; Yang et al. 1992; Shao et al. 2009; Brun et al. 2013; Pastore et al. 2017; Barros Filho 2017; Sousa et al. 2018; Da Silva et al. 2020; Zhang et al. 2022). We adopt a common terminology based on the process phase in which a conflict is detected. In this one, *merge* conflict corresponds to textual conflict. *Build* conflict corresponds to syntactic and static semantic conflicts. *Test* and *production* conflicts (and undetected ones) correspond to dynamic semantic conflicts.

Dynamic semantic conflicts, hereafter simply semantic conflicts, can negatively impact development productivity and the quality of software products, especially in projects involving divergent forks (Sung et al. 2020; Zhang et al. 2022) but also in projects with a single remote repository. As project tests and code reviews often fail to detect such conflicts, researcher (Horwitz et al. 1989; Binkley et al. 1995; Barros Filho 2017; Sousa et al. 2018; Da Silva et al. 2020) have proposed static analyses to detect them. In fact, as developer's desire (specification) is hard to capture and is often not available for automated tools, these techniques simply try to detect interference (we present a motivating example in Section 2). The techniques based on theorem proving (Sousa et al. 2018) and static analysis with System Dependence Graphs (SDGs) (Binkley et al. 1995; Barros Filho 2017) are computationally expensive. Although an existing technique based on testing (Da Silva et al. 2020; Silva et al. 2023; Da Silva et al. 2024) has been proven less expensive, it suffers from low recall.

As such, previous research explored an alternative approach for approximating interference when merging changes made by two developers (De Jesus et al. 2024; De Jesus et al. 2023): the use of lightweight static analysis simply on the merged version of the code—which is automatically annotated with line changing informa-

[1] State elements are program entities that hold or represent program state during execution, as any program variable or data structure that can be read from or written to during program execution. We consider the following state elements in Java: *instance fields*, *static fields*, *local variables*, *method parameters*, and *array elements*.

tion indicating instructions modified or added by each developer. Four variations of lightweight static analyses used in those studies (Interprocedural Direct Flow, Intraprocedural Direct Flow, Control Dependence and Program Dependence Graph) aim to identify data and control flows between instructions changed by both developers that contributed to a merge scenario. When one of the analsyis do find a data or control flow dependency, it reports an interference approximation. In those previous studies, the authors compared the performance of their lightweight static analysis to other approaches based on testing, formal verification, and a static analysis technique that requires the construction of a System Dependent Graph and considers the three versions of the system involved in a merge scenario. However, their empirical assessment considered only a single combination of these analyses by applying a logical OR to their results, reporting interference whenever at least one of the techniques suggest a conflict. This approach led to a high number of detected interferences, but it also resulted in a low precision, as many of these reported interferences were false positives.

To gain a deeper understanding of how lightweight static analysis can detect semantic conflicts, here we explore multiple analyses and combinations of the approach proposed in De Jesus et al. (2024), de Jesus et al. (2023). This enables a comprehensive assessment of their individual and combined effectiveness. To this end, we report the results of an experiment evaluating the *accuracy* and *computational efficiency* of different analysis in detecting interference. Our motivation stems from the fact that while static analysis techniques are generally fast in identifying semantic conflicts (De Jesus et al. 2024; De Jesus et al. 2023), combining certain methods may not always yield optimal performance. To determine the most effective configuration, we systematically explore all possible combinations of the nine analyses derived from five selected analyses, as discussed in Section 3. This approach allows us to identify the combination that achieve the best balance between accuracy and computational efficiency. Our study is guided by three key research questions: (1) How do different static analysis techniques lead to false positives when detecting semantic conflicts in a merge scenario? (2) Which combinations of static analysis techniques achieve the best performance for each metric in detecting semantic conflicts in a merge scenario? (3) How does the execution time of different static analysis techniques and their combinations impact the overall performance and scalability in detecting semantic conflicts in a merge scenario? We detail the setup of our empirical evaluation in Section 4.

The results of our analysis highlight the effectiveness of lightweight static analysis in detecting semantic conflicts, as detailed in Section 5. Our experiments validate the ability of these analyses to detect interference and their potential for identifying dynamic semantic conflicts. After showing how specific combinations of analyses optimize both accuracy and computational efficiency, we provide actionable insights for development teams (Section 6). Depending on the context, developers can prioritize precision, recall, or a balanced trade-off to best meet their needs. Integrating these analyses into development workflows represents a significant step toward reducing production errors and enhancing the overall reliability and robustness of software systems.

2 Motivating example

Here we present an motivating example to illustrate the importance of determining the ideal combinations of the set of lightweight static analysis algorithms (De Jesus et al. 2024; De Jesus et al. 2023) to identify semantic conflicts. Consider the merge commit code in the Fig. 1, which integrates changes made by two developers, say Left and Right, to a Base commit. In Line 6, highlighted in red, we see Left's change. In Line 8, highlighted in blue, we see Right's change. The rest of the code originates from the Base commit, which is the most recent common ancestor of the two integrated versions. No textual conflict happens in this case, since the changes made by Left and Right were not in the same or consecutive lines.

The resulting code declares the `Text` class, which has `text`, `words`, `spaces`, `comments`, and `fixes` fields representing the strings associated with objects of this class. The `Text` class also declares the `generateReport` method, which generates a report based on some text analysis. It does this by calling three dif-

```
1  class Text {
2    String text ;
3    int words, spaces, comments, fixes;
4
5    void generateReport() {
6      countDupWhiteSpaces(); //modify the variable fixes
7      countComments();
8      countDupWords(); //modify the variable fixes
9    }
10
11   void countDupWhiteSpaces() {
12     int countDupSpaces = 0;
13     for (int i = 0; i < text.length() - 1; i++) {
14       if (text.charAt(i) == '␣' && text.charAt(i+1) == '␣') {
15         countDupSpaces++;
16       }
17     }
18     fixes = fixes + countDupSpaces;
19   }
20
21   void countDupWords() {
22     String[] words = text.split("\\s+");
23     int consecutiveDups = 0;
24
25     for (int i = 0; i < words.length - 1; i++) {
26       if (words[i].equals(words[i+1])) {
27         consecutiveDups++;
28       }
29     }
30     fixes = fixes + consecutiveDups;
31   }
32  ...
```

Fig. 1 An example of semantic conflict, modified from de Jesus et al. (2023)

ferent methods, each responsible for a specific task related to text analysis. This task is accomplished by invoking other methods, such as `countDupWhiteSpaces`, `countComments`, and `countDupWords`.

To improve text cleaning, Left added a call to the `countDupWhitespaces` method, which counts duplicate whitespaces in the text. Duplicate whitespaces can occur when there is more than one space between words or sentences, which may indicate inconsistent or unnecessary formatting in the text. Removing or counting these spaces can be useful for improving the readability and formatting of the text. Contrasting, Right added a call to the `countDupWords` method, which is responsible for counting duplicate words in the text. Duplicate words can indicate redundancy or errors in writing. Identifying these duplications can help improve the clarity and conciseness of the text.

The desired outcome for each developer will not be achieved with the final `generateReport()` method implementation. According to Left's intention, the method should aggregate only the count of duplicate spaces with the number of comments, resulting in fixes reflecting these two concerns. Right's intention is that the method should aggregate only the count of duplicate words with the number of comments. However, when both branches are merged, the method calls both `countDupWhiteSpaces` and `countDupWords`, and since both methods modify the same state element, the variable `fixes`, by reading and incrementing it (as shown in the gray-highlighted lines at Line 18 and Line 30 in Fig. 1), the final value of `fixes` accumulates both the duplicate spaces count and the duplicate words count, plus comments.

For instance, when executing the Left version of the method with the text `"Hello□□□world!"`, there is no implementation of the `countDupWords` method in this branch. Therefore, when calling `countDupWhiteSpaces`, only the duplicate spaces will be counted, in addition to checking for comments when calling the `countComments` method. There are two duplicate spaces between *Hello* and *world*, so the expected result in Left for the variable `fixes` would be two.

When executing in the Right's branch with the same string, there is no implementation of the `countDupWhiteSpaces` method. Therefore, when calling `countDupWords`, only the number of duplicated words will be counted. Since there is no duplicate word, the expected result in Right for the variable `fixes` would be zero.

However, when executing the merged method from Fig. 1, there is no textual conflict, but the obtained result is two. While this numerically matches Left's expected result, it does not satisfy Right's specification. Right expects fixes to be zero (no duplicate words), but the merged version produces two because it also counts the duplicate spaces from Left's method. This asymmetric interference demonstrates how semantic conflicts can selectively violate one developer's intentions while appearing to preserve another's, making them particularly difficult to detect.

Semantic conflicts of this nature are often complex and costly to identify and resolve (Sousa et al. 2018; Da Silva et al. 2020). Code review can hardly detect such conflicts, letting them escape to production environments. Project tests are also hardly enough for detecting conflicts; this is what motivates previous work (Da Silva et al. 2020) that uses test generation tools to precisely compensate for that. Moreover,

a broken test in a merge version does not imply semantic conflict. This directly affects the quality and outcome of the products, as the final result is not as expected. [2]

Resolving these conflicts may require careful manual investigation, which can impact productivity. If not detected immediately after integration, addressing these conflicts can become even more challenging as it involves reconciling possibly subtle semantic and behavioral incompatibilities. In this example, it would be necessary to investigate whether the issue lies within the individual implementations of Left and Right, or in interference between them. This would require a thorough investigation that goes beyond the abstraction boundaries established by the methods that `generateReport` calls.

When analyzing the code in the motivating example using the lightweight static analysis algorithms designed by De Jesus et al. (2024), de Jesus et al. (2023), the Direct Flow analysis reported a potential semantic conflict because Left sets the variable `fixes`, which is read by Right in the `countDupWords` method. Another conflict was reported by the Override Assignment analysis, since both methods modify the instance field `fixes` with the number of needed fixes they counted. The remaining algorithms in de Jesus et al.'s lightweight analysis suite — namely Confluence and Program Dependence Graph — fail to detect this conflict. This underscores the importance of carefully selecting the combination and scope of analyses.

Consider the scenario where multiple analyses converge on the same case, leading to redundancy and inefficiency. This excess not only wastes computational resources but also complicates the identification of genuine conflicts. Additionally, there is a risk of running analyses that provide little to no valuable insights, hindering rather than improving the overall analysis process. An example of this is the work by De Jesus et al. (2024), de Jesus et al. (2023), which reported a high number of false positives when combining four analyses without considering the optimized individual application of each one. Each of the lightweight static analysis algorithms possesses unique strengths and limitations. While some excel in identifying specific conflict types, others may struggle due to their distinct algorithms and scope. Therefore, it's imperative to employ a diverse range of analyses and meticulously craft combinations. The focus should not solely be on the quantity of analyses used, but also on their complementary nature and ability to address a wide array of potential conflicts. A thoughtful selection of analysis methods can significantly enhance conflict detection mechanisms, leading to smoother merge processes and heightened software quality.

3 Static analysis combination approach

To detect interference and help mitigate the negative consequences of semantic conflicts, we proposed a new approach (herafter Static Analysis Combination Approach) that combines lightweight static analyses that execute only in the merged version of

[2] It is important to note, however, that a failing test in a merged version does not necessarily indicate a semantic conflict. For instance, consider a scenario where variables l and r are initialized as 0 in base. Left changes the base code to initialize l with 1, while Right changes it to initialize r with 1, and also adds a test that asserts r==1 && l==0. This test passes in Right and fails in merge, but there is no semantic conflict as both developers changed unrelated variables.

the code (De Jesus et al. 2024; De Jesus et al. 2023), which is automatically annotate with metadata that indicates the instructions that either the left or right developers modifed or added.

Static Analysis Combination Approach consists of a suite of static analyses that detect semantic conflicts through different combinations. While De Jesus et al. (2024), de Jesus et al. (2023) evaluated four analyses (Program Dependence Graph, Direct Flow Inter, Confluence Inter, and Override Assignment Inter), we expand the evaluation to nine analyses by introducing intraprocedural variants, exception-aware versions, and Control Dependence analysis.

Table 1 presents the nine static analysis evaluated in this study, showing their scope (interprocedural or intraprocedural) and exception handling capabilities. Each analysis represents a specific configuration designed to detect semantic conflicts. Cells marked with an "✓" indicate the algorithm's used this specific configuration. The symbol "✓" in the "New" column indicates analyses introduced in this study, expanding upon the four analyses previously evaluated in De Jesus et al. (2024), de Jesus et al. (2023).

The Direct Flow algorithms (DF Inter and DF Intra) extend traditional data flow analysis, commonly used in applications like taint analysis, with specialized adaptations for interference detection. Our implementation adds explicit flow edges from variable uses to conditional predicates and return statements. These additions capture how modifications affect control flow decisions and method outputs — relationships that are crucial for detecting semantic interference in the context of software merging.

The Control Dependence (CD) and Program Dependence Graph (PDG) algorithms implement well-known program analysis algorithms (Ferrante et al. 1987). We use them with some adaptations as a proxy for interference to identify semantic conflicts. For each algorithm, we provide two implementations: one that considers only regular control flow (CD and PDG) and another that includes exception handling (CD-e and PDG-e). This exception-aware extension was not present in the original formulations by Ferrante et al. (1987). The exception-aware variants capture control dependencies that arise from uncaught exceptions. For instance, if a statement `list.get(index)` can throw an uncaught exception, any statement following this call has a control dependency on it—the statement will only execute if the

Table 1 Configuration of the static analyses considering different levels (Inter for interprocedural and Intra for intraprocedural) and exception handling.

Analysis	Scope		Exception Handling		New
	Inter	Intra	With	Without	Analysis
DF Inter	✓				
DF Intra		✓			✓
CF Inter	✓				
CF Intra		✓			✓
OA Inter	✓				
CD-e		✓	✓		✓
CD		✓		✓	✓
PDG-e		✓	✓		✓
PDG		✓		✓	

list access completes normally. Without exception edges, this dependency is missed, potentially overlooking conflicts where one developer modifies the code that may throw exceptions while another modifies the subsequent code that depends on normal execution. CD and PDG are analyses known in the literature, and we apply them for interference detection, incorporating exception handling as an enhancement to better capture control flow scenarios.

The Override Assignment (OA Inter) and Confluence (CF Inter) algorithms were specifically designed for detecting interference patterns (De Jesus et al. 2024; De Jesus et al. 2023). We include intraprocedural variants (CF Intra) to explore whether a more lightweight analysis can still effectively identify interference.

The nine analyses try to explore likely interference situations by keeping track of the changes developers make and how they affect *state elements* such as global variables, object fields, and variables that hold method return values or raised exceptions, etc. The Static Analysis Combination Approach considers field accesses like `o.a` and `o.b` to refer to different state elements, even if `o` holds the same reference; the variable `o` also corresponds to a different state element than `o.a`.

For each of the nine analyses discussed here, the Static Analysis Combination Approach provides an implementation targeting the Java programming language, leveraging Soot as the underlying framework (Vallée-Rai et al. 2010). These analyses are lightweight because they do not rely on complex system abstractions—such as System Dependence Graphs (SDGs)—and execute exclusively on the merged version of the code. In contrast, previous static analysis-based techniques (Horwitz et al. 1989, 1990; Barros Filho 2017) rely on computationally heavier analyses (SDGs) or require running analyses on multiple program versions (base, left, and right).

3.1 Direct flow (DF Intra and DF Inter) analysis details

Direct Flow (DF) analysis captures how values propagate across state elements, both within a single method (intraprocedural) and across multiple methods (interprocedural). It provides the foundation for reasoning about data dependencies, enabling precise identification of semantic interactions caused by changes in state element definitions and uses. We adopt the formalization of Sparse Value-Flow Analysis (SVFA) introduced by Sui and Xue (2016) and extend it in two key ways: first, we adapt it from C/C++ to Java, accommodating Java's distinct constructs and semantics; second, we incorporate control-related flows through conditionals, loops, and return statements — constructs not explicitly modeled in the original SVFA definition.

We extend the SVFA model to include def-use edges for expressions used in control structures and return statements. While these constructs do not create new definitions, they represent critical use sites that influence program behavior. For conditional branches and loop conditions (such as `if`(v) or `while`(v)), we create def-use edges from all reaching definitions of *v* to the use at that program point, capturing dependencies where changes to *v*'s definition may alter control flow. Similarly, for return statements that return a variable *v*, we create def-use edges from all reaching definitions of *v* to that return point, ensuring that modifications to returned values are properly tracked across method boundaries.

For instance, DF analysis identifies potential semantic conflicts by marking state elements modified by either Left or Right and constructing a data-flow graph. **A conflict is detected when a path exists from a state element marked by Left to one marked by Right, or vice versa**. Figure 1 from Section 2 illustrates an example: Left introduces *countDupWhiteSpaces*() which modifies the fixes variable, while Right's *countDupWords*() method reads this value. Figure 2 provides a simplified visualization of this DF graph construction. The path from Left's definition of *fixes* to Right's use creates a data dependence that can lead to interference.

3.2 Control dependence (CD and CD-e) analysis details

Control dependence is a fundamental concept in program analysis. We adopt the classical definitions of control dependence introduced by Ferrante et al. (1987), which remain the standard in the field. Control Dependence (CD) analysis identifies potential semantic conflicts during software merging by constructing a control-dependence graph and marking nodes corresponding to state elements modified by Left or Right. **A path from a Left-marked node to a Right-marked node, or vice versa, indicates a possible conflict, similarly to direct flow analysis**. CD analysis captures how modifications to conditionals or loops affect dependent state elements and program behavior. It can include or exclude exception edges in the control-flow graph, resulting in two variants: CD-e (with exception edges) and CD (without).

For instance, consider the example in Fig. 3, adapted from the scenario in Section 2, illustrates a Control Dependence conflict where two developers independently modify a method. Right adds has *WeaselWords*() to the 'if' condition expression, while Left adds *text = loadFromFile()* before the conditional statement and *removeDuplicateWords*() inside the 'if' statement body. Since Left's addition depends on the conditional that Right modifies, the CD analysis detects interference through the control dependence paths, Left's code can only execute if Right's condition evaluates to true. Figure 4 provides a simplified visualization of this CD graph construction, showing the paths from the conditional that can lead to interference between Right and Left.

3.3 Program dependence graph (PDG and PDG-e) analysis details

The Program Dependence Graph (PDG), as originally defined by Ferrante et al. (1987), integrates both data and control dependencies to explicitly represent the

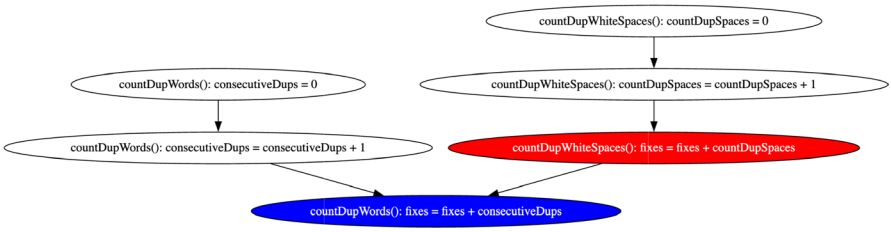

Fig. 2 Simplified graph showing Direct Flow (DF) conflict from Left to Right

```
33  ...
34    public void cleanText() {
35      text = loadFromFile();
36       removeComments();
37
38      if (text != null    && hasWeaselWords()){
39         removeWeaselWords();
40         removeDuplicateWords();
41      }
42    } ...
```

Fig. 3 Semantic conflict example. We highlight one developer's change in red and the other's in blue. Extracted from de Jesus et al. (2023)

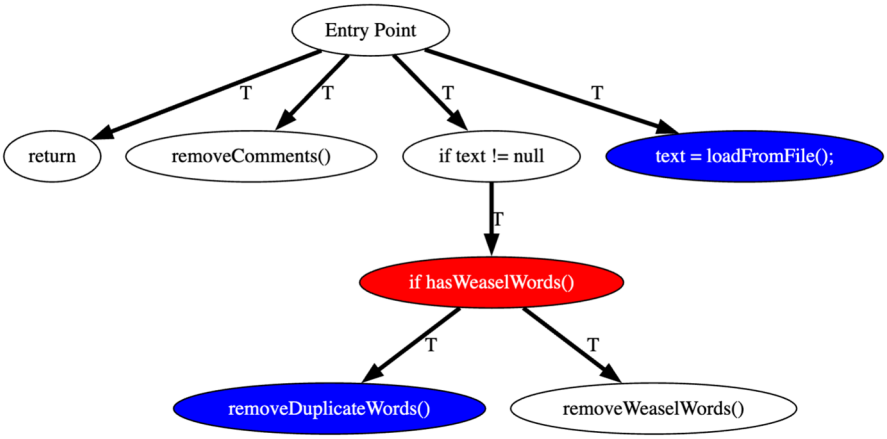

Fig. 4 Simplified graph showing Control Dependence (CD) conflict from Left to Right

semantic relations among program statements. Our implementation builds on the Direct Flow (DF) and Control Dependence (CD) analyses described earlier, and enriches the graph with additional edges to capture loop-carried dependencies and definition-order relations. Furthermore, the construction is performed with and without exception handling, depending on the configuration.

For instance, consider the same code example from the CD analysis in Fig. 3. The simplified PDG graph illustrates how both data and control dependencies are captured. Left's assignment creates a def-use edge (shown as a dotted line in Fig. 5) to the conditional that uses the `text` variable. Additionally, Left's *removeDupli-*

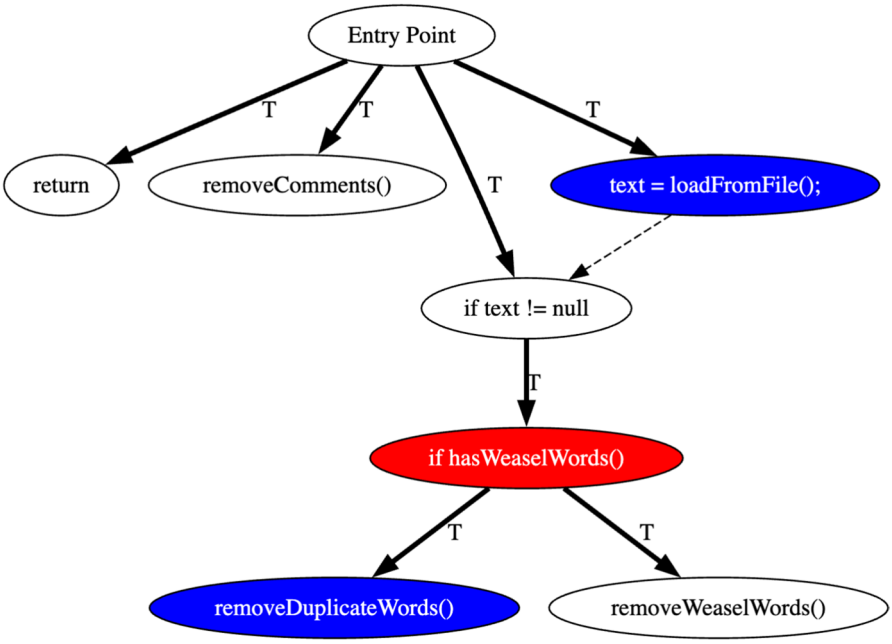

Fig. 5 Simplified graph showing Program Dependence (PDG) conflict from Left to Right and Right to Left

cateWords() call, placed inside the conditional body, has a control dependence on the conditional expression that Right modifies. The PDG graph reveals a path from Left's modifications to Right's changes, indicating a potential conflict, where both developers' changes affect the program's behavior in interconnected ways.

3.4 Interprocedural override assignment (OA Inter) analysis details

The Interprocedural Override Assignment (OA Inter) analysis detects conflicts in which a state element, modified by one developer is subsequently overwritten by changes introduced by another developer. A conflict is only reported if no intermediate modification from the trusted baseline occurs between the two writes, ensuring that the overwrite represents a genuine interference between concurrent modifications.

For instance, the motivating example in Section 2 illustrates a scenario where there is an OA Inter conflict. In this example, the Left developer adds a call to *countDupWhiteSpaces*(), which modifies the variable fixes by counting duplicate whitespaces. The Right developer adds a call to *countDupWords*(), which also modifies *fixes* by counting duplicate words. So the same state element fixes is written by both developers without any baseline modification in between, leading to a semantic conflict. This analysis do not create a graph representation, as it only tracks write operations and their ordering.

3.5 Interprocedural and intraprocedural confluence (CF Inter and CF Intra) analysis details

The Confluence (CF) analysis identifies where direct flows from different modifications converge, potentially causing interference. CF Intra operates within single methods, while CF Inter traces flows across method boundaries. This enables detection of semantic conflicts when changes from different developers in different state elements converge at a common program point, affecting the same state element, potentially leading to unexpected behavior.

The analysis executes dual DF analyses: once tracing changes from the Left branch to the Base, and once from the Right branch to the Base. Program points where paths from different developers converge are identified as confluence points, indicating potential interference in the merged program state when these paths affect the same state elements.

For instance, Fig. 6 illustrates a CF Inter conflict where Left adds *countDupWhiteSpaces*() to modify spaces, while Right adds *countDupWords*() to modify words. Since the final return statement combines both variables, their independent modifications converge at this point, creating a semantic conflict. Figure 7 provides a simplified visualization of this convergence pattern in the CF graph, leading to interference between Left and Right.

```
42        ...
43        public int countFixes(){
44             countDupWhiteSpace(); //affect the field's spaces
45             countComments();
46             countDupWords(); //affect the field's words
47             return spaces + words;
48        }
49        ...
50   }
```

Fig. 6 Confluence Flow (CF) example. Extracted from de Jesus et al. (2023)

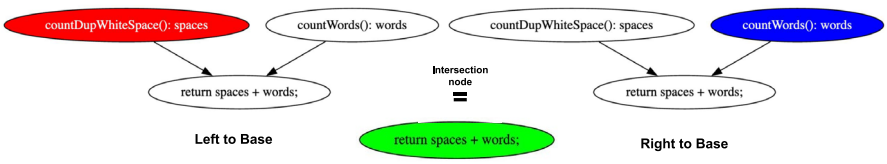

Fig. 7 Simplified graph showing Confluence Flow (CF) conflict

4 Evaluation method

Our **goal** is to understand how different anaylises of the Static Analysis Combination Approach affect interference detection performance, particularly in terms of precision and recall. To achieve this, we replicate the empirical study by De Jesus et al. (2024), de Jesus et al. (2023), adhering to their methodology for dataset construction, analysis execution, and data evaluation. However, we refine the evaluation process to mitigate the impact of excessive false positives.

This focus on false positives stems from the substantial influence of false positives on the findings of the original study. Specifically, De Jesus et al. (2024), de Jesus et al. (2023) reported a high number of false positives, which significantly reduced precision and ultimately compromised conclusions about their approach's effectiveness. To systematically explore Static Analysis Combination Approach's potential for interference detection, we follow the experimental design illustrated in Fig. 8 and detailed in this section. Our evaluation examines different analysis (interprocedural vs. intraprocedural, with and without exception handling), assessing their accuracy, computational efficiency, and suitability as a foundation for semantic merge conflict detection tools.

Our research is driven by the following **research questions** that focus on evaluating the effectiveness of different static analysis techniques in detecting semantic conflicts within merge scenarios. These questions aim to explore how each technique contributes to the occurrence of false positives and how their combinations impact detection performance, identifying the most effective sets of analyses for achieving optimal results across different evaluation metrics.

RQ(1) **How do different static analysis techniques lead to the occurrence of false positives in detecting semantic conflicts in a merge scenario?** To answer this question, we conduct an analysis of exclusive true positives (TP-e) and exclusive false positives (FP-e) individually for each static analysis technique, comparing their results with each other. TP-e refers to conflicts detected by only one specific analysis, while FP-e refers to false alarms reported by only one analysis. These exclusive metrics help us understand each analysis's unique contribution to the overall detection capability. We calculate the difference between TP-e and FP-e for each analysis. If the result of this difference is equal to or greater

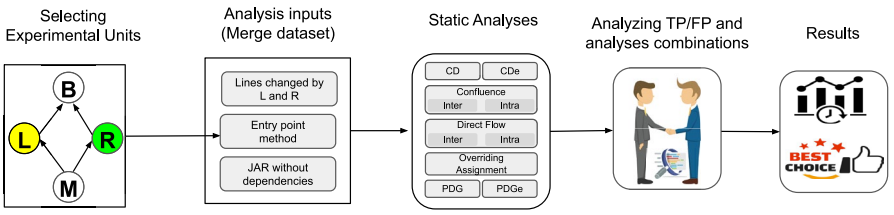

Fig. 8 Experiment Setup. We use the dataset from de Jesus et al. (2023), which includes merge scenarios from open-source Java projects. Each scenario is built into a JAR, extracting modified methods and lines for analysis. New static analyses are created, executed, and evaluated against a Ground Truth, testing all combinations of five techniques and nine variations to identify the best combination. Adapted from de Jesus et al. (2023)

than zero, it indicates that the technique does not hinder the results. Conversely, if the result is negative, it means that the technique may negatively impact the attainment of better results.

RQ(2) **What combinations of static analysis techniques demonstrate the best performance regarding each metric in detecting semantic conflicts in a merge scenario?** To address this question, we automatically and exhaustively analyze all possible combinations of analyses, forming an $n \times n$ set of combinations, where $n = 9$ available analyses. We evaluate the precision, recall, F1 score, and accuracy of each combination to ascertain which exhibited the best performance. Additionally, in cases where multiple combinations show identical performance and were subsets of another, we eliminate the larger subset.

RQ(3) **How does the execution time of different static analysis techniques and their combinations impact the overall performance and scalability in detecting semantic conflicts in a merge scenario?** To address this question, we systematically collect and analyze the execution time for each analysis technique and combination, considering the time taken for each individual execution and its impact on performance. Additionally, we explore how the execution time varies with the complexity of each combination, aiming to identify the most time-efficient approaches without sacrificing detection accuracy.

4.1 Experimental units

In our research, we adopt the same dataset of merge scenarios used in the original study (De Jesus et al. 2023), where pairs (*scenario*, *operation*) serve as *experimental units*. In more details, de Jesus et al. built the dataset of merge scenarios after mining 96 units that originally appeared in the datasets of previous studies: 35 units from the work of Silva et al. (2023), 31 from Barros Filho (2017), 30 units from Sousa et al. (2018), and three new units, totaling 99 units. These units are associated with 54 merge scenarios derived from 32 projects, previously identified in past research. All selected merge scenarios were from open-source Java projects available as GitHub repositories.

Following the approach of De Jesus et al. (2024), de Jesus et al. (2023), the experimental units exclude fast-forward and criss-cross merge commits (Chacon and Straub 2014). Their dataset focuses on scenarios where methods or constructors are modified by both developers, facilitating the establishment of interference ground truth. In contrast to previous work on semantic conflict detection with tests (Silva et al. 2023), de Jesus et al.'s dataset also excludes scenarios where only *field* declarations are changed by both developers. This criterion is also adopted by related work on information flow analysis (Barros Filho 2017) and verification (Sousa et al. 2018), as these approaches also require *method* or *constructor* declarations as the analysis entry point.

4.2 Data collection procedures

This study follows the methodology of de Jesus et al. (2023) for preparing the inputs required for static analyses. This process consists of building the merge commit version for each selected scenario, generating JAR files without external dependencies to optimize performance, and translating Java bytecode into the Jimple intermediate

representation using the Soot framework (Vallée-Rai et al. 2010). The Jimple representation is generated without optimizations (e.g., variable inlining) to preserve statement structure and correct line annotations. Additionally, they collect and use information about the modified lines of code and modified operations (methods and constructors), which serve as annotations and entry points for the analyses. To ensure accurate interference detection, they leverages the DiffJ tool[3] to obtain precise syntactic differences.

4.3 Running combinations

With the prepared analysis inputs, we execute the combination of static analyses to detect interference Section 3 for each experimental unit. For performance reasons, the interprocedural version of the analyses (that detect Direct Flow, Confluence and Overriding Assignments interferences) are all executed with nesting level five, meaning that they only analyze methods called up to five levels deep from the entry point. The same nesting level used by De Jesus et al. (2024), de Jesus et al. (2023). We also adopted the configuration from De Jesus et al. (2024), de Jesus et al. (2023), utilizing the Spark points-to analysis framework (Lhoták and Hendren 2003; Lhoták 2003), which enhances the precision of the call graph (Smaragdakis and Balatsouras 2015). Soot was configured similarly to the setup in De Jesus et al. (2024), de Jesus et al. (2023), retaining line numbers obtained from bytecode and excluding classes from standard Java libraries, except for basic classes such as String, RuntimeException, and those that simply encapsulate primitive types (Integer, Long, etc.). In addition to collecting analysis results that report interference or its absence, execution time data for each analysis combination is also gathered. Furthermore, combinations of all analyses are systematically generated in an $n \times n$ manner to collect their results, where $n = 9$ analyses. To achieve this, each analysis is run ten times for each of the 99 experimental units.

4.4 Data analysis procedures

In this final step of our experiment, we compare the interference ground truth with the combinations generated from the analysis results and compute precision, recall, F1 score, and accuracy metrics. When combining the nine static analysis algorithms (see Section 3), we explore 511 distinct possible combinations. This is because each combination of techniques can include any subset of the nine available techniques, leading to a large number of possible combinations. We perform the experiment for 511 possibilities, which highlights the difference from the previous study (De Jesus et al. 2024; De Jesus et al. 2023). For each configuration, we apply the logical OR operation to the set of techniques, resulting in a single outcome for each specific combination. Next, we calculate the metrics for each of these combinations. Additionally, we evaluate if the most effective combinations are subsets of other with the same outcome. If so, we eliminate the larger combination, retaining only the smaller and therefore more efficient ones. This procedure enables us to achieve the same result

[3] DiffJ compares Java files based on their ASTs and is available at https://github.com/jpace/diffj.

in less time. We adopted the same methodology and ground truth as de Jesus et al. (2023). Each experimental unit was manually analyzed by at least two researchers to identify interference based on defined conditions. In case of disagreement, a third researcher participated to reach a consensus. Each verdict was supported by a test case for interferences and an explanation for non-interferences, ensuring consistency and accuracy in the evaluation.

The manual analysis was performed by five researchers with 6-37 years of programming experience and 5-30 years of Java expertise. All researchers are familiar with semantic conflict analysis and followed a rigorous protocol that includes: examining code changes using diff analysis, analyzing data and control flow dependencies, executing test cases when available, identifying interference and documenting findings with annotated screenshots and structured descriptions.

To assess the contribution of each static analysis, we used the Exclusive True Positives (TPe) and Exclusive False Positives (FPe) metric. These metrics were developed to measure the effectiveness of each analysis in isolation, highlighting its specific contributions in terms of true positives (TP) and false positives (FP). The TPe represents the number of true positives identified exclusively by a given analysis, while the FPe corresponds to the false positives attributed exclusively to that analysis. The difference between TPe and FPe provides a balanced measure that highlights the net impact of each analysis. The higher the $TPe - FPe$ value, the greater the contribution of the analysis in providing accurate results without generating excessive noise, making it essential for evaluating the precision and usefulness of static analyses in detecting semantic conflicts.

The choice to focus on false positives (FP) stems from the high rate observed in previous works (De Jesus et al. 2024; De Jesus et al. 2023), which compromised the precision of their results and, consequently, the conclusions about the effectiveness of the approaches used. False positives occur when the system detects a semantic conflict that does not actually exist, generating false alarms. This can reduce trust in the analysis tool and overload developers with irrelevant alerts, potentially leading to the abandonment of the tool. Therefore, understanding how different static analysis techniques lead to the occurrence of false positives is crucial for assessing the real effectiveness of these techniques in merge scenarios.

We will present a comparison of the top-performing combinations identified for each metric of the confusion matrix in the next section. These combinations were selected based on their performance in accurately predicting classifications across various evaluation metrics, such as precision, recall, F1 score, and accuracy.

Finally, we summarize and analyze the execution time data using several approaches: the average execution time per unit, per analysis, and per combination. These calculations were performed across ten executions for each case, providing a comprehensive view of the time required for each analysis and combination. All execution times and corresponding metrics are available in our appendix (Appendix 2025).

We implemented all these data analysis steps and built a docker container for reproducibility purposes (see our online Appendix Appendix 2025).[4] This approach

[4] The activities in the first two steps are also automated, but require manual effort in a number of scenarios, as explained earlier. That is why the replication container contains only the activities in the last two steps of our experiment.

simplifies dependency management, enhances portability, and mitigates challenges associated with different runtime environments. Moreover, it grants access to the source code within the generated environment, safeguarding the integrity of the experiment.

5 Results

In this Section, we present the results of our investigation. In Section 5.1, we explore how different static analysis techniques influence the occurrence of false positives in detecting semantic conflicts in merge scenarios (research question RQ1). In Section 5.2, we examine which combinations of static analysis techniques lead to the best performance regarding their capability for detecting semantic conflicts (research question RQ2). In Section 5.3, we analyze the execution time for each analysis technique and combination, investigating how the time varies with the complexity of each approach and its impact on performance (research question RQ3).

5.1 How do different static analysis techniques lead to the occurrence of false positives in detecting semantic conflicts in a merge scenario?

We are investigating the occurrence of false positives because previous studies have indicated that static analysis tools often exhibit high rates of this type of error. Our goal is to identify which analysis techniques are most associated with the generation of false positives and understand their impact on the analyzed combinations.

Our dataset consists of 99 units, each labeled with a Locally Observable Interference (LOI) indicating the presence or absence of a semantic conflict. Among these, 29 units have conflicts, while 70 do not. Table 2 presents the totals and percentages of each confusion matrix component, false positives (FP), false negatives (FN), true negatives (TN), and true positives (TP), for each individually applied analysis technique. The PDG-e technique produced the highest number of false positives, with 27 occurrences, indicating a higher tendency to incorrectly classify non-existent conflicts as true. In contrast, the CD technique stood out for its precision, reporting only one false positive, making it the most conservative in terms of signaling false conflicts.

Table 2 Confusion matrix totals and percentages per analysis

Analysis	FP (%)	FN (%)	TN (%)	TP (%)
CD	1 (1.01%)	22 (22.22%)	69 (69.70%)	7 (7.07%)
CDe	13 (13.13%)	20 (20.20%)	57 (57.58%)	9 (9.09%)
DF-Intra	20 (20.20%)	22 (22.22%)	50 (50.51%)	7 (7.07%)
DF-Inter	23 (23.23%)	21 (21.21%)	47 (47.47%)	8 (8.09%)
OA Inter	6 (6.06%)	26 (26.26%)	64 (64.65%)	3 (3.03%)
Confluence Intra	8 (8.08%)	29 (29.29%)	62 (62.63%)	0 (0.00%)
Confluence Inter	18 (18.18%)	24 (24.24%)	52 (52.53%)	5 (5.05%)
PDG	18 (18.18%)	18 (18.18%)	52 (52.53%)	11 (11.11%)
PDG-e	27 (27.27%)	15 (15.15%)	43 (43.43%)	14 (14.15%)

To address this question, we conducted a detailed analysis of exclusive true positives (TP-e) and exclusive false positives (FP-e) across three combinations, using CD Fig. 10 and PDG Fig. 11 in combination with three other analyses, and also compared all nine analyses with one another Fig. 12. We then calculated the difference between TP-e and FP-e for each individual analysis. If this difference is zero or positive, it suggests that the analysis contributes positively or at least does not hinder the overall results. Conversely, a negative difference indicates that the technique might be more detrimental than beneficial. This calculation is performed based on the results of each analysis, compared to the respective ones, i.e., the TP-e that appears exclusively in one analysis and does not appear in any of the others. The same applies to FP-e, where one analysis generates an FP case that does not occur in any other.

Some analyses are subsets of others, and understanding these relationships is crucial to evaluating their individual contributions. For instance, the CD analysis is completely contained within CD-e, PDG and PDG-e, while DF Intra is a subset of DF Inter, PDG and PDG-e. This hierarchical relationship implies that the results (both positive and negative) of each contained analysis are inherently subsets of the results of their supersets. The Venn diagram in Fig. 9 illustrates the dependencies and intersections among all the analyses conducted. This diagram highlights how different analytical analyses overlap and contribute to the overall results.

Consequently, it is essential to analyze which analyses are contributing positively or interfering with the results. To achieve this, we summarize the results in Figs. 10, 11 and 12. These figures isolate and assess the specific impact of each analysis when compared to CD Fig. 10 and to PDG Fig. 11 individually, and performs an analysis of all the analyses together Fig. 12. The goal is to determine how each analysis contributes to or limits the accuracy for detecting semantic conflicts.

When analyzing the nine analysis Fig. 9, we identified that OA Inter, CF Inter, DF Inter, and PDG-e are the analyses with the largest sets of identified semantic conflicts and exhibit distinct characteristics compared to the others. However, it is crucial to assess the contribution of each configuration in relation to false positives. We performed a comparison using the DF Inter, OA Inter, CF Inter, and PDG-e analyses, as well as between OA Inter, CF Inter, and CD-e. However, we observed that the analyses with exceptions generated more false positives, resulting in negative values when calculating the $TPe - FPe$ of these comparisons. As a result, we decided to present, in the following subsections, a comparison exclusively between CD and PDG, without considering the exceptions, as they were generating an excessive number of false positives. We also included the analysis with the complete set of all the analyses.

5.1.1 Using CD to compare FP-e and TP-e

We analyzed the performance of the CD, DF Inter, OA Inter, and CF Inter analyses in the detection of semantic conflicts, considering both True Positives (TP) and False Positives (FP). The interaction and overlap between these analyses raise relevant questions about their roles and limitations.

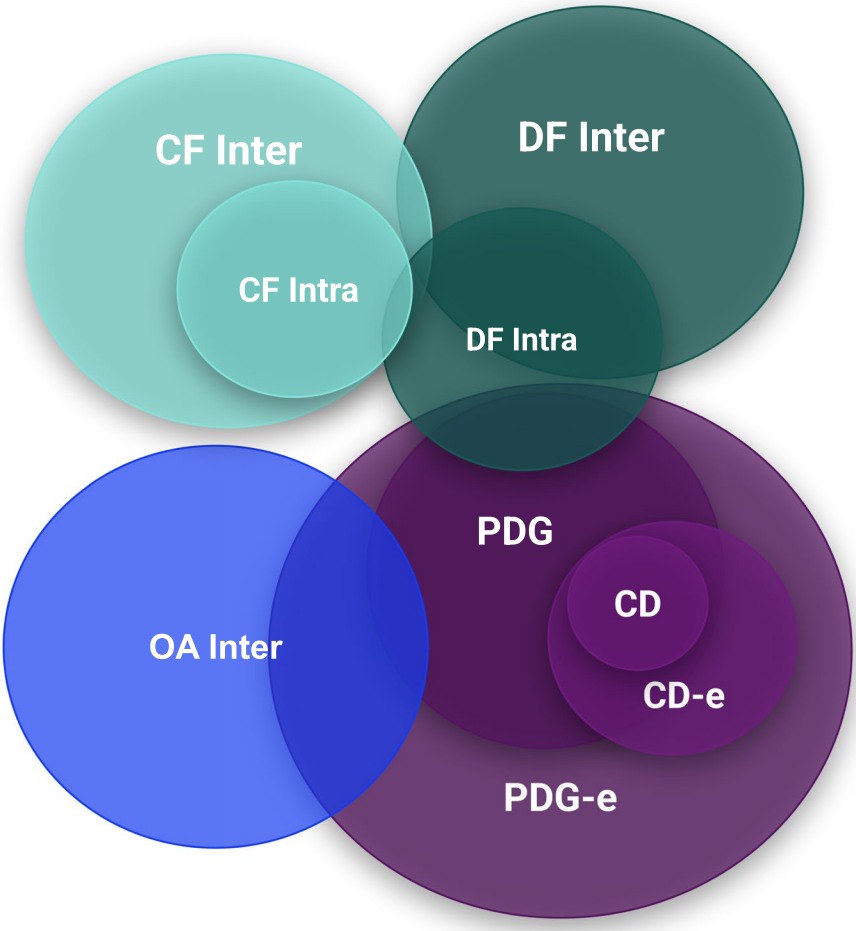

Fig. 9 Venn diagram showing the dependencies and intersections of all the analyses

In total, **13 True Positives** and **28 False Positives** were identified by the different analyses, as detailed in Fig. 10.

Regarding **True Positives**, we observed the following individual and overlap distribution: CD individually identified seven TPs; DF Inter identified eight TPs; CF Inter identified five TPs; OA Inter identified three TPs.

Only one unit was identified as a TP by **all** four analyses (CD ∩ OA Inter ∩ DF Inter ∩ CF Inter). Exclusive TPs exist for some analyses: CD identified three exclusive TPs, and DF Inter identified three exclusive TPs. CF Inter identified one exclusive TP. OA Inter did not identify exclusive TPs.

Several TPs were detected by combinations of analyses, but not by all, such as two TPs common only to CD and DF Inter. Other pair or trio overlaps (excluding the all-four overlap) include: one TP common to CD, OA Inter, and CF Inter; and one TP common to OA Inter, DF Inter, and CF Inter.

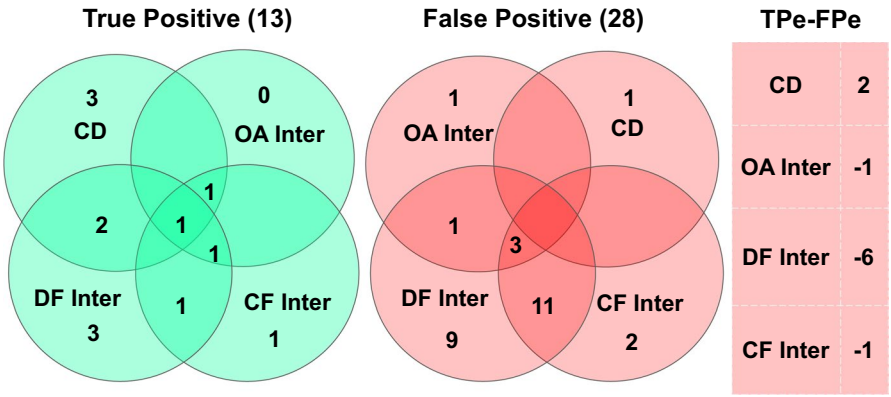

Fig. 10 Comparative of CD, OA Inter, DF Inter, and CF Inter analyses with the calculation of unique TP and FP scenarios.

Regarding **False Positives**, we observed a significantly higher total number (28). The distribution of FPs by analysis individually is: CD identified only one FP; DF Inter identified 24 FPs; CF Inter identified 16 FPs; OA Inter identified five FPs.

The analysis of exclusive FPs confirms that different analyses make distinct errors, reflecting their methodologies and scopes. DF Inter presents nine exclusive FPs, CF Inter two, OA Inter one, and CD one.

The overlap of FPs is also informative. Three FPs are common to OA Inter, DF Inter, and CF Inter (not CD); One FP is common only to OA Inter and DF Inter; 11 FPs are common only to DF and CF.

When evaluating the **exclusive** contribution of each analysis individually, considering the difference between their exclusive TP and their exclusive FP (TP-e - FP-e), we observed the following balances, as derived from the data presented in Fig. 10. CD is the only analysis to show a more significant positive balance (+2). The other analyses show negative balances: -1 for OA Inter, -6 for DF Inter, and -1 for CF Inter. This suggests that, while all analyses generate exclusive FPs, CD is comparatively more effective at identifying TPs unique to it than at generating unique FPs, showing the largest net exclusive contribution.

5.1.2 Using PDG to compare FP-e and TP-e

We analyzed the performance of the PDG, DF Inter, OA Inter, and CF Inter analyses, considering both TP and FP. In total, **15 True Positives** and **31 False Positives** were identified by the different analyses, as detailed in Fig. 11 and our online appendix (Appendix 2025).

Regarding **True Positives**, we observed the following individual and overlap distribution: PDG individually identifies 11 TPs; DF Inter identifies eight TPs; CF Inter identifies five TPs; OA Inter identifies three TPs.

Only one unit was identified as a TP by **all** four analyses (PDG ∩ OA Inter ∩ DF Inter ∩ CF Inter). Exclusive TPs exist for some analyses: PDG identified five exclu-

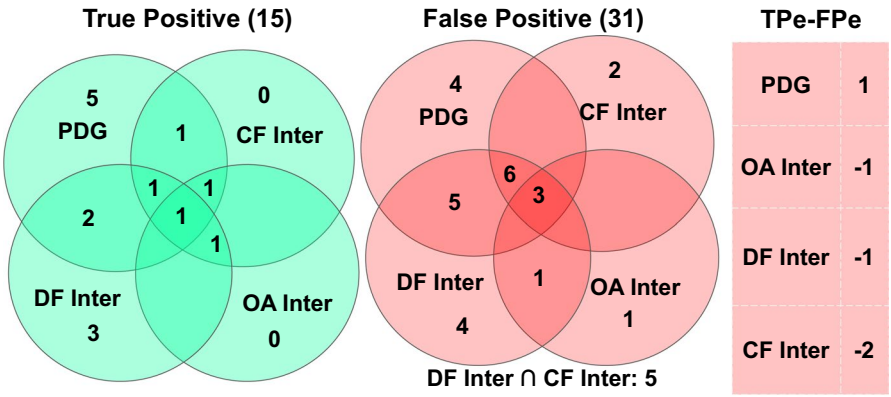

Fig. 11 Comparative of PDG, OA, DF, and CF analyses with the calculation of unique TP and FP scenarios

sive TPs, and DF Inter identified three exclusive TPs. OA Inter and CF Inter did not identify exclusive TPs; Two TPs common only to PDG and DF Inter, and one only to PDG and CF Inter.

Regarding **False Positives**, we observed a significantly higher total number (31). The distribution of FPs by analysis individually is: PDG identified 18 FPs; OA Inter identified five FPs; DF Inter identified 24 FPs; CF Inter identified 16 FPs.

DF Inter presents four exclusive FPs, PDG also four, CF Inter two, and OA Inter one. The overlap of FPs is also informative. Three FPs are common to **all** four analyses; Six FPs are common to PDG, DF Inter, and CF Inter (not OA Inter); Five FPs are common only to DF Inter and PDG; Five FPs are common only to DF and CF Inter; and one FP is common only to OA Inter and DF Inter.

When evaluating the **exclusive** contribution of each analysis individually, considering the difference between their results, presented in Fig. 11. PDG is the only analysis to show a positive balance (+1). The other analyses show negative balances: -1 for OA Inter and DF Inter, and -2 for CF Inter. This suggests that, while all analyses generate exclusive FPs, PDG is comparatively more effective at identifying TPs unique to it than at generating unique FPs. The total TP and FP values for each analysis are available in our online appendix (Appendix 2025).

5.1.3 Using all analyses to compare FP-e and TP-e

In Fig. 12, which compares all analyses, it is noticeable that most results show zero, as it is difficult for one result not to be contained within another. However, we can observe that two analyses, CF Inter and DF Inter, have two and one exclusive false positives, respectively. This indicates that, in the total set of combined analyses, these two contribute more to negative results than to positive ones.

Based on Figs. 10 and 11, we observe that CD and PDG are two promising analyses that contribute positively to the final result. Their lower number of false positives leads to higher precision, indicating that when they identify a conflict, it is more

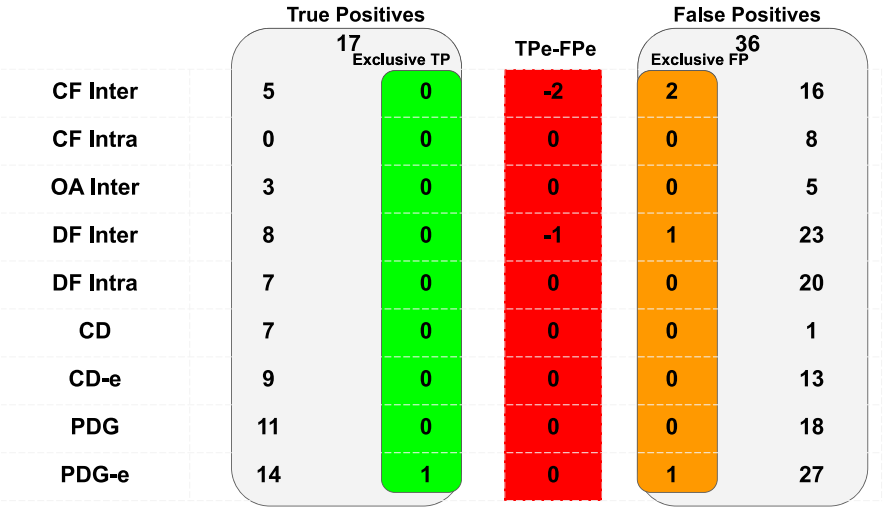

Fig. 12 Comparative of each analysis with the calculation of unique TP and FP scenarios

likely to be a true positive. A more in-depth analysis of their results and combinations is necessary to draw more accurate conclusions, which will be addressed in the next research question.

5.2 What combinations of static analysis demonstrate the best performance regarding each metric in detecting semantic conflicts in a merge scenario?

To identify which combinations of static analysis offer the best performance for each metric in detecting semantic conflicts, we systematically evaluated all possible 511 analyses combinations. These combinations were analyzed based on precision, recall, F1 score, and accuracy to determine which combinations are most effective. The results detailing the analyses with the best performance in each metric are presented in Table 3. When multiple combinations demonstrated equivalent performance but differed in complexity, we prioritized the simpler options to enhance practicality and implementation. While here we present only the best combinations out of the 511 we evaluated, the complete results can be found in our appendix (Appendix 2025).

Building on this, even though some analyses are subsets of others, our goal is to evaluate the best combinations across all analyses. This is because, as a subset, an analysis might be computationally faster while still producing the same results as its superset in certain scenarios. This potential trade-off between efficiency and

Table 3 Comparison of the best combinations identified for each metric of the confusion matrix

Metrics	Value	Analyses combination
Precision	0.88	CD
Recall	0.59	PDG-e or DF Inter; PDG-e or DF Intra
F1 Score	0.44	PDG-e or DF Intra
Accuracy	0.77	CD

effectiveness underscores the importance of examining all analyses. By doing so, we ensure a comprehensive understanding of which combinations yield optimal performance, balancing accuracy and computational efficiency. For instance, if we obtain the same precision result using CD and using the combination of CD or PDG, we would opt for using only CD, as it is contained within PDG and has a smaller analysis scope, thus being executed more quickly.

The combination with the highest precision was the CD, with a score of 0.88. This indicates that CD is particularly effective at identifying true positives while minimizing false positives, making it a robust choice for scenarios where precision is crucial.

In terms of recall, the best performance was observed in the combinations PDG-e or DF Inter and PDG-e or DF Intra, both with a score of 0.59. This suggests that these combinations are more effective in detecting a higher proportion of true positives among all actual positives, which is essential for comprehensive conflict detection. In this case, we could use only the combination of PDG-e or DF Intra, as DF Intra is contained within DF Inter, resulting in a faster combination while achieving the same result.

For the F1 score, which balances precision and recall, the combination PDG-e or DF Intra showed the best performance, with a score of 0.44. Although this combination is not the best in precision or recall individually, it provides a balanced performance between the two metrics.

Finally, the CD analysis also achieved the highest accuracy, with a score of 0.77. This highlights the overall effectiveness of CD in accurately identifying both true positives and true negatives across different scenarios.

Table 4 compares the metrics of the best combinations of static analysis. Although some combinations excel in precision, they do not perform well in other metrics, such as recall. For instance, the CD analysis is noted for its high precision and accuracy but has the lowest recall, with a score of 0.24. A detailed discussion of these results, including their implications and trade-offs, is presented in Section 6. This indicates that, while CD is effective at identifying true positives, it struggles to detect all semantic conflict cases, resulting in a higher number of false negatives. In contrast, the combinations PDG-e or DF Intra and PDG-e or DF Inter show superior performance in recall, reflecting a better ability to identify a higher proportion of true positives. However, these combinations do not stand out in precision and other metrics, suggesting that while they are better at detecting conflicts, they may generate more false positives. This analysis highlights the trade-off between precision and recall and emphasizes the need to balance these metrics to optimize the effectiveness of the analyses.

Table 4 Comparison of the results for the best combinations

Metrics	Analyses combination		
	PDG-e or DF Inter	PDG-e or DF Intra	CD
Precision	0.33	0.35	0.88
Recall	0.59	0.59	0.24
F1 Score	0.42	0.44	0.38
Accuracy	0.54	0.56	0.77

5.3 How does the execution time of different static analysis techniques and their combinations impact the overall performance and scalability in detecting semantic conflicts in a merge scenario?

To measure the execution time of each analysis, we recorded both configuration and execution phases to capture the full computational cost. For combinations using the OR condition, we computed the total time by summing the individual analysis times, though in practice an optimized approach could stop execution once a positive result is obtained.

We formulated the following null and alternative hypotheses:

- $\mathbf{H}_0$: There is no significant difference in execution time among the nine static analysis.
- $\mathbf{H}_1$: At least one static analysis exhibits significantly different execution time compared to the others.

Given the non-normal distribution of execution times (Shapiro-Wilk test), we employed non-parametric methods for our analyses. We leveraged the Kruskal-Wallis test to detect overall differences among the nine analyses. We conducted post-hoc pairwise comparisons using Dunn's test, which is specifically designed for non-parametric multiple comparisons following Kruskal-Wallis. To control for Type I error inflation due to multiple testing (36 pairwise comparisons), we applied Bonferroni correction, adjusting the significance threshold to maintain a family-wise error rate of 0.05.

The Kruskal-Wallis test revealed significant differences among the nine analyses, leading us to reject H_0. Table 5 presents key pairwise comparisons. The results confirm that OA Inter and CF Intra are significantly faster than both CD and DF Inter. The particularly large differences when compared to DF Inter highlight that interprocedural dataflow analysis incurs substantial computational overhead.

Comparing exception-aware versions reveals the cost of exception handling: CD-e is only 2.8% slower than CD, and PDG-e is 7.8% slower than PDG. Statistical tests show no significant difference between these pairs, confirming that incorporating exception flow analysis adds negligible computational overhead.

While individual analyses vary significantly in execution time Table 6, with CF Inter being 3.7x slower than OA Inter, computational cost does not determine combination effectiveness. We compared the three best-performing combinations identified in RQ2 and found that CD is significantly faster than both PDG-e or DF Inter and PDG-e or DF Intra. As shown in Table 4, CD also achieves superior precision compared to these more expensive combinations. When comparing PDG-e or DF

Table 5 Key statistical comparisons from Dunn's test (p-values after Bonferroni correction)

Comparison	Mean Diff (s)	p-value	Interpretation
OA Inter vs CD	-1.451	0.0001	OA significantly faster
OA Inter vs DF Inter	-8.662	<0.0001	OA significantly faster
CF Intra vs CD	-1.023	0.0013	CF Intra significantly faster
CF Intra vs DF Inter	-8.234	<0.0001	CF Intra significantly faster

Table 6 Descriptive statistics of execution times (in seconds) for each analysis technique

Analysis	N	Mean	SD	Median	Min	Max
CD	99	7.82	12.53	3.95	2.09	110.33
CD-e	99	8.04	12.52	4.10	2.09	110.72
CF Inter	99	23.84	106.13	4.17	1.48	1013.53
CF Intra	99	6.80	12.45	2.79	1.49	106.80
DF Inter	99	15.03	42.09	4.98	2.36	391.35
DF Intra	99	8.00	12.74	4.03	2.37	110.40
OA Inter	99	6.37	11.23	2.68	1.92	98.50
PDG	99	8.25	12.26	4.42	2.41	108.92
PDG-e	99	8.89	12.92	4.60	2.44	111.17

Intra against PDG-e or DF Inter, we found no significant difference despite PDG-e or DF Intra being faster. These results demonstrate that strategic use of simpler, faster analyses can outperform complex, expensive ones in both execution time and detection effectiveness.

To evaluate scalability, we examined the relationship between program complexity (measured by number of visited methods) and execution time using linear regression analysis. The regression analysis revealed a moderate positive linear relationship, indicating that 40% of execution time variance can be explained by the number of visited methods. The remaining 60% is attributable to other factors such as control flow complexity, call depth, and analysis-specific characteristics. Notably, 85% of scenarios involved fewer than 1,000 methods and completed within 120 seconds, demonstrating practical applicability. The approach successfully scaled to programs with up to 5,000 visited methods, though with increased time variance.

5.4 Comparison with previous work

We compared the results of the static analysis combination proposed in this work (SA) with those of related works, as detailed in Table 7. To ensure a fair comparison, we reused the datasets adopted in previous studies, enabling the application of all analyses over the same experimental units and facilitating the calculation of comparable metrics: precision, recall, F1 score, and accuracy. Our results are using the best combinations indentified in RQ2.

The column labeled SA presents the metrics obtained with our full dataset of 99 units. Our analysis achieved the highest precision (0.88) and the highest accuracy

Table 7 Accuracy metrics and comparison with previous work and their datasets. We use SA for the Static Analyses we propose in here, JSA for Static Analysis used in De Jesus et al. (2023), DA for the Dynamic Analysis used in Silva et al. (2023), IF for the Information Flow analysis used in Barros Filho (2017), and VA for the Verification Algorithm used in Sousa et al. (2018)

Metrics	Techniques							
	SA	JSA	SA	DA	SA	IF	SA	VA
Precision	0.88	0.33	0.85	0.66	0.86	0.50	0.33	0.22
Recall	0.59	0.52	0.58	0.08	0.56	0.36	0.17	0.33
F1 Score	0.44	0.41	0.44	0.14	0.53	0.42	0.18	0.27
Accuracy	0.77	0.56	0.85	0.77	0.68	0.60	0.70	0.63
Units	99	99	78	78	63	63	30	30

(0.77), suggesting a low incidence of false positives and a robust ability to correctly classify conflict scenarios. In contrast, the JSA technique, adapted from the static analysis presented in de Jesus et al. (2023) and applied to the same 99 units, yielded considerably lower precision (0.33) and accuracy (0.56), although with a higher recall (0.52), indicating a higher detection rate at the cost of more false positives.

To investigate how our approach compares with dynamic techniques, we also evaluated both our SA and the Dynamic Analysis (DA) proposed in Silva et al. (2023) over a shared subsample of 78 units. On this common dataset, SA achieved a precision of 0.85 and recall of 0.58, both close to the metrics obtained in the full dataset, while DA exhibited a precision of 0.66 and a significantly lower recall (0.08). These results show that, although DA is accurate in identifying a smaller set of true conflicts, it tends to miss a larger portion of them, suggesting a higher incidence of false negatives. Of the 78 units analyzed, 24 presented conflicts. However, only three conflicts were reported by tests, of which two were true positives and one was a false positive. This demonstrates that the test-based approach and test execution alone are not sufficient to guarantee the absence of conflicts in the code.

To investigate how our approach compares with other static techniques, we also evaluated both our SA and the Information Flow (IF) analysis proposed in Barros Filho (2017) over a shared subset of 63 units. Although IF was executed on all 99 units, it only produced results for 63 of them using its best configuration. On this common dataset, SA achieved a precision of 0.86, recall of 0.56, F1 score of 0.53, and accuracy of 0.68, while IF reached a precision of 0.50, recall of 0.36, F1 score of 0.42, and accuracy of 0.60. These results show that SA consistently outperforms IF across all evaluated metrics.

Finally, our comparison with the Verification Algorithm (VA) from Sousa et al. (2018), applied over 30 common units, revealed that both approaches performed modestly in this reduced sample. SA reached a precision of 0.33 and a recall of 0.17, while VA achieved 0.22 precision and 0.33 recall. The notably lower precision of SA compared to its full-dataset performance is due to the very small number of positive samples in the shared dataset, where missing five out of six true positives had a disproportionately large impact on the metric.

Taken together, these comparisons emphasize the robustness of our SA technique across different datasets. It consistently achieves high precision and competitive recall, which positions it as a promising approach for detecting semantic merge conflicts in software repositories. Moreover, the variability of results across datasets reinforces the importance of using common experimental units when comparing tools and techniques.

6 Discussion

Our comparative analysis of different combinations of static analysis revealed notable variations in the performance of classical evaluation metrics, including precision, recall, F1 score, and accuracy. While these metrics provide a quantitative basis for evaluating the effectiveness of each combination, they may not capture the full complexity of the scenarios involved. Therefore, a complementary manual analysis is essential to better interpret the results, understand the context behind false positives and false negatives, and refine the evaluation of each combination's practical applicability.

Of the 29 units labeled as containing conflicts, CD identified eight as conflictual. Among these, one case was a false positive, while the remaining seven were true positives. Notably, this unit exhibits a data dependency detected by CD but does not present a semantic conflict according to the LOI classification, this discrepancy arises from CD's conservative flow analysis, which may overapproximate potential interferences.

This means that CD correctly identified only seven out of the 29 actual conflicts, yielding a recall of 24%. However, its precision was 0.88, indicating that when CD does report a conflict, it is correct 88% of the time, i.e., out of the eight reported, it correctly identified seven. This is a strong result from a precision standpoint, suggesting that CD is a conservative analysis that minimizes false positives, a desirable property in contexts such as continuous integration pipelines, where false alarms can cause unnecessary rework and delays.

On the other hand, CD missed 22 out of the 29 actual conflicts, resulting in a false negative rate of 76%, which is concerning. Missing such a large proportion of real conflicts is critical, especially in safety- or security-sensitive systems, where undetected semantic conflicts can compromise software integrity. However, this shows that not all existing conflicts involve control flow, which leads to a low detection rate. That is why there is a need for other analyses and combinations.

A manual analysis of the 22 false negatives produced by the CD analysis revealed the following patterns:

- Five cases related to **exception edges**, where semantic changes involved exception flows not captured by CD;
- Four cases of **removed lines**, representing deletions that affected semantics, but we are unable to execute CD due to a limitation of our approach, which relies solely on the merge version;
- Four cases involving **dependencies**, meaning that the conflict depended on the use of external dependencies or on an interprocedural analysis, which CD was not designed to detect;
- Four cases involving **lack of conditional flow**, where logic changes occurred in regions without explicit conditional control structures;
- Two cases of **complex conditionals**, which CD was unable to resolve due to nested or non-obvious branching logic;
- Three cases involving **interface implementation**, where behavior changed through polymorphism, outside CD's resolution scope;
- Two cases related to **annotations**, in which we were unable to correctly identify the markings and they were not reflected in changes to the control flow.

These results show that most false negatives are caused by limitations in the CD analysis's ability to capture complex or implicit semantic behaviors, such as exception handling, polymorphism, and external dependencies, highlighting the need for more comprehensive analysis strategies to improve recall in semantic conflict detection.

Regarding the 70 units without conflicts, CD correctly classified 69 as true negatives and only misclassified one as a false positive. This further reinforces CD's high accuracy in filtering out non-conflicting cases, contributing to its overall reliability when it comes to avoiding irrelevant warnings.

6.1 Maximum coverage: PDG-e with DF-Inter

Of the 29 units labeled as containing conflicts, the PDG-e or DF-Inter combination identified 51 as conflictual. Among these, 17 were true positives, meaning real semantic conflicts correctly identified, while 34 were false positives, cases in which the combination incorrectly flagged a conflict that did not exist according to the LOI classification.

This means that the combination correctly identified 17 out of the 29 actual conflicts, yielding a recall of 59%, the highest among the evaluated combinations. However, its precision was only 0.33, indicating that only one-third of the flagged cases actually corresponded to real conflicts. This result reflects a highly inclusive (or aggressive) detection strategy, sacrificing precision for broader coverage.

A manual analysis of the 34 false positives revealed the following patterns:

- 13 cases of **refactoring**, where changes in the code did not impact the semantic behavior;
- Seven cases related to **harmless code changes**, such as insertions or modifications that did not affect the logic;
- Six cases attributed to **conservative flow**, where the analysis overestimates dependencies due to its conservative nature;
- Four cases related to **annotations**, where the analysis was unable to accurately identify the affected lines;
- Four cases of **harmless code insertion**, with no impact on method logic.

These findings show that many false positives result from structural or syntactical changes that do not affect the software's behavior, highlighting the importance of manual analysis to correctly interpret the alerts generated by automatic tools.

A manual analysis of the 12 false negatives revealed the following patterns: four cases of **removed lines**; four cases of **dependencies**; two cases of **complex conditionals**; and three cases of **interface implementation**.

These findings indicate that most false negatives stem from limitations in capturing implicit behavior or structural changes, emphasizing the challenges in achieving complete semantic coverage with static analysis.

Regarding the 70 units without conflicts, the analyses correctly classified 36 as true negatives, while 34 were incorrectly classified as false positives. This nearly balanced split highlights the analyses's low specificity, meaning it frequently flags conflicts where none exist. Still, its high recall indicates that it can be a valuable component in a detection pipeline where maximizing coverage is more important than achieving high precision.

6.2 F1 Score: balancing precision and recall

Of the 29 units labeled as containing conflicts, the combination PDG-e or DF-Intra identified 49 as conflictual. Among these, 17 were true positives, that is, actual semantic conflicts correctly flagged by the combination, while 32 were false positives, representing cases incorrectly classified as conflicts by the combination.

This means that the combination successfully detected 17 out of the 29 actual conflicts, yielding a recall of 59%. Its precision was 0.35, indicating that approximately

one out of every three reported conflicts was indeed correct. This trade-off between precision and recall results in an F1 Score of 0.44, the highest among all evaluated combinations. This reflects a more balanced approach to conflict detection, where neither recall nor precision is maximized independently, but both are maintained at reasonable levels.

A manual analysis of the 32 false positives revealed several recurring causes: 13 cases of **refactoring**; seven cases of **harmless code changes**; six cases of **conservative flow**; four cases related to **annotation usage**; and two two cases of **harmless code insertion**.

The 12 false negatives, actual conflicts not detected by the combination, were categorized as follows: three cases of **removed lines**; four cases involving **dependencies**; two cases of **complex conditionals**; and three cases of **interface implementation**.

These findings reveal that while PDG-e or DF-Intra achieves a balanced trade-off between precision and recall, it still struggles with specific conflict types, particularly those involving subtle behavioral changes such as deletions or polymorphism. The false positives predominantly stem from structural modifications or benign changes, underscoring the necessity of manual inspection to filter out irrelevant alerts.

Regarding the 70 units without conflicts, the combination correctly identified 38 as true negatives and misclassified 32 as false positives. This demonstrates a relatively low specificity, as nearly half of the non-conflicting units were incorrectly flagged. Nevertheless, the high recall makes this combination a strong candidate in contexts where coverage is more important than precision, such as early detection stages or collaborative development workflows.

6.3 Trade-offs and applicability

We observed that most FPs are associated with refactoring changes or harmless code modifications. The main causes include code modifications or insertions that do not affect method logic, limitations related to the conservative flow of the analyses, and difficulties in accurately identifying changes involving annotations. Some analyses that take exceptions into account may also capture flows that do not actually exist, contributing to FPs. Furthermore, interprocedural techniques, by analyzing flows across different methods, are more likely to detect lower-level dependencies, which can increase the incidence of false positives compared to intraprocedural analyses.

The CD analysis, in turn, shows a reduced number of false positives precisely because it does not consider scenarios commonly present in the other analyses. Moreover, since it operates in an intraprocedural manner and detects fewer conflicts, it is less likely to capture lower-level dependencies, which helps reduce FPs. However, this more conservative approach, while effective in minimizing false positives, may also result in a lower detection rate of real conflicts.

Table 8 summarizes the results of the manual analysis of false positives (FP) and false negatives (FN) across the three evaluated combinations: CD, PDG-e or DF-Inter, and PDG-e or DF-Intra. These findings highlight the trade-offs between precision, recall, and overall applicability of each combination.

False positives remain a challenge in our approach. Among the 34 FP cases identified in our best-performing analysis combination, 13 (38%) are caused by refactoring

Table 8 Manual analysis of false positives and false negatives by combination

Type	Category	CD	PDG-e or DF-Inter	PDG-e or DF-Intra
FP	Conservative flow	1	6	6
	Refactoring	0	13	13
	Harmless code changes	0	7	7
	Harmless code insertion	0	4	2
	Annotation related	0	4	4
Total		**1**	**34**	**32**
FN	Exception edges	5	0	0
	Removed lines	4	4	3
	Dependencies	4	4	4
	Lack of conditional flow	4	0	0
	Complex conditionals	2	2	2
	Interface implementation	3	3	3
	Annotation related	2	0	0
Total		**24**	**13**	**12**

operations that do not impact semantic behavior. Lira et al. (2025) demonstrate that integrating refactoring detection tools with semantic conflict analysis can effectively filter such cases. Applying their approach to our dataset, their solution successfully identified 10 of these 13 refactoring instances (77%), reducing our false positives from 34 to 24. This demonstrates that combining our semantic conflict detection with automated refactoring detection tools can improve precision and accuracy by 8-12 percentage points, significantly enhancing the practical applicability of the approach. Our approach provides substantial value: it detects semantic conflicts that currently go undetected by existing test suites and code review processes, reports exact conflict lines and complete interference paths for quick verification, and enables developers to identify critical issues during development rather than after deployment, where remediation costs are substantially higher.

7 Threats to validity

7.1 Construct validity

The way dynamic semantic conflicts are identified can influence the results. Similar to the work of De Jesus et al. (2024), de Jesus et al. (2023), we use the concept of locally observable interference to validate semantic conflicts. However, this approach has an inherent limitation: the analysis only considers interferences detected within the analyzed method and the methods it calls, making a global interference analysis across the entire project unfeasible.

The use of metrics such as precision, recall, F1-score, and accuracy may not fully capture the effectiveness of the techniques in detecting real conflicts in merge scenarios, as they do not necessarily reflect the true nature and impact of these conflicts in practical settings.

Another potential threat relates to the completeness of the ground truth. Since it was manually created, there is a risk that our dataset underestimates the actual

number of semantic interferences. Even with rigorous review by multiple experienced researchers following a detailed protocol, it is possible that certain subtle or complex interferences were not identified by any of the experts. This could lead to an overestimation of our approach's precision, as some reported conflicts we classified as false positives might actually be genuine interferences that we failed to recognize during manual analysis. To partially mitigate this threat, we developed detailed specifications and conducted training sessions to ensure consistent criteria for identifying semantic conflicts, required review by at least two experts per case with disagreement resolution, and made our complete dataset publicly available to enable verification and extension by the research community.

7.2 Internal validity

Small variations in the generated bytecode can affect the static analysis. If a different compiler or a distinct version of OpenJDK is used, the results may change. To mitigate this effect, we use OpenJDK 8 (1.8.0_202), one of the most widely adopted versions.

Some techniques may generate more false positives or negatives, which can influence the perception of their effectiveness. For instance, the Control Dependence (CD) technique is highly effective in reducing the number of false positives, having high precision and accuracy. However, it has very low recall, meaning it may not detect some relevant conflicts.

It is important to emphasize that the count of "exclusive" does not refer to unique semantic conflicts identified by a technique, but rather to units where an analysis detected a FP or FN differently from the other analyses. A unit can have multiple types of conflicts, and not all of them are identified by a single analysis. Thus, one analysis may find a type of conflict that another does not, meaning these conflicts may be different, not exclusive. Therefore, we use the terms exclusive FP and FN, not exclusive conflicts. This is because we rely on the final result of each analysis, i.e., the decision to return "true" or "false" for that specific unit, without comparing the data or control flows generated by each technique. Furthermore, these flows may not correspond, as different techniques may focus on different aspects of the code, making the notion of "exclusive conflict" inapplicable.

7.3 External validity

The experiment was conducted on specific Java projects, limiting its applicability to other programming languages that cannot be converted to bytecode. Although we analyzed 32 projects, they may belong to a specific domain, making it difficult to generalize the results to all types of projects. Since the study is based on a dataset specific to the work of De Jesus et al. (2024), de Jesus et al. (2023), this may limit the validity of the findings for other contexts.

Although the experiment uses real scenarios and projects, its real impact on developers' workflows has not been tested in an industrial environment. The techniques were not evaluated in companies, which could affect their practical adoption and actual effectiveness. The choice of techniques may depend on computational cost and

ease of adoption by developers. Theoretical results do not always translate directly into practical benefits.

8 Related work

Our study investigates the combination of static analysis techniques to provide a comprehensive evaluation of their effectiveness across different metrics and contexts. While various studies explore different approaches, we observed that many exhibit high false positive or false negative rates or a significant disparity between precision and recall. Thus, our work aims to identify technique combinations that can enhance results in real-world scenarios.

Horwitz et al. first studied dynamic semantic conflicts (Horwitz et al. 1989, 1990; Yang et al. 1992) and formalized the interference definition we use here. They, however, proposed the construction of PDGs (Horwitz et al. 1989) and SDGs (Horwitz et al. 1990) for the four program versions in a merge scenario, and compute differences between these to detect and resolve interference; to compute differences, they require the use of structured editors to identify AST nodes changed in each of the revisions. Although inspired by this seminal work, and also using PDG and CD, our technique only analyzes the merge version—which is annotated with line changing information—and avoids some of the heavyweight analysis needed to construct SDGs. More recent work (Barros Filho 2017) gets close to that, using the JOANA framework (Hammer and Snelting 2009) to build a single SDG for the merge version in a scenario. This, however, can take hours to construct, whereas our analyses run much faster. Overall, our technique presents better but comparable precision; we were not able to compare recall due to the related work experiment design.

Instead of static analysis, Da Silva et al. (2020, 2022, 2024) detect interference by generating unit tests, running them on the four program versions, and analyzing the results. They rely on test passing criteria that suggests interference—if a test breaks in the base version, and passes in one of the parents, and breaks again in the merge is suggestive evidence that the change carried on by this parent is not preserved in the merge version. They need no line change information, and can detect interference due to removed lines and changes in field declarations. Our results surpassed theirs by selecting the best combination for each metric, demonstrating superior performance across all evaluated dimensions, including precision, recall, F1 score, and accuracy. Our performance should also be much better than theirs, based on the little information (only test generation time, not test execution time) they provide about that. The combination of both techniques could be a promising direction for future work.

A third technique for interference detection is explored by Sousa et al. (2018), which statically infers relational post-conditions from the merge scenario code versions, establishing constraints on how state elements can be modified by different versions in such a way to avoid interference. It then applies theorem proving, to check if the constraints are satisfiable. Performance from this verification approach can be considerably higher than our approach with larger programs that were changed in a number of places.

A similar technique by Muylaert et al. (2023) employs symbolic execution to detect semantic conflicts. Their approach defines the program semantics as path conditions produced by a symbolic executor and checks whether these conditions meet the established rules that indicate a merge conflict. Specifically, the method determines whether the conditions defined in the Left, Right, and Merge versions of a merge scenario are satisfied. The authors validated their approach using a synthetic dataset of 438 scenarios, which they generated through mutation testing. After manually analyzing ten randomly selected scenarios, they found that the approach accurately classified three as true positives and the remaining as false negatives. Instead, we leverage a real-world dataset of GitHub scenarios and conduct a detailed manual analysis of the entire dataset to assess the accuracy of the results. Furthermore, our approach is applied exclusively to the merged version of the program, rather than to all three versions.

A related technique by De Jesus et al. (2024), de Jesus et al. (2023) employs four static analyses - Interprocedural Direct Flow, Interprocedural Confluence, Interprocedural Override Assignment, and Program Dependence Graph - to detect semantic conflicts. Their method reports interference whenever at least one of these techniques detects it, effectively applying a logical OR to the results. While this strategy increases the number of detected interferences, it also leads to a low precision due to the high number of false positives. Instead, our study implements additional techniques, introduces new analyses (five analyses), and systematically evaluates multiple combinations of these techniques. By doing so, we provide a more nuanced assessment of their individual and combined effectiveness, achieving better precision and recall metrics and leading to a deeper understanding of their impact.

Brun et al. (2011, 2013), Kasi and Sarma (2013) also addresses the detection of both static and dynamic semantic conflicts, with a greater emphasis on static detection. For dynamic analysis, they employ tests. Brun et al. (2011, 2013) present Crystal, a tool that speculatively merges local developers' repositories, builds, and tests the results, aiming to detect and warn developers early about potential conflicts. Similarly, Kasi and Sarma (2013) introduce Palantír, a tool that monitors ongoing changes in developers' personal workspaces and continuously shares information about changes that might lead to conflicts. However, the authors observe build conflicts in a small number of projects – three in one study and four in the other. It is, therefore, important to observe the frequency of conflicts in a broader context. In our research, we explore aspects of dynamic semantic conflicts that these previous works do not address. We alse analyze false positives and negatives to assess the accuracy of our approach.

The work of Thorsten and Andrzejak (Wuensche et al. 2020) employs static analysis to detect build and test conflicts. They identify 54 build conflicts over a period of 22 months in the development process of a single project. However, they do not manage to identify any conflicts that cause test failures during the analyzed period, highlighting that build conflicts are more common and that the tests used, which are the project's own tests, are not sufficient to capture all types of conflicts. In our work, we apply our approach to 32 real projects. They adopt different types and approaches to conflicts, treating some refactoring cases as conflicts, which differs from our approach. In the ground truth that we used, this type of change is not considered interference, as it involves refactorings that do not alter the system's behavior. Our

approach emphasizes the complexities and nuances of dynamic semantic conflicts, while their work focuses on static conflicts.

Shen et al. (2023a) employed a three-step method to analyze and classify types of conflicts. They identify merge scenarios in 208 GitHub projects and replicate the git merge process, classifying conflicts as textual, build, or test based on the phase where the failure occurs. The process includes running the tests created by the developers and executing the project's build process. Textual conflicts were reported more frequently than build and test conflicts, with the latter being more challenging to resolve. The total number of semantic conflicts was 153 (107 build and 46 test). Most of these conflicts were related to configuration files and build scripts, rather than Java code altered by developers. Additionally, the entire process is automated to check if a failure occurs at any stage. In contrast, our work focuses on code changes and dynamic semantic conflicts, rather than build scripts or configuration files, which are static semantic conflicts. We address other types of more complex conflicts that are not captured by tests after the build process, which cannot be automatically classified and require manual peer review.

Zhang et al. (2022), Dinella et al. (2022), Svyatkovskiy et al. (2021, 2022), Dong et al. (2023), Shen et al. (2023b) address automatic merge conflict resolution using machine learning and large language models. Zhang et al. (2022) explore GPT-3 for textual conflict resolution, demonstrating that the model can harmonize structural code differences and preserve static semantic correctness (passing the build process). Dinella et al. (2022) developed DeepMerge, a neural model for JavaScript that applies rearrangement and editing operations on conflicting lines identified by diff3, though it cannot synthesize new code when existing lines are insufficient. Svyatkovskiy et al. (2021, 2022) present MergeBERT, which applies diff3 at the token level instead of line level, enabling finer-grained resolution, the authors explicitly acknowledge their technique does not guarantee semantic correctness. Dong et al. (2023) present MergeGen, employing an encoder-decoder generative model that can produce new tokens when necessary, overcoming limitations of previous classification-based techniques that were restricted to combining pre-existing code elements. Shen et al. (2023b) proposes CHATMERGE, combining machine learning with ChatGPT to generate conflict resolutions. However, these approaches address only textual and build conflicts, not semantic interference. Our work detects dynamic semantic conflicts using static analysis to identify interference in the merge version.

Previous research has extensively addressed various aspects of conflict management in software development. Works such as those by Apel et al. (2011), Sarma et al. (2011), Apel et al. (2012), Gligoric et al. (2014), Cavalcanti et al. (2017), Cavalcanti et al. (2019) have focused on the prevention, analysis, detection, or resolution of textual and static semantic conflicts. These studies employ a variety of techniques, including structured merging, to address conflicts that arise during software development and integration processes. Their contributions have provided valuable insights into how to manage conflicts effectively within these specific contexts. Our work builds on and extends these foundational studies by addressing dynamic semantic conflicts, an area that has not been sufficiently explored in the previous literature. We aim to analyze conflicts that are not identified by the traditional types of conflicts (textual, build, and test). We focus on identifying conflicts that go unnoticed by these analyses and only manifest as system failures during its use.

9 Conclusions

We present a set of static analyses that show significant capability in detecting interferences and hold promising potential for identifying dynamic semantic conflicts. It is crucial to select the most appropriate analyses and their combinations, tailored to the specific context of software development.

The choice between precision, recall, and accuracy should be driven by the context and the problem being addressed. In applications where precision is critical, such as fraud detection or medical diagnosis, minimizing false positives is essential. In contrast, in scenarios where recall is paramount, such as security vulnerability detection or disaster alerts, ensuring the detection of all relevant cases is crucial. In practice, the choice of prioritizing precision, recall, F1 score, or accuracy depends on the specific problem being addressed and the impact of false positives and false negatives on business outcomes. It should also account for class balance within the context, ensuring an optimal trade-off that enhances decision-making effectiveness.

Our analysis demonstrates that the Control Dependence (CD) analysis excels in precision, while the Program Dependence Graph with exception edges (PDG-e), combined with Interprocedural Direct Flow (DF Inter) analysis, outperforms in recall. Furthermore, the PDG-e and DF Intraprocedural analyses strike an effective balance between precision and recall. Additionally, CD shows consistent efficiency with lower execution time, while the other combinations, PDG-e or DF Inter and PDG-e or DF Intra, display greater variability in execution time.

Despite variations in metrics, static analysis techniques offer significant value due to their lightweight nature and ability to detect potential conflicts early. Current merge tools do not handle these conflicts, as they focus only on textual, build, and test conflicts. This limitation increases the risk of undetected semantic conflicts, which can cause significant issues in later stages. These techniques stand out by providing detailed reports and pinpointing the exact lines of code where dynamic semantic conflicts may arise. This makes verification faster and more targeted, in contrast to situations where problems are only detected after they cause real impacts, making it challenging to trace their root cause.

Acknowledgements For partially supporting this work, we would like to thank INES (National Institute of Science and Technology for Software Engineering) and the Brazilian research funding agencies CNPq, FACEPE (grants IBPG-0029-1.03/20), and CAPES. We thank Rafael Alves, Léuson da Silva and Vinícius dos Santos for the support when creating our dataset.

Author Contributions Contributing authors: gsj@cin.ufpe.br; phmb@cin.ufpe.br; rbonifacio@unb.br; mbo2@cin.ufpe.br; G.S.J. performed data curation, conceptualization, investigation, methodology, software development, and writing – original draft, as well as writing – review and editing. P.B. contributed to conceptualization, funding acquisition, methodology, supervision, validation, and writing – original draft and review and editing. R.B. contributed to conceptualization, methodology, supervision, validation, and writing – original draft and review and editing. M.B.O. contributed to data curation, investigation, and software development. All authors reviewed and approved the final version of the manuscript.

Data Availability The dataset and scripts used to run our study are available in our Appendix (2025).

Declarations

Competing interests The authors declare no competing interests.

References

Apel, S., Leßenich, O., Lengauer, C.: Structured merge with auto-tuning: balancing precision and performance. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 120–129 (2012). https://doi.org/10.1145/2351676.2351694

Apel, S., Liebig, J., Brandl, B., Lengauer, C., Kästner, C.: Semistructured merge: rethinking merge in revision control systems. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 190–200 (2011). https://doi.org/10.1145/2025113.2025141

Appendix, O.: Data-Availability Statement. Last accessed 2 Mar 2025 (2025). https://spgroup.github.io/papers/interference-static-analyses-comparison.html

Barros Filho, R.S.M.D.: Using information flow to estimate interference between same-method contributions. Master's thesis, Federal University of Pernambuco (2017)

Binkley, D., Horwitz, S., Reps, T.: Program integration for languages with procedure calls. ACM Trans. Softw. Eng. Methodol. 3–35 (1995). https://doi.org/10.1145/201055.201056

Brun, Y., Holmes, R., Ernst, M.D., Notkin, D.: Early detection of collaboration conflicts and risks. IEEE Trans. Softw. Eng. 1358–1375 (2013). https://doi.org/10.1109/TSE.2013.28

Brun, Y., Holmes, R., Ernst, M.D., Notkin, D.: Proactive detection of collaboration conflicts. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 168–178 (2011). https://doi.org/10.1145/2025113.2025139

Cavalcanti, G., Borba, P., Accioly, P.: Evaluating and improving semistructured merge. Proceedings of the ACM on Programming Languages, 1–27 (2017). https://doi.org/10.1145/3133883

Cavalcanti, G., Borba, P., Seibt, G., Apel, S.: The impact of structure on software merging: semistructured versus structured merge. In: 34th IEEE/ACM International Conference on Automated Software Engineering, pp. 1002–1013 (2019). https://doi.org/10.1109/ASE.2019.00097

Chacon, S., Straub, B.: Pro Git. Springer, Book (2014)

Da Silva, L., Borba, P., Maciel, T., Mahmood, W., Berger, T., Moisakis, J.: Detecting semantic conflicts with unit tests. J. Syst. Softw. (2024). https://doi.org/10.1016/j.jss.2024.112070

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

Da Silva, L., Borba, P., Pires, A.: Build conflicts in the wild. J. Softw. Evol. Process. **2441** (2022). https://doi.org/10.1002/smr.2441

De Jesus, G.S., Borba, P., Bonifácio, R., De Oliveira, M.B.: Lightweight semantic conflict detection with static analysis. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, pp. 343–345 (2024). https://doi.org/10.1145/3639478.3643118

Dinella, E., Mytkowicz, T., Svyatkovskiy, A., Bird, C., Naik, M., Lahiri, S.: Deepmerge: Learning to merge programs. IEEE Trans. Software Eng. **49**(4), 1599–1614 (2022). https://doi.org/10.1109/TSE.2022.3183955

Dong, J., Zhu, Q., Sun, Z., Lou, Y., Hao, D.: Merge conflict resolution: Classification or generation? In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1652–1663 (2023). https://doi.org/10.1109/ASE56229.2023.00155 . IEEE

Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 319–349 (1987). https://doi.org/10.1145/24039.24041

Gligoric, M., Majumdar, R., Sharma, R., Eloussi, L., Marinov, D.: Regression test selection for distributed software histories. In: Computer Aided Verification: 26th International Conference, pp. 293–309 (2014). https://doi.org/10.1007/978-3-319-08867-9_19. Springer

Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Secur. 399–422 (2009). https://doi.org/10.1007/s10207-009-0086-1

Horwitz, S., Prins, J., Reps, T.: Integrating noninterfering versions of programs. ACM Trans. Prog. Lang. Syst. 345–387 (1989). https://doi.org/10.1145/65979.65980

Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Prog. Lang. Syst. 26–60 (1990). https://doi.org/10.1145/77606.77608

Jesus, G.S., Borba, P., Bonifácio, R., Oliveira, M.B.: Detecting Semantic Conflicts using Static Analysis (2023). https://arxiv.org/abs/2310.04269

Kasi, B.K., Sarma, A.: Cassandra: Proactive conflict minimization through optimized task scheduling. In: 2013 35th International Conference on Software Engineering, pp. 732–741 (2013). https://doi.org/10.1109/ICSE.2013.6606619. IEEE

Lhoták, O., Hendren, L.: Scaling java points-to analysis using spark. In: Hedin, G. (ed.) Compiler Construction, pp. 153–169. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/3-540-36579-6_12

Lhoták, O.: Spark: A flexible points-to analysis framework for java. PhD thesis (2003)

Lira, V., Borba, P., Bonifácio, R., Matheus barbosa, G.S.: RefFilter: Improving Semantic Conflict Detection via Refactoring-Aware Static Analysis (2025). https://arxiv.org/abs/2510.01960

Muylaert, W., Härtel, J., De Roover, C.: Symbolic execution to detect semantic merge conflicts. In: 23rd IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE International Working Conference on Source Code Analysis and Manipulation, Online (2023). https://doi.org/10.1109/SCAM59687.2023.00028

Pastore, F., Mariani, L., Micucci, D.: Bdci: Behavioral driven conflict identification. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 570–581 (2017). https://doi.org/10.1145/3106237.3106296

Sarma, A., Redmiles, D.F., Van Der Hoek, A.: Palantir: Early detection of development conflicts arising from parallel code changes. IEEE Transactions on Software Engineering, 889–908 (2011). https://doi.org/10.1109/TSE.2011.64

Shao, D., Khurshid, S., Perry, D.E.: Sca: a semantic conflict analyzer for parallel changes. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 291–292 (2009). https://doi.org/10.1145/1595696.1595747

Shen, B., Gulzar, M.A., He, F., Meng, N.: A characterization study of merge conflicts in java projects. ACM Trans. Softw. Eng. Methodol. 1–28 (2023a). https://doi.org/10.1145/3546944

Shen, C., Yang, W., Pan, M., Zhou, Y.: Git merge conflict resolution leveraging strategy classification and llm. In: 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS), pp. 228–239 (2023b). https://doi.org/10.1109/QRS60937.2023.00031

Silva, L.D., Borba, P., Maciel, T., Mahmood, W., Berger, T., Moisakis, J., Gomes, A., Leite, V.: Detecting Semantic Conflicts with Unit Tests (2023). https://doi.org/10.48550/arXiv.2310.02395

Smaragdakis, Y., Balatsouras, G.: Pointer analysis. Foundations and Trends® in Programming Languages, 1–69 (2015). https://doi.org/10.1561/2500000014

Sousa, M., Dillig, I., Lahiri, S.K.: Verified three-way program merge. Proc. ACM Program. Lang. (2018). https://doi.org/10.1145/3276535

Sui, Y., Xue, J.: Svf: interprocedural static value-flow analysis in llvm. In: Proceedings of the 25th International Conference on Compiler Construction, pp. 265–266 (2016). https://doi.org/10.1145/2892208.2892235

Sung, C., Lahiri, S.K., Kaufman, M., Choudhury, P., Wang, C.: Towards understanding and fixing upstream merge induced conflicts in divergent forks: An industrial case study. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice, pp. 172–181 (2020). https://doi.org/10.1145/3377813.3381362

Svyatkovskiy, A., Fakhoury, S., Ghorbani, N., Mytkowicz, T., Dinella, E., Bird, C., Jang, J., Sundaresan, N., Lahiri, S.K.: Program merge conflict resolution via neural transformers. In: Proceedings of the 30th ACM Joint European software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2022, pp. 822–833. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3540250.3549163

Svyatkovskiy, A., Mytkowicz, T., Ghorbani, N., Fakhoury, S., Dinella, E., Bird, C., Sundaresan, N., Lahiri, S.: MergeBERT: Program Merge Conflict Resolution via Neural Transformers (2021). https://www.microsoft.com/en-us/research/publication/mergebert-program-merge-conflict-resolution-via-neural-transformers/

Towqir, S.S., Shen, B., Gulzar, M.A., Meng, N.: Detecting build conflicts in software merge for java programs via static analysis. In: 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–13 (2022). https://doi.org/10.1145/3551349.3556950

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: A java bytecode optimization framework. In: CASCON First Decade High Impact Papers, pp. 214–224. Citeseer, Online (2010). https://doi.org/10.1145/1925805.1925818

Wuensche, T., Andrzejak, A., Schwedes, S.: Detecting higher-order merge conflicts in large software projects. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification, pp. 353–363 (2020). https://doi.org/10.1109/ICST46399.2020.00043

Yang, W., Horwitz, S., Reps, T.: A program integration algorithm that accommodates semantics-preserving transformations. ACM Transactions on Software Engineering and Methodology, 310–354 (1992). https://doi.org/10.1145/131736.131756

Zhang, J., Mytkowicz, T., Kaufman, M., Piskac, R., Lahiri, S.K.: Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper). In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 77–88 (2022). https://doi.org/10.1145/3533767.3534396

Authors and Affiliations

Galileu Santos de Jesus[1] · Paulo Borba[1] · Rodrigo Bonifácio[2] · Matheus Barbosa de Oliveira[1]

✉ Galileu Santos de Jesus
gsj@cin.ufpe.br

Paulo Borba
phmb@cin.ufpe.br

Rodrigo Bonifácio
rbonifacio@unb.br

Matheus Barbosa de Oliveira
mbo2@cin.ufpe.br

[1] Centro de Informática, Universidade Federal de Pernambuco, Recife, Pernambuco, Brazil

[2] Departamento de Ciência da Computação, Universidade de Brasília, Brasília, Distrito Federal, Brazil