MergirafSemi: A Language-Agnostic Semistructured Merge Tool

Pedro Lopes
Centro de Informática, UFPE
Recife, Brazil
phls2@cin.ufpe.br

Paulo Borba
Centro de Informática, UFPE
Recife, Brazil
phmb@cin.ufpe.br

Paola Accioly
Centro de Informática, UFPE
Recife, Brazil
prga@cin.ufpe.br

Guilherme Cavalcanti
Instituto Federal de Pernambuco
Belo Jardim, Brazil
guilherme.cavalcanti@belojardim.ifpe.edu.br

Abstract

Developers frequently face merge conflicts when integrating concurrent changes. Most merge tools rely on unstructured, line-based comparisons, often producing spurious conflicts and missing actual conflicts. To address these limitations, structure-aware merge tools have been proposed, which leverage syntactic representations to improve merge accuracy. However, fully structured tools may incur higher computational cost, and language-specific tools require significant development and maintenance effort. To balance these trade-offs, we propose *MergirafSemi*, a language-agnostic semistructured merge tool that captures structural information without requiring full structural modeling or language-specific implementations, and applies line-based merging within specific program regions, such as method bodies in Java. Our tool leverages lightweight Concrete Syntax Trees to guide merging decisions while preserving flexibility and efficiency across languages. We evaluate our tool through an empirical study on real-world merge scenarios across multiple programming languages, comparing it with unstructured, semistructured, and structured tools. Our results show that increasing structural granularity improves automatic conflict resolution but can also lead to more aggressive merge decisions, increasing the number of missed actual conflicts. In contrast, *MergirafSemi* achieves a more balanced trade-off, reducing spurious conflicts while maintaining competitive accuracy and better runtime performance in most common scenarios. Compared to an unstructured tool, it substantially reduces spurious conflicts, and when compared to a semistructured language-specific tool, it achieves comparable effectiveness while exhibiting more robust execution behavior and significantly lower runtime overhead.

Keywords

Merge Tools, Semistructured Merge, Code Integration, Collaborative Software Development.

1 Introduction

Developers frequently make changes to a codebase in parallel before merging their work. Version Control Systems (VCS) support this process by automatically merging different revisions of files. Most VCS mechanisms rely on unstructured, line-based comparison algorithms, such as *diff3* [20], which operate solely at the textual level and do not account for the syntactic or semantic structure of the code [18].

Although widely adopted and language-independent, line-based merging treats code as plain text, often producing spurious conflicts (false positives) that require unnecessary manual resolution of compatible changes. It may also fail to detect actual conflicts (false negatives), allowing syntactic or semantic inconsistencies to propagate into the merged artifact.

To address these limitations, structure-aware tools incorporate syntactic information into the merge process [1–3, 17, 19, 26]. They represent code trees and perform node matching and merging, improving accuracy over purely textual tools [23]. However, many of these tools rely on language-specific parsing, which increases development and maintenance effort, especially in multi-language projects.

More recently, language-agnostic structured tools, such as *Mergiraf* [1] and LastMerge [11], have been proposed to overcome this limitation by leveraging generic syntactic representations. While these tools improve generality, structured merging may incur higher computational cost and lead to overly aggressive merge decisions, resulting in more false negatives.

Rather than relying on full structural representations, semistructured tools use partial trees to guide merges and fall back to line-based merging when appropriate. This design reduces tree complexity while maintaining the benefits of structure, resulting in fewer false negatives and more false positives than structured tools.

In this work, we build on this and introduce *MergirafSemi*, a language-agnostic semistructured merge tool that captures structural information without requiring fully structured representations. It leverages lightweight Concrete Syntax Trees (CSTs) [4] and targeted line-based merging, enabling structure-aware decisions while maintaining accuracy, flexibility, and simplifying support for new languages.

We evaluate our tool using a multilingual dataset of real-world, compilable merge scenarios across five languages (Java, JavaScript, Go, Python, and Rust). The dataset construction and build validation methodology draws on prior work [22], which proposed a similar pipeline exclusively for Java. Using this dataset, we compare our tool (*MergirafSemi*) against unstructured (*diff3*), structured (*Mergiraf*), and language-specific semistructured (S3M) [5] tools, as well as *MergirafSemi+*, a variant with a different Java configuration for commutative contexts.

We assess the tools by measuring resolution rates, runtime performance, and merge accuracy. To ensure accurate metrics, we

[1] *Mergiraf*'s Website: https://mergiraf.org/

compile and test tools' outputs, assessing false positives and false negatives through pairwise comparisons, specifically in scenarios where tools disagree on the existence of conflicts.

Our evaluation is structured around three research questions: (i) the trade-offs between structural granularity, accuracy, and runtime, (ii) how our tool compares to semistructured language-specific and unstructured tools, and (iii) the impact of changing commutative contexts on merge accuracy.

Our results yield three main insights. First, greater structural granularity improves conflict resolution but raises false negatives; *MergirafSemi* offers a better balance among false positives, false negatives, and runtime. Second, it greatly reduces false positives compared to *diff3* and matches S3M's accuracy while running reliably (no timeouts or failures) with lower runtime overhead. Finally, relaxing commutative constraints in both tool versions has only a minor effect on conflict resolution, though it introduces a few false negatives in *MergirafSemi+*.

Overall, our findings suggest that semistructured merging can be effectively realized in a language-agnostic setting. Compared to unstructured tools, it substantially reduces false positives while reducing the structured tools' high number of false negatives. This positions language-agnostic semistructured merging as a viable alternative to other language-agnostic tools, achieving competitive accuracy with language-specific semistructured tools.

2 Unstructured, Semistructured and Structured Merge

To describe the limitations of unstructured merge, we present two representative scenarios in which line-based tools yield incorrect merge outcomes.

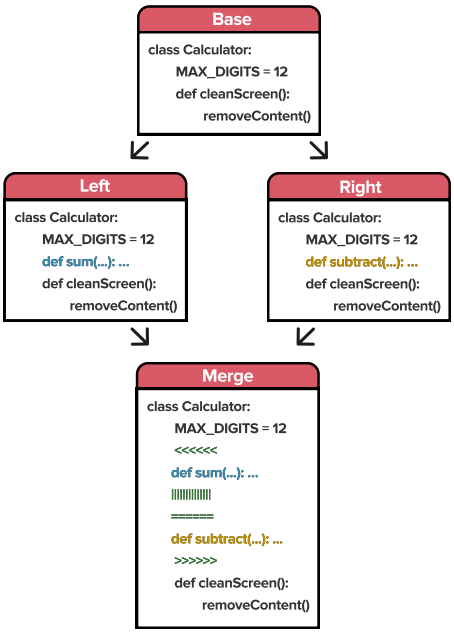

Figure 1: Unstructured Merge False Positive.

Figure 1 illustrates a scenario resulting in a spurious conflict. Two developers independently extend the same class by introducing new, semantically independent methods in the same textual region of the file. Because unstructured tools compare overlapping line ranges against a common base version, they interpret these edits as conflicting changes.

The merge output contains conflict markers that require manual resolution, even though the changes are compatible [14]. This example constitutes a false positive: unstructured tools incorrectly signal a conflict even though there is no semantic interference between the modifications.

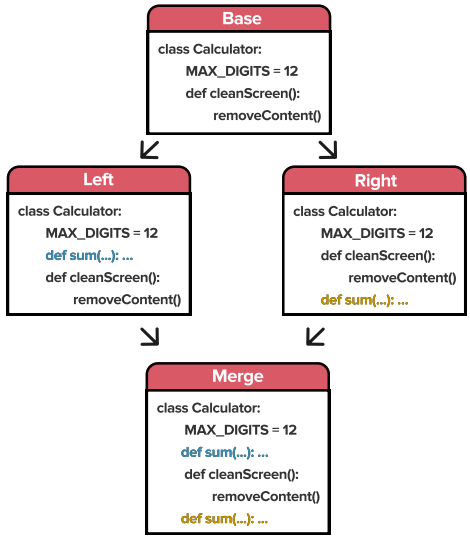

Figure 2: Unstructured Merge False Negative.

In contrast, Figure 2 presents a scenario that leads to an actual missed conflict. Two developers independently introduce a method with the same signature in different regions of the file. Again, as line-based merge detects conflicts based on overlapping lines [18], it treats these edits as independent and combines them automatically.

Although no conflict is reported, both changes affect the same logical entity, potentially resulting in invalid or unintended behavior. In languages such as Python, this can lead to unintended overriding depending on declaration order. This outcome represents a false negative.

These examples highlight a key limitation of unstructured tools: decisions are based solely on textual alignment rather than the syntactic organization of the code.

Structured merge tools address this limitation by reasoning about program elements such as classes and methods [2]. They represent code as trees, commonly using Abstract Syntax Trees (ASTs), aligning corresponding elements across revisions and reporting conflicts only when incompatible changes affect the same node [11].

Semistructured tools use a partial structure: high-level elements are structured, while method bodies are treated as raw text. This hybrid representation improves accuracy without requiring full syntactic processing [6].

Revisiting the examples from a structure-aware perspective highlights the effects of these tools. In Figure 1, where two developers add different methods to the same class, both structured and semistructured tools recognize that the changes involve distinct program elements. Thus, these modifications can be integrated automatically without conflict. In contrast, Figure 2 shows two developers introducing methods with the same signature in different file sections. Here, these tools detect that both changes refer to the same program element, correctly reporting a conflict, preventing inconsistent or even invalid code.

However, structured and semistructured tools typically depend on language-specific parsers, syntactic representations, and merging engines, increasing implementation and maintenance effort, especially in multi-language environments. While semistructured tools reduce structural detail, they still depend on those too.

Recent work shows that structured merge can be made language-agnostic [11], but this has not yet been fully explored for semistructured merging. This motivates the tool proposed in this work.

3 *MergirafSemi*

As discussed in Sections 1 and 2, structure-aware tools improve merge accuracy by reasoning about program elements, but introduce important trade-offs. Structured tools provide fine-grained analysis, reducing false positives, but may incur higher computational cost and lead to more false negatives. Semistructured tools reduce structural granularity, reducing false negatives but reporting more false positives; however, existing semistructured solutions still rely on language-specific representations, limiting their applicability.

Motivated by the limitations of unstructured merge tools and the trade-offs of structure-aware tools, this work builds on *Mergiraf*, a prior language-agnostic structured merge tool, and introduces *MergirafSemi*, a language-agnostic semistructured merge tool that combines semistructural reasoning with selective use of textual merging, preserving its language-agnostic nature through generic syntax tree representations while reducing structural granularity.

To support semistructured merging, we rebuild *Mergiraf*'s representation and use of structural information during merging. In particular, we introduce (i) selective structural parsing, analyzing specific regions structurally, (ii) line-based merging on specific nodes, and (iii) adaptations to existing language profiles, tree-building algorithms, matching procedures, and new unification mechanisms to operate correctly under partial structural information.

Importantly, we design these modifications to be modular and configurable. The semistructured behavior can be enabled via configuration flag (`--semistructured=diff3`) when desired, allowing the tool to revert to *Mergiraf*'s original fully structured behavior. This design preserves backward compatibility and enables controlled experimentation with different merging tools.

Our tool aims to balance merge accuracy and practical applicability by leveraging lightweight structural representations through *TreeSitter* parsers [4], configurable language profiles, and a multi-phase execution pipeline, which we describe in the following sections.

3.1 Generic Trees

MergirafSemi represents source code using generic tree structures derived from parsing input files with *Tree-sitter* [4], a fast and extensible parser framework currently in production at GitHub. These structures capture syntactic elements such as classes, methods, and declarations, serving as one of the bases for structure-aware merging.

In *Mergiraf*, all syntactic elements identified by the parser are represented explicitly as nodes in the tree. This enables the merge process to operate at a fine-grained level, including internal statements within method bodies.

In contrast, *MergirafSemi* employs a partial representation of the program tree in the form of Concrete Syntax Trees (CSTs). Unlike ASTs, CSTs preserve syntactic elements such as delimiters and formatting. While high-level constructs such as classes and method signatures are preserved as nodes, selected regions of the tree are treated as unstructured text. These regions are determined by a language-specific configuration that specifies the node types whose internal content should not be further decomposed.

Consequently, subtrees corresponding to these node types are truncated and represented as leaf nodes containing raw text, which we call unstructured merge later on in the merge process. This design enables the tool to retain structural awareness where it is most beneficial, while avoiding the complexity of fully structured analysis in regions where it is less critical.

3.2 Multi-Phase Merge Pipeline

MergirafSemi operates through a three-phase pipeline that incrementally increases merge granularity.

In the first phase, the tool performs an autotuned unstructured merge using a *diff3*-style algorithm. If the result is conflict-free, it is then parsed to detect duplicated signatures. If neither conflicts nor duplicated signatures are found, the merge is accepted, avoiding further analysis.

If conflicts or duplicated signatures are detected, the tool advances to a second phase, where semistructured merging is applied locally, parsing only conflicting regions into CSTs. Then, node correspondences are computed using *GumTree* [13] and an unstructured merge is applied on the raw text leaf nodes mentioned in Subsection 3.1. Because GumTree primarily aligns nodes against the base revision, elements newly added in both revisions remain unmatched. To prevent duplicated code, it executes a unification step.

This unification step collects all unmatched nodes from both parents and evaluates their logical identities using signatures. If an unmatched node in the left revision shares the exact same signature as one in the right revision, they are unified into a single logical entity.

Lastly, if conflicts persist or tree-related issues, such as parsing errors, are detected from Phase 2, a third phase applies semistructured merging globally by parsing the entire files into CSTs and repeating the matching, merging, and unification steps described in Phase 2.

This design enables efficient handling of simple cases while still supporting semistructural reasoning when needed. *Mergiraf* also shares this pipeline, but on Phases 2 and 3, instead of semistructured

tree-building, matching, and the unification step (which is exclusive to *MergirafSemi*), the whole process is fully structured.

3.3 Language Profiles

A key aspect of *MergirafSemi* is the use of configurable language profiles, which define how structural information is interpreted and used during merging. These profiles enable the tool to remain language-agnostic while still leveraging language-specific insights.

Specifically, language profiles define signatures, which capture the essential attributes of program elements (e.g., function names and parameters) and are used to identify corresponding nodes across revisions. They also specify commutative contexts, where the order of elements does not affect code semantics, allowing the merge process to integrate them without conflicts.

Most importantly, the language profile controls the level of structural granularity through tree-truncation rules. These rules determine which node types should be treated as text leaves. When such nodes are encountered, their internal contents are treated as raw text and merged using unstructured tools.

This lightweight and configurable design enables *MergirafSemi* to support multiple programming languages while significantly reducing the effort required for extension. In practice, adding support for a new language typically requires only a few hours, depending on the availability of an existing *Tree-sitter* parser and the developer's familiarity with the language. When structured support is already available within *Mergiraf*, the configuration effort is almost minimal, since you only need to configure the nodes where the CST will be truncated. When no prior support exists, adding a new language requires integrating a *Tree-sitter* parser and validating the merge behavior of the configurations cited in the previous paragraphs, but this effort is still relatively low compared to that required for language-specific tools.

3.4 Structure-Aware Merging

During the semistructured phases of the pipeline, our tool performs merging based on the alignment of syntactic nodes across the Base, Left, and Right revisions (as illustrated in Section 2), as determined by *GumTree* [13].

When corresponding nodes are modified independently in compatible ways, such as changes to different attributes of the same construct or identical modifications, the tool automatically integrates the changes, as described in Section 2 and illustrated in Figure 1. Similarly, modifications affecting different child nodes of a common parent are merged without conflict.

Conflicts are reported when concurrent changes affect the same aligned node in incompatible ways and no deterministic resolution can be inferred, as described in Section 2 and illustrated in Figure 2. In such cases, the merge process preserves both versions and explicitly signals the conflict.

This applies only to the structured portion of the representation. When truncated nodes are encountered, their contents are merged using unstructured tools, enabling the tool to distinguish between independent and interfering changes based on program structure, rather than relying solely on textual overlap.

3.5 Role of Textual Merging

Textual merging complements structural analysis at various stages of the *MergirafSemi* pipeline.

In the initial phase, the tool relies on line-based merging to efficiently handle simple scenarios. This fast-path mechanism, also adopted by language-specific structured merge tools such as *jDime* [1], avoids the overhead of structural analysis when changes can be safely integrated using unstructured merging.

In semistructured phases 2 and 3, textual merging is applied within regions intentionally treated as unstructured, as defined by the language profile. For example, method bodies may be represented as plain text, enabling the tool to reuse line-based merging in contexts where fine-grained structural reasoning is unnecessary.

Finally, textual merging serves as a fallback during structure-aware phases. When the structural merge process cannot reliably resolve a specific node, the tool locally reverts to line-based merging for that region while preserving the surrounding structural context.

4 Evaluation

To evaluate *MergirafSemi*, we performed an empirical study involving merge scenarios drawn from real-world projects across multiple programming languages. We aim to measure and compare its merge accuracy and runtime performance against unstructured (*diff3*), fully structured (*Mergiraf*), language-specific semistructured (S3M [5]), and *MergirafSemi+*, a variant of our tool with a different Java configuration within commutative contexts.

Our analysis centers on cases where different merge tools yield different results, allowing us to isolate instances in which tool selection directly influences the merge outcome.

4.1 Research Questions

We addressed the following Research Questions (RQs) to evaluate the effectiveness of *MergirafSemi*. The third RQ additionally considers *MergirafSemi+*.

4.1.1 ***RQ1**: What is the impact of structural granularity on merge accuracy?*

To investigate this question, we compare *MergirafSemi* with the fully structured tool, *Mergiraf*.

Fully structured merging relies on detailed syntax trees to align and integrate program elements across revisions. Semistructured merging reduces this granularity by treating selected regions of the code as text, allowing the merge process to rely on unstructured tools in contexts where fine-grained structure may not be necessary.

This comparison evaluates how structural granularity affects merge accuracy and whether reducing it helps avoid false positives or increases the likelihood of false negatives.

4.1.2 ***RQ2**: How does a language-agnostic semistructured merge tool compare to language-specific and line-based tools?*

To address this question, we compare *MergirafSemi* against two baselines: S3M, a language-specific semistructured merge tool for Java, and *diff3*, a widely used line-based merge tool. The first comparison is restricted to Java scenarios, as S3M supports only Java, limiting cross-language evaluation.

Semistructured language-specific tools, such as S3M, leverage detailed knowledge of a target language to guide merge decisions,

thereby improving precision but requiring substantial implementation and maintenance effort, limiting applicability across languages. In contrast, *MergirafSemi* adopts a language-agnostic design, relying on generic parsing and configurable language profiles rather than language-specific logic.

This comparison evaluates whether a language-agnostic tool can achieve accuracy comparable to that of a language-specific one, without significant impact on merge runtime performance and accuracy, while also assessing its advantages over traditional line-based merging. In particular, we investigate whether removing language-specific aspects compromises merge accuracy relative to a specialized tool.

4.1.3 ***RQ3**: What is the impact of the commutative configuration on the accuracy of MergirafSemi and MergirafSemi+?*

We compare *MergirafSemi* and *MergirafSemi+* based on their handling of commutative constraints for class body declarations. This analysis focuses on Java, where certain declarations are treated as non-commutative, unlike other language profiles that do not enforce this constraint.

In the Java default language profile, elements such as methods and variable declarations are treated as non-commutative. As a result, if Left adds a method and Right adds an attribute (or vice versa) to the same class body, even when the changes are semantically independent and do not compromise program correctness, the merge results in a conflict.

MergirafSemi+ relaxes this constraint by treating such elements as commutative, allowing the merge process to integrate them without reporting conflicts. This modification is motivated by the observation that, in Java, the relative ordering of these declarations does not affect program semantics, although it may impact code organization.

This comparison evaluates how relaxing commutative constraints affects merge accuracy and if it introduces trade-offs, such as less-structured output due to interleaved declarations.

4.2 Metrics

We evaluate merge tools based on merge accuracy and runtime performance. To assess merge accuracy, we analyze how different tools behave in scenarios where their outputs differ on whether conflicts exist; in these cases, the tool choice affects the merge outcome.

Our analysis of merge accuracy is based on pairwise comparisons between tools. For each scenario, we compare the outputs produced by different tools and examine how they differ. These comparisons allow us to identify whether a tool introduced an unnecessary conflict or failed to detect an actual conflict.

We reference the merge results in the repository as the expected outcome, as they were produced and accepted by developers. However, since a merge can be correct even if it differs from the repository, we also conduct builds and tests for further verification.

We employ a standard methodology from recent empirical studies on merge tools [11, 22], combining repository comparisons with behavioral validation. By integrating various signals—such as syntactic equivalence, build success, and test outcomes—we obtain a more reliable approximation of the expected merge result, thereby minimizing misclassification risk.

4.2.1 *Automatic Conflict Resolution Rate.*

The automatic conflict resolution rate shows the percentage of merge scenarios a tool successfully resolves. We calculate this by dividing the number of conflict-free merges by the total valid scenarios. It serves as an initial indicator of the tool's ability to autonomously integrate changes, thereby reducing manual effort. However, a high rate mainly reflects conservativeness rather than accuracy, as it may merge code with errors. We interpret this alongside false positives and negatives (explained in the next section) to avoid hidden errors.

4.2.2 *Added False Positives and Added False Negatives.*

We identify added false positives (aFPs) and added false negatives (aFNs) through pairwise comparison of tool outputs, following prior work [11]. Given a pair of tools A and B, we say that A incurs an aFP candidate if it reports a conflict in a scenario where B produces a conflict-free result, and B incurs an aFN candidate if it produces a conflict-free result in a scenario where A reports a conflict.

To validate these cases, we rely on the repository merge as a reference and complement it with build and test execution. For aFPs, we require that B's output is either syntactically identical to the repository merge or, if different, it successfully compiles and passes all tests. For aFNs, we require that B's output differs from the repository merge and fails to compile or causes test failures.

These definitions capture two types of undesirable behavior: spurious conflicts (aFPs), which introduce avoidable manual effort, and silent integration of actual conflicts (aFNs), where incorrect merged artifacts are produced without signaling a conflict.

4.2.3 *Runtime Performance.*

To assess runtime performance, we measure the execution time of all merge tools for each merge scenario.

Each scenario is executed six times per tool. The first execution is discarded as a warm-up run to mitigate initialization overhead (e.g., JVM startup in Java projects). The reported execution time corresponds to the mean of the remaining five runs.

To provide a comprehensive view of runtime behavior, we also analyze the distribution of execution times using raincloud, which capture variability, median values, and the presence of outliers across scenarios.

4.3 Experimental Setup

This section describes the tools, execution environment, and experimental procedures used to evaluate the proposed tool.

4.3.1 *Merge Tools.*

We evaluate five merge tools: *Mergiraf*, *MergirafSemi*, *MergirafSemi+*, S3M, and *diff3*. The Mergiraf-based tools rely on version 0.12.1 of the original implementation, with *MergirafSemi* and *MergirafSemi+* built on top of it. For *diff3*, we use the native implementation provided by the `git merge-file` command.

S3M does not provide strict versioning. Therefore, we use a specific version identified by a commit hash[2]. To ensure reproducibility, we provide a replication package containing the exact binaries of all evaluated tools, along with the execution scripts and the complete dataset used in our experiments [12].

[2]S3M's version via commit: https://github.com/guilhermejccavalcanti/s3m/tree/bcf0a43cbc14d5aa21757686737c2434084e2165

Our baseline tool choices for each RQ are guided by the specific variable being controlled. For RQ1, we select *Mergiraf* as the structured baseline because it shares an algorithmic foundation with *MergirafSemi*. This allows us to attribute differences in accuracy and runtime to structural granularity rather than implementation choices. Using other structured tools like LastMerge [11], *jDime* [1], or *Spork* [19] would introduce confounding factors related to heuristics and tree-building.

For RQ2, we choose S3M [5] as the language-specific semistructured baseline, as it shares *MergirafSemi*'s principle of combining structural and textual strategies, differing primarily in language-agnosticism. Tools like SESAME [7], which use syntactic separators, would not address RQ2 appropriately.

4.3.2 *Execution Environment.*

All experiments are executed within Docker containers, using the `eclipse-temurin:17-jre-focal` image to standardize the runtime environment and Java dependencies (needed to execute S3M). The host machine runs Ubuntu 20.04.2 LTS (Kernel 5.4.0-74-generic x86_64) and is equipped with an Intel Xeon E31220 @3.10GHz processor and 32GB of RAM.

To prevent anomalous executions from blocking the evaluation pipeline, we set a 30-minute timeout for both the merge process and the subsequent build and test step.

4.3.3 *Execution Pipeline.*

The evaluation uses a framework that clones repositories, filters merge scenarios, and isolates revisions. It is extended with custom components to run merge tools, normalize outputs, and perform build and test validation [16]. We normalize merge outputs to enable comparison by removing formatting differences, such as spaces, tabs, and line breaks introduced by pretty-printing in tree-based tools.

For each selected scenario, we extract the corresponding revisions into a clean environment and execute all tools via command-line interfaces.

Normalized outputs are compared to identify disagreements, and builds and tests are triggered following a validation procedure discussed in the previous sections. This is done using GitHub Actions with language-specific workflows.

To prevent biases introduced by project configurations such as linters and dependencies, we use generic workflows tailored to each language's build system. This ensures failures are due to issues in merged outputs, such as compilation or testing errors, rather than unrelated constraints.

4.3.4 *Dataset.*

The dataset construction in this study is inspired by prior work on large-scale evaluation of merge tools [22]. While that study focused on Java repositories from Reaper [21] and GitHub Greatest Hits [15], our goal requires a multilingual setup including Java, JavaScript, Python, Go, and Rust.

Due to limited language support in Reaper, we rely solely on repositories from the GitHub Greatest Hits dataset, which are popular and actively maintained. We apply a multi-stage filtering pipeline to ensure the selected merge scenarios are suitable for evaluation.

Initially, we keep repositories containing build or dependency management files at the root, such as `pom.xml`, `build.gradle`, `package.json`, `Cargo.toml`, or `go.mod`, ensuring they are automatically compilable and testable.

We find the earliest successful build commit to set a stable baseline for merge evaluation. From there, we collect non-fast-forward merge commits where both parents modify the same files, ensuring meaningful merge decisions.

This filtering process yields a dataset of real-world merge scenarios that are both structurally relevant and empirically testable. In total, our dataset comprises **21,615** merge scenarios extracted from **513** repositories.

The distribution across languages is as follows: **7,181** scenarios from Go, **1,479** from Java, **2,842** from JavaScript, **5,860** from Python, and **4,253** from Rust.

By ensuring that all selected scenarios originate from buildable project states, we enable reliable validation of the expected merge result through compilation and test execution.

5 Results and Discussion

In this section, we discuss the findings from our evaluation of *MergirafSemi*. We focus on merge accuracy and runtime performance, highlighting how design choices affect outcomes. The results are organized by research questions, covering structural granularity, comparisons with existing merge tools, and commutative configurations.

5.1 RQ1: What is the impact of structural granularity on merge accuracy and runtime?

To answer this question, we compare *MergirafSemi* (language-agnostic semistructured) with *Mergiraf* (language-agnostic structured) and analyze their behavior across multiple programming languages.

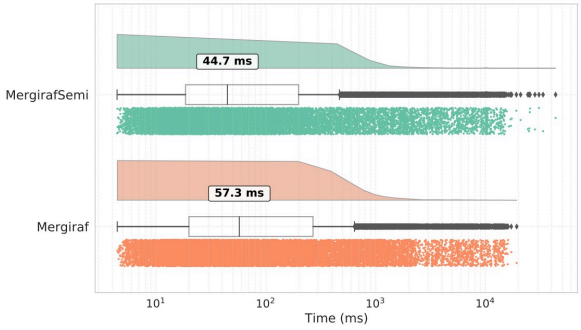

Figure 3: Runtime Efficiency Across Programming Languages

Figure 3 presents the aggregated runtime distribution on a logarithmic scale. The global median execution time shows that *MergirafSemi* (44.7 ms) is 22% faster than *Mergiraf* (57.3 ms), indicating better performance in most common scenarios. Both tools also have many extreme outliers, with some cases exceeding 10^4 ms.

A closer look at pipeline logs shows that during structural analysis phases, *MergirafSemi* significantly reduces runtime. In Phases

Table 1: *MergirafSemi* vs *Mergiraf* – Merge Accuracy

		MergirafSemi			*Mergiraf*		
Language	**Total Scenarios**	**Res. Rate**	**aFPs**	**aFNs**	**Res. Rate**	**aFPs**	**aFNs**
Go	7181	76,66%	153	65	79,81%	5	143
Java	1479	81,27%	44	15	85,67%	0	36
JavaScript	2842	82,51%	61	5	86,14%	4	53
Python	5860	79,88%	244	19	84,78%	10	90
Rust	4253	83,92%	144	26	88,38%	9	81

2 and 3, the tree-building steps, semistructured merging is at least 60% faster than structured merging. For example, in Phase 2 (local conflict resolution), median execution time in Rust drops from 781.9 ms to 89.5 ms, and in Java from 153.4 ms to 25.7 ms. Similar improvements occur in Phase 3, with *MergirafSemi* reducing median runtime by about 61% in Python and 63% in Go.

This performance advantage comes at the cost of increased variability. While median runtimes are lower, average runtimes are affected by a few expensive scenarios, especially in Phase 3 of the merge process. This variability mainly stems from the unify operation (described in Section 3.2), which is costly in large files with many declarations due to the increased number of candidate comparisons. Sometimes, unify accounts for over 90% of total time, impacting mean runtime and standard deviation.

As shown in Table 1, across all evaluated languages, *Mergiraf* resolves 84.1% of merge scenarios without conflicts, outperforming *MergirafSemi* by 4.1% (80.0%). Java's success rate rises from 81.27% to 85.67%, Python's from 79.88% to 84.78%. Similar gains are seen in Go, JavaScript, and Rust. These results indicate that structured merging is better at automatically resolving conflicts. *Mergiraf* uses complete structural information to integrate concurrent changes that semistructured merging cannot, often completing in Phase 2 and avoiding further processing.

For merge accuracy, we examine aFPs and aFNs to clarify trade-offs. Fully structured merging sharply reduces false positives in all languages. For example, in Python, aFPs drop from 224 in *MergirafSemi* to 10 in *Mergiraf*, with similar trends in other languages. This shows fine-grained structure helps precisely identify independent changes.

However, fewer aFPs come with more aFNs: *Mergiraf* often merges real conflicts without flagging them. For instance, in Go, aFNs rise from 65 to 143, and in Python, from 19 to 90.

This indicates that the structured tool's higher resolution rate results from aggressive merging, which silently resolves conflicts. Structured merging reduces aFPs and boosts automation but risks more aFNs, while semistructured merging is conservative: it reports more aFPs but avoids many aFNs.

> ***Answer to RQ1:*** *Neither tool is strictly better; each reflects different priorities. Structured merging favors automation and less manual effort, while semistructured merging prioritizes safer merges by preserving potential conflicts. Higher structural granularity improves conflict resolution and reduces aFPs, but doesn't guarantee better correctness or runtime.*

5.2 RQ2: How does *MergirafSemi* compare to a language-specific semistructured tool and a line-based merge tool?

To answer this question, we compare *MergirafSemi* (language-agnostic semistructured) against S3M (language-specific semistructured), focusing only on Java scenarios due to the language-specific nature of S3M, and against *diff3* (unstructured) across all languages.

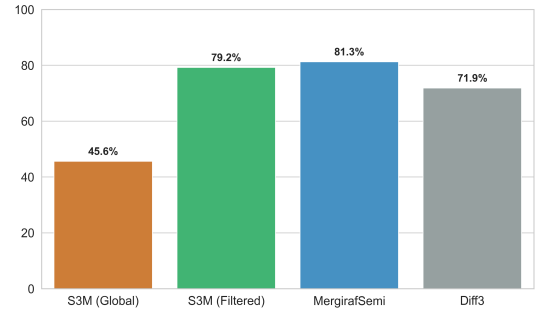

Figure 4: Automatic Conflict Resolution Rate (Java only).

In Figure 4, *MergirafSemi* achieves a resolution rate of 81.3%, outperforming both *diff3* and S3M under default conditions. S3M's initial global resolution rate of 45.6% is strongly influenced by execution interruptions rather than merge effectiveness alone: it failed to generate a merge in 627 of 1,479 scenarios (approximately 42%) due to timeouts and runtime errors.

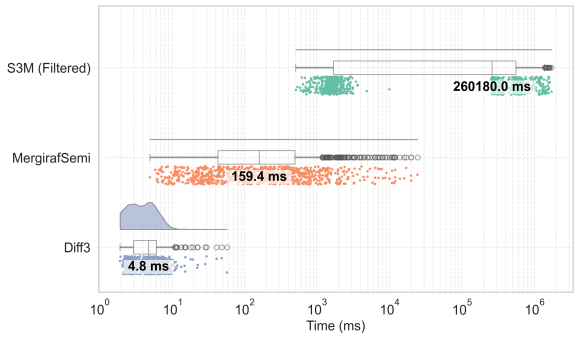

Figure 5: Runtime Comparison Across Merge Tools (Java only).

To better evaluate the tool, we now restrict the analysis to the 852 scenarios in which S3M completes successfully. Under these conditions, its resolution rate increases to 79.2%, approaching the performance of *MergirafSemi*, though still slightly lower.

Despite restricting our analysis to the mentioned scenarios, *MergirafSemi* successfully processed all 1479 scenarios without execution failures under the same experimental setup.

Figure 5 shows the runtime distribution of the evaluated tools. As expected, *diff3* has the lowest median execution time at approximately 4.8 ms, reflecting its textual nature. In contrast, *MergirafSemi* has a median runtime of 159.4 ms due to the overhead from parsing and structure-aware processing. Even in optimal scenarios, our tool incurs runtime overhead compared to standalone *diff3* due to the parsing step required to detect duplicated signatures.

Even when considering only scenarios where S3M completes successfully, its median execution time reaches 260,180.0 ms (approximately 4.3 minutes) per merge. However, this comparison should be interpreted with caution, as the tools are implemented in different programming languages and runtime environments, which can significantly affect execution time.

Rather than focusing on absolute runtime differences, our goal is to assess whether a language-agnostic semistructured tool imposes a prohibitive runtime overhead. In this respect, the results indicate that *MergirafSemi* remains efficient in practice, with execution times that are compatible with real-world usage.

Table 2: *MergirafSemi* vs S3M – Merge performance

Language	aFPs	aFNs
MergirafSemi	6	27
S3M	11	13

A deeper analysis of aFPs and aFNs offers insight into these tools' behavior. Table 2 compares the two semistructured tools across scenarios in which S3M completes successfully, showing a trade-off between conflict reduction and merge conservativeness.

MergirafSemi has fewer false positives (6 vs. 11), indicating an ability to avoid unnecessary conflicts, while S3M has fewer false negatives (13 vs. 27), indicating its language-specific design favors conservative conflict handling. Although differences are significant, they should be interpreted cautiously, as implementation, configuration, and heuristics may influence outcomes, requiring manual analysis to clarify the specific reasons for these discrepancies.

Table 3: *MergirafSemi* vs *diff3* - Merge Accuracy

	MergirafSemi		***diff3***	
Language	**aFPs**	**aFNs**	**aFPs**	**aFNs**
Go	20	292	65	9
Java	3	110	33	1
JavaScript	45	44	89	1
Python	1	172	30	44
Rust	0	244	17	15

When comparing *MergirafSemi* with *diff3* across all languages, our tool significantly reduces aFPs by utilizing structural information, but it also leads to a notable increase in aFNs. While *diff3* has few aFNs in most languages (except Python), *MergirafSemi* generates many more, indicating that automating conflict resolution raises the risk of incorrectly merging changes without notifying developers.

Please note that *MergirafSemi* has 6 aFPs when compared to S3M and 3 when compared to *diff3*. This difference is expected because these metrics are computed through pairwise comparisons, as explained in Subsection 4.2.2. The set of aFPs attributed to *MergirafSemi* may differ when comparing different tools because they successfully resolve different subsets of scenarios.

To directly address whether this trade-off holds when restricted to the subset comparable with S3M, we recompute the aFPs and aFNs for *MergirafSemi* and *diff3* on the same 852 scenarios in which S3M completes successfully. Under this restricted setting, *MergirafSemi* yields 1 aFP and 55 aFNs, while *diff3* yields 13 aFPs and 1 aFN. This confirms that the same pattern observed across the full dataset persists on this subset: *MergirafSemi* continues to substantially reduce aFPs relative to *diff3*, at the cost of a much higher aFN count.

The trade-off is evident when considering structured merging. Inspired by the high *MergirafSemi*'s aFNs number in Table 3, we compared *Mergiraf*'s accuracy against *diff3*. As expected, *Mergiraf* has more aFNs. For instance, structured merging in Python and Rust yields 377 and 432 aFNs, whereas semistructured merging yields 172 and 244. Additionally, *diff3* produces more aFPs compared to *Mergiraf* than *MergirafSemi*, but this is also expected.

This result highlights an important role for semistructured merging: although it increases the risk of silent integration errors compared to conservative line-based tools, it mitigates the substantially higher number of incorrect automatic merges observed in fully structured tools.

> ***Answer to RQ2:*** *Language-specific semistructured merging (S3M) improves correctness by leveraging domain knowledge but is computationally intensive. Language-agnostic semistructured merging (MergirafSemi) reduces unnecessary conflicts while maintaining competitive accuracy and better runtime. Line-based merging (diff3) is conservative, often exposing conflicts rather than resolving them. However, it may lead to false negatives and over-report conflicts.*

5.3 RQ3: What is the impact of the commutative configuration on the accuracy of *MergirafSemi* and *MergirafSemi+*?

We compare *MergirafSemi* and *MergirafSemi+*, which handle Java declaration order differently. *MergirafSemi+* treats method and variable declarations as commutative, allowing conflict-free merges.

Both configurations show similar conflict resolution rates, with *MergirafSemi+* slightly improving over *MergirafSemi* (81.54% vs. 81.27%). This indicates that loosening ordering constraints can reduce some unnecessary conflicts. Overall, this specific type of ordering-related conflict seems infrequent, and relaxing constraints has a limited but consistent effect.

A more detailed analysis of merge correctness, measured by the number of aFPs and aFNs, reveals a subtle trade-off introduced by relaxing ordering constraints.

Table 4: Merge Performance: *MergirafSemi* vs *MergirafSemi+*

Tool	Res. Rate	aFPs	aFNs
MergirafSemi	81.27%	0	0
MergirafSemi+	81.54%	0	4

In Table 4, both *MergirafSemi* and *MergirafSemi+* produce no additional false positives, indicating that treating declarations as commutative does not introduce false positives. However, *MergirafSemi+* introduces a small number of false negatives (four cases), whereas the default configuration does not introduce any.

Overall, the results suggest that language-agnostic semistructured merging provides a practical balance between effectiveness and efficiency. While it does not eliminate the trade-offs inherent to language-specific semistructured tools in terms of accuracy, it achieves competitive results while significantly improving upon line-based tools in terms of conflict reduction, without incurring prohibitive runtime costs.

> ***Answer to RQ3:*** *MergirafSemi+ resolves scenarios that MergirafSemi flags as conflicting, showing a trade-off between reducing conflicts and maintaining conservative detection. These cases are few but illustrate more liberal merge decisions. Both configurations have almost identical runtime, as the modification only affects local merge decisions. Given minor differences, the choice between MergirafSemi and MergirafSemi+ is a matter of user preference.*

6 Threats to Validity

We discuss threats to validity in accordance with standard guidelines for empirical software engineering studies [25].

Construct Validity. Our analysis of added false positives (AFPs) and false negatives (AFNs) follows prior work [11], combining comparison with repository merges and build/test validation. While useful, these signals are imperfect: repository merges only approximate expected results, and tests are only as strong as the project's test suite. Our tool also depends on Tree-sitter parsers, whose limitations may affect structural representations and merge behavior. To reduce infrastructure-related noise on false negatives, we apply a multi-stage validation process, including CI log inspection and validation of parent commits to distinguish merge-induced failures from external issues. Despite these precautions, the process remains heuristic-based, though it reflects the current gold standard in merge tool evaluation [22].

Internal Validity. Threats to internal validity concern factors that may affect the correctness of the experimental procedure. Although the evaluation pipeline is fully automated, it relies on custom extensions implemented in a framework [16]. Bugs or misconfigurations in this infrastructure may influence the results. Additionally, differences in tool configuration and execution environments may introduce unintended bias. We mitigate this risk by standardizing execution through Docker containers and applying consistent normalization procedures across all tools.

External Validity. Our dataset consists of merge scenarios extracted from publicly available repositories across multiple programming languages. Although this increases diversity compared to single-language studies, it still represents only a subset of possible development contexts. Moreover, the selection criteria focus on non-fast-forward merges involving concurrent modifications, which may not capture all types of merge scenarios encountered in practice. As a result, our findings may not generalize to all repositories, programming languages, or development workflows. The dataset, however, is significant and comparable to the strongest datasets in related work.

Conclusion Validity. Our evaluation focuses on scenarios where tools disagree on the presence of conflicts. While this allows us to focus on cases where merge strategies differ, it may not fully capture the tools' overall behavior. Additionally, build and test execution is only performed for these scenarios. As a result, incorrect merges may go undetected when all tools agree. Finally, although execution time is measured using multiple runs to reduce noise, variability in system performance and workload may still influence runtime results.

7 Related Work

Software merging has been extensively studied, with various tools proposed to improve concurrent change integration. We review existing work, starting with semistructured merge tools, which are mostly related to our proposal, then structured and textual tools.

Semistructured Merge.

Semistructured merge tools aim to balance structure-aware merging with the flexibility of textual tools by selectively incorporating syntactic information.

Apel et al. [2] proposed *FSTMerge*, a generic semistructured merge tool based on Feature Structure Trees (FSTs), instantiated for languages such as C#, Java, and Python. Its core idea of combining structural and textual representations inspires our work. However, *FSTMerge* requires manually annotated grammars, which demand language-specific expertise and limit scalability. In contrast, *MergirafSemi* leverages existing parser infrastructure and lightweight configuration to reduce the effort required to support new languages. Additionally, it compares tools through conflict-resolution metrics. We adopt a more comprehensive evaluation methodology that considers not only conflict resolution but also merge accuracy through compilation and test validation across a broader set of programming languages.

Cavalcanti et al. [5] introduced S3M, a Java-specific extension of FSTMerge that refines merge results through heuristic handlers, improving accuracy in cases such as renaming. However, it remains tied to a single language. In this work, we compare the two tools to assess whether a language-agnostic semistructured tool can match the performance of language-specific solutions.

Cavalcanti et al. [7] proposed SESAME, which combines semistructured and textual strategies using language-specific syntactic separators. While it reduces false positives compared to line-based merging, it may increase missed conflicts.

Cavalcanti et al. [8] directly compared semistructured and structured merge in language-specific settins. Results show that the two strategies differ mainly in conflicting scenarios and exhibit a clear trade-off: semistructured merge reports more false positives, whereas structured merge introduces more false negatives due to more aggressive merge decisions. Our findings align with this observation, observing the same trade-off despite focusing on language-agnostic tools.

Seibt et al. [23] compared unstructured to semistructured and structured tools, as well as combinations of them. Results found that increasing structural granularity reduces reported conflicts but raises computational cost and false negatives. They suggest hybrid strategies combine the strengths of both structured and simpler techniques. Our results complement this by showing similar trade-offs.

Tavares et al. [24] found semistructured merge offers limited benefits for JavaScript, requiring complex adaptations and yielding only modest conflict reduction. This shows semistructured merge effectiveness depends on language, underscoring the need for flexible designs.

Structured Merge.

Structured merge tools operate on explicit program representations, typically ASTs, to integrate concurrent changes, more accurately distinguishing independent from interfering modifications than unstructured tools.

Apel et al. [1] introduced *jDime*, an AST-based merge tool for Java that combines structured and line-based methods through an auto-tuning mechanism. Subsequent work improved its matching through look-ahead strategies and similarity-based heuristics, enhancing alignment and handling refactorings such as renaming and code movement [26]. Despite these advances, *jDime* relies on language-specific parsing, limiting its applicability.

Larsen et al. [19] proposed *Spork*, which also relies on tree matching (via GumTree [13]) and emphasizes output quality through high-fidelity pretty-printing. While it preserves code structure more effectively than *jDime*, it shares the same limitation of relying on language-specific representations. The core matching and merging algorithms adopted by *Mergiraf* are inspired by *Spork*. As a result, *MergirafSemi* also inherits from this work.

Duarte et al. [11] introduced LastMerge, a language-agnostic structured merge tool built on Tree-sitter, with algorithms inspired by jDime. Their results show that generic structured merging can achieve accuracy and performance comparable to language-specific tools, supporting the feasibility of language-agnostic designs.

Textual Merge.

Textual merge tools operate directly on raw source code, typically relying on line-based algorithms such as *diff3* to integrate concurrent changes. Because they ignore program syntax, they may produce imprecise merge decisions, motivating tools that refine textual matching.

Clementino et al. [9] introduced *CSDiff*, a lightweight tool that enhances line-based merging by incorporating language-aware separators without relying on explicit structural representations. It preprocesses code by isolating syntactic separators (e.g., braces and parentheses) into separate lines, improving alignment across revisions. As an extension, *SepMerge* [3] applies this transformation selectively, refining only conflicting regions identified by *diff3*. However, introducing extra lines can affect LCS computations, potentially leading to unstable results. Additionally, both tools depend on manually defined separator sets, which may not generalize well across languages.

MergeGen [10] uses generative models for conflict resolution, relying on training data rather than explicit program structure. Our tool instead uses deterministic, semistructured merging, but it could be useful for future investigation.

In our design, *MergirafSemi* incorporates line-based merging as a modular component within unstructured regions, allowing it to accommodate and emulate such tools, such as *CSDiff*.

8 Conclusions

In this paper, we propose a language-agnostic semistructured merge tool, *MergirafSemi*, balancing structural awareness with language-independent flexibility. Using lightweight syntactic information via CSTs and semistructural reasoning, it avoids the complexity of structured merging and improves on traditional line-based tools.

Our evaluation provides three main findings. First, increasing structural granularity improves conflict resolution and reduces aFPs, but leads to more aggressive merge decisions and a higher risk of aFNs. In contrast, *MergirafSemi* achieves a more balanced trade-off, maintaining competitive accuracy while providing better runtime performance in most common scenarios. Second, when compared to *diff3* and S3M, *MergirafSemi* reduces aFPs relative to *diff3* and achieves comparable accuracy to S3M with no prohibitive runtime overhead, despite being language-agnostic. Third, relaxing context constraints has only a marginal effect on conflict resolution while introducing few aFNs.

Overall, our results show that semistructured merging can be effectively realized in a language-agnostic setting. This positions language-agnostic semistructured merging as a viable alternative to language-agnostic tools, achieving competitive accuracy with language-specific semistructured tools.

Artifact Availability

To support reproducibility, we provide a replication package containing all artifacts required to reproduce our results [12]. The package includes the dataset of merge scenarios, the exact versions of all evaluated tools (*Mergiraf*, *MergirafSemi*, *MergirafSemi+*, and S3M), the experiment infrastructure, and the scripts used to execute the experiments.

Acknowledgements

We thank the participants of our survey and interviews. For partially supporting this work, we would like to thank INES (National Software Engineering Institute) and the Brazilian research funding agencies CNPq, FACEPE, and CAPES.

References

[1] Sven Apel, Olaf Leßenich, and Christian Lengauer. 2012. Structured merge with auto-tuning: balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE '12)*. Association for Computing Machinery, New York, NY, USA, 120–129. doi:10.1145/2351676.2351694

[2] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*. doi:10.1145/2025113.2025141

[3] Felipe Araujo, Paulo Borba, and Guilherme Cavalcanti. 2024. Refinando a Precisão da Detecção de Conflitos: Uma Análise do CSDiff com Abordagem Focalizada. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 246–256. doi:10.5753/sbes.2024.3395

[4] Max Brunsfeld, Amaan Qureshi, Andrew Hlynskyi, Patrick Thomson, ObserverOfTime, Will Lillis, Josh Vera, dundargoc, Phil Turnbull, Timothy Clem, Douglas Creager, Andrew Helwer, Rob Rix, Daumantas Kavolis, Hendrik van Antwerpen, Michael Davis, Christian Clason, Ika, Amin Ya, Riley Bruins, Tun-Anh Nguyn, Stafford Brunk, Matt Massicotte, bfredl, Niranjan Hasabnis, Mingkai Dong, Samuel Moelius, Steven Kalt, and Kolja. 2025. *tree-sitter/tree-sitter: v0.25.3*. doi:10.5281/zenodo.14969376

[5] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and improving semistructured merge. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 59 (Oct. 2017), 27 pages. doi:10.1145/3133883

[6] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and improving semistructured merge. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 59 (Oct. 2017), 27 pages. doi:10.1145/3133883

[7] Guilherme Cavalcanti, Paulo Borba, Leonardo dos Anjos, and Jonatas Clementino. 2024. Semistructured Merge with Language-Specific Syntactic Separators. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. doi:10.1145/3691620.3695483

[8] Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. 2019. The Impact of Structure on Software Merging: Semistructured Versus Structured Merge. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1002–1013. doi:10.1109/ASE.2019.00097

[9] Jônatas Clementino, Paulo Borba, and Guilherme Cavalcanti. 2021. Textual merge based on language-specific syntactic separators. In *Anais do XXXV Simpósio Brasileiro de Engenharia de Software* (Joinville). SBC, Porto Alegre, RS, Brasil. https://sol.sbc.org.br/index.php/sbes/article/view/18820

[10] Jinhao Dong, Qihao Zhu, Zeyu Sun, Yiling Lou, and Dan Hao. 2023. Merge Conflict Resolution: Classification or Generation?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1652–1663. doi:10.1109/ASE56229.2023.00155

[11] Joao Pedro Duarte, Paulo Borba, and Guilherme Cavalcanti. 2025. LastMerge: A language-agnostic structured tool for code integration. arXiv:2507.19687 [cs.SE] https://arxiv.org/abs/2507.19687

[12] Lopes et al. 2026. MergirafSemi: A Language-Agnostic Semistructured Merge Tool. Zenodo. doi:10.5281/zenodo.21422483

[13] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 313–324. doi:10.1145/2642937.2642982

[14] Gleiph Ghiotto, Leonardo Murta, Marcio Barros, and Andre van der Hoek. 2020. On the Nature of Merge Conflicts: A Study of 2,731 Open Source Java Projects Hosted by GitHub . *IEEE Transactions on Software Engineering* 46, 08 (Aug. 2020), 892–915. doi:10.1109/TSE.2018.2871083

[15] GitHub, Inc. 2020. Greatest Hits — GitHub Archive Program. https://archiveprogram.github.com/greatest-hits/. Accessed: 2026-05-05.

[16] Software Productivity Group. 2026. MiningFramework. https://github.com/predohenr/miningframework. Accessed in: 29/04/2026.

[17] J.J. Hunt and W.F. Tichy. 2002. Extensible language-aware merging. In *International Conference on Software Maintenance, 2002. Proceedings.* 511–520. doi:10.1109/ICSM.2002.1167812

[18] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. 2007. A Formal Investigation of Diff3. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, V. Arvind and Sanjiva Prasad (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 485–496.

[19] Simon Larsén, Jean-Rémy Falleri, Benoit Baudry, and Martin Monperrus. 2022. Spork: Structured Merge for Java with Formatting Preservation. doi:10.48550/arXiv.2202.05329

[20] T. Mens. 2002. A State-of-the-Art Survey on Software Merging. *IEEE Trans. Softw. Eng.* 28, 5 (May 2002), 449–462. doi:10.1109/TSE.2002.1000449

[21] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Softw. Engg.* 22, 6 (Dec. 2017), 3219–3253. doi:10.1007/s10664-017-9512-6

[22] Benedikt Schesch, Ryan Featherman, Kenneth J Yang, Ben Roberts, and Michael D. Ernst. 2024. Evaluation of Version Control Merge Tools. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 831–83. doi:10.1145/3691620.3695075

[23] Georg Seibt, Florian Heck, Guilherme Cavalcanti, Paulo Borba, and Sven Apel. 2021. Leveraging Structure in Software Merge: An Empirical Study. *IEEE Transactions on Software Engineering* 48 (01 2021), 1–1. doi:10.1109/TSE.2021.3123143

[24] Alberto Trindade Tavares, Paulo Borba, Guilherme Cavalcanti, and Sérgio Soares. 2020. Semistructured merge in JavaScript systems. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 1014–1025. doi:10.1109/ASE.2019.00098

[25] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Analysis and Interpretation.* Springer Berlin Heidelberg, Berlin, Heidelberg, 123–151. doi:10.1007/978-3-642-29044-2_10

[26] Fengmin Zhu, Fei He, and Qianshan Yu. 2019. Enhancing Precision of Structured Merge by Proper Tree Matching. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSECompanion)*. 286–287. doi:10.1109/ICSE-Companion.2019.00117