The Impact of Static Analysis Depth Limit on Semantic Conflict Detection

Pedro Patriota
Centro de Informática, Universidade Federal de Pernambuco
Recife, Brazil
pasp@cin.ufpe.br

Matheus Barbosa
Centro de Informática, Universidade Federal de Pernambuco
Recife, Brazil
mbo2@cin.ufpe.br

Paulo Borba
Centro de Informática, Universidade Federal de Pernambuco
Recife, Brazil
phmb@cin.ufpe.br

Abstract

Collaborative software development requires periodic code integration of independent contributions into a shared codebase, a process that may introduce dynamic semantic conflicts (behavioral interference between changes not captured by textual merge tools). Static-analysis-based approaches for detecting such conflicts rely on configuration parameters that affect both detectability accuracy and computational cost. A key parameter is the method traversal depth limit, which determines how deeply chains of method calls are followed when tracking interference. In this paper, we analyze how varying the depth limit affects both the number of reported interferences and the execution performance of the Overriding Assignment (OA) analysis for interference detection. We evaluate it on two datasets from previous studies: the MergeDataset, a manually curated benchmark, and the RefDataset, a broader and more diverse dataset covering a wider range of project sizes and merge patterns. Both datasets comprise Java open-source projects from GitHub, and our findings are consistent across them. We execute the analyses limiting the depth to 5, 10, and 20 method calls. Our results show that increasing the depth limit does not consistently improve detection: the relationship is non-monotonic. The gain is most notable when moving from depth 5 to 10, where the number of newly reported scenarios increases. Beyond depth 10, however, the number of detected scenarios and total interferences both decrease considerably. These gains come at a measurable cost: average analysis time and memory consumption increase noticeably at greater depths, with diminishing returns. Nevertheless, accuracy and precision tend to improve from depth 5 to 10, but degrade when moving further to depth 20.

Keywords

Merge Tools, Semantic Conflict, Static Analysis, Depth Tuning, Pointer Analysis

1 Introduction

Software projects depend on frequent merges, and resolving conflicts remains a significant challenge in practice [4, 18, 20]. Beyond textual clashes reported by tools such as `diff3` [13], parallel edits can alter program behavior and cause dynamic semantic conflicts. Early detection is therefore critical to integration quality [10, 19].

Detecting these conflicts requires reasoning about behavior. Early static analyses for semantic conflicts relied on Program Dependency Graphs and System Dependence Graphs [12], which are impractical for moderate-size codebases [3]. Barbosa et al. [3] proposed *Overriding Assignment* (OA) analysis, a lightweight static analysis [6] that reports interference when two developers write the same state element without a base write in between. Their interprocedural OA (iOA) variant traverses called methods recursively to a configurable *depth limit*, but their evaluation fixed it at 5 without examining how the choice affects recall or execution cost.

The depth limit is a critical, yet underexplored, configuration parameter of iOA. When the limit is too shallow, the analysis halts before reaching write operations buried in called methods, missing interferences and reducing recall. When it is too deep, the traversal covers an increasingly large portion of the call graph, raising execution time and memory to potentially prohibitive levels. There is thus a need for principled guidance on how to configure it.

In this paper, we address this gap by empirically investigating how varying the depth limit (set to 5, 10, and 20) affects both the detectability of OA interferences and the execution performance of the analysis, using two datasets from prior work. The *MergeDataset* [7] contains 99 three-way merge experimental units comprising methods mutually modified by both developers across Java open-source GitHub projects, with manual ground-truth interference labels. The *RefDataset* [17] is a larger collection of 907 experimental units (of which 846 pass an additional build filter, described in Section 4.1) mined from a broader set of Java projects, covering a wider range of project sizes and merge patterns, without per-scenario manual ground truth. Both datasets comprise real merge scenarios from complete projects at different scales and validation levels.

To investigate this trade-off, we run iOA using the SPARK pointer-analysis framework (built on top of Soot) [14], following the most prominent configuration in Barbosa et al. [2].[1] For each depth limit, we execute the analysis ten times per experimental unit to gather stable performance measurements. Because timeouts are a known concern in prior work [2], we adopt a *partial-result strategy* that caps analysis at the point of timeout explosion, allowing us to include partial results up to that boundary rather than discarding the entire scenario.

From these executions we collect execution time both at the dataset level and per scenario, as well as the number of scenarios for which iOA reports at least one interference, and the overall set of detected interferences. For the MergeDataset, we further evaluate the behavior of newly reported scenarios at each depth, computing precision, recall, accuracy, and F1-score to enable a direct comparison across depth configurations.

The main contributions of this paper are: *i.* an empirical evaluation of the impact of depth tuning on OA interference detectability; *ii.* a structural characterization of interference locality and call-depth distribution across the studied datasets; *iii.* a characterization

[1] SPARK restricts callee resolution to classes actually instantiated via `new`, pruning the call graph to executable paths [2].

of the corresponding performance costs across two datasets of distinct scales; *iv.* a practical recommendation that depth values close to 10 represent a suitable configuration for the analyzed datasets, reducing the share of entirely missed scenarios while remaining computationally feasible for most scenarios; and *v.* a *partial-result strategy* that caps analysis at timeout boundaries, allowing timed-out scenarios to still contribute detections rather than being discarded entirely.

2 Motivating Example

To demonstrate how the depth limit affects the detection of interferences, consider the merge scenario shown in Figure 1.

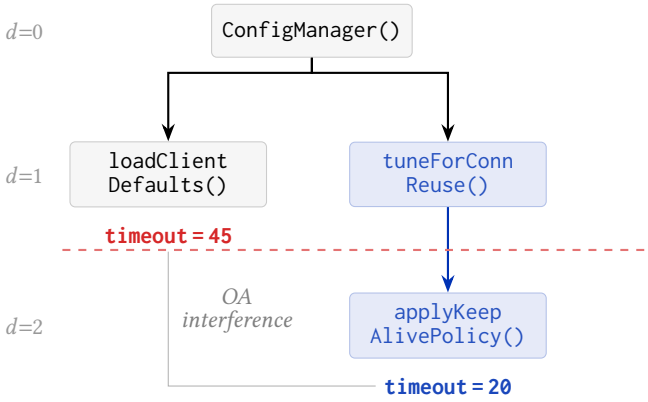
```
1  public class ConfigManager {
2    private int timeout;
3    private int retries;
4
5    public ConfigManager() {
6      loadClientDefaults();
7      tuneForConnectionReuse();
8    }
9
10   public void loadClientDefaults() {
11     this.timeout = 60 45;
12     this.retries = 3 2;
13     ...
14   }
15
16   public void tuneForConnectionReuse() {
17     applyKeepAlivePolicy();
18     ...
19   }
20   public void applyKeepAlivePolicy() {
21     this.timeout = 20;
22     ...
23   }
24 }
```

Developer Left's change Developer Right's change

Figure 1: Merged `ConfigManager` code from a three-way merge.

It shows the Java code of the `ConfigManager` class resulting from a three-way merge in which two developers modified the base version in parallel. Developer Left modified the existing values in `loadClientDefaults`, changing `timeout` from 60 to 45 and `retries` from 3 to 2. Developer Right introduced a call to `tuneForConnectionReuse` in the constructor and provided its implementation, which eventually sets `timeout = 20` inside `applyKeepAlivePolicy`.

This constitutes an OA interference because Developer Right's write operation to `timeout` that is defined inside `applyKeepAlivePolicy` overrides Developer Left's write operation `timeout = 45` with no textual conflict. The merged program silently applies a 20-second timeout instead of the 45-second value Developer Left intended. No textual conflict is reported because the two changes do not touch the same or consecutive lines.



Depth limit	Writes observed	OA detected?
1	`timeout = 45` only	No
≥ 2	`timeout = 45`, `timeout = 20`	**Yes**

Figure 2: Depth-limited analysis traversal of `ConfigManager`.

To detect semantic interference in merged code, iOA traverses method calls up to a configurable *depth limit (d)*, recording every write to state elements it encounters along the way.[2] Figure 2 illustrates how that limit affects what the analysis can observe. Starting at the constructor (d=0), the analysis follows each call one level deeper and, at depth 1, observes the left-side write in `loadClientDefaults` and enters `tuneForConnectionReuse`, but halts before descending into `applyKeepAlivePolicy`, so the right-side write remains unseen. With limit 2 it reaches `applyKeepAlivePolicy`, sees both writes to `timeout` without an intervening base write, and detects the OA interference.

This example illustrates the core trade-off: deeper traversal reaches more write sites, reducing the chance a genuine interference goes undetected, but at the cost of more analysis work and longer execution times.

In practice, some scenarios escalate into exponential growth of analysis states and trigger a *timeout explosion*: a single subtree of the call graph can expand into exponentially many paths, exhausting the analysis budget before other subtrees are visited; an interference located in a different branch may never be reached despite falling within the declared depth limit. Like the depth limit, the per-scenario timeout is a configurable parameter: without it, analysis could run indefinitely on pathological call graphs. Taken together, timeout explosion and depth-limit sensitivity make the choice of depth limit a trade-off between *detectability* and *computational cost*. Understanding how different depth configurations shift this balance, across precision, recall, and execution time, motivates our study.

[2]State elements refer to program entities that hold or represent the state of a program during its execution. These include global variables, object fields, system files and streams, variables that store method return values, raised exceptions, and any other entities that maintain state information across different execution contexts.

3 Static Analysis

Throughout this paper, "merge" means a three-way merge with a common ancestor and two divergent versions (left and right), regardless of whether they originate from local branches, remote branches, or pull requests; OA analysis runs on the merged version after the textual merge is performed.

An OA interference occurs, for example, when a merge integrates two developers' changes that write to the same state element with no base-version write in between, causing one write to silently override the other. This condition is a conservative approximation of a dynamic semantic conflict: it signals a possible behavioral interference, but it does not guarantee that the merged program will exhibit one. Static analysis can detect these interferences as an approximation to finding semantic conflicts.

One approach to this problem, which we focus on, is the *Overriding Assignment* analysis [3], a lightweight static analysis that detects interferences in merged code. The OA analysis scans the merged code, which is annotated so that each instruction carries the identity of the developer who introduced it or marks it as base code. The algorithm maintains two definition sets, S_L and S_R, one per developer. As it scans the merged code, each write instruction adds its target state element to the corresponding set; an OA interference is detected as soon as a write targets an element already present in the *other* set, since this means both developers modified the same state without a base write in between, exactly the overriding condition. When the analysis encounters a base-version write (a write not marked as left or right), it removes that state element from both S_L and S_R, resetting the interference boundary for that element.

The iOA analysis extends this scan across method-call boundaries: starting from the annotated entry method modified by both developers, the iOA follows each call recursively up to a configurable *depth limit*, applying the same S_L/S_R logic in each visited callee. The call graph that governs which callees to visit is built using SPARK [14].

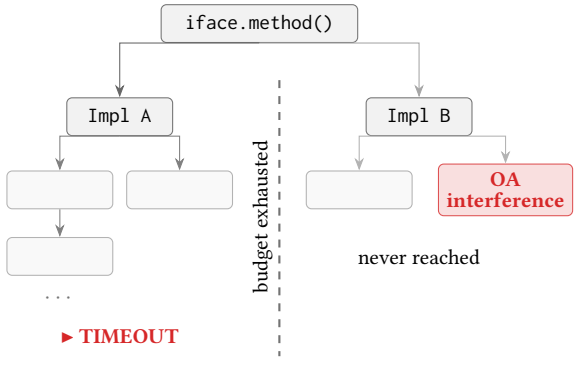


Figure 3: Call-graph timeout explosion.

A compounding factor is that virtual dispatch introduces sibling subtrees: both CHA (Class Hierarchy Analysis, a conservative algorithm that includes all subtypes as potential callees) and SPARK include multiple implementations as potential callees of a call site. The analysis explores these sequentially; if one subtree exhausts the time budget, the remaining siblings go unexplored, silently missing any interferences they contain. Figure 3 illustrates this: *Impl A*'s large subtree consumes the budget before *Impl B* is visited, so *Impl B*'s OA interference is missed (a false negative caused not by insufficient depth but by budget exhaustion in a sibling subtree). This is especially problematic because iOA follows a depth-first traversal order: it fully explores one callee subtree to the configured depth limit before backtracking to sibling branches, allowing a deep or combinatorially large subtree to consume the entire per-scenario time budget and leave shallower siblings unexplored.

4 Study Setup

We investigate the impact of varying the iOA depth limit on interference detection and performance, evaluating changes in false positives (FP: reported when none exists) and false negatives (FN: missed actual interferences) across depth configurations, alongside execution time and memory usage. We address the following research questions:

RQ1. How does the depth limit affect the number of reported OA interferences?
RQ2. How is accuracy affected by depth limitation?
RQ3. How do different depth configurations affect computational performance?

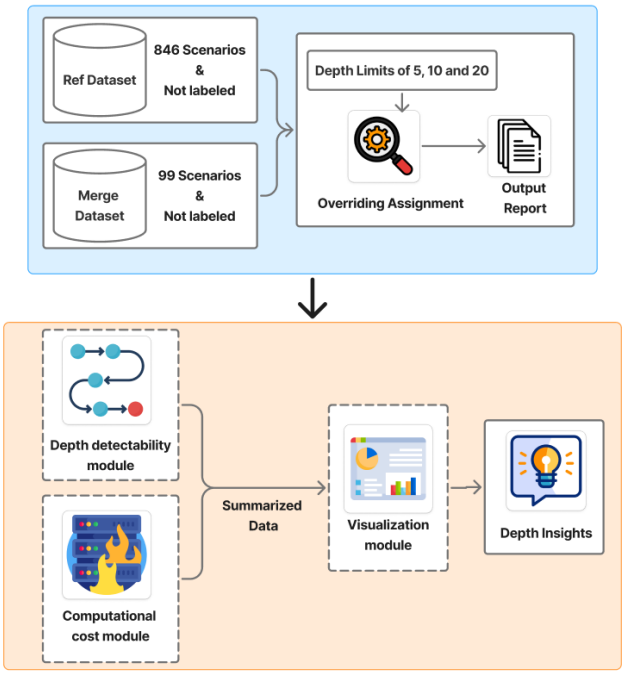


Figure 4: Overview of the study methodology.

Together, these questions provide insights into the structural properties of OA interferences across depth configurations, including how interferences distribute across call depths and whether they occur within the same method, class, or across class boundaries

(characteristics we map empirically and present as part of our results). This enables practitioners and researchers to make informed decisions when tuning the depth parameter to achieve better results in their semantic conflict detection analyses (Figure 4).

4.1 Dataset

We use two datasets from prior work. The *MergeDataset* [7] comprises merge scenarios (base, left, right, and merge versions) in which two contributors changed the same methods or constructors. It provides a manually labeled set of 99 experimental units associated with 54 merge scenarios from 32 open-source Java systems. As a merge scenario might have more than one declaration that was independently changed by both developers, we consider pairs (scenario, declaration) as our experimental units. The MergeDataset scenarios [5, 7, 21, 23] have been used in previous studies on semantic conflicts.

Similarly, the RefDataset [17] contains methods or constructors modified by two contributors. The RefDataset consists of 907 experimental units from 295 projects, selected and balanced satisfying the following criteria: GitHub's greatest hits repositories with at least 10 stars, using Maven or Gradle for build automation, and with the latest main branch commit successfully compiling and passing tests within 30 minutes on JDK 8, 11, or 17. This ensures a higher likelihood of automated `.jar` file generation, which is needed for running the analyses. Lastly, we apply an additional filter to confirm that the target class and method changed by two developers are included in the final build's .jar file. After this filter, 846 experimental units remain and form the dataset used in this study.

For each scenario in both datasets, we obtain the JAR file of the merge commit version without external dependencies. This reduces the amount of code analyzed, improving execution time, although it may decrease accuracy; this trade-off is promising for practical uses of static analysis in detecting interference, and thus we adopt this design. Along with this, we collect the line numbers corresponding to the changes introduced by the Left and Right sides. We focus on cases where two developers modified the same method, as such scenarios are more prone to behavioral interference.

4.2 Study Procedures

For RQ1, our goal is to understand how the depth limit shapes the composition of reported OA interferences within merge scenarios, that is, how many interferences are found, at which depths they surface, and how they are distributed across scenarios. Because the *RefDataset* does not provide per-scenario manual labels for dynamic semantic conflicts, we analyze the number of reported interferences as the observable output of the analysis rather than the number of true dynamic semantic conflicts.

To address this, we execute iOA using SPARK at depth limits of 5, 10, and 20 over both the *MergeDataset* and *RefDataset* (our labeled dataset and its extension, respectively). We select these depth values to investigate the impact of increasing traversal depth: depth 5 represents the default established in prior work [3], while depths 10 and 20 allow us to explore how deeper method call traversals affect interference detection. We choose the SPARK pointer-analysis framework as our call-graph construction strategy, following the most prominent configuration evaluated in prior work on pointer analysis refinements to iOA [2]. For each configuration, we monitor the analysis output per scenario, tracking not only whether iOA detects an interference, but also the specific depth at which each interference occurs, along with the classes and methods involved. Since a single experimental unit may contain multiple interferences, we also examine the composition of interferences within each scenario, analyzing how individual interferences relate to one another and how that composition changes across depth configurations.

For RQ2, our goal is to assess how accurately each depth configuration identifies true semantic conflicts, understanding whether deeper traversal improves or degrades the quality of reported results. To measure accuracy, we restrict this analysis to the *MergeDataset*, which provides per-scenario ground truth based on the locally observable interference (LOI) criterion [3]. We compute precision, recall, and F1-score for each depth configuration by comparing the analysis output against this ground truth.

For RQ3, we characterize the computational cost of each depth configuration by monitoring execution time (total and per-scenario, as reported by iOA) and memory utilization (sampled at one-second intervals).

4.3 Experimental Setup

All experiments use Soot 4.5.0 with default settings from Barbosa et al. [2]. We preserve original variable names and line numbers in bytecode, exclude standard Java libraries (except basic classes), and analyze only project code. Analyses run in 10 Docker containers on Ubuntu 22.04 LTS, each with 20 GB memory and 4 CPU cores. We classify a scenario as a positive result when iOA detects at least one OA interference before the analysis either completes or reaches the predefined timeout. Given the timeout explosion behavior described in Section 3, we impose a time limit of 20 minutes per scenario. Rather than discarding scenarios that hit this limit, we adopt a *partial-result strategy*: the algorithm caps the analysis at the timeout boundary and records whatever interference the traversal reaches up to that point, so timed-out scenarios still contribute to detection counts and accuracy evaluation.

We repeat computational cost analyses 10 times with low variability across runs. To ensure reproducibility, all experimental artifacts are publicly available [1].

5 Results and Discussion

5.1 Dataset Characteristics: Interference Locality and Symmetry

Since each depth configuration may surface different interferences (deeper traversals can reach write sites invisible at shallower depths), understanding the structural characteristics of interferences in our datasets provides context for interpreting the RQ findings.

We characterize interference composition by analyzing whether interferences in a given scenario occur within the same class and method across the left and right changes. Table 1 shows the percentage of scenarios in which more than 99% of their interferences share the same class and method across both developers' changes.

Most interferences occur within the same class and method across the left and right changes, suggesting that the studied conflicts are largely localized to modified methods. We also characterize

Table 1: Percentage of scenarios where >99% of interferences share the same class and method across both changes.

	MergeDataset		*RefDataset*	
Depth	Class (%)	Method (%)	Class (%)	Method (%)
5	100.0	93.3	99.5	94.8
10	100.0	85.7	99.5	94.5
20	100.0	92.3	98.5	93.7

interference symmetry by computing the *depth diff*, the absolute difference between the call depth at which the left interference and the right interference occur. In the motivating example, Developer Left's write occurs at depth 1 (`loadClientDefaults`) and Developer Right's at depth 2 (`applyKeepAlivePolicy`), yielding a depth diff of 1. Table 2 reports, for each depth configuration, the percentage of scenarios in which at least 99% of their interferences have a depth diff of zero.

Table 2: Percentage of scenarios where ≥99% of interferences have depth diff = 0 (left and right interference at the same call depth), with scenario counts.

Depth	*MergeDataset*	*RefDataset*
5	66.7	83.6
10	60.0	78.4
20	69.2	77.1

In most cases, both sides of the merge access the conflicting state from the same call depth. This suggests that, in these datasets, OA interferences tend to be symmetric: both branches modify the same resource through call paths of equivalent length. The pattern holds across depth configurations and both datasets, suggesting it is not an artifact of a particular depth setting.

5.2 RQ1: Interference Detection Across Depth Configurations

Table 3 shows the number of scenarios for which iOA using SPARK reported at least one interference (detected) and those for which no interference was reported (not detected), for each depth configuration across both datasets. Note that a scenario is counted as detected when at least one OA interference is found within it; Table 4 separately counts the total number of individual interferences reported, so one detected scenario can contribute multiple interferences.

Table 3: Number of experimental units with at least one detected OA interference vs. not detected, per depth.

	MergeDataset		*RefDataset*	
Depth	Detected	Not Detected	Detected	Not Detected
5	20	79	256	590
10	18	81	263	583
20	16	83	247	599

The results reveal a non-monotonic relationship between depth and detection. Increasing the depth from 5 to 10 allows the analysis to reach write operations buried deeper in call chains, bringing previously unreachable write sites into scope and raising the number of detected scenarios in the *RefDataset*. However, pushing the depth further to 20 reverses this trend: the number of detected scenarios decreases in both datasets. This is consistent with the timeout explosion behavior described in Section 3: deeper traversals increase the probability of exhausting the per-scenario time budget before the analysis completes, causing scenarios that would have been detected at shallower depths to time out and produce no result. This tension illustrates that depth tuning requires finding a balance: too shallow a limit misses interferences hidden in deep call chains, while too deep a limit risks exceeding the computational budget and silently dropping detections altogether. Notably, in the *MergeDataset*, this timeout-driven suppression is already visible at depth 10, where detected scenarios drop from 20 to 18: even a moderate depth increase can trigger timeout explosions in scenarios that would otherwise be detectable.

Table 4 complements this view by showing the total number of individual interferences reported across all scenarios, rather than just the scenario count.

Table 4: Total number of OA interferences reported per depth configuration and dataset.

Depth	*MergeDataset*	*RefDataset*
5	22	468
10	31	477
20	19	423

To understand why total interference counts can increase even when fewer scenarios are detected, Table 5 shows the distribution of scenarios by their number of interferences, grouped into depth ranges. Percentages are relative to the number of detected scenarios at each configuration.

Table 5: Distribution of detected scenarios by number of interferences per scenario (% of detected scenarios).

	Interferences per scenario				
Configuration	0–1	2–3	4–6	7–10	11+
depth 5, *Merge*	26.7%	6.6%	66.7%	—	—
depth 10, *Merge*	26.7%	6.6%	46.7%	20.0%	—
depth 20, *Merge*	30.8%	7.7%	53.8%	7.7%	—
depth 5, *Ref*	36.2%	37.5%	26.3%	—	—
depth 10, *Ref*	35.3%	30.7%	22.1%	11.9%	—
depth 20, *Ref*	37.1%	30.2%	22.0%	6.8%	3.9%

The distribution reveals a key mechanism behind the total interference counts in Table 4. At depth 5, all detected scenarios are confined to the 0–6 interference range. As the depth increases to 10 and 20, a portion of scenarios shifts into higher ranges (7–10 and beyond), meaning that the scenarios which do complete without hitting the timeout are able to reach more interferences per run. This explains why depth 10 yields more total interferences than

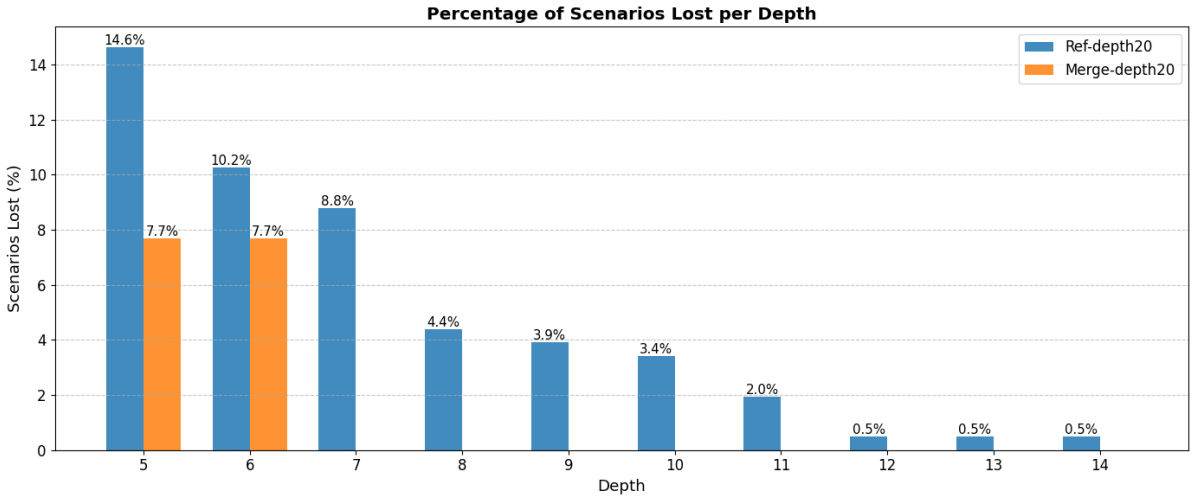


Figure 5: Percentage of scenarios lost per depth configuration, with depth 20 fixed as reference to isolate the effect of call-graph traversal depth on detection.

depth 5 despite detecting fewer scenarios in the *MergeDataset*: scenarios that avoid timeout explosion reach more interferences per run. At depth 20 the timeout pressure outweighs this gain, reducing both detected scenarios and total interferences.

To isolate the effect of the depth limit itself, *independent of timeout explosion*, we fix depth 20 as a reference and measure, for each shallower depth value, the percentage of *scenarios lost*: scenarios in which every detected interference occurs at a depth greater than the candidate limit, meaning a shallower setting would have reported nothing for that scenario at all. Figure 5 shows this loss curve from depth 5 to 14. This analysis assumes depth 20 is sufficient to surface all reachable interferences; interferences beyond depth 20 would be missed at all evaluated configurations and are outside our study's scope.

The figure shows that shallow depth limits impose a meaningful detection penalty solely because of the depth restriction, before any timeout effect is considered. At depth 5, the default used by Barbosa et al. [3], 14.6% of *RefDataset* scenarios and 7.7% of *MergeDataset* scenarios are entirely missed because all their interferences reside deeper in the call graph. The loss decreases sharply as depth grows: at depth 10, scenario loss drops to 3.4% for *RefDataset* and 0% for *MergeDataset*, indicating that most detectable interferences are accessible within 10 levels of call-chain traversal. Beyond depth 10, the *MergeDataset* shows no further losses, and *RefDataset* losses fall to 0.5% or below.

RQ1 Answer

The relationship between depth and detection is non-monotonic: increasing the limit from 5 to 10 raises the number of detected scenarios and total interferences, but pushing further to 20 reverses these gains due to timeout explosion. Depth 10 captures nearly all reachable interferences while limiting timeout exposure.

5.3 RQ2: Accuracy Across Depth Configurations

To evaluate accuracy, we use the *MergeDataset*, which provides per-scenario ground truth (see Section 4.2). Each detected scenario is classified as a true or false positive by comparing the analysis output against that ground truth, allowing us to compute precision, recall, and F1-score for each depth configuration. Table 6 reports the classification summary for each depth.

Table 6: Classification summary for *MergeDataset* across depth configurations

	Depth 5	Depth 10	Depth 20
TP	5	5	4
FP	15	13	12
TN	55	57	58
FN	24	24	25

iOA using SPARK is one of several approaches for detecting semantic conflicts, and the *MergeDataset* contains scenarios that are inherently outside its detection scope regardless of depth. Absolute metric values therefore reflect both technique limitations and dataset composition; the meaningful signal lies in how metrics *shift* across depth configurations.

Several patterns emerge from Table 6. First, the number of true positives remains stable from depth 5 to 10 (TP = 5), meaning the additional call-chain reach gained by going deeper does not convert any missed true interference into a detection: the newly reachable write sites at depth 10 correspond to false positives, not ground-truth interferences. Second, the reduction in false positives from depth 5 to 10 (FP: 15 → 13) is best explained by the timeout explosion effect described in Section 3: scenarios that were previously reported as false positives at depth 5 now exhaust the time budget before producing any output, causing them to silently move from the detected set to TN. This is exactly the timeout-driven artifact

that the RQ2 accuracy analysis highlights in Table 7. This FP reduction is therefore coincidental (an artifact of the timeout suppressing output, not of the analysis becoming more precise), which tempers the apparent accuracy gain at depth 10. This is confirmed by the corresponding TN increase (55 → 57), which accounts for exactly the two FP scenarios that dropped out. The FN count remains unchanged (24), consistent with the observation from Figure 5 that depth 10 loses no true scenarios relative to the depth-20 baseline. Going from depth 10 to 20, the pattern shifts: one TP is lost (5 → 4) and FN grows by one (24 → 25), meaning a scenario with a genuine interference now times out before that interference is reached, a direct instance of the timeout explosion silencing a true detection. The continued FP decrease (13 → 12) follows the same mechanism: one more false-positive scenario exhausts its budget and exits the detected set. Together, these shifts indicate that accuracy changes across depths are driven less by the analysis becoming inherently more or less precise, and more by the timeout explosion selectively removing scenarios from the result set as traversal depth grows.

Table 7 makes these effects explicit through the derived metrics.

Table 7: Accuracy metrics for *MergeDataset* across depth configurations. Values in parentheses show relative change from the previous depth.

	Depth 5	Depth 10	Depth 20
Precision	0.25	0.28 (+11.2%)	0.25 (−10.1%)
Recall	0.17	0.17 (0.0%)	0.14 (−19.8%)
F1-Score	0.20	0.21 (+4.4%)	0.18 (−16.4%)
Accuracy	0.61	0.63 (+3.3%)	0.63 (0.0%)

The absolute metric values are modest because OA is only one algorithm for detecting one class of interferences, and the ground-truth labels capture all interferences globally, including those beyond OA's scope. This relatively low recall (around 17%) is consistent with the original evaluation of OA analysis by Barbosa et al. [3], which reports similarly modest recall values, reflecting an inherent limitation of the technique's detection scope rather than an artifact of depth tuning. The meaningful signal for depth-limit tuning lies in the *shifts* in metrics across configurations.

From depth 5 to depth 10, precision improves slightly while recall remains flat. The precision gain is driven by the timeout explosion effect: false-positive scenarios that previously completed now exhaust their budget and exit the detected set, reducing reported positives without losing any true ones. Recall stays unchanged because the additional write sites reached at greater depth do not correspond to ground-truth interferences, consistent with the partial-capture nature of iOA. F1 therefore improves marginally, reflecting the precision shift rather than any genuine gain in interference detection quality.

Going from depth 10 to depth 20, both precision and recall degrade. Precision does not continue improving because the timeout explosion now also claims a true positive: a scenario with a genuine interference times out before that interference is reached, the classic outcome where a sibling subtree exhausts the budget before the relevant write site is visited. Recall falls as a result of this true positive loss. F1 reaches its lowest point across the three configurations, and accuracy remains stable, masking the opposing TP and FP shifts that cancel out in the aggregate count.

RQ2 Answer

Accuracy improves modestly from depth 5 to 10 (higher precision, stable recall, best F1-score), but degrades from 10 to 20 as timeout explosion silences a true detection. The apparent gain at depth 10 is driven largely by timeout-induced removal of false positives rather than genuine analysis improvement.

5.4 RQ3: Computational Performance Across Depth Configurations

To characterize the computational cost of depth tuning, we focus on the *RefDataset*, which, with 846 scenarios (after build filtering) spanning a broader range of project sizes and merge patterns, provides a more representative and diverse workload than the *MergeDataset* for assessing performance behavior at scale.

Table 8: Execution time analysis for *RefDataset* per depth: total time and per-execution statistics (s).

Depth	Total (s)	Avg	Med.	75th	90th	95th	99th	Max
5	7,624	6.51	2.52	4.14	7.12	12.88	116.41	551.04
10	23,784	22.59	2.63	4.59	10.37	45.47	537.74	1200.22
20	55,885	54.24	2.62	4.62	14.04	143.27	1200.04	1200.11

Aggregate cost grows super-linearly: depth 10 requires 3.1× as much time as depth 5, and depth 20 requires 7.3× as much, well above the 4× expected under linear scaling with depth. The steepest unit increase falls in the 5→10 transition, exactly the step with the largest detection gain per RQ1. The median execution time is virtually unchanged across depths (2.52 s, 2.63 s, and 2.62 s), while the mean climbs from 6.5 s to 22.6 s to 54.2 s. This divergence reveals a bimodal execution profile: most runs complete in seconds regardless of depth, while a growing tail of pathological runs, caused by timeout explosion (Section 3), drives the aggregate upward. At depth 20, both the 99th-percentile (1200.04 s) and the maximum (1200.11 s) coincide with the timeout cap, confirming that at least 1% of runs exhaust the budget. Because the *RefDataset* uses the *partial-result strategy*, these runs still produce output and are not recorded as hard timeouts, though their full cost is reflected in the distribution. Tables 9 and 10 show how memory and CPU utilization are distributed across wall time (elapsed real time), exposing the resource regime behind this cost growth.

Table 9: Memory working-set distribution (% of wall time) per depth.

Depth	0–4 GB	4–8 GB	8–12 GB	≥12 GB
5	5.8%	53.1%	40.3%	0.8%
10	1.7%	6.3%	31.0%	61.0%
20	0.1%	9.3%	51.2%	39.4%

Table 10: CPU utilization distribution (% of wall time) per depth.

Depth	0–30%	30–50%	50–70%	70–100%
5	5.0%	37.5%	27.9%	29.6%
10	50.2%	33.3%	13.9%	2.6%
20	74.0%	20.3%	4.6%	1.1%

At depth 5, the analysis remains lightweight and compute-bound: most of its wall time is in the 4–8 GB range and CPU utilization is generally high, indicating that the call graph fits comfortably in memory. At depth 10, memory pressure jumps sharply (61.0% of wall time moves above 12 GB) while CPU activity drops, with half the run below 30% CPU and only 2.6% above 70%; by depth 20, 74.0% of wall time sits below 30% CPU as the pointer-analysis working set pushes toward RAM saturation and the processor idles on memory and garbage-collection overhead. This apparent easing at depth 20 is not a sign that the analysis becomes cheaper; it reflects the fact that more runs now hit the timeout cap before the heaviest memory phases complete, biasing the observed profile toward earlier, truncated executions.

These findings paint a consistent picture of the performance cost of depth tuning. At depth 5, the analysis is lightweight and compute-bound: most scenarios finish in seconds, CPU utilization is high, and memory stays in the 4–8 GB range. The jump to depth 10 crosses a threshold: call-graph expansion drives memory into saturation, CPU efficiency falls, and total time multiplies by 3.1×, yet this step also yields the greatest detection gain per RQ1, making it the best cost-to-benefit trade-off. Beyond depth 10, cost more than doubles again (7.3× over depth 5) as the analysis enters a CPU-idle, memory-saturated regime where timeout explosion actively silences detections rather than enabling them.

RQ3 Answer

Computational cost grows super-linearly with depth (3.1× at depth 10, 7.3× at depth 20, relative to depth 5), and the execution regime shifts from compute-bound to memory-bound as depth increases, with a growing share of runs hitting the timeout cap.

6 Study Implications

The combined findings of RQ1, RQ2, and RQ3 form a coherent picture with concrete implications for safer code integration. Developers who merge code daily often have limited visibility into how their changes interact with concurrent modifications made by colleagues: an undetected behavioral interference can silently introduce bugs that pass textual review and reach production. Properly configured static analysis lowers this risk, giving teams more confidence that integrated code behaves as intended. The findings below translate our empirical results into guidance for practitioners deploying iOA and for researchers building on it.

Depth 10 is the practical optimal value. Across all three research questions, depth 10 consistently emerges as the most balanced configuration. From a detection standpoint (RQ1), moving from depth 5 to depth 10 reduces the share of entirely missed *RefDataset* scenarios from 14.6% to 3.4%, and eliminates scenario loss entirely in the *MergeDataset*, meaning nearly all reachable interferences fall within 10 levels of call-chain traversal. From an accuracy standpoint (RQ2), precision improves from 0.25 to 0.28 while recall remains stable, yielding the highest F1-Score (0.21) across the three depths. From a performance standpoint (RQ3), depth 10 multiplies total execution time by 3.1× relative to depth 5, a meaningful increase, yet one that still keeps most scenarios in a compute-bound regime where the analysis finishes in seconds. Practitioners configuring iOA for new codebases can therefore treat depth 10 as a principled default, revisiting it only if project-specific evidence points to a different call-chain structure. We note, however, that this recommendation is contingent on the hardware and timeout budget used in our experimental setup: the timeout explosion effect that penalizes depth 20 is fundamentally a function of available memory and CPU allocated per scenario, so the depth at which this penalty dominates would likely shift on more resource-constrained or more powerful hardware. Additionally, since we only evaluated depth limits of 5, 10, and 20, our data cannot precisely locate the sweet spot between these values; the true optimum may lie anywhere in the 6-19 range, or shift outside it under different hardware, and pinpointing it would require a finer-grained sweep.

Timeout explosion is a first-class design concern. A recurring mechanism across all three RQs is timeout explosion, the phenomenon where a single deep call subtree exhausts the per-scenario time budget before sibling subtrees, which may contain actual interferences, are ever visited. At depth 20, this effect reduces detected scenarios and lowers both precision and recall relative to depth 10, turning increased depth into a net liability rather than a benefit. Crucially, a timeout may be conclusive (no output at all) or inconclusive (partial output available up to the boundary). In the former case, absence from the result set is indistinguishable from a genuine absence of interferences; in the latter, the recorded output is partial and should be interpreted as a lower bound. Tool designers should treat the timeout as a configurable, first-class parameter rather than a safety valve, and document clearly that silence under timeout pressure does not imply safety. A promising direction to mitigate this specific failure mode is exploring a breadth-first traversal strategy, which would prioritize visiting all sibling branches before committing the time budget to a single deep subtree, at the cost of requiring additional bookkeeping to track pending call sites and their associated definition sets.

Partial results are better than no results. Our adoption of a *partial-result strategy* (capping the analysis at the timeout boundary and recording whatever the traversal reached) proved its value: scenarios that exceeded the time limit still contributed output to RQ1 and RQ3, and their partial interference sets were used in the RQ2 accuracy evaluation. Discarding such scenarios entirely, as prior implementations did, would have systematically under-reported detection rates and distorted performance distributions. We recommend that future implementations of interprocedural static analyses adopt analogous partial-result semantics and expose the completeness degree of each result, allowing downstream users to reason about uncertainty rather than treating all outputs as equally complete.

Variant selection interacts with depth tuning. All findings in this study are specific to the PA variant of OA (the pointer-analysis-based variant using SPARK, as opposed to the CHA-based noPA variant [2]). Depth tuning can only recover interferences that the chosen call-graph algorithm makes visible in the first place; if call edges are trimmed by SPARK's open-world assumption, no depth value will surface the corresponding interferences. Practitioners and researchers should therefore treat call-graph algorithm selection and depth-limit tuning as joint decisions, and evaluate the cost-benefit trade-off of each combination rather than optimising depth in isolation.

7 Threats to Validity

We organize threats following the four-category taxonomy of construct, internal, external, and conclusion validity [26].

Construct Validity. Our precision and recall use the LOI criterion (defined in Section 4.2), which captures locally visible interference but may miss semantically equivalent writes through aliases or cross-method calls. All experiments use the PA variant of OA, which relies on SPARK pointer analysis; Barbosa et al. [2] showed that this variant achieves lower recall than the noPA/CHA variant because SPARK's open-world assumption trims call edges. Thus, reported metrics reflect both OA's partial detection scope and the chosen call-graph algorithm, and our depth-tuning conclusions characterise SPARK-based OA in particular. Ground truth labels were produced via double-blind labelling.

Internal Validity. Depth and timeout interact as confounders: complex call graphs (likely at depth 20) risk timeout before full exploration. Our *partial-result strategy* makes depth 20 detections a lower bound on unconstrained analysis. SPARK's unsoundness with Java reflection may cause missed interferences in reflection-heavy code.

External Validity. Scenarios are GitHub Java projects filtered by build success and textual conflicts, favoring well-maintained code; results may not transfer to frameworks or reflection-heavy projects. The same-method filter over-represents interference-prone scenarios, potentially inflating precision and recall compared to real-world distributions [3]. Only three depth values were evaluated, leaving precise inflection points unlocated. The MergeDataset's 99 units from limited projects constrain statistical power; RefDataset lacks per-scenario ground truth.

Conclusion Validity. Small true-positive counts on MergeDataset make precision, recall, and F1 sensitive to individual outcomes. No formal significance tests were applied; differences are descriptive. F1 comparisons should be interpreted as directional.

8 Related Work

The challenges of collaborative software development and code integration have been extensively studied. Ghiotto et al. [11] investigated the nature of merge conflicts across thousands of open-source projects, categorizing the types of textual conflicts developers face. Similarly, Elias et al. [9] proposed approaches to recommend resolution strategies for such conflicts, and Campos Junior et al. [8] investigated how code composition strategies and line-combination patterns affect manual merge conflict resolution. While these studies provide valuable insights into textual conflicts and their resolution, our work focuses specifically on dynamic semantic conflicts, behavioral interferences that occur even when textual merges succeed without warnings.

To detect such semantic conflicts, some approaches rely on testing. Da Silva et al. [21] proposed an automated behavior change detection approach using unit test generation to identify semantic conflicts. Unlike test-based methods, which are inherently limited by the quality and coverage of the project's test suite and can be computationally expensive to run repeatedly, our study evaluates a static analysis that does not rely on the existence of tests, but instead systematically traverses the program's call graph to find overriding assignments.

Other researchers have explored formal and heavyweight static analysis. Horwitz et al. [12] introduced the use of Program Dependency Graphs (PDGs) and System Dependency Graphs (SDGs) for integrating non-interfering program versions. Sousa et al. [23] proposed SafeMerge, a formal verification approach to guarantee that three-way merges do not introduce unwanted behaviors. Although theoretically sound and providing formal guarantees, these techniques often require building complete dependency graphs or formal models, struggling to scale to large, real-world codebases. In contrast, our work builds upon the lightweight Overriding Assignment (OA) analysis [3, 6], which performs a targeted scan directly on the merged code. Instead of focusing on formal verification, we empirically investigate how tuning the traversal depth limit of this lightweight analysis impacts its detectability and performance.

Another related technique is program slicing [25], which identifies statements that affect or are affected by a given statement. Slices can expose semantic dependencies, but they do not capture overriding assignments such as $x = 0;$ followed by $x = 1;$ because neither write depends on the other.

The most closely related work to ours is by Barbosa et al. [2], who investigated the effect of different call-graph construction algorithms on the accuracy and execution time of the OA analysis. While they focused on the precision of the underlying call graph, they kept the method traversal depth fixed at a default limit of 5. Our study directly complements their work by fixing the call-graph algorithm to the most prominent configuration they evaluated (SPARK) and systematically exploring the orthogonal dimension of traversal depth. Together, these studies provide a comprehensive empirical understanding of how to configure lightweight static analysis for practical semantic conflict detection.

The broader literature on static analysis configuration [15, 16, 22, 24] documents a precision-scalability trade-off for parameters like context sensitivity, where deeper settings generally improve precision until the cost becomes prohibitive. Our study contrasts this classical view by showing a non-monotonic detectability-cost relationship in semantic merge-conflict detection, driven by timeout explosion under per-scenario budgets.

9 Conclusion

This paper presented an empirical investigation into how varying the call-graph traversal depth limit (set to 5, 10, and 20) affects both

the detectability and the computational cost of the Overriding Assignment (OA) static analysis for semantic merge conflict detection. Evaluating on two datasets of Java open-source merge scenarios, we found that the relationship between depth and detection is non-monotonic: moving from depth 5 to depth 10 reduces the share of entirely missed scenarios and yields the highest accuracy score on the *MergeDataset* (tied with depth 20), while pushing depth to 20 reverses these gains as timeout explosion silences detections before they are reached. On the performance side, total execution time grows super-linearly (3.1× at depth 10, 7.3× at depth 20 over depth 5), and the resource regime shifts from compute-bound to memory-bound as deeper call-graph traversal drives the pointer-analysis working set into RAM saturation. Together, these findings establish depth 10 as a principled default that best amortises detection gain against computational cost for the configurations and datasets studied.

Artifact Availability

To support the replication of our study and encourage further research, we provide all datasets, scripts, and experimental results in an online appendix [1].

Acknowledgements

For partially supporting this work, we would like to thank INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, and CNPq grants 408817/2024-0 and 401032/2025-6. We also thank the members of the Software Productivity Group for their valuable feedback, insightful suggestions, and overall support throughout this research.

References

[1] 2026. Online Appendix. available at: https://doi.org/10.17605/OSF.IO/BAXRP.
[2] Matheus Barbosa, Paulo Borba, Rodrigo Bonifácio, Victor Lira, and Galileu Santos. 2025. The Effect of Pointer Analysis on Semantic Conflict Detection. *arXiv preprint arXiv:2507.20081* (2025). https://arxiv.org/abs/2507.20081
[3] Matheus Barbosa, Paulo Borba, Rodrigo Bonifacio, and Galileu Santos. 2022. Semantic Conflict Detection with Overriding Assignment Analysis. In *Proceedings of the Brazilian Symposium on Software Engineering*.
[4] Christian Bird and Thomas Zimmermann. 2012. Assessing the Value of Branches with What-if Analysis. In *Proceedings of the International Symposium on Foundations of Software Engineering*.
[5] Roberto Souto Maior de Barros Filho. 2017. *Using information flow to estimate interference between same-method contributions.* Master's thesis. Universidade Federal de Pernambuco.
[6] Galileu Santos de Jesus, Paulo Borba, Rodrigo Bonifácio, and Matheus Barbosa. 2024. Lightweight Semantic Conflict Detection with Static Analysis. In *Companion Proceedings of the International Conference on Software Engineering*.
[7] Galileu Santos de Jesus, Paulo Borba, Rodrigo Bonifácio, and Matheus Barbosa de Oliveira. 2025. Comparing static analyses for improved semantic conflict detection. *Automated Software Engineering* (2025).
[8] Heleno de S. Campos Junior, Gleiph Ghiotto L. de Menezes, Márcio de Oliveira Barros, André van der Hoek, and Leonardo Gresta Paulino Murta. 2024. How code composition strategies affect merge conflict resolution? *Journal of Software Engineering Research and Development* (2024).
[9] Paulo Elias, Heleno de S. Campos, Eduardo Ogasawara, and Leonardo Gresta Paulino Murta. 2023. Towards accurate recommendations of merge conflicts resolution strategies. *Information and Software Technology* (2023).
[10] H. Christian Estler, Martin Nordio, Carlo A. Furia, and Bertrand Meyer. 2014. Awareness and Merge Conflicts in Distributed Software Development. In *Proceedings of the International Conference on Global Software Engineering*.
[11] Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and Andre Van Der Hoek. 2018. On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github. *IEEE Transactions on Software Engineering* (2018).
[12] Susan Horwitz, Jan Prins, and Thomas Reps. 1989. Integrating Noninterfering Versions of Programs. *ACM Transactions on Programming Languages and Systems* (1989).
[13] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. 2007. A Formal Investigation of Diff3. In *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science*.
[14] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using SPARK. In *Proceedings of the International Conference on Compiler Construction*.
[15] Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Proceedings of the International Conference on Compiler Construction*.
[16] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Transactions on Programming Languages and Systems* (2020).
[17] Victor Lira, Paulo Borba, Rodrigo Bonifácio, Galileu Santos, and Matheus Barbosa. 2025. RefFilter: Improving Semantic Conflict Detection via Refactoring-Aware Static Analysis. *arXiv preprint arXiv:2510.01960* (2025). https://arxiv.org/abs/2510.01960
[18] Wardah Mahmood, Moses Chagama, Thorsten Berger, and Regina Hebig. 2020. Causes of merge conflicts: a case study of ElasticSearch. *arXiv preprint arXiv:2010.01960* (2020). https://arxiv.org/abs/2010.01960
[19] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. 2017. Software Practitioner Perspectives on Merge Conflicts and Resolutions. In *Proceedings of the International Conference on Software Maintenance and Evolution*.
[20] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. 2001. Parallel Changes in Large-Scale Software Development: An Observational Case Study. *ACM Transactions on Software Engineering and Methodology* (2001).
[21] Leuson Da Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and João Moisakis. 2020. Detecting Semantic Conflicts via Automated Behavior Change Detection. In *Proceedings of the International Conference on Software Maintenance and Evolution*.
[22] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
[23] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. 2018. Verified Three-Way Program Merge. In *Proceedings of the ACM on Programming Languages*.
[24] Frank Tip and Jens Palsberg. 2000. Scalable Propagation-Based Call Graph Construction Algorithms. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.
[25] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* 4 (1984), 352–357.
[26] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*.