ProMerge: Enhancing Software Merging through Conflict Proactive Detection

Carlos Eduardo Carbonera[a], Kleinner Farias[a], Toacy Cavalcante de Oliveira[b]

[a]*PPGCA, University of Vale do Rio dos Sinos, Av. Unisinos, 950, Sao Leopoldo, RS, Brazil*
[b]*Instituto Politécnico Jean Piaget do Sul, Avenida Jorge Peixinho, n.º30, Almada, Portugal*

Abstract

Context: Proactive source code conflict detection tries to help the efficiency and productivity of software development teams in conflict detection and resolution. Existing approaches often fail to address the challenges of conflict resolution in modern software engineering. Empirical studies suggest proactive strategies help developers resolve conflicts before integrating modified source code. However, semantic conflicts and compilation errors persist, requiring new approaches for continuous development. **Objectives:** We propose ProMerge, an approach designed to assist developers in applying contextual information to detect and resolve source code conflicts during merge tasks proactively. We defined rules to evaluate conflict complexity and resolution efforts. **Methods:** We conducted an empirical study with 32 developers performing ten tasks, five using ProMerge and five with a traditional approach across 320 evaluation scenarios. **Results:** Compared with the traditional approach, ProMerge reduced task completion time by 5.49%. It increased the merge correctness rate by 12.18% and lowered the error rate by 30.76%. **Conclusion:** Results demonstrate ProMerge potential to help development teams resolve conflicts proactively, reduce errors, and improve code merge accuracy.

Keywords: Source Code conflict, Direct Conflict, Higher-order Conflict Severity Definition, Error Rate, empirical study, Correctness Rate
PACS: 0000, 1111
2000 MSC: 0000, 1111

1. Introduction

Managing source code is a critical aspect of software development [10, 13, 25]. Developers often modify and merge code fragments while collaborating in distributed teams, which can lead to conflicts when multiple developers alter the same code snippets [17, 37]. These conflicts are both common and disruptive, frequently causing problems during the software merging process [15, 51]. Such issues arise when changes from different contributors are merged incorrectly, resulting in source code errors or unexpected behaviors in the software system under development [2, 3, 19] Source code conflicts account for approximately 10% to 20% of merges in large projects [25]. According to [48], the KDiff3 algorithm[1] results in incorrect merges in 17% of cases. Consequently, pull requests are often halted due to errors in continuous merges, bugs, or security vulnerabilities [27]. These issues can delay development tasks by hours or even days, negatively impacting the overall software development process [30].

Syntactic conflicts arise when changes challenge language grammar, while semantic conflicts are related to understanding software behavior [1, 7, 39]. These conflicts are complex and challenging to resolve, often discouraging developers from attempting to merge conflicting source code fragments. Additionally, there are concerns regarding the reliability of results produced from questionable source code merges, where the information generated may deviate from the expected behavior, negatively impacting software usability [1, 50, 58]. These issues have been increasingly recognized and require substantial effort to address and correct [36, 57].

Proposed approaches for merging source code fragments have traditionally relied on textual analysis [25, 37, 44]. Currently, version control systems for source code files can be classified into two main types: (1) text-based artifact evaluation: These approaches analyze source code as plain text, independent of the programming language used. This method is widely applied but lacks semantic understanding [4, 18, 37]; and (2) abstract and Structured document representation: These approaches use structured representations of source code, enabling a higher-level analysis that accounts for the relationships and dependencies within the code. These techniques are important when different developers add or modify code fragments within the same functionalities, leading to potential merge conflicts that require resolution [5, 6, 7, 17, 47].

This article introduces *ProMerge*, a tool-supported approach for proactive detection of higher-order conflicts through contextual history analysis. *ProMerge* differs from other approaches by providing comprehensive support for various source code merge conflict types. It assists developers in conflict resolution using concepts such as effort, severity level, and commit timing, all implemented within an Eclipse

[1]KDiff3: `https://kdiff3.sourceforge.net/`

plugin [21]. To assess its effectiveness, we conducted an empirical study in two controlled environments: one utilizing the proposed tool and the other relying on the traditional approach without technical assistance. Developers performed ten similar tasks (five in each scenario) with equivalent complexity levels, resulting in 320 evaluation scenarios. The results obtained with *ProMerge* demonstrated significant improvements in effort reduction, correctness, and error rates compared to the traditional approach. Statistical analyses support these findings, revealing that developers required less time to resolve conflicts as they could quickly identify conflicting fragments. Moreover, the study underscores the importance of precision and attentiveness in conflict resolution. Using *ProMerge*, the effort required for resolution tasks decreased by 5.49%. The correctness rate improved, showing a reduction of 12.18% in errors. Finally, the error rate decreased by 16.42% compared to the traditional approach.

The primary contribution of this work is introducing an approach that assists in proactive conflict detection and resolution using context-sensitive information. Proactive approaches enable earlier detection of potential conflicts while changes are still being developed in local workspaces, notifying developers of conflicts as soon as parallel changes interact [46]. The work presents a scientific contribution to using context histories to infer the detection of direct and higher-order conflicts. According to [29, 9], context-sensitive information refers to data or annotations highlighting critical aspects necessary for a comprehensive understanding of conflict resolution. Such information can be modeled to align with developers' needs, enhancing the resolution process's accuracy and efficiency.

Contextual information was used to increase familiarity improving accuracy in conflict resolution between merging modified or accommodated source code fragments. The empirical knowledge generated benefits: (1) the research community – insights that improve understanding and promote future best practices; (2) software developers – accurate approach to detect and address source code conflicts improving the source code quality and maintainability. This evaluation provides the necessary understanding of the effects generated by *ProMerge* from the software developer's point of view regarding time (effort), correction, and error rate.

This study is organized as follows: Section 2 details the theoretical foundation necessary for a correct understanding of technical terms and the motivation for this study. Section 3 analyzes the related studies filtered. Section 4 presents *ProMerge*, an approach for proactive source code conflict detection using context-sensitive information. Sections 5 and 6 discuss the evaluation and the obtained results. Finally, Section 7 outlines some conclusions and future directions.

2. Concepts and Motivations

This section introduces some concepts (Section 2.1) and presents a motivating scenario (Section 2.2).

2.1. Conflict Detection and Resolution Approaches

Simultaneous modifications to source code files or fragments can lead to merge conflicts, which occur when developers change the same source code fragments [53]. Resolving such conflicts is a complex and time-consuming task [48], prompting developers and researchers to explore new approaches to minimize their occurrence and impact. In [29, 40, 49], the authors describe proactive conflict detection as a speculative approach where modified source code fragments in local workspaces are pulled and analyzed in the background. This contrasts with reactive approaches, where conflict resolution occurs post hoc. The study in [46] highlights that proactive conflict detection and resolution can assist developers in understanding and resolving source code conflicts. Early notification of conflicts enables quicker resolution, reducing unnecessary effort and improving efficiency.

Merge conflicts are an inherent challenge in collaborative software development [38]. While individual modifications may be correct, their combinations can produce inconsistencies [13]. According to [35], most commits merge seamlessly, but parallel changes may result in conflicts. The primary approaches for conflict detection and resolution currently available are discussed as follows.

Direct conflicts. Figure 1 illustrates an example of a direct conflict, which arises when multiple developers modify the same source code fragments or functionality in their local branches [8]. In Figure 1.1, developer A adds lines of source code numbered 12 to 16 in the constructor method of the *Person* class within their local branch. Similarly, in Figure 1.2, developer B adds a different set of lines, also numbered 12 to 16, in the constructor method of the same class within their local branch. This demonstrates that while the modifications occur in the same code segment, the changes differ between the two branches, leading to a direct conflict.

Figure 1: An example of direct conflict.

In this scenario, developers do not know about the modifications in the same source code fragments in other local branches. The first developer to commit changes to the remote branch encounters no conflicts. However, other developers who try to commit modifications to the same fragments in the Person class will face source code conflicts due to the initial commit. To address this, developers should be proactively notified of potential conflict in shared source code fragments that have been modified by others

Figure 2 illustrates an example of a higher-order conflict. This conflict arises when developers modify different source code fragments in their local branches with a relational or hierarchical dependency. In such cases, changes made to one fragment can influence the behavior or results of another. As shown in Figure 2.1, developer A, working in the Sellers class within the *ExportSells* function, added lines of source code from 125 to 129 and line 133 to their local branch. Meanwhile, in Figure 2.2, developer B, working in the *ExportExcel* class within the *writeCsv* method, added lines of code between 68 and 77 to their local branch. Although the modifications occur in different source code files, the hierarchical dependency between these changes creates a conflict that remains unknown to the developers due to the lack of proactive synchronization.

Developer A

Sellers.java

```
121
122     public void ExportSells() {
123         ArrayList<Sellers> _list = this.ListSellersToExport();
124
125         // 5% of discount
126         for (Sellers sellers : _list) {
127             sellers.setAmountSell(sellers.TotalSell * 0.95);
128         }
129
130         ExportExcel _excel = new ExportExcel();
131
132         _excel.writeCsv("C:\\CSV\\Products.csv", _list);
133         System.out.println("Export OK ");
134     }
```

Developer B

ExportExcel.java

```
61      public void writeCsv(String filePath,
62              ArrayList<Sellers> _pedidos)
63      {
64          FileWriter fileWriter = null;
65
66          try {
67              fileWriter = new FileWriter(filePath);
68              fileWriter.append(
69                  "Sequence, Description, Ststus\n");
70
71              for (Sellers u : _pedidos) {
72                  fileWriter.append(String.valueOf(
73                      u.getCodPedido())).append(",");
74                  fileWriter.append(String.valueOf(
75                      u.getDescricao())).append(",");
76                  fileWriter.append(String.valueOf(
77                      u.getStatus())).append(",");
78              }
79
80          } catch (Exception ex) {
81              ex.printStackTrace();
82          }
83      }
```

Figure 2: An example of higher-order conflict.

Thus, source code conflicts will not be detected when the commit is applied. However, a hierarchical dependency exists between the modified functionalities. For example, in the Sellers class, the Export function references the *writeCsv* function, which is declared in the *ExportExcel* class at line 132. This modification can potentially affect the results produced by the Export function when invoked. To address this, higher-order conflicts must be proactively detected, and developers must be notified in advance when modified functionalities depend on or call other functions located in different source code files that have also been altered by other developers in their local branches [7, 17, 46].

2.2. Motivating Scenario

Source code merge research has significantly increased in the last two decades [10]. The goal of resolving source code conflicts is associated with software quality [12, 37, 47]. However, source code modifications are applied to a timeline of the software project development. We characterize a modification management model developed using the traditional approach. Figure 3 characterizes the scenario that motivates this research.

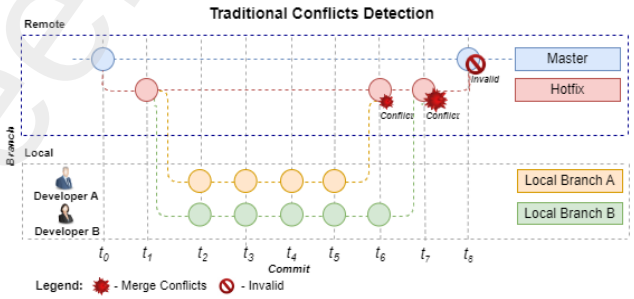

Figure 3: Late detection of higher order conflicts.

We detail a modification management model using the traditional approach. In this environment, t_0 represents the master branch. So, t_1 represents the hotfix branch created to correct bugs using the master branch as the header. Each developer copied the hotfix branch to the local environment to develop new features. From t_2 to t_5 the developers started the activities in the local branch, when each task was finalized the modifications were committed into the local branches without interaction with other developers, in an isolation mode.

When the developers finish the tasks and commit to the local branch, the remote hotfix branch must be updated with the changes, as detailed in t_6 and t_7. We conclude, that at this moment there exists a high probability of source code conflicts during the integration from each developer's local branch to branch hotfix remote, demanding a lot of effort in source code conflict resolution. In t_8 the merge from the hotfix branch with the master branch Is invalid due to the many conflicts created by the source code commits applied.

Figure 1 details a direct conflict example. It occurs when different developers change the same source code fragments or functionalities into local branches. This conflict type is so common in a collaborative environment that the developers do

not know how the tasks are evolved and related to the source code fragments. However, to avoid this problem the developers must be notified about this merge conflict type possibility occurrence, applying corrections to minimize future problems.

Figure 2 we detail a higher-order conflict example that occurs when the developers change different source code fragments, with another source code file in the branch having a direct dependency relationship. This conflict type influences other functionalities referenced by the source code fragment change. This conflict type is common in collaborative environments too. However, to avoid this problem the developers must be notified about this merge conflict type possibility occurrence too.

The main challenge was to develop a proactive approach to source code conflict detection, providing a user-friendly interface with detailed information to resolve conflicts. Development of technical resources that help to detect conflicts, and evaluate the severity level of functionality in a software development project environment. Finally, analyze the technical viability of *ProMerge* presenting indexes of productivity, correctness, and error rates analyzing the applied commits.

3. Related Work

This section presents how this study relates to prior studies, identifying approaches, techniques, and tools applied to analyze source-code conflicts. Related research results were identified in digital repositories such as Google Scholar[2], ACM[3], IEEE[4], Scopus[5] and by applying the search string "(Merge OR Integration OR Fusion) AND (Development OR Maintenance OR Integration) AND (Software OR Application OR Product OR Project) AND (Effort OR Cost OR Pricing OR Evolution) AND (Consolidation OR Productivity OR Time)". In total, ten relevant studies were selected for further analysis.

3.1. Analysis of Related Work

Larsén et al. (2023) [31]. The authors claim that the use of abstract syntax trees (AST-tree) for pure textual analysis reduces the occurrence of merge conflicts. Tending to alter the formatting of merged source codes and presenting excessive execution times about other existing methods for comparison were presented in the SPORK. A tool that preserves the formatting to a significantly higher degree. Evaluations were applied in 119 open-source code projects with 1.740 modified files. Results indicated that future research about structured merging must investigate the preservation of the formatting, contributing to the evolution of quality results without merge limits.

Wu et al. (2021) [56]. The authors defined a differential fact database and a uniform interchangeable representation that supports efficient queries and manipulations based on the existing concepts in the modified software artifacts. The modifications are represented as first-class objects with links between intra-version facts, providing insights into how artifacts evolve, and developing several differential fact extractors that support different programming languages. Results highlight the viability and benefits of the proposed approach in terms of sharing and reusing intermediate analysis results and interoperability between languages. Among the main benefits, an average reduction of time of 44% using differential facts was verified.

Shen et al. (2019) [49]. The authors defined a structured refactoring-aware merge approach based on graph theory, resolving conflicts detected in modified source code fragments. The authors evaluated 1.070 merge scenarios in 10 open-source projects. It observed a reduction in the number of conflicts in 58.90% with the GitMerge approach, proving the effectiveness and practicality of the approach. It defined some future directions: (1) investigating other types of refactoring such as those that influence the control flow; (2) presenting the different types of dependencies between conflict blocks, facilitating conflict resolution by the developers.

Brito et al. (2017) [9]. It presented a refactoring software imputing into GitHub diff context-sensitive information used as a web browser plugin. The authors conducted a controlled experiment with eight professional developers for three months. The authors confirm that source code fragments can be removed wrongly when developers apply several source code fragment modifications. Finally, the authors evidenced a reduction in the cognitive effort required and the average number of lines to be reviewed for the detection and conflict resolution originated by modified Source Code using Git diff textual.

Dias et al. (2017) [18]. It proposed the DeltaImpactFinder, an approach model (not implemented) for proactive detection of semantic conflicts. That analyzes the impact of modifications in the source code branches. The authors affirm the necessary support for correct impact analysis and dependency between the branches obtained using the AST–Tree algorithms. Finally, to evaluate the prototype (if developed), the authors would conduct a controlled experiment with developers with different levels and qualification skills, analyzing results statistically proving the prototype's viability and usability.

Pastore et al. (2017) [40]. The authors claim that the SVN server provides extensive textual conflict detection and resolution widely. However, these higher-order conflicts are still in their infancy and are only partially supported. A new approach to higher-order conflict detection is presented that elevates the analysis of conflicts from the source code level to the functional behavior of modified or merged source code fragments. The proposed approach and textual merge can be analyzed by effectively addressing major higher-order conflicts, and detecting the interfering modifications when implemented with the versioning system, even without introducing any conflict types. The result analysis (by authors) suggests that the proposed software can effectively detect higher-order conflicts and detail how the modifications influence context-sensitive information.

[2]Google Scholar: `https://scholar.google.com/`
[3]ACM Dl: `https://dl.acm.org/`
[4]IEEE: `https://ieeexplore.ieee.org`
[5]Scopus (Elsevier): `https://www.elsevier.com/pt-br`

Table 1: Comparative analysis of related works

Study	Purpose	Conflict Type	Support Conflict Resolution	Platform	CVS Applied	Target Source Code	Research Method	Sample Size	Participant Profile	Insights/Future Directions	Main Contributions	RQ Number	Hypotheses Evaluated	Software Support	Replication Package
Larsén et al. [31] (2023)	Software preserving source code structure from abstract syntax trees (AST)	Syntactic and Semantic	No	Standalone	None	Java [28]	Controlled Experiment	Not Defined	Not Defined	Develop new techniques to preserve the formatting AST tree of objects	software (plugin) for evaluating modified source code	3	9	Yes	Software and Data analysis
Wu et al. [56] (2021)	Presented a differential fact base and a uniform and interchangeable representation based on the existing fact concepts of modified source code fragment	Syntactic and Semantic	No	Standalone	Git	Not Defined	Case Study	Not Defined	Not Defined	New approaches development for sharing and reusing of intermediate analysis results, interoperability between languages/platforms	Commit evaluation software	4	Not defined	No	Unavailable
Shen et al. [49] (2019)	Graph-based refactoring-aware merging algorithm	Syntactic and Semantic	No	Eclipse [21]	Git	Java [28]	Case Study	Not Defined	Not Defined	Identifying dependencies between detected conflicts in source code blocks	Syntactic analysis	3	Not defined	Yes	software
Pastore et al. [40] (2017)	Source code conflict evaluation software by analyzing behavioral models detailing the interfering modifications.	Syntactic and Semantic	No	No	Git	None	Controlled Experiment	Not Defined	Not Defined	Automatic generation of models to represent the behavior of a source code	Syntactic/Semantic Analysis	Not Defined	Not Defined	No	Unavailable
Brito et al. [9] (2017)	Approach that evaluates Git diff intelligently and with refactoring awareness through Chrome browser plug-in	Syntactic	No	No	Git	Not Defined	Controlled Experiment	8	Professionals	Refactoring awareness using Git diff with model analysis to detect behavioral deviations in branches	Google Chrome plugin to detect modifications into source code files (Client/Server)	4	Not Defined	Yes	software
Dias et al. [18] (2017)	Semantic analysis technique for static source codes	Semantic	No	No	Git	Pharo	Case Study	Not Defined	Not Defined	A new approach development	Not Defined	Not Defined	Not Defined	Yes	software
Costa et al. [14] (2016)	Recommendation most qualified developers for branch merge	Not found	No	NetBeans	Git	Java [28]	Case Study	5	Professionals	Recommendation most qualified developers to perform merge by analyzing contributions in branches, considering potential direct or higher-order conflicts that can arise	Commit analysis approaches	Not defined	Not defined	Yes	software and Data analysis
Kasi et al. [29] (2013)	Proactive technique to minimize direct and higher-order conflicts	Not found	No	Eclipse [21]	Not defined	Java [28]	Controlled Experiment	Not defined	Not defined	Implementation and technical evaluation proposed approach	Eclipse [21]plugin detecting dependencies between modified source codes	Not defined	Not defined	No	Unavailable
Sarma et al. [46] (2011)	Proactive detection and resolution software for direct and higher-order conflicts	Syntactic	No	Eclipse [21]	CVS, RCS	Java [28]	Case Study	40	Students	Higher detection and conflicts resolution compared to not using them but involving a reasonable overhead of effort.	Source code merge software	Not defined	Not defined	No	Unavailable
Dewan et al. [16] (2007)	Proactive conflict detection software (working asynchronously)	Syntactic and Semantic	Yes	Not defined	Git	Not Defined	Case Study	16	Not defined	Early conflicts detection by identifying the dependency relationship between conflicting source code fragments, without evaluating the time effort estimation and correctness rate.	Proactive conflict detection plugin working in distributed environments	Not defined	Not defined	No	Unavailable

Costa et al. (2016) [14]. It investigated the difficulties of merging modified source code fragments in large projects in a distributed environment. The authors conclude that it is not hiring the appropriate developer to perform these activities. The main goal is to indicate the most qualified developer analyzing the complexity and number of commits applied. They evaluated 28 projects and found that in 85% the developers that conducted the merge were correctly included, compared with the previous merge, and in 82% cases, developers were correctly selected.

Kasi et al.(2013) [29]. It presented a new technique for proactively minimizing potential direct and higher-order conflicts in source codes, defining them as constraints, and recommending a set of recommendations to reduce the number of these occurrences. Four open-source projects were evaluated to define and characterize the distribution of conflicts and the efforts required for resolution. The results showed that Cassandra can resolve most sets of constraints avoiding conflict. The authors conclude that future research should focus on applying data mining techniques to automatically generate context-sensitive information and dependency analysis of source code fragments, refining the proposed approach to conflict detection.

Sarma et al. (2011) [46]. The approach proposed detects and resolves potential conflicts using analysis of modifications of source code fragments from the developer. Two qualitative and quantitative experiments were applied in a controlled environment to evaluate Palantir's effectiveness and robustness. So, new research opportunities appear mainly investigating different types of existing syntactic and semantic conflicts.

Dewan et al. (2007) [16]. The authors presented software for proactive detection and resolution of direct and higher-order conflicts with support for communication between developers. Allowing the modifications applied to source code fragments in an asynchronous scenario can be resolved in a synchronous environment. They investigated 16 developers with different levels of qualification and age, distributed in pairs performing a defined number of tasks over sixty minutes.

Initially, the participants received 15 minutes of training to understand the usability and functionalities available. So, regarding the new definitions, dependency reports were refined, configured, and activated. Qualitative data were defined so the developers could resolve possible conflicts at the time change and not later, significantly reducing the effort and cognitive load. Table 1 details the principal comparative aspects between the ten studies evaluated.

3.2. Comparative Analysis and Research Opportunity

Table 1 evaluates ten studies carefully selected, detailing the differences and similarities detected. So, we can define the following aspects: (1) the absence of qualitative and quantitative analyses of empirical evidence regarding the time to resolve conflicts in distributed teams; (2) Evaluating dependent and independent variables that influence the artifacts merge and finally; (3) Absence of empirical evaluations evaluating quality results control and versioning software currently available.

So, we defined some research opportunities highlighted and classified according to: (1) Research of new conflict resolution approaches; (2) Evaluation results from empirical experiment executions, and these gaps will be investigated and analyzed in Section 4 detailing the goal of this study. Finally, developers can apply good practices in source code fragments to avoid source-code conflict.

4. ProMerge Approach

This section introduces *ProMerge*, a tool-supported approach for proactive detection of higher-order conflicts using analysis of context history. Section 4.1 presents the main requirements. Section 4.2 revisits the motivating scenario to demonstrate how the proposed ProMerge functionalities mitigate software merge issues reported. Section 4.3 outlines the component-based architecture. Section 4.4 details how developers interact with the *ProMerge* approach. Finally, Section 4.5 presents the implementation aspects.

4.1. Main Requirements

The main requirements of the proposed approach, grounded in findings from previous studies [2, 7, 19, 39], are presented. The requirements are based on the premise that conflict detection and resolution must be done in advance, enabling a collaborative and proactive approach to prevent bad merges.

R1) Proactive and context-aware detection of conflicts. One key requirement of the proposed approach is the proactive detection of severe conflicts using analysis of context history. When conflicts arise, they can ripple through many source code snippets. When conflicts arise, they can ripple through many source code snippets. A tool-supported solution can leverage context-sensitive information to pinpoint these knock-on effects, avoiding the spreading and tangling of conflict problems. ProMerge analyzes direct and higher-order conflicts in advance. This enables early identification of integration issues that could disrupt software development. By incorporating context history analysis, ProMerge effectively propagates conflicts. We seek to ensure that dependencies and interactions among changes are accurately detected and addressed before they escalate. This proactive and context-aware approach not only aims to reduce integration efforts but also to foster collaboration among developers, ultimately minimizing the risk of bad merges and making everyone feel included in the process.

R2) Automated merge error detection and notification. The proposed approach includes a software requirement that focuses on detecting and notifying developers of errors resulting from incorrectly performed merges, referred to as bad merges. This functionality is designed to automatically analyze the merged code and identify bugs, compilation, or building errors that arise from incorrect merge operations. When such errors are detected, the approach promptly notifies developers, enabling them to address the issues before propagating further through many modules. By providing early feedback on bad merges, this requirement helps maintain code quality, reduces

debugging effort, and minimizes the risk of introducing failures into the software system. R2 benefits developers by enhancing code quality, reducing the effort required for debugging, and minimizing the production of bad merges. It can promote a more efficient and error-free development process.

R3) Storage of context-sensitive merge information for conflict and error resolution. The ProMerge approach supports capturing and storing context-sensitive information that arises at the moment in which developers run software merge. When source code is merged into the repository, ProMerge analyzes the modified code fragments and generates detailed context information. This includes the compilation time, commit date and time, the date and time of the merge scenario, the developers involved, and the specific source code snippets affected. Storing this information enables ProMerge to provide valuable insights during conflict resolution and bug identification, such as compilation errors or build issues. By maintaining a comprehensive record of merge contexts, we believe that developers can better understand the sequence of changes, facilitating more effective resolution of conflicts and errors encountered in later stages of development.

R4) Conflict severity measurement. The proposed approach includes a software requirement to measure the severity of conflicts, particularly those affecting highly coupled methods or code snippets. The severity is determined based on the number of dependencies associated with the affected functionality. Conflicts involving methods or functionalities with high coupling can lead to significant propagation of errors if resolved incorrectly. To quantify the severity, four levels were defined based on the number of references to a functionality: (1) *Low*: Values up to fifty, considering only one reference; (2) *Medium*: Values greater than fifty and less than or equal to one hundred, or considering up to two references; (3) *High*: Values greater than one hundred and less than or equal to three hundred, or considering three to six references; and (4) *Serious*: Values greater than three hundred, or involving more than six references. This severity classification is intended to help developers prioritize conflict resolution efforts, ensuring that more critical conflicts, especially those with high coupling, are addressed first to minimize the risk of widespread issues in the codebase. Table 2 presents the different degrees:

Degree	Range Values
Low	≤ 50
Medium	$> 50 \leq 100$
High	$> 100 \leq 300$
Serious	> 300

Table 2: degree of severity.

R5) Conflict analysis and commit timing. The ProMerge approach includes a software requirement for analyzing conflicts and measuring commit timing to assess developer productivity and code quality during development activities. The approach evaluates critical time intervals and the frequency of commits, storing this information in the Merge Storage (PostgreSQL [41]). This analysis focuses on three key aspects: (1) the date and time interval between the first and last submission of source code to the repository, ensuring the integrity of the artifacts; (2) the interval between submissions and commit returns; and (3) the time between acceptance and commit. These metrics provide valuable insights into the productivity and efficiency of development teams and individual developers. Additionally, they enable the calculation of strategic indexes that can support decision-making processes improving the quality of source code merges. The lack of studies addressing these aspects, as highlighted in [30], underscores the importance of systematically analyzing these parameters to identify bottlenecks, improve development workflows, and ensure higher-quality integrations.

4.2. Revisiting the Motivating Scenario

After presenting the requirements of the proposed approach, we revisit the motivating scenario to demonstrate how these requirements address or mitigate the issues caused by the lack of prior synchronization of conflicting changes (described in Section 2.2). Previous studies [25, 48] highlights that between 10% and 20% of source code merges fail, with some cases exceeding 50%. They suggest that new approaches or research could aid developers in resolving source code conflicts. Similarly, in [33], 22% of conflict cases are identified as involving code refactoring, a more complex type of conflict. This increases the difficulty of conflict resolution, influenced by factors such as effort, features, or development complexity. Addressing this gap, *ProMerge* offers an effective solution for synchronizing local branches containing contradictory changes, a common challenge in source code management. Figure 4 shows the proactive detection of higher-order conflicts.

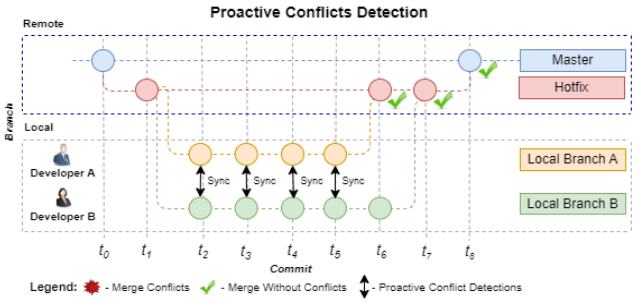

Figure 4: Proactive detection of higher-order conflicts.

ProMerge detects when a developer modifies or accommodates a source code fragment in their local branch. These changes are logged by the *ProMerge Server*, which stores the modification history for future evaluation. If another developer makes similar changes to the same fragment, ProMerge stores a second modification history. The system compares both histories and identifies potential conflicts, whether direct or higher-order. When such conflicts are detected, the *ProMerge View* notifies the developers, prompting them to synchronize their workspaces, ensuring that all developers work with the same source code version. This proactive detection eliminates integration issues and prevents conflicts when merging into the master or hotfix branches.

From t_2 to t_5, conflicting changes between the local workspaces of *Developer A* and *Developer B* are proactively synchronized by ProMerge, ensuring that both developers' environments are consistent and up to date with the most recent modifications.

Additionally, ProMerge ensures that developers' local branches are aligned with the main development branch before committing changes, eliminating the possibility of conflicts during the merge process. Using context-sensitive information, ProMerge provides developers with tools to detect and resolve conflicts early, ensuring the integrity, efficiency, and quality of the development process.

4.3. Component-Based Architecture

Figure 5 presents the component-based architecture of the proposed approach, which aims to modularize its features into distinct software components. The provided component diagram visually represents the architecture of ProMerge, an approach for proactive conflict detection in source code using contextual information. Below is a detailed description of each component in alignment with the diagram:

Conflict View (ProMerge). It is a user-facing architecture component implemented as a plugin for the Eclipse IDE. Its primary role is to display context-sensitive information related to direct and higher-order conflicts in source code files. The Conflict View provides developers with clear messages about conflicts, highlights impacted code fragments, and offers two functionalities: overwriting local files with the repository version and performing local merges. Additionally, it includes a severity evaluation feature to help developers prioritize their actions.

Conflict Engine (ProMerge Server). It serves as the core processing unit of the architecture, hosted on the ProMerge Server. It is responsible for proactively detecting conflicts and generating context-sensitive information. Upon receiving commit data from the *Change Observer*, the Conflict Engine processes modified source code files and produces valuable insights, such as commit history, compilation results, complexity levels, and referencing impacts. It also classifies detected conflicts based on severity and performs automatic conflict resolution while analyzing the time and effort associated with the conflict detection process.

Change Observer (ProMerge Hook). It operates as a *ProMerge Hook* integrated into the SVN Server. Its role is to monitor post-commit events in the version control system. When a commit occurs, the *Change Observer* extracts information such as modified files and timestamps and sends these parameters to the Conflict Engine for processing. Acting as an event listener, this component ensures timely detection and reporting of conflicts.

Persistence (ProMerge Core). It is responsible for managing and storing context-sensitive information processed by the *Conflict Engine*. It ensures the proper formatting and standardization of data before storage while enforcing security protocols and business rules. This component executes a central role in bridging the Conflict Engine and the storage layer, ensuring the consistent and secure handling of conflict-related data.

Storage Connector (JDBC Connector [42]). It is implemented as a JDBC Connector and acts as middleware between the Persistence component and the *Merge Storage*. It aims to establish a reliable, secure, and agnostic connection for transmitting data. By adhering to JDBC standards, the Storage Connector ensures compatibility between the Persistence layer and the underlying database system.

Merge Storage (PostgreSQL [41]). It is implemented using a PostgreSQL database and serves as the persistent storage layer for all context-sensitive information generated by the *Conflict Engine*. It securely stores conflict data, commit history, and related metadata, making this information accessible for visualization in the Conflict View. The Merge Storage is critical for maintaining the historical context and supporting the proactive detection process.

Finally, our components work together to implement the requirements detailed in Section 4.1, which are divided into two stages: *processing* and *visualization*. In the processing stage, the *Conflict Engine* (*ProMerge Server*) receives requests containing parameters from the *Conflict View* (*ProMerge*) and executes the steps necessary to store the resulting information in Merge Storage (PostgreSQL [41]). In the visualization phase, the processed data is presented as user-friendly messages through the Conflict View (ProMerge). When source code fragments at the repository are modified and saved into local branches, a post-commit event is triggered. The *Change Observer* (*ProMerge Hook*) captures the request and initiates processing in the *Conflict Engine* (*ProMerge Server*). The resulting information is stored in *Merge Storage* (PostgreSQL [41]). The *Conflict View* (*ProMerge*) then delivers context-sensitive merge information to developers.

4.4. Developer–ProMerge Environment Interaction Scheme

Figure 6 illustrates the interaction scheme between developers and developers to support proactive conflict detection and resolution. This diagram captures the interaction between two developers (*Developer A* and *Developer B*) and the *ProMerge Server* during software development to identify and resolve conflicts before committing changes to the repository. The process begins with the initialization of the *ProMerge Server*, which waits for requests from the *ProMerge View*. In Step 1, developers save changes to their source code fragments in local branches without committing them. This triggers the server to analyze the changes in the source code, as described in Step 2. Notably, this analysis occurs automatically each time a developer saves changes locally.

In Step 3, the server stores the changes and compares them with other source code fragments previously modified by other developers. At this phase, ProMerge checks for direct or higher-order conflicts in Step 4. If no conflicts are detected, the process continues uninterrupted. However, if conflicts are identified, Step 5 records the conflict, and in Step 6, the server notifies the affected developers about the conflicting changes. Developers must resolve these conflicts proactively in Step 7 before proceeding further. Once conflicts are resolved, developers return to saving their changes, and the proactive conflict detection cycle repeats. At this point,

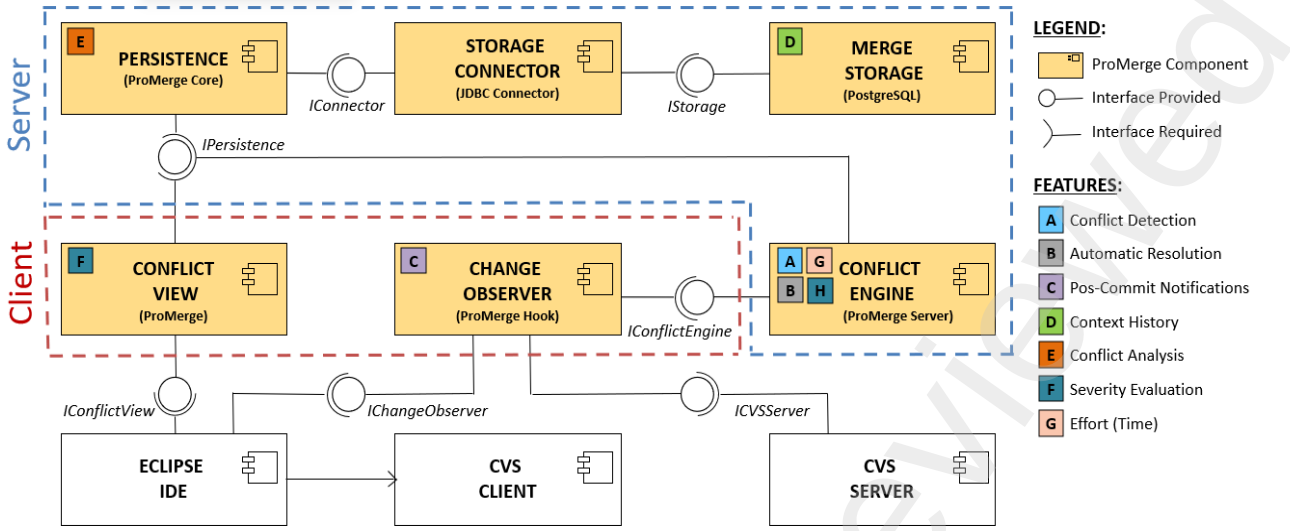

Figure 5: *ProMerge* component-based architecture.

developers can proceed with commits of the source code changed fragments to the repository, as shown in Step 8. Following the commit operation, the ProMerge Server compiles the project files to detect high-order errors in Step 9. If no errors are found, the process concludes. Otherwise, in Step 10, developers are notified about the high-order errors so they can take corrective action. This proactive detection and resolution process ensures that developers remain synchronized and conflicts are addressed early, reducing the likelihood of integration errors during the commit phase.

4.5. Implementation Aspects

Figure 7 introduces *ProMerge* tool (available at our replication package [11]), a support tool for proactive conflict detection using context-sensitive information as Eclipse [21] plugin. We highlight six key aspects: (1) project organization and source code files displayed in the local workspace using Eclipse [21] (Figure 7.1); (2) source code files modified or adjusted by the developer; (3) identification of the developer who last saved or committed changes to the SVN Server; (4) the source code file name containing the conflict; (5) the conflict type, classified as direct or higher-order (proactively detected) or as a syntactic or semantic compilation error; and (6) the conflict complexity level, with detailed classifications provided in Table 2. We highlight that all requirements explained previously were fully implemented. Two possible actions for developers in *Conflict View (ProMerge)* can be run: (1) integrate Source: The source code file can be merged with the repository source code file; and finally (2) synchronize source: The source code file in the current workspace can be overwritten with the version in SVN repository.

Conflict Engine (*ProMerge Server*) processes context-sensitive information directly related to proactive conflict detection. It processes the definition of the complexity level, commit history, and change history, providing developers with the necessary informations to correct the interpretation and resolution of conflicts. The architecture client/server is fully componentized and exists independently and interrelated, allowing a complete isolation of functionalities.

This resource must be referenced on the server machine, also as *Storage Connector (JDBC Connector [42])* and *Merge Storage (PostgreSQL [41])*. The components were developed using the Eclipse platform [21] (Juno Service Release 2) with Java JDK 8 [28]. The *Storage Connector (JDBC Connector)* is represented by the most recent version of JDBC (Java Database Connector) [42]. An open-source JDBC driver written in pure Java, using a native network protocol. Finally, *Merge Storage (PostgreSQL [41])* are represented by the last version of PostgreSQL [41] used as a data repository.

ProMerge tool provides a suite of resources to facilitate the proactive detection of conflict using context-sensitive information helping the developers to source code conflict resolution (as detailed in Figure 7). The main *ProMerge* features includes:

- **Direct and higher-order conflicts:** Performs proactive detection of direct and higher-order conflicts using context-sensitive information histories stored in *Merge Storage (PostgreSQL [41])* and available to the developers by *Conflict View (ProMerge)*.
- **Semantic Errors:** When modified source code files are committed on the SVN Server. The project directory with source code files is compiled to detect syntactic or semantic conflicts. The compilation error messages are available to the developers by *Conflict View (ProMerge)*.
- **Effort:** We provide team managers the possibility to evaluate the effort applied during the development of tasks. Check the time interval between commits applied in the SVN Server, providing a history of users and the time interval between commits, allowing performance evaluation of developers.
- **Complexity Level:** Developers can validate the number of references of a functionality. This allows them to

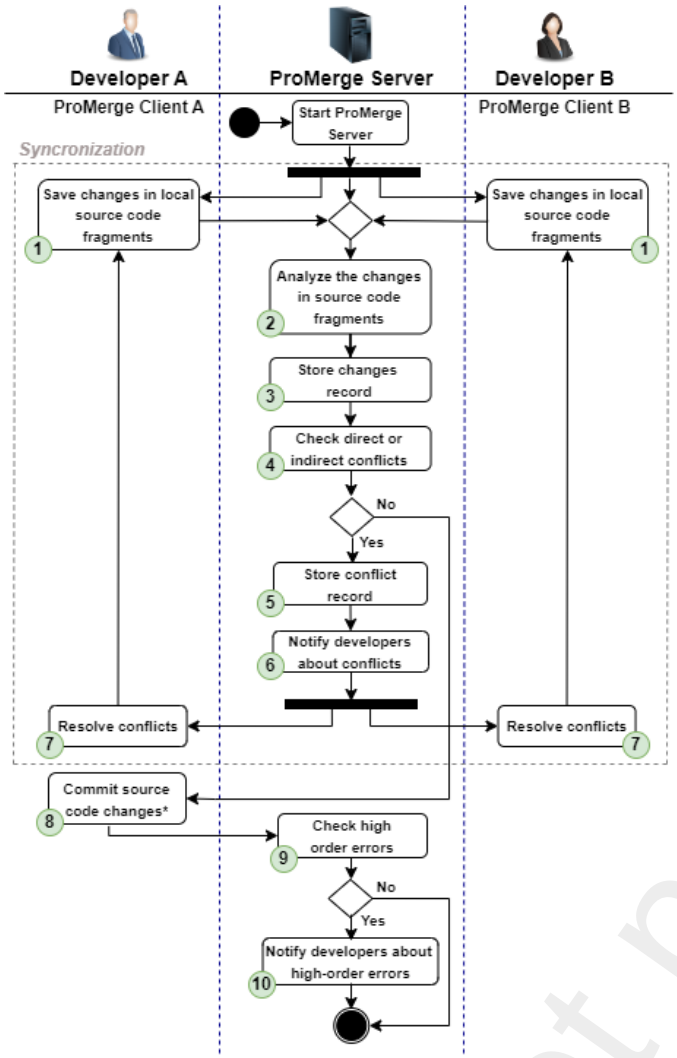

Figure 6: Interaction scheme between developers and *ProMerge* environment. Activities with (*) can be executed by all developers.

define which developers are more qualified to perform maintenance or improvements. By assessing the number of references, it is possible to develop the necessary attention to mitigate potential source code conflicts.

5. Evaluation

We defined two distinct environments for the controlled experiments of *ProMerge*. Each one contains five activities similar in content and at the same complexity level involving source code modifications and evaluating the technical aspects implemented. In the first environment, the developers had the support of *ProMerge*. In the second environment, the developers do not have any technical support to comprehend conflict resolutions, except the resources provided by Eclipse [21].

5.1. Objective and Research Questions

We analyzed three hypotheses, and the environment variables related to each one were evaluated in the results. The controlled experiment investigated important aspects resulting from *ProMerge* tasks. For more rigorously directives, we define the objective of our study, we use GQM (Goal-Question-Metric) template [55], structured as follows:

Analyze merge approaches
for the purpose of *investigate their effects*
with respect to *effort, correctness and error rate*
from the perspective of *software developers*
in the context of *evolving source code*.

We propose an experiment to evaluate the effects of understanding and resolving the detected conflicts in source code excerpts from a controlled experiment. The analysis results are segmented into two principal groups: the evaluation proposed approach and qualitative. Five activities involving source code implementations with different complexity levels were applied. This study evaluated the effects of *ProMerge* on the developer's effort, correctness, and error rates in specifying source code merge conflicts. Thus, we formulate three research questions (RQs) to explore this goal better:

- **RQ1:** How much effort do developers invest in source code conflict resolutions?
- **RQ2:** What is the developer's correctness rate in source code merge conflict resolutions?
- **RQ3:** What are the developer's error rates in source code merge conflict resolutions?

5.2. Hypotheses Formulation

We defined hypotheses to evaluate the activities development using *ProMerge* to proactive conflict detection from source code fragments merged in a controlled experiment. Three hypotheses were formulated in this stage: hypothesis 1 analyzed the early conflict detection by evaluating the effort to resolve the conflicts with and without the support *ProMerge*. Hypothesis 2 analyzed the correctness rate of conflict resolution detected by the participants. Finally, hypothesis 3 evaluated the error rate detected after executing the tasks proposed in the experiments, comparing the two approaches. Table 3 presents the hypotheses investigated.

Null Hypothesis	Alternative Hypothesis
H_{1-0}: $\mathbb{ET}_{ProMerge} \geq \mathbb{ET}_{Traditional}$	H_{1-1}: $\mathbb{ET}_{ProMerge} < \mathbb{ET}_{Traditional}$
H_{2-0}: $\mathbb{CR}_{ProMerge} \geq \mathbb{CR}_{Traditional}$	H_{2-1}: $\mathbb{CR}_{ProMerge} < \mathbb{CR}_{Traditional}$
H_{3-0}: $\mathbb{ER}_{ProMerge} \geq \mathbb{ER}_{Traditional}$	H_{3-1}: $\mathbb{ER}_{ProMerge} < \mathbb{ER}_{Traditional}$

Table 3: Analyzed Hypotheses

Hypothesis 1 (H1): Effort ($\mathbb{ET}$). We evaluated the interval between the source code conflict resolution and the effort corrections that ensure the integrity evaluated in minutes. This measurement was defined in a technical mode and is not based on any research or publication and this time scale was applied by another empirical experiment type. That can be fully auditable and customized according to the evaluation

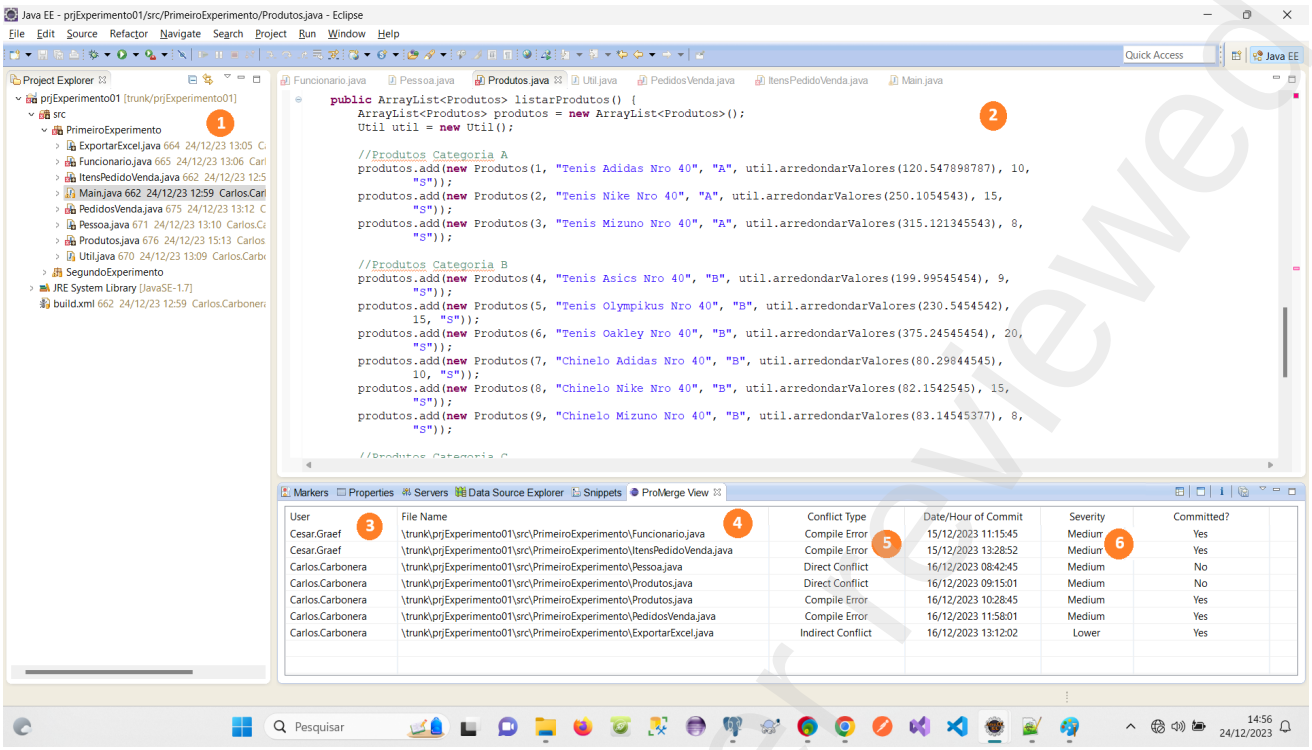

Figure 7: *ProMerge* tool into the Eclipse[21] platform.

criteria between the experiments, facilitating the analysis indicators. However, this hypothesis can not be valid. For example, in many occurrences multiplied by the number of developers, the final effort can be considered much higher than team development standards. Therefore, hypothesis 1 assessed whether *ProMerge* presents significant reductions in effort compared to heuristic-based approaches. The null and alternative hypotheses were presented as follows:

Null Hypothesis 1, H_{1-0}**:** *ProMerge* does not present significant reductions in effort conflict resolution.
H_{1-0}**:** $\mathbb{ET}_{\ ProMerge} \geq \mathbb{ET}_{\ Traditional}$
Alternative Hypothesis 1, H_{1-1}**:** A significant reduction in effort were founded in conflict resolution using *ProMerge*.
H_{1-1}**:** $\mathbb{ET}_{\ ProMerge} < \mathbb{ET}_{\ Traditional}$

Hypothesis 2: Correctness Rate ($\mathbb{CR}$). We investigated the utilization of pair programming techniques for conflict resolution. This approach proved to be a robust alternative, increasing the assertiveness level in conflict resolution. The developers involved cannot have the technical knowledge required (individual or collective) or the understanding necessary for the impact of modifications. This occurs partly because it´s impossible to identify test scenarios required for the necessary corrections to the source code files. The paired analysis has proven to be more effective than the individual analysis. The correctness rate analyzes the assertiveness of merging source code fragments modified. As evaluation criteria, the number of technical problems multiplied by the effort to resolve source code conflicts. We applied heuristics to compose indicators with a high accuracy degree, it is not obvious that there is an objective approach to reducing the time to correct conflicts. Most problems can be detected between execution or in the results evaluations from modified functionalities. Hypothesis 2 evaluated if the correctness rate of *ProMerge* degree presented significant reductions compared to not using heuristic-based approaches. Finally, the null and alternative hypotheses were presented as follows:

Null Hypothesis 2, H_{2-0}**:** *ProMerge* does not present significant reductions in the correctness rate in conflict resolution.
H_{2_0}**:** $\mathbb{CR}_{\ ProMerge} \geq \mathbb{CR}_{\ Traditional}$
Alternative Hypothesis 2, H_{2-1}**:** A significant reduction in the correctness rate in conflict resolution was observed with *ProMerge*.
H_{2-1}**:** $\mathbb{CR}_{\ ProMerge} < \mathbb{CR}_{\ Traditional}$

Hypothesis 3: Error Rate ($\mathbb{ER}$). We proposed in *ProMerge* evaluation approaches to provide the error rates for each developer participating in the controlled experiment regarding the tasks executed. Provide the understanding necessary for the complexity of source code files. The evaluation error rates are based on the complexity analysis to provide a systematic option to compose this index. So, there can be many advantages in functionalities evaluation into complex evolution scenarios. However, the applicability is not enough to reduce the error rates quickly and accurately compared to results of one minus the correctness rate defined in Table 4. Hypothesis 3 evaluated if the error rates produced by *ProMerge* presented significant

reduction rates about the traditional approach. The null and alternative hypotheses are presented as follows:

Null Hypothesis 3, H_{3-0}: *ProMerge* does not present a significant reduction in the error rate in conflict resolution.
H_{3-0}: $\mathbb{ER}_{\ ProMerge} \geq \mathbb{ER}_{\ Traditional}$
Alternative Hypothesis 3, H_{3-1}: A significant reduction in the error rate in conflict resolution was observed with *ProMerge*.
H_{3-1}: $\mathbb{ER}_{\ ProMerge} < \mathbb{ER}_{\ Traditional}$

Finally, in hypothesis H1 we investigated if the time-based approach implies or not in minors effort rates related with the application of heuristic-based approaches. In hypothesis H2 we produced empirical knowledge about the correctness rate in modifications applied to the source code files about the inconsistency level. Finally, in hypothesis H3 we obtained empirical knowledge related to the error rate in the execution of tasks with the help *ProMerge* compared to the traditional approach and the heuristic-based approaches.

5.3. Study Variables

We investigated if the variables applied to the hypotheses formulation evaluated the effort to merge the modified source code files and provided the understanding necessary for conflicts resulting from merges of source code fragments. The effectiveness and relevance of knowledge of prior variables can increase the understanding levels of developers when applying other experiments. The independent variables are represented by the *ProMerge* and Traditional approaches. For each task proposed, exists a corresponding task in the other scenario, similar in structure and complexity level. If a developer begins the experiment using *ProMerge* scenario, the next developer uses the traditional approach. The results investigated the dependent variables, detailed in Table 4.

We defined that the controlled experiment must be composed of three dependent variables: Effort ($\mathbb{ET}$), Correctness ($\mathbb{CR}$), and Error Rate ($\mathbb{ER}$). (1) The $\mathbb{EF}$ evaluates the resolution time tasks in minutes proposed as part execution or conflicts generated after the merge source code files. (2) The $\mathbb{CR}$ evaluates the correct execution activities by the developers. The advantages and disadvantages of using *ProMerge* were analyzed. The $\mathbb{CR}$ is the number of correct answers by the total of tasks executed correctly, represented by the formula: $\mathbb{CR}$ = Total of Correct Answers / Total of Responses. (3) the $\mathbb{ER}$ evaluates the result of one minus the $\mathbb{CR}$ found, represented by the formula: $\mathbb{ET}$ = 1 - $\mathbb{CR}$. Finally, about the independent variables, *ProMerge* investigated the relation between dependent variables with the traditional approach, other independent variables will be disregarded by this study too. Table 4 details the variables involved and analyzed to validate *ProMerge*.

Type	Variable	Scale
Dependents	Effort ($\mathbb{ET}$)	Range [0..n]
	Correctness Rate ($\mathbb{CR}$)	Range [0..1]
	Error Rate ($\mathbb{ER}$)	Range [0..1]
Independents	Merge Technique	Nominal: *ProMerge*, Traditional

Legend: Time in Minutes *(n)*

Table 4: Dependent and Independent Variables.

5.4. Experimental Process

We divided the controlled experiment into three steps. The participants were instructed about the experiment enabling the capacity necessary to perform the tasks. In step 1 a characteristics questionnaire was applied to the participants, receiving some technical information before the experiment. Step 2 consisted of explaining *ProMerge* individually to the participants, starting to execute and record the experiment tasks collecting information from each of them in the two proposed scenarios. Finally, in step 3 the participants answered another questionnaire to understand the technical perception experiment conducted, applied after finishing the experimental tasks following the TAM model (Technology Acceptance Model) (link to access in [11]), that are presented:

Phase 1: Training. The participants were trained about *ProMerge* utilization ensuring full understanding proposed tasks. Examples were applied to detailing to the participants regarding the experimental environment. After answering the characteristics questionnaire, the controlled experiment workspace was available individually with instructions and activities to be conducted.

Phase 2: Execution of Experimental Tasks. The tasks in the two proposed controlled experiment scenarios were similar with the same levels of complexity. The technical criteria evaluation was the same in scenarios, mitigating the creation of problems. In the first scenario, the developers had support of *ProMerge* in the activities. The second scenario applied the traditional approach to conflict detection, with only the resources available in the Eclipse [21].

Phase 3: Analysis and Evaluation of Results. The participants resolved two questionnaires. The first was applied before the experiment and the second when the experiment was concluded. The characteristics questionnaire tried to get information about the participant's profile, including professional experience, academic background, age, current position, experience with software development, and others. Finally, the last questionnaire contained sixteen questions that evaluate the developer's perceptions regarding the experiments conducted and the acceptance *ProMerge*.

The tasks of the controlled experiment evaluated the performance and quality of each participant's development. This involves the development tasks with direct, higher-order, syntactic, and semantic conflicts. Finally, the source code files resulting from proposed tasks were identified and cataloged (by the participant), and evaluated with the standard response model defined as ideal defining the error degree and correctness questions. The participants executed the activities individually

and without external interruption, avoiding any distortion in results. Finally, the process applied to evaluate the steps of the controlled experiment is presented in Figure 8.

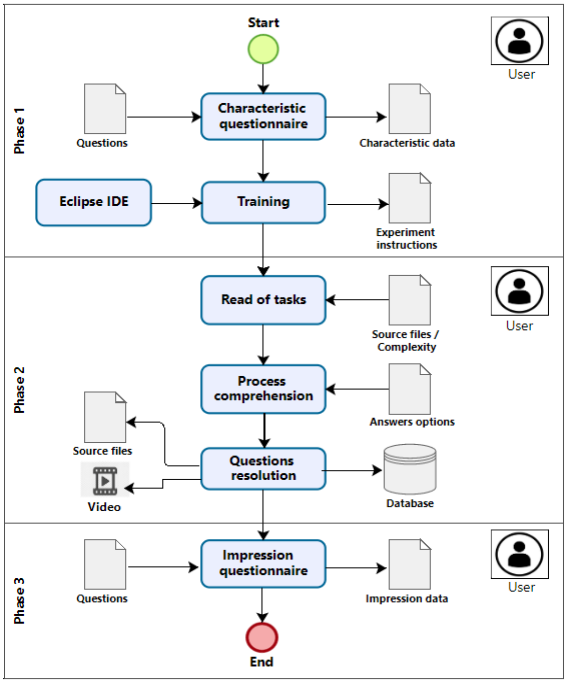

Figure 8: *ProMerge* experimental process.

5.5. Selection of Participants

The technical characteristics of participants impact the controlled experiment findings in software engineering [20, 32, 23, 43]. Guidelines emphasize the importance of recruiting the correct participants. There exist limitations on the guidance on achievement this effectively [32]. So, to increase the generalization of obtained results, we followed very known and applied guidelines related in [32, 43] to support our context definition and participant selection. This defines the desired population characteristics and identifies a sample of sources to minimize the difference in results, referring to the degree of dissimilarity between the sample and the population.

So, we recruited 32 participants by technical nearest, including students and professionals (with industry experience) invited by email or Linkedin[6] professional network. Students were recruited to the controlled experiment by their knowledge of software merge tools and Java [28] development language. This ensures a consistent baseline of skills and availability, providing a controlled and homogeneous sample for evaluating *ProMerge* effectiveness. Convenience samples, often used in research, may not accurately represent the population under study [22, 45]. We limited our findings to the population considered. This participant pool encompassed individuals holding MSc and bachelor's degrees or equivalents and substantial proficiency in software maintenance tasks and object-oriented programming.

[6]Linkedin: `https://www.linkedin.com/`

Convenience samples are used in controlled experiments, but cannot accurately represent the population under study [12, 24]. We limited our findings to the population considered. This diversified participant group encompassed developers with MSc and bachelor's degrees or equivalents and substantial proficiency with software maintenance development and object-oriented programming. The participant selection process tried to ensure a diversification of profiles and expertise levels among participants, enriching the study findings with important perspectives and insights.

We applied the controlled experiment using the same stringent quality standards equal to practical laboratory exercises. Each participant received substantial training about *ProMerge* usability and the experimental process, ensuring the validity and reliability of the application of tasks. This approach mitigates potential biases to ensure the consistency necessary for the participant's understanding and engagement throughout the evaluation process. Table 5 details the controlled experiment participant profiles.

Characteristics (n=32)	**Answers**	**Total**	**%**
Education level	Graduation	21	65.63%
	Specialization	7	21.88%
	Master degree	4	12.5%
Under graduation	System analysis	9	28.13%
	System information	4	12.5%
	Computer science	16	50%
	Computer engineering	3	9.38%
Years of experience	< 2 years	1	3.13%
	2–4 years	7	21.88%
	5–6 years	10	31.25%
	7–8 years	7	21.88%
	> 8 years	7	21.88%
Age	20–25 years	5	15.63%
	26–30 years	8	25%
	31–35 years	2	6.25%
	36–40 years	5	15.63%
	41–45 years	6	18.75%
	> 45 years	6	18.75%
Position years	< 2 years	3	9.38%
	2–4 years	2	6.25%
	5–6 years	7	21.88%
	7–8 years	1	3.13%
	> 8 years	19	59.38%
Current position	Developer	16	50%
	System analyst	7	21.88%
	Software architect	5	15.63%
	Test analyst	1	3.13%
	Director	1	3.13%
	Teacher/Instructor	1	3.13%
	Quality Assurance	1	3.13%
Experience position	< 2 years	6	18.75%
	2–4 years	7	21.88%
	5–6 years	5	15.63%
	> 8 years	14	43.75%
Type	Male	30	93.75%
	Female	2	6.25%

Legend: Total of Participants *(n)*

Table 5: Characteristics Questionnaire Result.

In this sense, we applied 32 participants with technical experience, including professionals and researchers (with software industry experience) were invited. Experiment samples are frequently used in research and can not accurately

represent the population under study [54]. So, we restrict our findings to the population considered. This group was formed by participants as MSc or PhD students with a bachelor's degree or equivalent and substantial technical proficiency in software maintenance or object-oriented programming tasks. The selection process presented a heterogeneous profile and knowledge levels among participants, enriching the controlled experiment findings containing varied perspectives and insights.

Applying this approach we tried to mitigate the potential biases, ensuring the consistency necessary for the participants understanding and engagement during the evaluation of the controlled experiment. A detailed discussion of their experience with source code merge is essential to interpret the results obtained. Addressing this aspect, we try to provide interesting insights into the obtained results, enhancing the understanding of how developers' expertise may affect their interaction with the tool.

5.6. Analysis Procedure

Quantitative analysis. We applied descriptive statistics to analyze the normal distribution and the statistical inference to test the hypotheses. We conducted a descriptive analysis to evaluate the normal distribution, dispersion, and trends, such as means and medians, of the data collected. So, we applied statistical analyses to investigate the hypotheses, with a significance level of $\alpha = 0.05$. The single sample student's t-test was performed to analyze hypotheses H1, H2, and H3.

Wilcoxon test were applied to analyze the differences between two paired samples, especially when there are two available measurements from the same sample of information. The participants were evaluated in two conditions, as the main techniques for evaluating paired samples. The results were classified by comparing the difference between two time intervals, applied in this study for hypotheses H1 and H3. The paired t-test investigates whether the means of two related outcome measures are classified as statistically different. The null hypothesis of a t-test in paired samples is defined by the mean differences between the measures equal to zero, therefore there is no difference between the measures.

McNemar's evaluation model was chosen to define whether the results of the paired proportions are different. Evaluated in two different approaches are compared by performing the same experiments evaluating the results between *ProMerge* compared to the traditional approach (hypotheses H1 and H3). Non-parametric tests and descriptive analysis were used to assess the results to identify the corresponding (paired) data or repeated measurements on nominal (categorical) and ordinal (classified) scales, especially when there are small samples of data that parametric techniques cannot evaluate.

The single-sample Students t-test, McNemar, and Wilcoxon test were selected by appropriateness with analyzing the mean difference between a sample and hypothesized population mean aligning with the experimental guidelines well-defined in [55]. The standard deviation was applied to express the dataset dispersion degree indicating the uniform data distribution. More near the standard deviation from 0 to the data are homogeneous, indicating the value in the middle of a dataset when ordered. The median indicates that half (50%) of values present in a set of values are major or minor, presenting a measure of central tendency. The mean was defined as the value that demonstrates the concentration of data in a distribution, the equilibrium point frequencies result in a histogram, interpreted as a significant value in a sequence of results.

Therefore, we can define the mean by measuring values in a specific method generating significant results. Represent robust evidence to reject hypothesis 1. The null hypotheses 2 and 3 cannot be ignored, the results are presented in The WINKS Statistics[7] were used to evaluate the descriptive and inference statistics test were analyses using the data resulting from a controlled experiments executed by the developers. Table 6 grouped by dependent variables.

	ProMerge			**Traditional**		
	ET	**CT**	**ER**	**ET**	**CR**	**ER**
Min	194	0.25	0.00	210	0.25	0.00
25th	245.50	1.00	0.00	289	0.75	0.00
Mean	298	1.00	0.00	323.50	1.00	0.00
75th	375.50	1.00	0.00	395.75	1.00	0.25
Max	470	1.00	0.75	461	1.00	0.75
Med	311.65	0.90	0.09	338.75	0.87	0.13
SD	72.67	0.22	0.28	65.64	0.22	0.21

Legend: Minimum *(Min)*, Average *(Mean)*, Median *(Med)*, Maximum *(Max)*, Standard Deviation *(SD)*, Effort seconds *(*$\mathbb{ET}$*)*, Correctness Rate *(*$\mathbb{CR}$*)*, Error Rate *(*$\mathbb{ER}$*)(n)* Participants

Table 6: Descriptive Statistics of Results (n=32).

So, we concluded that the dependent variables *ProMerge* presented better results than traditional variables. The information was segmented by scenario and task, displaying the average values of each one. Therefore, we observed a significant difference between the two scenarios controlled experiment for each task in Table 7, supported by the statistics presented in the following sections. Finally, the Effort ($\mathbb{ET}$) showed a reduction of time an average of 5.49% to task resolutions, and the Correctness Rate ($\mathbb{CR}$) presented a reduction of 12.18%. Finally, the Error Rate ($\mathbb{ER}$) showed a lower rate of 16.42% compared with the traditional approach.

Qualitative analysis. Data sources contributed to the qualitative data collection questionnaires, audio and video recordings, transcriptions, comments, and interviews. This multifaceted and multi-step approach enabled the acquisition of complementary evidence to elucidate the quantitative findings. Finally, we created baseline evidence to systematically align the quantitative and qualitative data avoiding the derivation of conclusions.

To evaluate the acceptance by developers and the usability of *ProMerge* were used the Technology Acceptance Model (TAM) questionnaire [34] (accessible in [11]). This model

[7]WINKS Statistics: `https://www.alanelliott.com/TEXASOFT/`

Variable	Approach	TA1	TA2	TA3	TA4	TA5	Average
Effort ($\mathbb{ET}$)	*ProMerge*	312	245	223	134	93	201
	Traditional	339	254	233	138	95	212
Correctness	*ProMerge*	0.91	0.91	0.86	0.92	0.94	0.91
Rate ($\mathbb{CR}$)	Traditional	0.88	0.89	0.80	0.88	0.90	0.87
Error Rate	*ProMerge*	0.09	0.09	0.14	0.08	0.06	0.09
($\mathbb{ER}$)	Traditional	0.13	0.11	0.20	0.12	0.10	0.13

Legends: (s) - Time in Seconds, TA - Task

Table 7: Average Values per Task.

evaluates three criterias: *acceptance of use*, which measures the user's impression the technology reduces their effort; *perceived usefulness*, which measures how much the technology can enhance their development activities; and *behavior intention*, which seeks to capture the usability of the proposed approach. Each question had response options on a standard 5-point Likert scale[8], allowing the developers to express their views as follows: Completely Disagree, Partially Disagree, Neutral, Partially Agree, and Completely Agree.

5.7. Replication Package

All our artifacts are available in the replication package [11] with access public. The package includes several key elements: Software, installers, detailed setup instructions for replicating the experimental environment, and the necessary code and scripts to execute the controlled experiments. We have also provided thorough documentation explaining the procedures followed during the study, including the tasks applied to the scenarios. The package contains the source code and the data collected during the experiments, allowing a comprehensive verification of the presented results and the reproducibility of our research by the academic community to validate, replicate, or develop other controlled experiments over our findings. We hope this package can encourage further research and lead to new questions, methodological improvements, and detailed insights into the concepts explored in our study.

6. Results

Our evaluation scenarios were chosen to investigate different aspects of the execution activities involving source code modifications or the accommodation of new functionalities. The analysis of results was executed using the concept of the dependent variable detailed in Table 4.

6.1. RQ1: ProMerge and Effort

We investigated the effort necessary to resolve the tasks proposed for the two experimental scenarios. Table 8 presents the results of descriptive statistics based on the information. The main finding indicates that the effort applied by the participants resolving the tasks using *ProMerge* was more minor than when using the traditional approach, represented by Figure 9. Scenario A details the correct answers from the task executions between the *ProMerge* and the Traditional approach. Scenario B details the incorrect answers from the task executions in both scenarios. The scenarios presented the same scoring levels for the tasks considered in the controlled experiment.

So, when the activity is correct, the items evaluated must be considered correctly. For each task of the controlled experiment in two possible scenarios, result data were compared with answers defined correctly. On the correct answers, we detected that the effort applied to resolve the task with the help *ProMerge* was lower compared with the traditional approach. Identifying the necessary effort to correct interpretation *ProMerge* was important to obtain higher assertiveness levels about the Traditional approach. Figure 10 details the results of the answers:

Table 8 details that *ProMerge* presented a major accuracy level compared to the traditional approach. The findings reveal the statistics and p–values for the pairwise comparisons. In major cases, the p–value was lower ($p \leq 0.05$). Therefore the null H1 hypothesis can be rejected. Wilcoxon test were applied to evaluate the non-normalized distribution data. They were applied as an alternative non-parametric static hypothesis to combine the means of two related samples, as pairwise difference tests. The paired t-test uses the average variation between the experiment data measures within the different complexity levels and details the p–values for the pairwise correlation.

Tasks	Wilcoxon		Paired t-test		
	p–value	W	p–value	DF	t
All	0.001	278.65	0.001	159	7.799
Task 1	0.001	512.5	0.001	31	7.376
Task 2	0.201	406	0.004	31	3.087
Task 3	0.107	410.5	0.001	31	3.804
Task 4	0.042	341	0.019	31	2.457
Task 5	0.110	349	0.116	31	1.614

Legend: Degree of Freedom *(DF)*.

Table 8: Statistical Analysis Experimental Tasks Resolution.

Our results demonstrated the existence of a significant variation between the proportions of correct answers in the source code files using *ProMerge*. The evaluations were compared with the traditional approach detailing the pairwise p–values for each measure. The p–value indicates the statistically significant results and rejects the null hypothesis. The sum signed ranks (RT) determines the direction in which the results are relevant. In task 4, W has a value of 341, and the p–value is less than 0.05 (p = 0.042) in the Wilcoxon test for the measurement between the two approaches. The inconsistency levels are significantly lower when compared with the group with the traditional approach. Finally, to the H2 Hypothesis exists statistical evidence that *ProMerge* presented a positive impact over effectiveness in several scenarios.

[8]Likert scale: `https://en.wikipedia.org/wiki/Likert_scale`

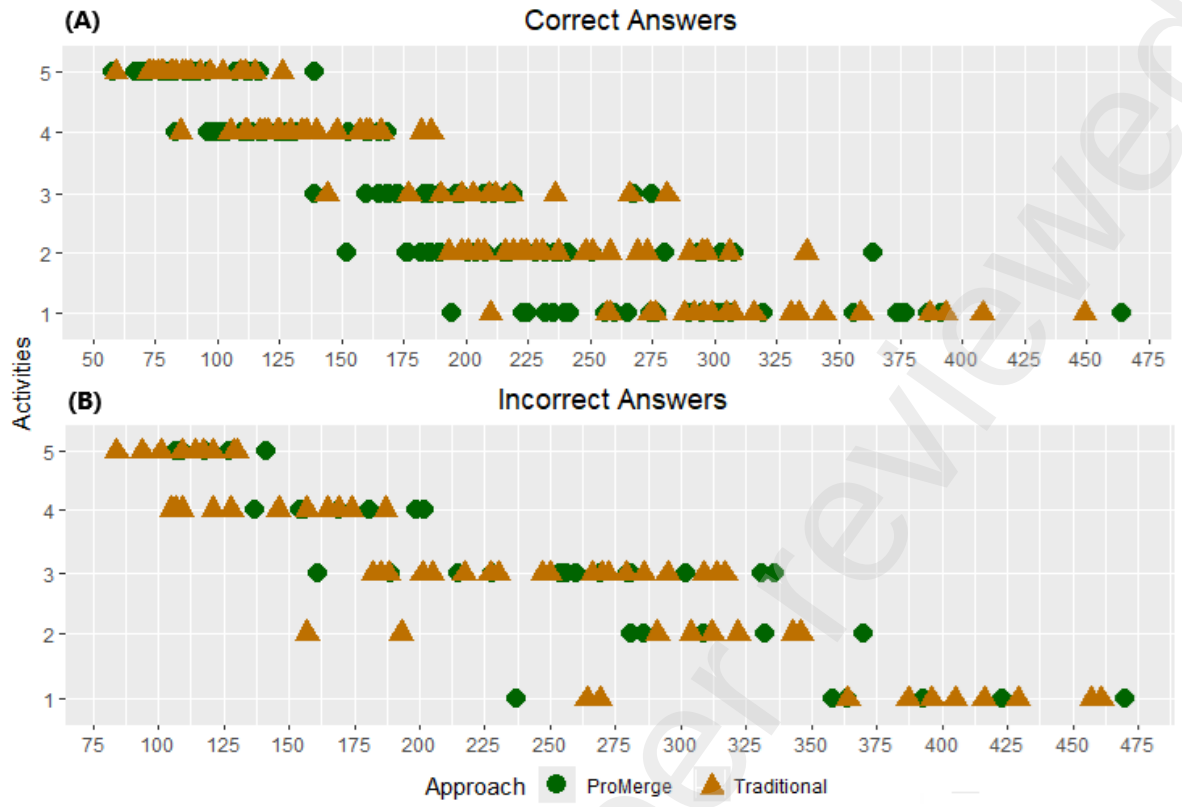

Figure 9: Effort evaluation – *ProMerge* and Traditional approaches.

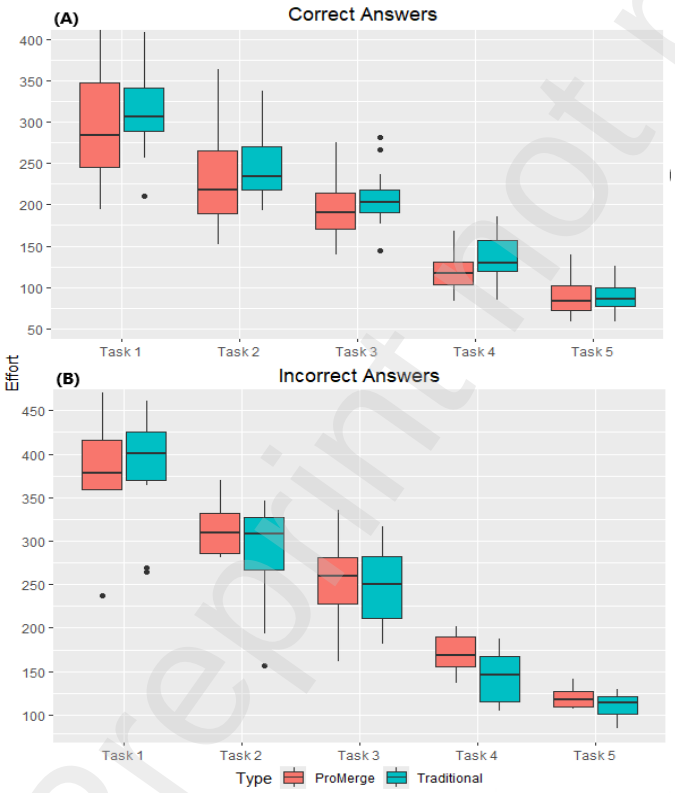

Figure 10: Comparative effort evaluation–*ProMerge* and Traditional.

Summary of RQ1: We conducted the Wilcoxon Signed-Rank Test and Paired t-test to determine whether there was a statistically significant difference at dependent variables: $\mathbb{ET}$, $\mathbb{CR}$ and $\mathbb{ER}$ evaluates *ProMerge* and Traditional approach. The results indicated that the $\mathbb{CR}$ average values were 12.18% larger about other dependent variables as evaluation criteria when compared to the traditional approach. The p–values are equal to 0.001 in the tasks, much lower than 0.005. These results suggest that other experiments applying more complex tasks with professionals classified according to a professional rank can present interesting and robust results.

6.2. RQ2: ProMerge and Correctness Rate

We analyzed the results from *ProMerge* about the impact of the correctness rate compared with the traditional approach. Table 9 and Figure 11 shows the results from applying descriptive statistics detailed by each task. In scenario A, are detailed the correct answers and in scenario B the incorrect answers for ProMerge and traditional approaches. The y-axis represents the quantity of correct and incorrect answers for each task, the x-axis represents the tasks conducted in the two scenarios that make up the experiment. The histogram demonstrates the results by task evaluating the degrees of complexity of each one.

Our results indicate the existence of a significant variation between the correct responses percent generated by *ProMerge* evaluated about the traditional approach. Therefore, there

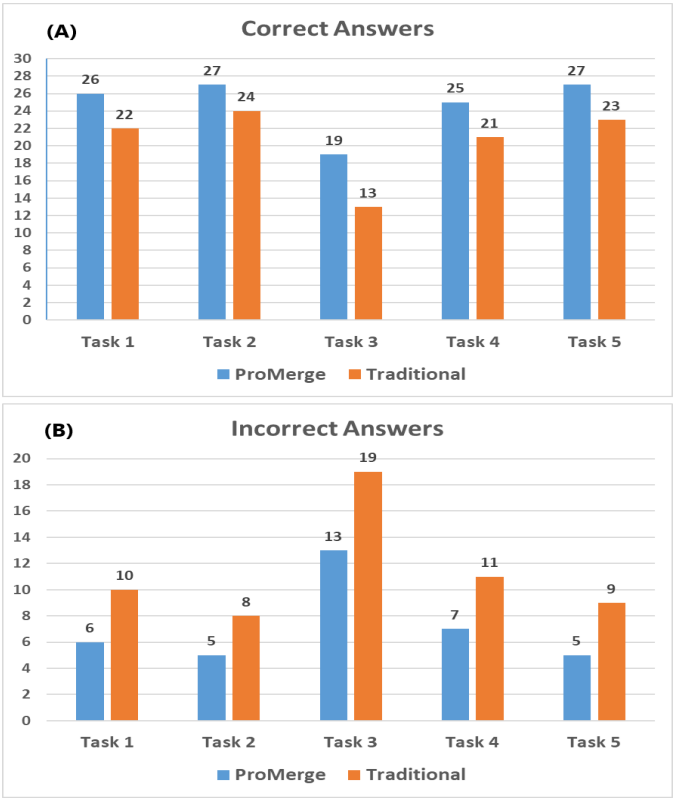

Figure 11: Correctness rate per tasks of controlled experiment.

Tasks	ProMerge		Traditional		Results	
	CO	IC	CO	IC	Chi-Square	p–value
All	124	36	103	57	12.90	0.001
Task 1	26	6	22	10	1.50	0.221
Task 2	27	5	24	8	0.80	0.372
Task 3	19	13	13	19	3.13	0.078
Task 4	25	7	21	11	1.13	0.289
Task 5	27	5	23	9	2.25	0.134

Legend: Correct *(CO)*, Incorrect *(IC)*.

Table 9: McNemar evaluation Comparison Results – Hypothesis 3.

is sufficient statistical evidence to conclude that *ProMerge* indicated a positive impact on the effectiveness of the evaluation scenario data from the tasks. Finally, the execution and evaluation of other experiments segmented and categorized according to the technical experience participants can present different results presented in Table 9.

Summary of RQ2. We performed the McNemar test to evaluate and detect significant changes in variables: p–values and chi-square. The values showed that the Chi-Square (n = 32) was equal to 12.9 and p–values equal to 0.001 in the tasks. *ProMerge* presented better results than the traditional approach.

6.3. RQ3: ProMerge and Error Rate

We conducted the hypothesis H3 to analyze the error rates of tasks by the controlled experiment with the data collected from *ProMerge* and the traditional approach(presented in Section 5.2). Each task was evaluated and the source code files resulting from experiments were compared with source code files defined as the response standard evaluated separately and applied as an analysis model. This evaluation was important to avoid creating any statistical problems by mitigating possible failures resulting from a partial analysis, defined in Step 3 of Section 5.4. Table 10 evaluated the descriptive statistics data collected from the tasks grouped in a single environment.

Tasks	Wilcoxon		paired t-test		
	p–value	W	p–value	DF	t
All	0.001	49.15	0.022	31	2.395
Task 1	0.218	17.5	0.103	31	1.678
Task 2	0.312	12	0.183	31	1.359
Task 3	0.040	40.5	0.018	31	2.489
Task 4	0.164	35	0.096	31	1.716
Task 5	0.062	15	0.022	31	2.395

Legend: Ranks Sum *(RT)*, Degree of Freedom *(DF)*.

Table 10: Statistical Analysis of Experimental Error Rates.

Our results proved only task 3 presented a p–value lower than 0.05 ($p \leq 0.05$). However, in the evaluation data from the activities, the p–value was lower than 0.05 ($p \leq 0.05$), it is a significant result because there exists a difference between the inconsistency rates *ProMerge* about the traditional approach. The degree of freedom (DF) shows the direction in which the results are relevant. Finally, statistical tests were applied to evaluate the hypotheses and provide more support for the results.

Summary of RQ3. We conducted the Wilcoxon Signed-Rank Test to evaluate the mean values of samples and analyze the paired difference tests. The results showed that the *p–values* found in most tasks were higher than 0.05 ($p > 0.05$). However, $\mathbb{ER}$ presented a significant reduction of 30.76% compared with the traditional approach. These findings suggest that new features to *ProMerge* applying another combination of datasets into dependent variables can present promising results.

We collected qualitative data using two questionnaires: Characteristics and Impressions, including audio/video recordings and transcripts. The collected data helped to analyze several aspects and details from quantitative results and conclusions based on a chain of evidence and in the systematic alignment of quantitative and qualitative data.

6.4. Contributions

The main contributions of our study include: (1) Definition of an approach for proactive conflict resolution, where aspects

related to direct and higher-order conflicts were evaluated. (2) Conducting a controlled experiment with thirty-two developers to validate *ProMerge*, generating empirical knowledge and statistical results upon three hundred twenty valid scenarios. (3) Implementing concepts related to commit time and severity level are the main contributions *ProMerge*. The relationship between time (effort) and the number of commits conducted was analyzed using robust indicators were presented based on the information collected. Finally; (4) Severity level details the importance of a functionality or method to a software project and compares it with other functionalities.

6.5. Study Limitations

We recognize some limitations in analysis and evaluation results, particularly activities involving new controlled experiment executions or new approaches. This study addressed the effort applied to post-commit conflict resolution and proactivity approaches to detect and resolve conflicts. Finally, it is possible to classify the main study weaknesses and present some alternatives for research.

Proposed Approach. We developed a robust and reliable evaluation environment. However, it was projected to evaluate situations similar to software industry environments. Potential threats can be highlighted, mainly regarding data input. A componentized architecture eliminates possible threats to validate the proposed model and the data collected and evaluated. Therefore, it is necessary to execute other evaluations of conflicting scenarios to apply the corrections, defined as atypical. Even after these precautions, we can state that the approach is not exempt from scenarios that were not evaluated or considered.

Controlled Experiment. We investigated the environment variables that could influence any experiment involving source code merge. First, we need to increase the number of participants, segmented by knowledge range or technical experience. Thirty-two developers were selected, which is a robust and reliable sample for the controlled experiment. This mitigates any bias types in the experiment replication for future evaluations. Another important aspect involves the evaluation results collected from experiments applied with *ProMerge*. We used Wilcoxon, McNemar, and Paired t-test statistical analyses ensuring integrity and data validation, expanding the scope indicators produced, defining new evaluation models, and obtaining more assertive results.

7. Conclusions and Future Work

We presented *ProMerge* an approach for proactive source code conflict detection using context-sensitive information, generated from analyses of source code fragments modified or accommodated, helping the developers to gain a comprehensive understanding of the conflict resolution. For evaluation, we applied a controlled experiment and the results were analyzed to validate the usability of the approach developed.

Therefore, we executed a controlled experiment using thirty-two participants. Each performed ten activities divided into two scenarios. The first scenario used *ProMerge* and the second did not use any resource exception of Eclipse [21] resources natives, totaling three hundred and twenty robust evaluation scenarios. The following dependent variables were evaluated: Effort, Correctness, and Error Rate are defined in Table 4.

In each controlled experiment scenario, five similar tasks were applied with the same level of complexity and evaluation criteria. The collected results were carefully analyzed to generate key metrics and insights. Supported by statistical tests, *ProMerge* helped developers to correctly understand the conflicts arising from the development activities, reducing the effort applied to the traditional approach. The results obtained after applying the characteristics and impressions questionnaires indicated that the usability of *ProMerge* is possible, showing a user-friendly interface that is easy to interpret by the developers participating in the experiment.

The committing time definition is represented by the date and time intervals between the commits during the execution tasks. This important aspect can be fully audited representing a productivity indicator and quality of the execution tasks. We observed that the regularity of commits by task was one per task, in a few cases three or more. Commits applied in the repository update the context-sensitive information, helping the developers to evaluate the quality of tasks. Another important aspect is the evaluation severity level, which represents the importance level of functionality in the environment as detailed in Table 2. Finally, we evidenced three new research opportunities for future research as detailed:

- **Proposed Approach.** Client/server technology allows the developed components to be extended and improved with new and important functionalities, possibiliting migrations to a more robust and current architecture by integrating with microservices or external APIs. An important aspect is enabling source code to merge with other repository types including Github [26] and TFS [52] or other source code repositories used by the software industry or academic community.
- **Controlled Experiment.** We evidenced other experiments were realized with a quantity higher than our thirty-two participants. The participants were classified into five professional segments according to experience levels and technical knowledge. Initially, ten tasks were proposed with a complexity level between medium and low divided among the two existing scenarios, with five for each scenario. The first scenario presented a related task to the second scenario, similar to the same complexity evaluation criteria level. The average execution time for experiments was fifty-two minutes per participant. Therefore, as a future improvement, it is proposed to carry out evaluations with a greater quantity of activities and with a higher complexity level possibiliting executing complex and robust validations.
- **Psychophysiological Data (Biometric).** We detected a lack of studies that evaluate data collected using biometric equipment. EEG (Electroencephalogram) will be applied

to collect information on electrical brain activity, making it possible to measure the level of stress and anxiety of developers between the merge of modified source code fragments. Applying other research to execute controlled experiments can highlight the main reasons and causes of an incorrect merge. Therefore, collecting this information makes it possible to mitigate these problems increasing the productivity and assertiveness of developers.

Acknowledgement

This work was supported in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), under Grant No. 314248/2021-8 and Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul (FAPERGS), under Grant No. 21/2551-0002051-2. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 88887.897203/2023-00.

References

[1] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. 2018. Understanding Semi-structured merge conflict characteristics in open-source Java projects (journal-first abstract) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 955. https://doi.org/10.1145/3238147.3241983

[2] Nader Ale Ebrahim, Shamsuddin Ahmed, and Zahari Taha. 2009. Virtual teams: A literature review. *Australian journal of basic and applied sciences* 3, 3 (2009), 2653–2669.

[3] Hayward P Andres. 2002. A comparison of face-to-face and virtual software development teams. *Team Performance Management: An International Journal* (2002).

[4] 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. https://doi.org/10.1145/2351676.2351694

[5] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured merge: rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/2025113.2025141

[6] Sven Apel, Jörg Liebig, Christian Lengauer, Christian Kästner, and William Cook. 2010. Semistructured Merge in Revision Control Systems.

[7] Daniel Bruno Armino. 2016. *PACCS uma ferramenta para detecção proativa e resolução colaborativa de conflitos de código fonte*. Mestrado em Computação Aplicada. Universidade do Vale do Rio dos Sinos, São Leopoldo.

[8] Jae Young Bang and Nenad Medvidovic. 2015. Proactive detection of higher-order software design conflicts. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*. IEEE, 155–164.

[9] Rodrigo Brito and Marco Tulio Valente. 2021. RAID: Tool Support for Refactoring-Aware Code Reviews. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 265–275. https://doi.org/10.1109/ICPC52881.2021.00033

[10] Carlos Carbonera, Kleinner Farias, César Graeff, and Robson Silva. 2023. Software Merge: A Two-Decade Systematic Mapping Study. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. 99–108.

[11] Carbonera, Carlos E. and Farias, Kleinner. (2024). ProMerge: Proactive Conflict Detection Using Context-Sensitive Informations. https://doi.org/8t2mry5sdc.1 Mendeley Data Repository.

[12] Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. 2015. Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–10. https://doi.org/10.1109/ESEM.2015.7321191

[13] C. Costa, J. Figueiredo, G.L.M. Ghiotto, and L. Murta. 2014. Characterizing the Problem of Developers' Assignment for Merging Branches. *International Journal of Software Engineering and Knowledge Engineering* 24 (12 2014), 1489–1508. https://doi.org/10.1142/S0218194014400166

[14] Catarina Costa, Jair Figueiredo, Anita Sarma, and Leonardo Murta. 2016. TIPMerge: recommending developers for merging branches. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 998–1002. https://doi.org/10.1145/2950290.2983936

[15] Prasun Dewan and Rajesh Hegde. 2007. Semi-synchronous conflict detection and resolution in asynchronous software development. In *10th European Conference on Computer-Supported Cooperative Work*. Springer, 159–178.

[16] Prasun Dewan and Rajesh Hegde. 2007. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. *ECSCW 2007 - Proceedings of the 10th European Conference on Computer Supported Cooperative Work*, 159–178. https://doi.org/10.1007/978-1-84800-031-5_9

[17] Martín Dias, Guillermo Polito, Damien Cassou, and Stéphane Ducasse. 2015. DeltaImpactFinder: Assessing Semantic Merge Conflicts with Dependency Analysis. In *Proceedings of the International Workshop on Smalltalk Technologies* (Brescia, Italy) *(IWST '15)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/2811237.2811299

[18] Martín Dias, Guillermo Polito, Damien Cassou, and Stéphane Ducasse. 2015. DeltaImpactFinder: Assessing Semantic Merge Conflicts with Dependency Analysis. In *Proceedings of the International Workshop on Smalltalk Technologies* (Brescia, Italy) *(IWST '15)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/2811237.2811299

[19] Kevin Dullemond, Ben van Gameren, and Rini van Solingen. 2014. Collaboration spaces for virtual software teams. *IEEE Software* 31, 6 (2014), 47–53.

[20] Riya Dutta, Diego Elias Costa, Emad Shihab, and Tanja Tajmel. 2023. Diversity awareness in software engineering participant research. In *(ICSE-SEIS)*. IEEE, 120–131.

[21] Eclipse. [n. d.]. *Eclipse , https://www.eclipse.org/, accessed: 08.2024*.

[22] Davide Falessi et al. 2018. Empirical software engineering experts on the use of students and professionals in experiments. *ESE* 23 (2018), 452–489.

[23] Kleinner Farias, Alessandro Garcia, Jon Whittle, Christina von Flach Garcia Chavez, and Carlos Lucena. 2015. Evaluating the effort of composing design models: a controlled experiment. *Software & Systems Modeling* 14 (2015), 1349–1365.

[24] Ana M Fernández-Sáez, Marcela Genero, and Michel RV Chaudron. 2013. Empirical studies concerning the maintenance of UML diagrams and their use in the maintenance of code: A systematic mapping study. *Information and Software Technology* 55, 7 (2013), 1119–1142.

[25] Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and André 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, 8 (2020), 892–915. https://doi.org/10.1109/TSE.2018.2871083

[26] GitHub. 2024. *GitHub, https://github.com/, accessed: 08.2024*.

[27] Ray Hyman. 1982. Quasi-experimentation: Design and analysis issues for field settings (book). *Journal of Personality Assessment* 46, 1 (1982), 96–97.

[28] Java. [n. d.]. *Java, https://www.java.com/, accessed: 08.2024*.

[29] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: proactive conflict minimization through optimized task scheduling. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 732–741.

[30] Gunnar Kudrjavets, Aditya Kumar, Nachiappan Nagappan, and Ayushi

Rastogi. 2022. Mining Code Review Data to Understand Waiting Times Between Acceptance and Merging: An Empirical Analysis. *arXiv preprint arXiv:2203.05048* (2022).

[31] Simon Larsén, Jean-Rémy Falleri, Benoit Baudry, and Martin Monperrus. 2023. Spork: Structured Merge for Java With Formatting Preservation. *IEEE Transactions on Software Engineering* 49, 1 (2023), 64–83. https://doi.org/10.1109/TSE.2022.3143766

[32] Valentina Lenarduzzi, Oscar Dieste, Davide Fucci, and Sira Vegas. 2021. Towards a methodology for participant selection in software engineering experiments: A vision of the future. In *(ESEM)*. 1–6.

[33] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. 2019. Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 151–162. https://doi.org/10.1109/SANER.2019.8668012

[34] Nikola Marangunić and Andrina Granić. 2015. Technology Acceptance Model: a Literature Review from 1986 to 2013. *Universal Access in the Information Society* 14 (2015), 81–95.

[35] S. McKee, N. Nelson, A. Sarma, and D. Dig. 2017. Software Practitioner Perspectives on Merge Conflicts and Resolutions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 467–478. https://doi.org/10.1109/ICSME.2017.53

[36] José William Menezes, Bruno Trindade, João Felipe Pimentel, Alexandre Plastino, Leonardo Murta, and Catarina Costa. 2021. Attributes that may raise the occurrence of merge conflicts. *Journal of Software Engineering Research and Development* 9, 1 (Oct. 2021), 1–14. https://doi.org/10.5753/jserd.2021.1911

[37] Tom Mens. 2002. A state-of-the-art survey on software merging. *IEEE transactions on software engineering* 28, 5 (2002), 449–462.

[38] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. 2019. The life-cycle of merge conflicts: processes, barriers, and strategies. *Empirical Softw. Engg.* 24, 5 (oct 2019), 2863–2906. https://doi.org/10.1007/s10664-018-9674-x

[39] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu Lahiri, and Mike Kaufman. 2021. Can Program Synthesis be Used to Learn Merge Conflict Resolutions? An Empirical Analysis. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 785–796. https://doi.org/10.1109/ICSE43902.2021.00077

[40] Fabrizio Pastore, Leonardo Mariani, and Daniela Micucci. 2017. BDCI: behavioral driven conflict identification. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 570–581. https://doi.org/10.1145/3106237.3106296

[41] PostgreSQL. [n. d.]. *PostgreSQL, https://www.postgresql.org/, accessed: 08.2024*.

[42] PostgreSQLJDBC. [n. d.]. *PostgreSQL JDBC Driver, https://jdbc.postgresql.org/, accessed: 08.2024*.

[43] Austen Rainer and Claes Wohlin. 2022. Recruiting credible participants for field studies in software engineering research. *IST* 151 (2022), 107002.

[44] Thomas Reiter, Kerstin Altmanninger, Alexander Bergmayr, Wieland Schwinger, and Gabriele Kotsis. 2007. Models in conflict-detection of semantic conflicts in model-based development. In *Proceedings of International Workshop on Model-Driven Enterprise Information Systems@ ICEIS*, Vol. 7. 29–40.

[45] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are students representatives of professionals in software engineering experiments?. In *(ICSE)*, Vol. 1. 666–676.

[46] Anita Sarma, David F. Redmiles, and André van der Hoek. 2012. Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes. *IEEE Transactions on Software Engineering* 38, 4 (2012), 889–908. https://doi.org/10.1109/TSE.2011.64

[47] Georg Seibt, Florian Heck, Guilherme Cavalcanti, Paulo Borba, and Sven Apel. 2022. Leveraging Structure in Software Merge: An Empirical Study. *IEEE Transactions on Software Engineering* 48, 11 (2022), 4590–4610. https://doi.org/10.1109/TSE.2021.3123143

[48] Bowen Shen and Na Meng. 2024. ConflictBench: A benchmark to evaluate software merge tools. *Journal of Systems and Software* 214 (2024), 112084.

[49] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. 2019. IntelliMerge: a refactoring-aware software merging technique. *Proceedings ACM Program Language* 3, OOPSLA, Article 170 (oct 2019), 28 pages. https://doi.org/10.1145/3360596

[50] Leuson Da Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and João Moisakis. 2020. Detecting Semantic Conflicts via Automated Behavior Change Detection. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 174–184. https://doi.org/10.1109/ICSME46990.2020.00026

[51] Chungha Sung, Shuvendu K Lahiri, Mike Kaufman, Pallavi Choudhury, and Chao Wang. 2020. Towards understanding and fixing upstream merge induced conflicts in divergent forks: An industrial case study. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 172–181.

[52] TFS. [n. d.]. *Team Foundation Server, https://visualstudio.microsoft.com/downloads/, accessed: 08.2024*.

[53] Gustavo Vale, Claus Hunsen, Eduardo Figueiredo, and Sven Apel. 2022. Challenges of Resolving Merge Conflicts: A Mining and Survey Study. *IEEE Transactions on Software Engineering* 48, 12 (2022), 4964–4985. https://doi.org/10.1109/TSE.2021.3130098

[54] Heather Wild, Aki-Juhani Kyröläinen, and Victor Kuperman. 2022. How representative are student convenience samples? A study of literacy and numeracy skills in 32 countries. *Plos one* 17, 7 (2022), e0271191.

[55] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[56] Xiuheng Wu, Chenguang Zhu, and Yi Li. 2021. DIFFBASE: a differential factbase for effective software evolution management *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 503–515. https://doi.org/10.1145/3468264.3468605

[57] Ryohei Yuzuki, Hideaki Hata, and Kenichi Matsumoto. 2015. How we resolve conflict: an empirical study of method-level conflict resolution. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*. 21–24. https://doi.org/10.1109/SWAN.2015.7070484

[58] Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K. Lahiri. 2022. Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/3533767.3534396