



Pós-Graduação em Ciência da Computação

Guilherme José de Carvalho Cavalcanti

Should We Replace Our Merge Tools?



Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
<http://cin.ufpe.br/~posgraduacao>

Recife
2019

Guilherme José de Carvalho Cavalcanti

Should We Replace Our Merge Tools?

Tese de Doutorado apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Doutor em Ciência da Computação.

Área de Concentração: Engenharia de Software

Orientador: Paulo Henrique Monteiro Borba

Recife
2019

Catálogo na fonte
Bibliotecária Mariana de Souza Alves CRB4-2106

C376s Cavalcanti, Guilherme José de Carvalho
Should We Replace Our Merge Tools? – 2019.
92f.: il., fig., tab.

Orientador: Paulo Henrique Monteiro Borba
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn,
Ciência da computação. Recife, 2019.
Inclui referências.

1. Engenharia de Software. 2. Integração de Software. 3.
Desenvolvimento colaborativo. 4. Sistema de controle de versões.
I. Borba, Paulo Henrique Monteiro (orientador). II. Título.

005.1

CDD (22. ed.)

UFPE-MEI 2019-150

Guilherme José de Carvalho Cavalcanti

“Should We Replace Our Merge Tools?”

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

Aprovado em: 02/10/2019.

Orientador: Prof. Dr. Paulo Henrique Monteiro Borba

BANCA EXAMINADORA

Prof. Dr. Marcelo Bezerra d’Amorim
Centro de Informática/UFPE

Prof. Dr. Leopoldo Motta Teixeira
Centro de Informática/UFPE

Prof. Dr. Breno Alexandro Ferreira de Miranda
Centro de Informática/UFPE

Prof. Dr. Leonardo Gresta Paulino Murta
Instituto de Computação / UFF

Prof. Dr. Sven Apel
Saarland Informatics Campus / Saarland University

I dedicate this work to everyone who gave me the necessary support to get here.

ACKNOWLEDGEMENTS

A minha família, amigos e namorada pelo apoio nos momentos em que necessitei e pela compreensão nos momentos em que estive ausente. Ao professor Paulo Borba pela inquestionável competência com a qual orienta seus alunos e pelas oportunidades oferecidas. Aos membros do SPG pelas experiências e conhecimentos compartilhados, bem como os bons e divertidos momentos vividos. Aos pesquisadores Paola Accioly, Sven Apel e Georg Seibt pela fundamental colaboração nos artigos que foram frutos deste trabalho. Aos membros de minha banca Marcelo d'Amorim, Leopoldo Teixeira, Breno Miranda, Leonardo Murta e Sven Apel pela paciência, disponibilidade e comentários que certamente contribuirão para este e trabalhos futuros. Aos competentes profissionais do ambulatório de Doenças Gastrointestinais do Hospital das Clínicas de Pernambuco por permitirem que eu fosse capaz de conduzir este trabalho de uma forma saudável. Finalmente, agradeço a FACEPE por financiar minha pesquisa e ao Centro de Informática da Universidade Federal de Pernambuco por todo recurso que me ofereceu.

ABSTRACT

Merge conflicts often occur when developers concurrently change the same code artifacts. While state of practice unstructured merge tools (e.g git merge) try to automatically resolve merge conflicts based on textual similarity, semistructured and structured merge tools try to go further by exploiting the syntactic structure and semantics of the involved artifacts. Previous studies compare semistructured and structured merge with unstructured merge concerning the number of reported conflicts, showing, for most projects and merge situations, a reduction in favor of semistructured and structured merge. This evidence, however, might not be sufficient to justify industrial adoption of advanced merge strategies such as semistructured and structured merge. The problem is that previous studies do not investigate whether the observed reduction on the number of reported conflicts actually leads to integration effort reduction (Productivity) without negative impact on the correctness of the merging process (Quality). Besides, it is unknown how semistructured merge compares with structured merge. So, to decide whether we should replace our state of practice unstructured merge tools, we need to compare these merge strategies and understand their differences. We then first compare unstructured and semistructured merge. Our results and complementary analysis indicate that the number of false positives is significantly reduced when using semistructured merge when compared to unstructured merge. However, we find no evidence that semistructured merge leads to fewer false negatives. Driven by these findings, we implement an improved semistructured merge tool that further combines both approaches to reduce the false positives and false negatives of semistructured merge. Semistructured merge has shown significant advantages over unstructured merge, especially as implemented by our improved tool. However, before deciding to replace unstructured tools by semistructured merge, we need to investigate whether structured merge is a better alternative than semistructured merge. So, we compare semistructured and structured merge. Our results show that semistructured and structured merge differ on 24% of the scenarios with conflicts. Semistructured merge reports more false positives, whereas structured merge has more false negatives. Finally, we observe that adapting a semistructured merge tool to resolve a particular kind of conflict makes semistructured and structured merge even closer. Overall, our findings suggests that semistructured merge is a better replacement of unstructured tools for conservative developers, having significant gains with a closer behavior to unstructured tools than structured merge. Besides that, practitioners might be reluctant to adopt structured merge because of the observed performance overhead and its tendency to false negatives. So, when choosing between semistructured and structured merge, semistructured merge would be a better match for developers that are not overly concerned with semistructured extra false positives.

Keywords: Software merging. Collaborative development. Version control systems.

RESUMO

Conflitos de integração frequentemente ocorrem quando os desenvolvedores alteram simultaneamente os mesmos artefatos de código. Enquanto que as ferramentas de integração não-estruturadas, que representam o estado da prática (por exemplo, git merge), tentam resolver conflitos automaticamente, baseadas em semelhança textual, as ferramentas de integração semiestruturada e estruturada tentam ir além explorando a estrutura sintática e a semântica dos artefatos envolvidos. Estudos anteriores comparam as estratégias semiestruturada e estruturada com a não-estruturada em relação ao número de conflitos reportados, mostrando, para a maioria dos projetos e situações de integração, uma redução a favor das estratégias semiestruturada e estruturada. O problema desses estudos anteriores é que eles não investigam se a redução observada no número de conflitos realmente leva à redução do esforço de integração (Produtividade) sem impacto negativo na correção do processo de integração (Qualidade). Além disso, não se sabe como a estratégia semiestruturada se compara com a estruturada. Para ajudar os desenvolvedores a decidir que tipo de ferramenta usar e entender melhor suas diferenças, conduzimos dois estudos empíricos. No primeiro, comparamos a integração não-estruturada e a semiestruturada. Nossos resultados e análises complementares indicam que o número de falsos positivos é significativamente reduzido ao usar a estratégia semiestruturada quando comparado à não-estruturada. No entanto, nós não encontramos evidências de que a integração semiestruturada leva a menos falsos negativos. Motivados por essas descobertas, implementamos uma ferramenta de integração semiestruturada que combina ainda mais as duas estratégias para reduzir os falsos positivos e os falsos negativos da integração semiestruturada. No segundo estudo, comparamos as integrações semiestruturada e estruturada. Nossos resultados mostram que as integrações semiestruturada e estruturada diferem em 24% dos cenários com conflitos. A estratégia semiestruturada reporta mais falsos positivos, enquanto a estruturada tem mais falsos negativos. Finalmente, observamos que adaptar uma ferramenta semiestruturada para resolver um determinado tipo de conflito torna-a ainda mais próxima à integração estruturada. No geral, nossas descobertas sugerem que a estratégia semiestruturada é uma melhor alternativa à não-estruturada para desenvolvedores conservadores, possuindo ganhos significativos com um comportamento mais próximo ao das ferramentas não-estruturadas do que ferramentas estruturadas. Além disso, os usuários podem relutar em adotar a estratégia estruturada por causa do seu impacto no desempenho e sua tendência a falsos negativos. Dessa forma, a estratégia semiestruturada seria mais apropriada para desenvolvedores que não são muito preocupados com seus falsos positivos extras.

Palavras-chaves: Integração de Software. Desenvolvimento colaborativo. Sistema de controle de versões.

LIST OF FIGURES

Figure 1 – Centralized version control paradigm.	17
Figure 2 – Distributed version control paradigm.	17
Figure 3 – Superimposition of Stack class declarations, based on a example of Apel e Lengauer (2008). Bold lines indicate added nodes.	21
Figure 4 – Sets of conflicts reported by unstructured and semistructured merge. Notation explained in the text.	25
Figure 5 – Unstructured merge additional false positive: <i>ordering conflict</i>	26
Figure 6 – Semistructured merge additional false positive: <i>renaming conflict</i>	27
Figure 7 – Unstructured merge additional false negative: <i>duplicated declaration error</i>	28
Figure 8 – Semistructured merge additional false negative: <i>type ambiguity errors</i>	29
Figure 9 – Semistructured merge additional false negative: <i>initialization blocks</i>	30
Figure 10 – Semistructured merge additional false negative: <i>new element referencing edited one</i>	30
Figure 11 – Intercepting the <i>FSTMerge</i> tool to find <i>renaming</i> false positives (<i>aFP(SS)</i>) candidates.	35
Figure 12 – Box plots describing the percentage, per project, of additional false positives in terms of merge scenarios and conflicts. Unstructured merge in red, semistructured in blue.	39
Figure 13 – Crosscutting ordering conflicts.	40
Figure 14 – Box plots describing the percentage, per project, of the additional false negatives in terms of merge scenarios and conflicts. Unstructured merge in red, semistructured in blue.	42
Figure 15 – Unstructured merge conflicts classified as semistructured merge addi- tional false negatives.	43
Figure 16 – Observed new element referencing edited one.	44
Figure 17 – Merging with Semistructured and Structured Merge (False Positive).	57
Figure 18 – Merging with Semistructured and Structured Merge (True Positive).	58
Figure 19 – Equivalent conflicts with different granularity.	63
Figure 20 – Building and testing merge commits. A green check mark indicates no conflict with one strategy, a red cross indicates conflict with the other strategy.	64
Figure 21 – Semistructured merge conflict from project GLACIERUPLOADER	68
Figure 22 – Structured merge conflict from project MVVMFX	69
Figure 23 – Structured merge conflict from project EDITORCONFIG-NETBEANS	70
Figure 24 – Structured merge conflict from project RESTY-GWT	72

LIST OF TABLES

Table 1	– Comparing conflict numbers of unstructured, semistructured, and improved tools.	49
Table 2	– Comparing false positives and false negatives numbers of the improved and original tools with unstructured merge. Arrows indicate whether the number is underestimated (\uparrow , meaning the numbers should be bigger in practice) or overestimated (\downarrow).	50
Table 3	– Numbers for merge scenarios with false positives and false negatives. . .	73

CONTENTS

1	INTRODUCTION	12
2	BACKGROUND	16
2.1	VERSION CONTROL SYSTEMS	16
2.2	MERGING SOFTWARE ARTEFACTS	18
2.3	COMPARING MERGE STRATEGIES	22
2.4	CONTINUOUS INTEGRATION	22
3	EVALUATING AND IMPROVING SEMISTRUCTURED MERGE .	24
3.1	HEURISTICS FOR FALSE POSITIVES AND FALSE NEGATIVES	24
3.1.1	Additional False Positives of Unstructured Merge	25
3.1.2	Additional False Positives of Semistructured Merge	26
3.1.3	Additional False Negatives of Unstructured Merge	27
3.1.4	Additional False Negatives of Semistructured Merge	28
3.1.5	Common False Positives and False Negatives	30
3.2	EMPIRICAL EVALUATION	31
3.2.1	Mining Step	32
3.2.2	Execution and Analysis Steps	33
3.2.2.1	Computing the Underestimated Number of Additional False Positives of Unstructured Merge— $aFP(UN)$	33
3.2.2.2	Computing the Overestimated Number of Additional False Positives of Semistructured Merge— $aFP(SS)$	34
3.2.2.3	Computing the Underestimated Number of Additional False Negatives of Unstructured Merge— $aFN(UN)$	34
3.2.2.4	Computing the Overestimated Number of Additional False Negatives of Semistructured Merge— $aFN(SS)$	35
3.3	RESULTS AND DISCUSSION	36
3.3.1	When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort by reporting fewer false posi- tives?	37
3.3.1.1	Additional False Positives of Semistructured Merge are Easier to Analyze and Resolve	38
3.3.2	When compared to unstructured merge, does semistructured merge compromise integration correctness by having more false negatives? .	41
3.3.2.1	Additional False Negatives of Semistructured Merge are Harder To Detect and Resolve	44

3.4	IMPROVING SEMISTRUCTURED MERGE	45
3.4.1	Improvements	45
3.4.1.1	Handling Renaming Conflicts	45
3.4.1.2	Handling Type Ambiguity Errors	46
3.4.1.3	Handling Matching of Initialization Blocks	47
3.4.1.4	Handling Accidental False Negatives	48
3.4.2	Usability	48
3.4.3	Gains	49
3.4.4	Performance Evaluation	51
3.5	THREATS TO VALIDITY	52
4	THE IMPACT OF STRUCTURE ON SOFTWARE MERGING: SEMISTRUC- TURED VERSUS STRUCTURED MERGE	55
4.1	SEMISTRUCTURED AND STRUCTURED MERGE	56
4.2	RESEARCH QUESTIONS	58
4.3	EMPIRICAL EVALUATION	61
4.3.1	Mining Step	61
4.3.2	Execution Step	62
4.4	RESULTS	65
4.4.1	How many conflicts arise when using semistructured and structured merge?	66
4.4.2	How often do semistructured and structured merge differ with re- spect to the occurrence of conflicts?	66
4.4.3	Why do semistructured and structured merge differ?	67
4.4.4	Which of the two strategies reports fewer false positives?	70
4.4.5	Which of the two strategies has fewer false negatives?	71
4.4.6	Does ignoring conflicts caused by changes to consecutive lines make the two strategies more similar?	74
4.5	DISCUSSION	74
4.6	THREATS TO VALIDITY	76
5	CONCLUSIONS	77
5.1	CONTRIBUTIONS	78
5.2	FUTURE WORK	79
5.3	RELATED WORK	81
5.3.1	Strategies and Tools Assisting Conflict Detection and Resolution . .	81
5.3.2	Evidence on Conflicts and Their Impact	83
	REFERENCES	87

1 INTRODUCTION

In a collaborative software development environment, developers often independently perform tasks, using individual copies of project files. So, when integrating contributions from each task, one might have to deal with conflicting changes, and dedicate substantial effort to resolve conflicts. These conflicts might be detected during merging, building, and testing, impairing development productivity, since understanding and resolving conflicts often is a demanding and tedious task (ZIMMERMANN, 2007; BRUN et al., 2011; BIRD; ZIMMERMANN, 2012; KASI; SARMA, 2013). Perhaps worse, conflicts might not be detected during integration and testing, escaping to production releases and compromising correctness.

To better detect and resolve code integration conflicts, researchers have proposed tools that use different strategies to decrease integration effort and improve integration correctness. For merging software code artifacts, unstructured, line-based merge tools are the state of practice, relying on purely textual analysis to detect and resolve conflicts (MENS, 2002; ZIMMERMANN, 2007; KHANNA; KUNAL; PIERCE, 2007). Structured merge tools are programming language specific and go beyond simple textual analysis by exploring the underlying syntactic structure and static semantics when integrating programs (APEL; LESSENICH; LENGAUER, 2012). Semistructured merge tools attempt to hit a sweet spot between unstructured and structured merge by *partially* exploring the syntactic structure and static semantics of the artifacts involved (APEL et al., 2011; CAVALCANTI; BORBA; ACCIOLY, 2017). For program elements whose structure is not exploited, like method bodies in Java, semistructured merge tools simply apply the usual textual resolution adopted by unstructured merge.

Previous studies compare these merge strategies with respect to the number of reported conflicts, showing, for most but not all projects and merge situations, reduction in favor of semistructured (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015) and structured merge (APEL; LESSENICH; LENGAUER, 2012). For instance, in merge situations where semistructured merge reduces the number of reported conflicts, Apel et al. (2011) show an average reduction of 34% compared to unstructured merge. In a replication of this study, we find a larger average reduction of 62%, again in favor of semistructured merge (CAVALCANTI; ACCIOLY; BORBA, 2015). This reduction is mainly due to the automatic resolution of obvious unstructured merge false positives that are reported when, for example, developers add different and independent methods to the same file text area.

This evidence, however, is not enough to justify industrial adoption of semistructured or structured merge, and to convince practitioners to replace their merge tools. The problem is that previous studies do not investigate whether the observed reduction of reported conflicts actually leads to integration effort reduction (productivity) without negative

impact on the correctness of the merging process (quality). Although one might expect only accuracy benefits from the extra structure exploited by semistructured and structured merge, we have no guarantees that this is the case. This means that the observed reduction could have been obtained at the expense of missing actual conflicts between developers changes (false negatives). In that case, semistructured and structured merge users would be simply postponing conflict detection to other integration phases such as building and testing, or even letting more conflicts escape to users. Moreover, given that the set of conflicts reported by semistructured and structured merge in previous studies is often smaller but not a subset of the set reported by unstructured merge, they could even be introducing other kinds of false positives that might be harder to resolve than the ones they eliminate. Finally, it is unknown how semistructured merge compares with structured merge. Thus, if we want to move forward on the state of the practice on merge tools, and figure out whether we should replace them, it is important to have solid evidence and further knowledge about the differences among these merge strategies.

So, to help developers decide which kind of tool to use, and to better understand how merge tools could be improved, we first compare unstructured and semistructured merge, identifying false positives (conflicts incorrectly reported by one strategy but not by the other) and false negatives (conflicts correctly reported by one strategy but missed by the other) (CAVALCANTI; BORBA; ACCIOLY, 2017). In a sample of more than 30,000 merge scenarios from 50 projects, we found that the number of false positives is significantly reduced when using semistructured merge, and we found evidence that its false positives are easier to analyze and resolve than those reported by unstructured merge. On the other hand, we found no evidence that semistructured merge leads to fewer false negatives, and we argue that they are harder to detect and resolve than unstructured merge false negatives. Although our comparison process favors unstructured merge whenever we are not able to precisely classify a reported conflict, this last finding, and the associated lack of evidence in support of semistructured merge, might justify non adoption of semistructured merge in practice. Nevertheless, our findings shed light on how merge tools can be improved. So, we benefit from that and implement an improved semistructured merge tool that further combines both merge strategies to reduce the false positives and false negatives of semistructured merge. We found evidence that the improved tool, when compared to unstructured merge in our sample, reduces the number of reported conflicts by half, has no additional false positives, has at least 8% fewer false negatives, and is not prohibitively slower.

Although we found evidence that semistructured merge has significant advantages over unstructured merge, it is imperative to investigate how semistructured merge compares to structured merge before deciding which kind of tool to use. So, in a second empirical study, we assess how often semistructured and structured merge report different results, and we also identify false positives and false negatives (CAVALCANTI et al., 2019). Surprisingly, in

a sample of more than 40,000 merge scenarios from more than 500 projects, we found that the two strategies rarely differ for the scenarios in our sample. Considering only scenarios with conflicts, however, the tools differ in about 24% of the cases. A closer analysis reveals they differ when integrating changes that modified the same textual area in the body of a declaration, but the modifications involve different abstract syntax tree (AST) nodes in the structured merge representation of that body. They also differ when changes in the same AST node correspond to different text areas in the semistructured merge representation of the same declaration body. We also found that semistructured merge reports false positives in more merge scenarios (66) than structured merge (6), whereas structured merge has more scenarios with false negatives (45) than semistructured merge (8).

Overall, our findings suggest that semistructured merge is a better replacement of state of practice unstructured tools for conservative developers, having significant gains with a closer behavior to unstructured tools than structured merge. Besides that, practitioners might be reluctant to adopt structured merge because of the observed performance overhead and its tendency to false negatives. So, when choosing between semistructured and structured merge, semistructured merge would be a better match for developers that are not overly concerned with semistructured extra false positives. Finally, adapting a semistructured merge tool to report conflicts only when changes occur in the *same* lines (resolving conflicts caused by changes to *consecutive* lines) might be the way to achieve a sweet spot in the relation between structure and accuracy in non-semantic merge tools.

The remainder of this document is organized as follows:

- In Chapter 2, we present the essential concepts used throughout this work;
- In Chapter 3, we describe our first empirical study, comparing unstructured and semistructured merge with respect to the occurrence of false positives and false negatives. We then present our improved semistructured tool and how it compares to unstructured and original semistructured merge. This chapter is published in Cavalcanti, Borba e Accioly (2017), with the co-authoring of Paola Accioly and Paulo Borba. They both reviewed and guided this work. They helped in understating semistructured merge in detail too. Paola Accioly also helped with the elaboration of the mining scripts.
- In Chapter 4, we describe our second empirical study, assessing how often semistructured and structured merge report different results, and identifying false positives and false negatives between these two merge strategies. This chapter is published in Cavalcanti et al. (2019), in collaboration with Paulo Borba, Georg Seibt and Sven Apel. They reviewed and provided essential feedback and guidance. Georg Seibt and Sven Apel were fundamental in understanding structured merge in depth. Georg Seibt and Paulo Borba also helped with the extensive manual inspections necessary in this work.

- In Chapter 5, we present our concluding remarks, and future and related work.

2 BACKGROUND

In this chapter, we explain the main concepts used in this work. Initially, in Section 2.1, we discuss the fundamentals of version control systems (VCSs). In this context, we explain how software artifacts are merged in Section 2.2. Afterwards, we describe characteristics of the unstructured, semistructured, and structured merge tools, and define the criteria we adopt to compare these merge strategies in Section 2.3. Finally, we present the role of Continuous Integration in Section 2.4.

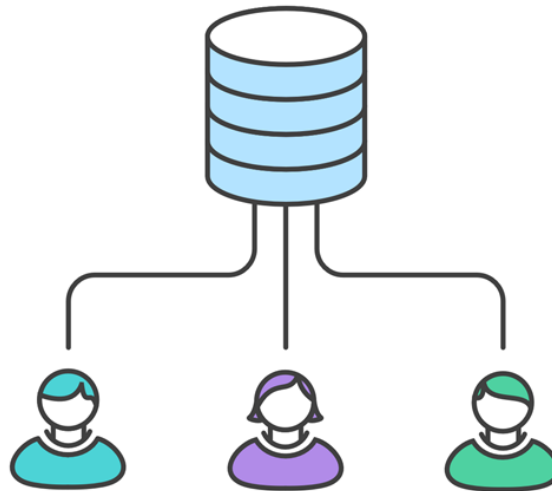
2.1 VERSION CONTROL SYSTEMS

Collaborative software development is only possible thanks to software configuration management (SCM) and the consequent use of VCSs. In particular, SCM manages the evolution of large and complex software systems (TICHY, 1988). It provides techniques and tools to assist developers in performing coordinated changes to software products. These techniques include version control mechanisms to deal with the evolution of a software product into many parallel versions and variants that need to be kept consistent and from which new versions may be derived via software merging (CONRADI; WESTFECHTEL, 1998). This became necessary when there were several developers working together in projects, and a standardized way of keeping track of the changes were needed. If there would be no control, the developers would overwrite each other's changes (ESTUBLIER et al., 2002). In practice, VCSs allow developers to download and modify files in their local working area, which is periodically synchronized with the repository that contains the main version of the files. How the repositories are disposed and how the developers access them determine the version control paradigm.

In a *Centralized Version Control System (CVCS)*, such as CVS (CVS, 2019) and Subversion (SUBVERSION, 2019), there is one central repository, which can accept code, and everyone synchronizes their work with it (see Figure 1). Relying on a client-server architecture, a number of developers are consumers of that repository and synchronize to that one place. This means that if two developers are working based on the same repository and both make changes, the first developer to send their changes can do so with no problems. The second developer must merge in the first one's work before sending changes, so as not to overwrite the first developer's changes.

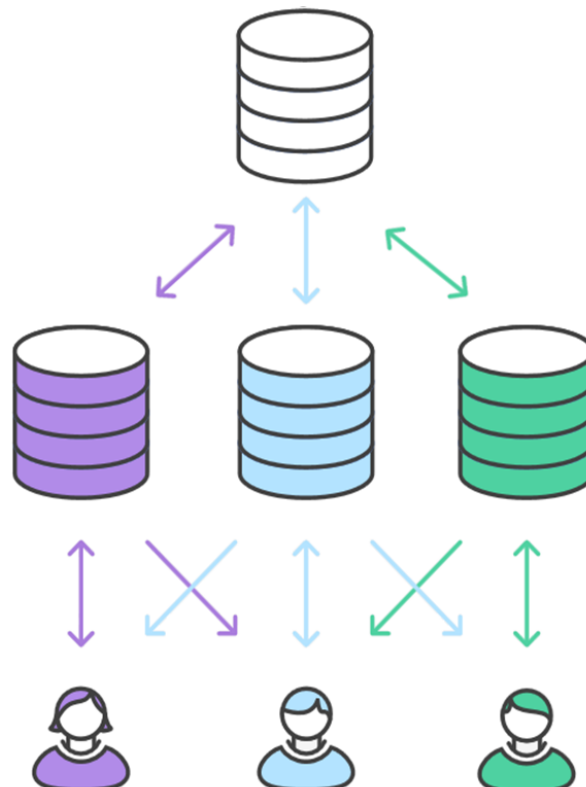
Conversely, a *Distributed Version Control System (DVCS)*, such as Mercurial (MERCURIAL, 2019) and Git (GIT, 2019), relies on a peer-to-peer architecture. The main idea is, instead of getting and sending data to a single server, each developer holds its own repository, including project data and history, and synchronizes on demand with repositories maintained by other developers (see Figure 2). Rigby et al. (1996) makes an interesting

Figure 1 – Centralized version control paradigm.



differentiation between CVCSs and DVCSs: *"With CVCSs changes flow up and down (and publicly) via a central repository. In contrast, DVCSs facilitate a style of collaboration in which work output can flow sideways (and privately) among collaborators, with no repository being inherently more important or central."* DVCSs have seen an increase in popularity relative to traditional CVCSs, mainly on the open source community (BIRD et al., 2009; BRINDESCU et al., 2014; GOUSIOS; PINZGER; DEURSEN, 2014).

Figure 2 – Distributed version control paradigm.



Regardless of the VCS paradigm, they allow the creation of parallel versions of a

software system. The challenge, however, is in figuring out how to merge them back into a single version (PERRY; SIY; VOTTA, 2001). While the VCS's synchronization protocol allows rapid parallel development, it also allows developers to make conflicting changes inadvertently.

2.2 MERGING SOFTWARE ARTEFACTS

To merge multiple source code artifacts, it is essential to compare them and extract the differences. For this purpose, a *two-way merge* attempts to merge two revisions directly by comparing two files without using any information from the VCS. Therefore, each difference between the two revisions leads to a conflict since it cannot decide whether only one of the revisions introduced a change to the code or both. It also cannot determine whether a certain program element has been created by one revision or has been deleted by the other one. In turn, with *three-way merge*, which is used in every practical VCS, the information in the common ancestor is also used during the merging process. As a consequence, *three-way merge* has more information to decide where a change came from and whether it creates a conflict or not (PERRY; SIY; VOTTA, 2001; O'SULLIVAN, 2009).

Conflicts appear not only as overlapping textual edits, but also as subsequent build and test failures (ZIMMERMANN, 2007; BRUN et al., 2011; aES; SILVA, 2012; KASI; SARMA, 2013). These conflicts emerge because developers are not aware of others' changes, and conflicts become more complex as changes grow without being integrated and as further developments are made. Some developers even do not merge as frequently as desirable because of difficult merges, and rush their tasks to avoid being the ones responsible for the merge (SOUZA; REDMILES; DOURISH, 2003). To prevent conflicts, tools using different strategies have also been proposed (BRUN et al., 2011; Sarma; Redmiles; van der Hoek, 2012; aES; SILVA, 2012; KASI; SARMA, 2013). However, it is not always possible to detect conflicts before code integration, and as a consequence, when the developers decide to merge their changes, one likely has to dedicate substantial effort to resolve the conflicts, often using some merge tool.

Although VCSs have evolved over the years, together with tools and practices to better support collaborative software development, merge tools have not evolved much. The state of the practice is still textual, line-based, *unstructured merge* based on the *diff3* algorithm (MENS, 2002). When merging files, unstructured merge tools typically compare, line by line, two modified files in relation to their common ancestor (the version from which they have been derived) and detect sets of differing lines (chunks), during the three-way merging process. For each chunk, such merge tools check whether the three revisions have a common text area that separates chunks' content. If such separator is not found, the tool reports a conflict (KHANNA; KUNAL; PIERCE, 2007). The benefits of a textual merge are its generality and its performance. It can be applied to all non-binary files, even to very large ones, so there is only one tool needed regardless of which programming languages are used

within a project. If the amount of changes is very small in comparison to the input files, or if there are no changes at all, this strategy is very effective. However, as this kind of merge does not utilize knowledge about the structure of the input documents and the syntax of respective languages, it might report too many unnecessary conflicts, and it might produce syntactically incorrect output (HORWITZ; PRINS; REPS, 1989; BUFFENBARGER, 1995). For instance, (re)formatting of code produces unnecessary conflicts in Java when unstructured merge is used, because the position of brackets and the indentation style (e.g., tabs or spaces) might be different between the merged versions.

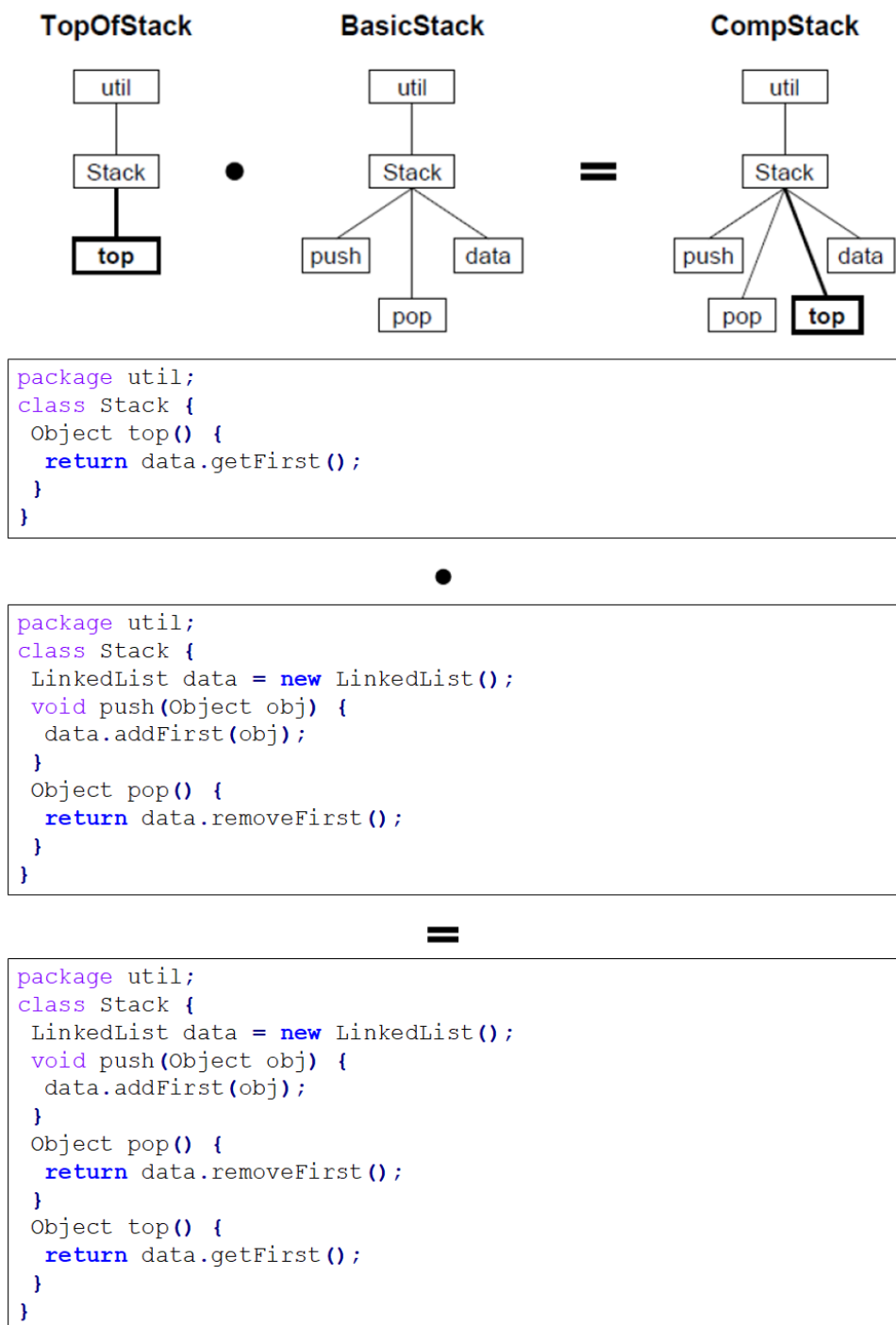
Conversely, tools that leverage and take advantage of information on the syntax and semantics of the programs involved in the merging process to resolve as many conflicts as possible have been proposed (WESTFECHTEL, 1991; GRASS, 1992; BUFFENBARGER, 1995; APIWATTANAPONG; ORSO; HARROLD, 2007; APEL; LESSENICH; LENGAUER, 2012; LESSENICH et al., 2017; ZHU; HE; YU, 2019). *Structured merge* exploits language-specific knowledge and attempts to compare and merge software artifacts more precisely than unstructured merge. The underlying data structure for structured merge is usually ASTs, which requires the merge tool to parse the programs in advance and generate the corresponding trees (MENS, 2002). Comparing two programs for structured merge means traversing trees and finding the differing nodes (APEL; LESSENICH; LENGAUER, 2012). As the merge is applied to the trees after parsing the program, code formatting is no longer relevant. When the merging process has finished, the output document is generated by pretty-printing the AST. However, this strategy has disadvantages as well. Structured merge is much slower than textual merges, mostly due to the complexity of their tree algorithms. It requires the input files to be syntactically correct, otherwise the parser is not able to build the AST. Furthermore, it is restricted to certain file types since it uses syntactic knowledge of the programming languages in which the programs to be merged are written. So, supporting additional programming languages might require a lot of work. Changes in the specification of a language might break the correctness of a merge tool, which then must be subsequently adapted to the specification (e.g. from Java 6 to 8). Also, by default, the original code formatting done by the developers is lost after the structured merge.

Finally, in an attempt to hit a sweet spot between unstructured and structured merge, *semistructured merge* (APEL et al., 2011) exploits only part of the language syntax and semantics. Such tools represent part of the program elements as trees, and rely on information about how nodes of certain types (methods, classes, etc.) should be merged. Such trees include some but not all syntactic structural information. Concerning Java, for example, classes, methods and fields appear as nodes in the tree, whereas statements and expressions in method (or constructor) declarations appear as plain text in tree leaves.¹ To merge leaves, semistructured merge simply invokes unstructured merge. The semistructured

¹ From now on, we use method declarations to refer both to method and constructor declarations.

merge algorithm is itself implemented via *superimposition* of trees, working recursively and beginning at the root nodes (APEL; LENGAUER, 2008). The principle of superimposition is illustrated in Figure 3. The example illustrates the process of trees superimposition with the corresponding code. The original version (BasicStack) is composed with a derived version (TopOfStack). The result is a new version (CompStack), that is represented by the superimposition of the previous trees. The nodes `util` and `Stack` are composed with their counterparts, and their subtrees (i.e., their methods and fields) are merged in turn. Note that nodes are matched and merged based on structural and nominal similarities. This approach implies that method bodies only have to be merged if the signature of two methods is identical. In this case, unstructured merge is launched to merge the method bodies.

Figure 3 – Superimposition of Stack class declarations, based on a example of Apel e Lengauer (2008). Bold lines indicate added nodes.



2.3 COMPARING MERGE STRATEGIES

To compare these merge strategies, and decide if we should replace our state of practice unstructured tools by semistructured or structured merge, we could simply measure how often they are able to merge contributions. The preference would be for the strategy that most often generates a syntactically valid program that integrates both contributions. Under this criteria, semistructured and structured merge would be superior to traditional unstructured merge because they often report fewer conflicts, as shown in previous studies (APEL et al., 2011; APEL; LESSENICH; LENGAUER, 2012; CAVALCANTI; ACCIOLY; BORBA, 2015).

Given that merging contributions is the main goal of any merge tool, in principle that criterion could be satisfactory. However, in practice merge tools go slightly beyond that and might detect other kinds of conflicts that do not preclude them from generating a valid program, but would lead to build or execution failures. So, from a developer's perspective, it is important to use a comparison criterion that considers not only the capacity of generating a merged program, but also the possibility of missing or early detecting conflicts that could appear during building or execution.

Consequently, we adopt, and we are guided by a broader notion of conflict. We rely on the notion of interference defined by Horwitz, Prins e Reps (1989): two contributions (changes) to a base program interfere when the specifications they are individually supposed to satisfy are not jointly satisfied by the program that integrates them; this often happens when there is, in the integrated program, data or control flow between the contributions. We then say that two contributions to a base program are conflicting when there is no valid program that integrates them and has no interference.

2.4 CONTINUOUS INTEGRATION

Since it is not always possible to prevent the occurrence of conflicts, nor always properly resolve them automatically, regardless of the kind of adopted merge tool, practitioners need other tools and practices to better support collaboration. In the context of collaborative software development, Continuous Integration (CI) is, nowadays, one of the mostly adopted software engineering practices.

(FOWLER, 2006) defines Continuous Integration (CI) as *"a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily— leading to multiple integrations per day"*. To enable this practice, each merged program should be verified by an automated build — including tests — to detect build and test errors as quick as possible. According to (ZHAO et al., 2017), CI practices have the potential to speed up development and help maintain code quality. CI has originally arisen as one of the twelve Extreme Programming (XP) practices (BECK, 2000), and it is seeing a broad adoption with the increasing popularity of decentralized VCSs such as Git and their

web-based repository hosts such as GitHub. Among the most popular GitHub-compatible, cloud-based CI tools are Travis CI (TRAVIS, 2019) and Jenkins (JENKINS, 2019).

A CI *build process* may be composed of many phases, these include a build and compilation phase, and a testing phase. The process is sequential; if a phase fails, all subsequent phases are aborted. The build and compilation phase basically attempts to translate a program into a form in which it can be executed by a computer. Problems such as source code not respecting the surrounding programming language syntax (for instance, a missing semicolon in the end of a line in Java), or an attempt to use a method or variable not declared yet in the program, are verified in this phase, and would lead the build process to fail. During the build and compilation phase, the dependencies of a program are also verified aiming to build the software as an artifact. If a dependency could not be satisfied (for instance, due to unavailability of dependency repositories), the build process will break. (SEO et al., 2014), for instance, verifies that around 65% of all build process break due to problems related to unsatisfied dependencies.

After the build and compilation phase, the testing phase is responsible for verifying whether the merged program presents the expected behavior. Testing phase is not mandatory allowing some projects to skip this important quality verification. A report, using Travis information, shows that 20% of all projects do not include tests in their build process (in Java this percentage increases to 31%) (BELLER; GOUSIOS; ZAIDMAN, 2017). For those that include, a number of test types, such as unit, integration and system tests might be executed. If any test fails or presents errors during its execution (for example, an exception not handled), the build process breaks and reports the failed test case.

Among the services and tools to support CI, Travis CI (TRAVIS, 2019) is the most used service offering a free and online service supporting projects hosted on GitHub. Its use and popularity have increased over time because of its simplicity and support for different programming languages. A Travis build process starts with an external event (*pushes* or *pull requests*) sent to a GitHub repository properly configured to use Travis CI. When such event occurs, Travis tries to build the project according to the new changes in the last commit, executing the phases explained before. Pushes and pull requests might involve a collection of commits, but only the most recent is built in the moment they are sent to GitHub. If a problem occurs in the build and compilation phase, Travis returns an *errored* status. In case a problem occurs in the testing phase, Travis returns a *failed* status. Finally, if the build process does execute all phases without any problem, the *passed* status is achieved, meaning the Travis CI build process was successfully performed. Besides the final build process status, for each build, a log is generated showing how the build process performed. Part of this log is composed by the output generated by the project's adopted build manager (e.g Maven, Gradle and Ant).

3 EVALUATING AND IMPROVING SEMISTRUCTURED MERGE

To decide whether we should replace our merge tools, we need to compare the merge strategies and understand their differences, strengths and weaknesses. So, we first compare unstructured and semistructured merge. Previous studies (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015) provide evidence that semistructured reduces the number of reported conflicts when compared to unstructured merge. This reduction is mainly due to the automatic resolution of obvious unstructured merge false positives. However, semistructured merge might have its own false positives, or misses actual conflicts (false negatives). This can make developers reluctant to replace unstructured by semistructured merge. So, we go further than those previous studies, and we compare these two merge strategies in terms of false positives and false negatives, based on the comparison criteria described in Section 2.3.

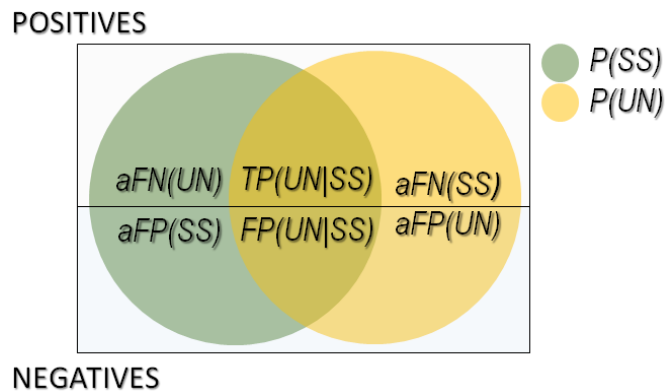
The challenge associated to the adopted comparison criteria is establishing ground truth for integration conflicts (and therefore false positives and false negatives) between development tasks, as this is not computable in this context (BERZINS, 1986; HORWITZ; PRINS; REPS, 1989). Semantic approximations through static analysis are imprecise and often too expensive, especially in the case of information flow analysis. Experts who understand the integrated code (possibly developers of each analyzed project) could determine truth, but not without the risk of misjudgment. As these options would imply into a reduced sample and limited precision guarantees, we prefer to relatively compare the two merge strategies with regard to the occurrence of false positives and false negatives of one strategy *in addition* to the ones of the other. We do that by simply analyzing when the merge strategies report different results for the same merge scenario— each scenario comprehends the three revisions involved in a three-way merge. We identify conflicts reported by one strategy but not by the other (false positives), and conflicts reported by one strategy but missed by the other (false negatives).

3.1 HEURISTICS FOR FALSE POSITIVES AND FALSE NEGATIVES

To better understand that notion of *additional* false positives and false negatives, consider the diagram in Figure 4. We illustrate the set of conflicts reported by unstructured ($P(UN)$) and semistructured ($P(SS)$) merge. Unstructured merge’s set (in yellow) includes its false positives, represented in the bottom part of the yellow circle. This is the union of the false positives reported by both strategies ($FP(UN|SS)$) with the false positives reported only by unstructured merge ($aFP(UN)$). So we say $aFP(UN)$ is the set of additional false positives of unstructured merge. Semistructured merge’s set (in green) includes conflicts detected by this strategy but missed by unstructured merge ($aFN(UN)$);

these are the additional false negatives of unstructured merge. Similarly, semistructured merge's set includes its additional false positives ($aFP(SS)$), and unstructured merge's set includes conflicts detected by this strategy but missed by semistructured merge ($aFN(SS)$). Finally, the sets also include true positives (actual conflicts) common to both strategies ($TP(UN|SS)$). For simplicity, we do not name the sets of true and false negatives common to both strategies. As our comparison is relative, the common cases do not interest us.

Figure 4 – Sets of conflicts reported by unstructured and semistructured merge. Notation explained in the text.



To guide our relative comparison, we first tried to understand the differences in the behavior of representative tools of both strategies. As semistructured merge tool, we use the original implementation of *FSTMerge* (APEL et al., 2011)—at the time of the experiment, the only available semistructured merge tool—with the annotated Java grammar they provide supporting Java 5, following previous studies (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015). Besides that, we arbitrarily chose the *Kdiff3* tool, which is one of the many unstructured merge tools available, but is a representative implementation of the *diff3* algorithm.¹ We systematically analyzed their implemented algorithms, and empirically assessed a small sample of Java merge scenarios to observe when they behave differently, and how this might lead to additional false positives and false negatives. In particular, for each conflict reported by unstructured merge, we checked whether semistructured merge also reported that conflict, and vice versa. In case of divergence, we judged if the conflict was a false positive or a false negative. In the following sections, we describe the observed kinds of additional false positives and false negatives of each merge strategy. Although we use toy examples for simplicity, the inspiration comes from concrete merge scenarios from non trivial open source projects.

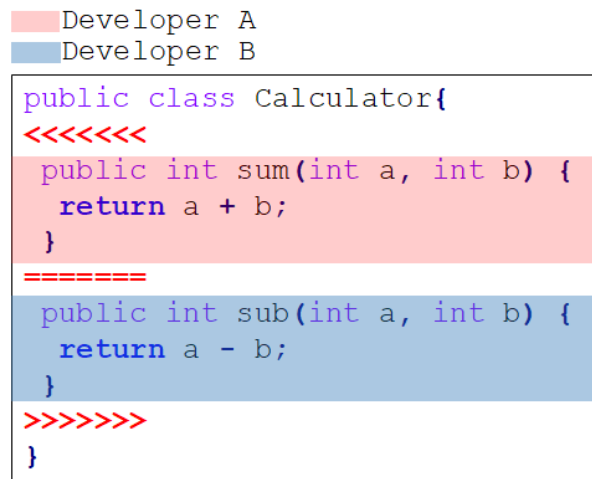
3.1.1 Additional False Positives of Unstructured Merge

One of the main weaknesses of unstructured merge is its inability to detect rearrangeable declarations. In Java, for instance, a change in the order of method and

¹ <<http://kdiff3.sourceforge.net/>>

field declarations has no impact on program behavior, but unstructured merge might report false positives—the so-called *ordering conflicts*—when developers add declarations of different elements to the same part of the text. Figure 5 illustrates this situation: a reported conflict caused by different developers adding two different methods (`sum` and `sub`, separated by typical conflict markers) to the same text area. In contrast, this is not reported as a conflict by semistructured merge. By exploiting knowledge about Java syntax and static semantics, semistructured merge identifies commutative and associative declarations, and understands that the changes to be merged can be integrated because they are related to different nodes. As discussed later, not all ordering conflicts are (additional) false positives of unstructured merge. For example, import declarations are often, but not always, re-arrangeable.

Figure 5 – Unstructured merge additional false positive: *ordering conflict*.



```

Developer A
Developer B

public class Calculator{
<<<<<<<
    public int sum(int a, int b) {
        return a + b;
    }
=====
    public int sub(int a, int b) {
        return a - b;
    }
>>>>>>>
}

```

3.1.2 Additional False Positives of Semistructured Merge

Renamings challenge semistructured merge, as illustrated in Figure 6, which shows a false positive *renaming conflict*: one of the developers renamed the `calculate` method to `sum`, whereas the other developer kept the original signature but edited the method body. Semistructured merge reports this as a conflict because its algorithm interprets method renaming as method deletion, consequently assuming that a developer deleted the method (`calculate` in this case) changed by the other. In particular, to check whether a base declaration was changed by both developers, the merge algorithm tries to match nodes by the type and identifier of the corresponding declaration. In Java, for instance, a method is identified by its name and the types of its formal parameters. So, when an element is renamed, the merge algorithm is not able to map the base element to the newly named element in the changed version of the file, and assumes the element was deleted. Note that the `sum` method does not appear in the conflict; that is, it is not surrounded by the conflict markers (the vertical bars and equal signs separate base code from code to be merged).

Figure 6 – Semistructured merge additional false positive: *renaming conflict*.

Base	
1	<code>public class Calculator{</code>
2	<code><<<<<<<</code>
3	<code>public int calculate(int a, int b)</code>
4	<code>{</code>
5	<code>return (a + b)*2;</code>
6	<code>}</code>
7	<code> </code>
8	<code>public int calculate(int a, int b)</code>
9	<code>{</code>
10	<code>return a + b;</code>
11	<code>}</code>
12	<code>=====</code>
13	<code>>>>>>>></code>
14	<code>public int sum(int a, int b)</code>
15	<code>{</code>
16	<code>return a + b;</code>
17	<code>}</code>
18	<code>}</code>

In the illustrated case, merging the new signature with the new body would be a safe valid integration of both contributions; there is no conflict because the changes do not affect the developers' expectations: the method will behave as desired by one developer, and will be called as wanted by the other developer. This is exactly what unstructured merge does. It does not report a conflict because the changes occur in distinct text areas; in the example, the "{" in line 9 separates the two changed areas. If this character was in line 8, unstructured merge would also report a conflict, and this would be a common false positive. This helps to explain why a renaming conflict involving a field declaration is an additional false positive only when the declaration is split into multiples lines of code.

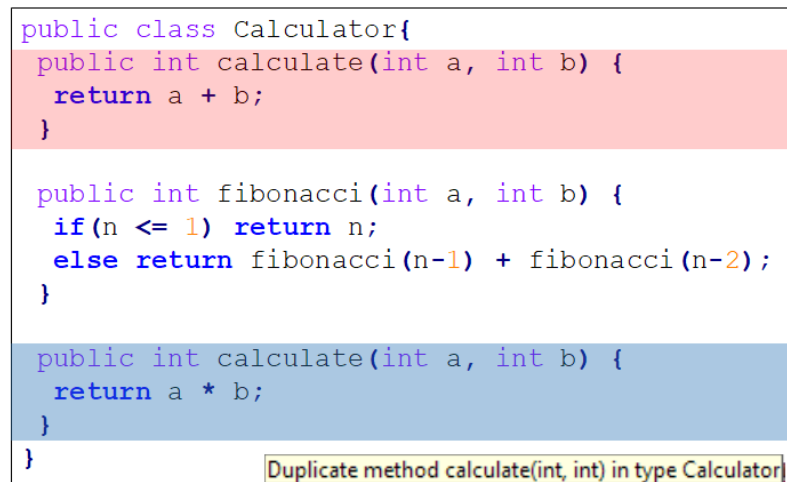
Renaming conflicts are often false positives, but they might be true positives too. For example, consider the case where one of the developers renames a method, and the other developer not only changes the same method body but also adds new calls to it. Merging the new signature with the new body, which corresponds to the integration of both contributions, would lead to an invalid program that calls an undeclared method. Although semistructured merge does not actually realize that the merge would lead to an invalid program, it soundly does not perform the merge and reports a conflict. In contrast, unstructured merge would unsoundly merge the contributions provided there is a separator as in the illustrated example.

3.1.3 Additional False Negatives of Unstructured Merge

The additional false negatives of unstructured merge are mostly caused by failing to detect that the contributions to be merged add duplicated declarations. For example,

unstructured merge reports no conflict when merging developers contributions that add declarations with the same signature to different areas of the same class. This leads to an invalid resulting program with a compiler *duplicated declaration error*. Figure 7 illustrates this situation: both developers added methods with the same signature but with different bodies, in clearly separated areas, not leading to an unstructured merge conflict. As semistructured merge matches elements to be merged by their type and identifier, it would detect the problem and correctly report a conflict.

Figure 7 – Unstructured merge additional false negative: *duplicated declaration error*.



```
public class Calculator{
    public int calculate(int a, int b) {
        return a + b;
    }

    public int fibonacci(int a, int b) {
        if(n <= 1) return n;
        else return fibonacci(n-1) + fibonacci(n-2);
    }

    public int calculate(int a, int b) {
        return a * b;
    }
}
```

Duplicate method calculate(int, int) in type Calculator

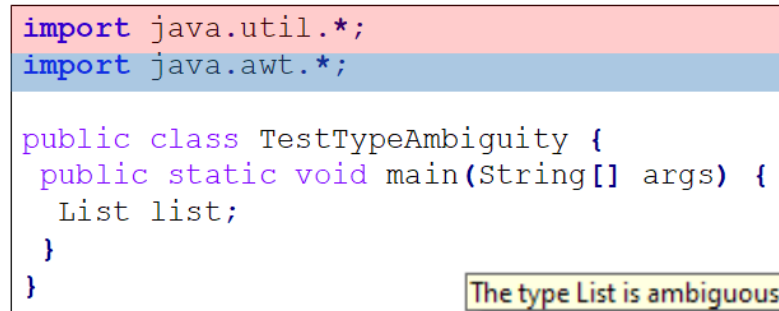
Besides that, renaming conflicts might also be additional false negatives of unstructured merge. As explained at the end of the previous section, renaming conflicts detected by semistructured merge might be true positives. But these would only be detected by unstructured merge in case the changes occur in the same text area. For instance, consider an example similar to the one in Figure 6, but where developer A also added a call to method `calculate` in an area not changed by developer B. As there are separators between developers changes, unstructured merge would erroneously not report a conflict.

3.1.4 Additional False Negatives of Semistructured Merge

As mentioned in Section 3.1.1, ordering conflicts involving import declarations are often, but not always, false positives. In fact, merging developers contributions that add import declarations to the same text area (so unstructured merge reports conflict) might lead to additional false negatives of semistructured merge, as it assumes that such declarations are always re-arrangeable. For example, this might lead to a *type ambiguity error* (see Figure 8) when the import declarations involve members with the same name but from different packages. In the illustrated case, both imported packages have a `List` class. As the import declarations appear in the same or adjacent lines of code, unstructured merge correctly reports a conflict, whereas semistructured merge lets the conflict escape. In other situations, semistructured merge's assumption about import declarations might

lead to behavioral errors instead. In the illustrated example, suppose that developer A had written `import java.awt.List`. As the ambiguous members might share methods with the same signature but different behaviors, the presence of both declarations might affect class behavior. In the example, both `List` members have `add` methods behaving differently. Again, semistructured merge would miss the conflict, which would be reported by unstructured merge.

Figure 8 – Semistructured merge additional false negative: *type ambiguity errors*.



```
import java.util.*;
import java.awt.*;

public class TestTypeAmbiguity {
    public static void main(String[] args) {
        List list;
    }
}
```

The type List is ambiguous

Besides the issue with import statements, semistructured merge has additional false negatives due to the way it handles *initialization blocks*. Since it uses elements types and identifiers to match nodes, the algorithm is unable to match nodes without identifiers. This leads to problems like the one illustrated in Figure 9; as the two independently added initialization elements have no identifier, semistructured merge cannot match them, and therefore keeps both. A conflict should have been reported so that developers could negotiate how the class field should be initialized. In case the initialization blocks are added to the same text area, unstructured merge reports a conflict.

Semistructured merge has also other kinds of additional false negatives. Although they do not conform to a small set of recurring syntactic patterns, they all result from unstructured merge *accidentally* detecting conflicts that would otherwise escape if changes were performed in slightly different text areas. For example, this might occur when developers change or add, in the same text area, different but dependent declarations. In particular, we observed that when one developer added a new declaration that references an existing one edited by the other developer. In such cases, the developer who added the new declaration might not be expecting the changes made to the referenced one, possibly leading to an improper behavior on the merged program. This is illustrated in Figure 10, where the new method `triple` references the `calculate` method, which was changed by the other developer. Although one might assume that methods provide the lowest level of information hiding and modularity (with its signature as interface), in practice, an unchanged method interface is not sufficient to ensure that the method behavior is also unchanged. So, we consider this situation as a semistructured merge additional false negative. In fact, as the changes correspond to different elements (technically, different nodes in the semistructured merge tree), semistructured merge reports no conflict. In

Figure 9 – Semistructured merge additional false negative: *initialization blocks*.

```

public class TestInitializationBlocks{
    static String _name;

    static {
        _name = "Bob";
    }

    static {
        _name = "Alice";
    }

    public static void printName() {
        System.out.println(_name);
    }

    public static void main(String[] args) {
        printName();
    }
}

```

contrast, unstructured merge might accidentally detect such conflicts when the changes are in the same text area.

Figure 10 – Semistructured merge additional false negative: *new element referencing edited one*.

```

public class Calculator{
    public int calculate(int a, int b) {
        return (a + b) * 2;
    }

    public int triple(int a, int b) {
        return calculate(a,b)*3;
    }
}

```

3.1.5 Common False Positives and False Negatives

The kinds of false positives and false negatives described in the previous sections correspond only to the differences between semistructured and unstructured merge algorithms. As our interest here is to compare both strategies relatively— not to establish how accurate they are in relation to a general notion of conflict— we do not need to measure the occurrence of false positives and negatives when both strategies behave identically. For example, when processing changes inside method bodies, semistructured merge actually

invokes unstructured merge, so they present common false positives and false negatives. As an example of a common false positive, consider that developers edit consecutive lines in a method body, but one of them does not change behavior (simply refactors or changes spacing). A common false negative would be edits to different method lines, in different areas of the body, but with a harmful data flow dependency between the statements in such lines. Besides that, it is important to note that unstructured merge might accidentally report renaming false positives in case changes occur in the same area, or miss the same kinds of semistructured merge false negatives described in the previous section, in case changes are not in the same area. However, these cases do not interest us because both merge tools would behave identically. Although important for establishing accuracy in general, these are not useful for relatively comparing merge strategies.

3.2 EMPIRICAL EVALUATION

Our evaluation investigates whether the reduction in the number of conflicts by using semistructured merge, in relation to unstructured merge (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015), actually leads to integration effort reduction (productivity) without negative impact on the correctness of the merging process (quality). We do that by reproducing merges from the full development history of different GitHub projects, while collecting evidence about the occurrence of conflicts, and the kinds of additional false positives and false negatives described in the previous section. In particular, we investigate the following research questions:

- **RQ1** *When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort by reporting fewer false positives?*
- **RQ2** *When compared to unstructured merge, does semistructured merge compromise integration correctness by having more false negatives?*

To answer **RQ1**, we compute the metric *overestimated number of additional false positives of semistructured merge*— $aFP(SS)$ (false positives reported by semistructured merge and not reported by unstructured merge). We also compute the metric *underestimated number of additional false positives of unstructured merge*— $aFP(UN)$ (false positives reported by unstructured merge and not reported by semistructured merge). As the metrics names suggest, they are approximations. To consider a large sample, as discussed in Section 3.1, we aim to evaluate the merge strategies by computing the number of diverging false positives and false negatives of each strategy. However, precisely computing that is hard, as discussed later in this section. Nevertheless, if we find that an overestimated number is inferior to an underestimated number, we can conclude that the precise value represented by the overestimated number is lower than the precise value represented by the underestimated one. This was indeed observed in dry runs of our study, giving us confidence that we

could adopt this design. As different reported conflicts might demand different resolution effort (MENS, 2002; PRUDÊNCIO et al., 2012; SANTOS; MURTA, 2012), comparing conflict numbers might not be sufficient for understanding the impact on integration effort. So, to better understand the effort required to resolve different kinds of false positives, we conduct a number of complementary analyses to estimate the impact on integration effort. Our goal with these analyses is to simply check that the computed metrics are not obviously bad choices as proxies for integration effort.

For answering **RQ2**, we compute the metrics *overestimated number of additional false negatives of semistructured merge*— $aFN(SS)$ (conflicts missed by semistructured merge and correctly reported by unstructured merge), and the *underestimated number of additional false negatives of unstructured merge*— $aFN(UN)$ (conflicts missed by unstructured merge and correctly reported by semistructured merge). We also discuss the integration effort impact of the kinds of false negatives of both strategies, but this does not require further elaborated analyses.

To answer these questions and compute the related metrics, we adopt a three-step setup: mining, execution, and analysis. In the *mining* step, we use tools that mine GitHub repositories to collect merge scenarios — each scenario is composed by the three revisions involved in a three-way merge. In the *execution* step, we use an unstructured and a semistructured merge tool to merge the selected scenarios and to find potential additional false positives and false negatives. In the *analysis* step, we confirm the occurrence of additional false positives and negatives. Finally, we use R scripts to perform data analysis and to generate reports. We now detail these steps, jointly explaining the execution and analysis steps for simplicity.

3.2.1 Mining Step

To select meaningful projects, we first searched for the top 100 Java projects with the highest number of stars in GitHub’s advanced search page.² From this search result, due to execution time constraints, we selected 50 projects having a certain degree of diversity (NAGAPPAN; ZIMMERMANN; BIRD, 2013) with respect to a number of factors described later. We restricted our sample to Java projects because the execution and analysis steps demand language dependent tool implementation and configuration. After selecting the sample projects, we used tools to mine their GitHub repositories and collect merge scenarios from their full histories. In particular, we used the *GitMiner* tool to convert the entire development history of a GitHub project into a graph database.³ Subsequently, we implemented scripts to query this database and retrieve a list of the identifiers of all merge commits— commits having a true value for their `isMerge` attribute— and their parents. As a result, we obtained 34,030 merge scenarios from the 50 selected Java projects.

² <<https://github.com/search/advanced>>

³ <<https://github.com/pridkett/gitminer>>

Given that part of the execution and analysis steps are language dependent, we only process the Java files in these scenarios, missing conflicts in non-Java files. As the compared tools could easily be adapted to behave identically for non-Java files, this is not a major problem. Moreover, the number of non-Java files not merged corresponds to 1.73% of the total number of files in the sample (we discuss this threat later in Section 3.5).

Although we have not systematically targeted representativeness or even diversity (NAGAPPAN; ZIMMERMANN; BIRD, 2013), we believe that our sample has a considerable degree of diversity with respect to, at least, the number of developers, source code size, and domain. It contains projects from different domains such as databases, search engines, and games. They also have varying sizes and number of developers. For example, RETROFIT, a HTTP client for Android, has only 12 KLOC, while OG-PLATFORM, a solution for financial analytics, has approximately 2,035 KLOC. Moreover, MCT has 13 collaborators, while DROPWIZARD has 141. Besides that, our sample includes projects such as CASSANDRA, JUNIT, and VOLDEMORT, which are analyzed in previous studies (BRUN et al., 2011; KASI; SARMA, 2013; CAVALCANTI; ACCIOLY; BORBA, 2015). The list of the analyzed projects, together with the tools we used, is in our online appendix (CAVALCANTI, 2019).

3.2.2 Execution and Analysis Steps

After collecting the sample projects and merge scenarios, we use the *KDiff3* unstructured tool, and the *FSTMerge* semistructured tool to merge the selected scenarios. We then identify and compare the occurrence of additional false positives and false negatives, as described in Section 3.1. These tools take as input the three revisions that compose a merge scenario (here we call them as *base*, *left*, and *right* revisions) and try to merge their files. To identify false positives and false negatives candidates, we intercept *FSTMerge* during its execution. Given that the tool is structure-driven, we are able to inspect the source code and the conflicts in terms of the syntactic structure of the underlying language elements. This would not be possible with a textual tool. To confirm the occurrence of the false positives and false negatives, we use a number of scripts; some of them rely on the parsing and compiler features of the Eclipse JDT API.⁴ For brevity, here we overview how we compute the metrics, and leave the detailed explanation to the online appendix (CAVALCANTI, 2019), where we also point to the version of *FSTMerge* that contains the interceptors we implemented for our study.

3.2.2.1 Computing the Underestimated Number of Additional False Positives of Unstructured Merge— aFP(UN)

The additional false positives of unstructured merge are due to its inability to perceive that some declarations are commutative and associative, and therefore avoid the ordering

⁴ <<http://www.eclipse.org/jdt/>>

conflicts (see Section 3.1.1). We found no specific patterns of ordering conflicts that would allow us to identify them by systematically inspecting the reported conflicts. We can, however, compute this metric in terms of the others. As explained earlier, Figure 4 illustrates the set of conflicts reported by unstructured and semistructured merge. For instance, unstructured merge set includes its additional false positives ($aFP(UN)$). The sets also include true and false positives common to both strategies, denoted by $TP(UN|SS)$ and $FP(UN|SS)$. Based on the diagram, we can infer that

$$aFP(UN) = P(UN) - (FP(UN|SS) + TP(UN|SS)) - aFN(SS)$$

Observe that we can estimate $FP(UN|SS) + TP(UN|SS)$ in terms of $P(SS)$ (in green in the diagram). To compute a lower bound of $aFP(UN)$, we need an upper bound of $FP(UN|SS) + TP(UN|SS)$ because it is a subtractive factor. This upper bound is reached when $aFP(SS)$ is at its minimum (zero). Finally, we can derive the underestimated number of unstructured merge additional false positives as follows:⁵

$$aFP(UN) \geq P(UN) - P(SS) + aFN(UN) - aFN(SS)$$

Note that $P(SS)$ and $P(UN)$ can be simply computed by running the merge tools and observing the reported conflicts.

3.2.2.2 Computing the Overestimated Number of Additional False Positives of Semistructured Merge— $aFP(SS)$

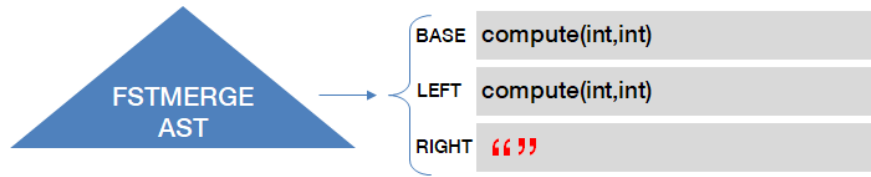
The additional false positives of semistructured merge, as explained in Section 3.1.2, are due to renamings. To identify these false positives, we first intercept conflicts detected by *FSTMerge*. We then check whether the involved triple of base, left and right elements contains a non-empty base, and an empty left or right element. Not having the left or right version, represented by the red empty string in Figure 11, indicates that the semistructured merge algorithm could not map the element (method in this case) to its previous version. This happens when the element was renamed or deleted. Since we cannot precisely guarantee there was a deletion (the best option would be a similarity analysis on method bodies), we conservatively analyze all such cases. For simplicity, we also conservatively assume that unstructured merge did not report the same conflict, in case there were no separators between the changes.

3.2.2.3 Computing the Underestimated Number of Additional False Negatives of Unstructured Merge— $aFN(UN)$

The main pattern of unstructured merge additional false negatives occurs, as explained in Section 3.1.3, when developers independently introduce duplicated declarations in

⁵ We formally derive the formula in the online appendix.

Figure 11 – Intercepting the *FSTMerge* tool to find *renaming* false positives (*aFP(SS)*) candidates.



different areas of the program text. To identify such situations, we intercept *FSTMerge* when it matches triples of elements with an empty base, and non-empty left and right elements. This indicates the addition of two new elements with the same signature. To confirm that the elements were added to different areas, we merge the files with the unstructured tool and verify that there is no reported conflict that contains the duplicated declaration. In case there is such a conflict, the additional false negative candidate is discarded. If there is no such a conflict, we compile the resulting file and search the compiler output for duplicated declarations errors related to the identified elements.

Whereas we are able to precisely compute the number of duplicated declaration errors, it would be harder to precisely compute the other kinds of additional false negatives—such as true renaming conflicts— of unstructured merge. So, we actually compute the *underestimated* number of additional false negatives of unstructured merge.

3.2.2.4 Computing the Overestimated Number of Additional False Negatives of Semistructured Merge— *aFN(SS)*

As explained in Section 3.1.4, the additional false negatives of semistructured merge are related to three major causes: reordering import statements that involve types with the same identifier, not matching initialization blocks, and unstructured merge accidental conflict detection.

To compute the first kind of false negative, we use *FSTMerge* to identify attempts to merge trees that contain at least a pair of introduced or modified import declaration nodes. We then try to merge the corresponding files with unstructured merge, and check if it reports a conflict involving the pair of imports statements. If it does not report, we have a common true or false negative of both strategies; so no further action is needed. If it does report such a conflict, we further check if the import declarations lead to type ambiguity errors. We do that by compiling the resulting file merged by semistructured merge and searching for type ambiguity compilation errors. This works for when both developers add or edit imports to packages, or members. We also check if the import statements might lead to behavioral issues— when one developer adds or edits imports to packages, and the other imports to members— we search for the name of the member imported by one developer in the changes introduced by the other, and vice versa. This is a grep-based analysis, having as search scope the file containing the import statements. With a positive

result in one of the checks, we conservatively consider the pair of import statements as a conflict missed by semistructured merge (additional false negative).

To identify the second kind of false negative, we use *FSTMerge* to identify all nodes representing initialization blocks in the different tree versions. Using textual similarity, based on the *Levenshtein distance algorithm* (LEVENSHTEIN, 1966) with 80% of degree of similarity, we group triples of similar initialization nodes. These triples are then merged with unstructured merge. If they conflict, we conservatively compute the conflicts as additional false negatives. As the same block might have been substantially changed by both developers, they might not satisfy our similarity threshold. This way we could miss false negatives. To make sure this is not the case, we carry on a manual analysis. We discuss this threat in Section 3.5. Choosing a lower threshold could be too conservative, and not substantially reduce the number of cases to be manually analyzed. An alternative metric could consider the number of edited initialization blocks as the number of additional false negatives of this kind. However, this might not be conservative as changes to initialization blocks might lead to more than one conflict.

All other conflicts reported by unstructured merge but missed by semistructured merge are conservatively classified as additional false negatives, except in the following two cases. In both cases, we are able to parse the text of the unstructured merge conflict and then classify the reported conflict. First, when the reported conflict contains only field declarations that do not reference each other. Such reported conflicts are unstructured false positives instead because there is no dependence among the field declarations. Second, when the conflict resolution keeps all changes from both left and right revisions, and adds no new code. We assume that the developer correctly analyzed the conflict and decided there was no problem (we discuss this threat later in Section 3.5); so that would be an unstructured false positive, not a semistructured false negative. We check that by parsing and inspecting the original merge commit in the project repository.

More precise analyses, such as those based on testing or information flow, could possibly reduce our upper bound of semistructured merge false negatives, but would still be imprecise and reduce the analyzed sample, as explained earlier.

3.3 RESULTS AND DISCUSSION

By analyzing a total of 34,030 merge scenarios from 50 Java projects, we identified 19,238 conflicts when using unstructured merge, and 14,544 using semistructured merge. This represents a semistructured merge reduction of approximately 24% in the total number of reported conflicts. As each merge scenario might have conflicts reported by both tools, only one, or none, we observed that the 19,238 conflicts reported by unstructured merge occurred in 8.8% of the sample merge scenarios. Moreover, the 14,544 conflicts reported by semistructured merge occurred in 7.1% of the sample merge scenarios. We also observed that in 54.6% of the sample merge scenarios having at least one conflict, regardless of

the tool, semistructured merge reported fewer conflicts than unstructured merge. This is similar to previous studies, differing at most by 5% (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015). In these scenarios, the observed reduction in the total number of conflicts was of $71\% \pm 30\%$ (average \pm standard deviation), compared to $62\% \pm 24\%$ in our previous study (CAVALCANTI; ACCIOLY; BORBA, 2015), and $34\% \pm 21\%$ in the study of Apel et al. (2011). This evidence of conflict numbers, however, is not enough for justifying the adoption of a merge tool because of the risk of missing actual conflicts (false negatives), or introducing new kinds of false positives. Considering the merge scenarios having conflicts, we found that in 27.1% of them one strategy detected at least one conflict, and the other none. In 17.1% of them the strategies reported the same conflicts, and in 25.9% the strategies detected only different conflicts. So, when they report conflicts, they differ more substantially than we originally expected. In the remaining of the section, we further present descriptive statistics, structured according to our research questions, and discuss their implications. Detailed results for the analyzed projects are available in the online appendix (CAVALCANTI, 2019).

3.3.1 When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort by reporting fewer false positives?

To answer RQ1, we compare the number of additional false positives of each merge strategy. Our results show that, in our sample, when using an unstructured merge tool, $6.58\% \pm 6.07\%$ of the merge scenarios have at least one estimated additional false positive ($aFP(UN)$). Moreover, $43.47\% \pm 19.01\%$ of the reported conflicts are additional false positives according to our metric ($aFP(UN)$). This is bigger than the percentage of semistructured merge additional false positives ($aFP(SS)$): $30.21\% \pm 20.68\%$. In addition, only $3.12\% \pm 3.55\%$ of the merge scenarios have at least one additional false positive ($aFP(SS)$). So, considering the aggregated scenarios of all projects, we conclude that semistructured merge has fewer additional false positives and fewer scenarios with additional false positives. In practice, we should expect a bigger difference in favor of semistructured merge since $aFP(UN)$ is underestimated and $aFP(SS)$ is overestimated. However, these findings do not uniformly hold across projects: $aFP(SS)$ is greater than $aFP(UN)$ in 26% of our sample projects. In contrast, in only 2% of the projects, semistructured merge had more merge scenarios with more additional false positives.

The large error bounds, and the lack of uniformity of the results across projects, are caused by variations in the analyzed projects. For example, project histories containing renaming of directories might significantly increase the number of renaming conflicts reported by semistructured merge. We observed that in projects such as OG-PLATFORM and EQUIVALENT-EXCHANGE-3, which have a greater number of scenarios containing directory renamings, and consequently show semistructured merge false positive rates substantially above the average. In such cases, unstructured merge reports, for each file

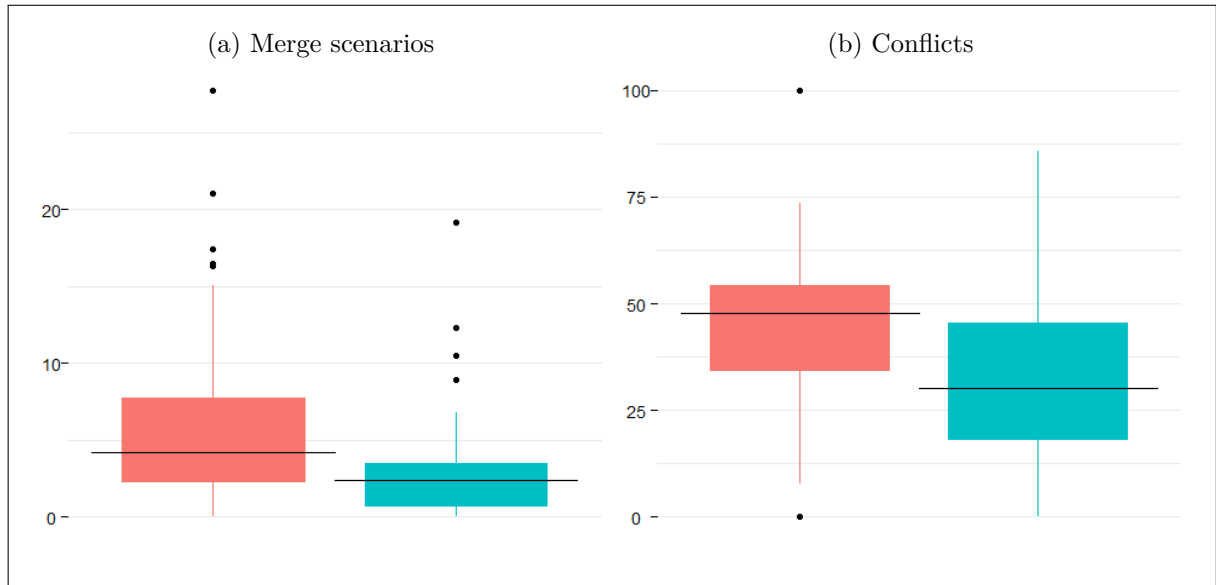
of the renamed directory, a single large conflict; it cannot map the files of the renamed directory to the corresponding files of the other revision. Conversely, due to semistructured merge finer granularity, the conflicts are reported per element (method, constructor, etc.) in the files of the renamed directory, substantially increasing the number of reported conflicts. When ignoring these directories renamings, the average number of semistructured additional false positives drops by about 5%. The error bounds are also explained by projects such as `ANDEngine`, `MCT`, and `VOLDEMORT`, in which most conflicts occur inside method bodies. In these situations, the tools behave identically and report a large number of common conflicts, decreasing the percentage of additional false positives.

Given that our data is paired, and deviates from normality, we analyze differences in the computed metrics with the paired Wilcoxon Signed-Rank test. It shows that the merge strategies present statistically significant different means of percentages of merge scenarios with false positives ($p\text{-value} = 1.13\text{e-}09 < 0.05$). Besides that, we observed a large effect size ($r = 0.82 > 0.5$, based on the Pearson Correlation Coefficient). There is also significant difference between the percentages of additional false positives ($p\text{-value} = 0.001047 < 0.05$), with medium effect size ($r = 0.46 > 0.3$). This tendency can also be observed in the box plots of Figure 12. Note, for instance, that in both cases (merge scenarios and conflicts) the 3rd quartile in the box plots of the semistructured strategy is inferior to the median in the box plots of unstructured merge. The difference on the shapes of the boxplots suggests that, overall, projects have a high level of agreement with each other in terms of merge scenarios with semistructured merge additional positives (Figure 12(a) in blue), but the projects hold quite different number of semistructured merge additional false positives (Figure 12(b) in blue). This means that, among projects, although the number of scenarios with semistructured merge additional false positives is close, the number of semistructured merge additional false positives is not. Conversely, unstructured merge results in a close number of scenarios with its additional false positives, and overall number of additional false positives among projects.

3.3.1.1 Additional False Positives of Semistructured Merge are Easier to Analyze and Resolve

Semistructured merge reduction in the number of additional false positives for most projects and scenarios suggests that it might reduce integration effort. However, a more accurate comparison would measure the actual effort required for analyzing and discarding false positives. This is important because different conflicts often demand different effort to be analyzed, and then discarded or resolved. Thus, to better understand the impact of false positives on integration effort, we manually analyzed 100 randomly selected merge scenarios: 50 containing semistructured merge additional false positives (renaming conflicts), and 50 with unstructured merge additional false positives (ordering conflicts). These are appropriate sample sizes for estimating proportion (ENG, 2003) considering margins of error of 10% and 15%, respectively, for renaming and ordering conflicts (we

Figure 12 – Box plots describing the percentage, per project, of additional false positives in terms of merge scenarios and conflicts. Unstructured merge in red, semistructured in blue.



further discuss this in Section 3.5). For each scenario, we observed the reported conflicts and attempted to resolve them to understand how easily they could be analyzed and discarded. We also observed the corresponding merge commit, in the project history, to understand how each developer contribution was integrated. Similarly to Menezes et al. (2018), we assume that resolutions including only changes from the merged contributions (without new code, nor combination of contributed code) demand less effort. We believe this is a fair approximation of the time needed to fix the code— which is part of the total integration effort— but not of the time needed to reason about the conflict and then decide how to fix it. The manual analysis considers this last part.

The manual analysis revealed that semistructured merge false positives are easy to analyze and resolve. As explained before and illustrated in Figure 6, this kind of reported conflict shows the original element name with its new body (as left, for example), and its original body (as base). The integrator can then easily find a corresponding element with a new name and original body.⁶ Resolution basically consists of declaring a single element, with the new name and the new body. However, we have also observed cases where the integrator discarded the new name or the new body.

With respect to unstructured merge false positives, only part of the manually analyzed cases was easy to analyze and resolve. We believe that reported conflicts caused by the introduction of declarations (methods or fields) in the same text area can often be analyzed and resolved with little effort. The integrator simply has to choose one of the

⁶ Not finding such an element indicates deletion (instead of renaming), which implies into a true positive, not being useful for our analysis.

declarations, or decide to keep them all. However, we also observed a challenging kind of ordering conflict that does not respect the boundaries of Java syntactic structures—so we name it a *crosscutting conflict*. Such reported conflicts involve incomplete parts of different language structural elements (methods, fields, etc.). We illustrate this in Figure 13, observed in a merge scenario from project CASSANDRA. Note that parts of the `getColumn` and `validateMemtableSetting` methods conflict because the changes occurred in the same text area. Such conflicts are more difficult to analyze and resolve because they demand one to map code chunks to corresponding syntactic structures; in the illustrated example, it is not clear which method contains the `for` and `if` statements. As such kind of conflict involves different syntactic elements (and then different nodes in a parse tree), semistructured merge automatically resolves them.

Figure 13 – Crosscutting ordering conflicts.

```

<<<<<<
public static void validateMemtableSettings(org.apache.cassa...
=====
public ColumnDefinition getColumnDefinition(ByteBuffer name) {
    ...
public ColumnDefinition getColumnDefinitionForIndex(String indexName) {
    ...
>>>>>>
<<<<<<
    if(cf_def.memtable_flush_after_mins != null)
        ...
    if(cf_def.memtable_throughput_in_mb != null)
        ...
    if(cf_def.memtable_operations_in_millions != null)
        ...
}
public ColumnDefinition getColumnDefinition(ByteBuffer name) {
    ...
public ColumnDefinition getColumnDefinitionForIndex(String indexName) {
    for(ColumnDefinition def : column_metadata.values())
=====
    for(ColumnDefinition def : column_metadata.values())
>>>>>>

```

Developer A
Developer B

To understand how these findings are related to our entire sample, we carried on further automatic analysis. By trying to parse code of the unstructured merge additional false positives, we found that 44.81% of the conflicts are crosscutting; that is, we could not parse the conflict text because it does not correspond to a single valid language element. When analyzing how these conflicts were resolved, we found that 92.76% of the resolutions involved no new code. This suggests that a significant part of unstructured merge false positives might be hard to analyze, but their resolution is rarely hard. In fact, conflict

analysis might be so hard that resolution might simply correspond to discarding one of the contributions.

Summary: In our sample, though not uniformly across projects, semistructured merge reduced the overall number of reported conflicts and has fewer additional false positives than unstructured merge. Furthermore, we argue that semistructured merge additional false positives are easier to understand and resolve.

3.3.2 When compared to unstructured merge, does semistructured merge compromise integration correctness by having more false negatives?

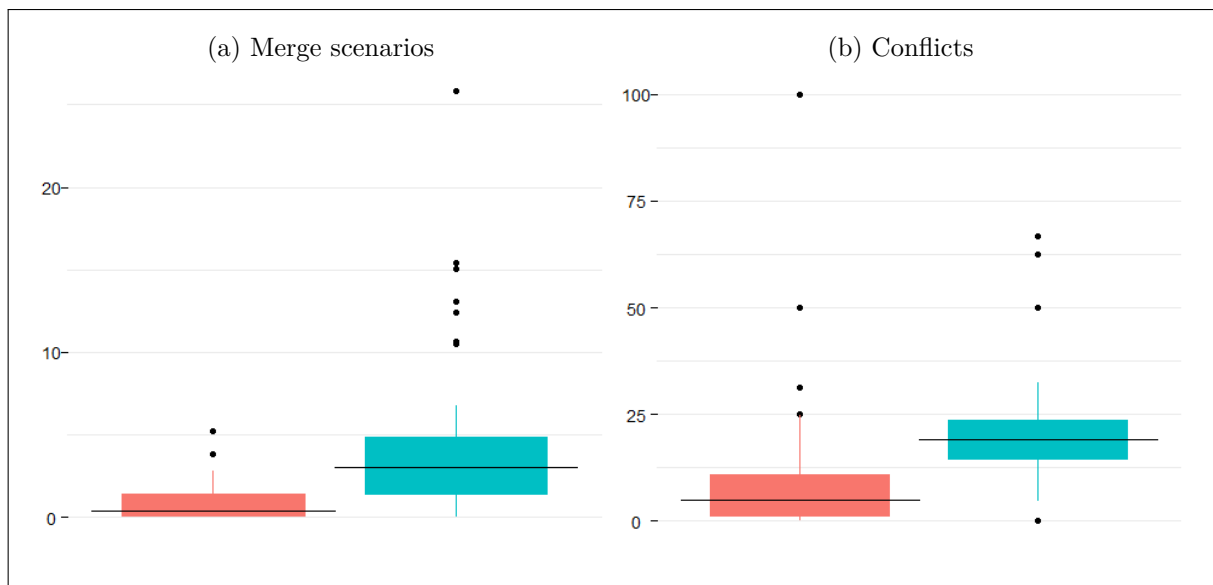
To answer RQ2, we compare the number of additional false negatives of each merge strategy. Our results show that the number of semistructured merge additional false negatives ($aFN(SS)$) is $20.60\% \pm 21.30\%$ with respect to the total number of reported conflicts, compared to $9.62\% \pm 16.29\%$ of unstructured merge ($aFN(UN)$). We also observed that $4.42\% \pm 5.53\%$ of the merge scenarios have at least one semistructured merge additional false negative ($aFN(SS)$), compared with $0.88\% \pm 1.08\%$ of unstructured merge ($aFN(UN)$). So, considering the aggregated merge scenarios of all projects, semistructured merge has more additional false negatives and more scenarios with additional false negatives. However, in practice, we should expect a lower, or even no advantage in favor of unstructured merge since $aFN(SS)$ is overestimated and $aFN(UN)$ is underestimated. So, in this aspect, looking only at numbers, our study brings no conclusive evidence to answer RQ2, then we resort to manual analysis as explained later. Besides, as for RQ1, this does not uniformly hold across projects: $aFN(UN) > aFN(SS)$ in 18% of the sample projects. Additionally, in only 2 projects (CLOSURE-COMPILER and ESSENTIALS), unstructured merge had more merge scenarios with more additional false negatives ($aFN(UN)$).

As in the previous section, the observed error bounds are partly explained by some projects having high rates of common conflicts. But here, they are additionally influenced by some projects having high rates of accidental detection of conflicts by unstructured merge. In fact, projects such as ANTENNAPOD and MOCKITO presented considerably above the average percentage of additional false negatives ($aFN(SS)$) due to accidental detection of conflicts by unstructured merge (when unstructured merge detects actual conflicts that would otherwise escape if changes were performed in slightly different text areas). Most of these conflicts were conservatively classified as semistructured additional false negatives, but turned into false positives ($aFP(UN)$) after further analysis. Conversely, analyzing unstructured merged additional false negatives ($aFN(UN)$), we found projects with a higher incidence of duplicate simple methods declarations such as getters and setters, or methods containing common words from developers' vocabulary such as initialize, execute, run, or load. We also found that copy and paste across repositories was a common practice

in some projects. For example, in a certain commit one developer added a method. Then, on a divergent branch, another developer copied this method and made a few changes. When merging these changes, the conflict occurred with semistructured merge, but not with unstructured merge. We even found examples where, instead of copying one method, the developer copied entire files from one repository to the other. All these situations led to above average percentage of additional false negatives ($aFN(UN)$) in projects such as ATMOSPHERE, CLOUDIFY and GRADLE.

Wilcoxon Signed-Rank tests show that there is statistically significant difference when comparing the two strategies, both in terms of merge scenarios and in terms of conflicts (p -value equals to, respectively, $4.18e-09$ and $5.54e-06 < 0.05$). We also observed a large effect size in terms of merge scenarios ($r = 0.8 > 0.5$), and in terms of conflicts ($r = 0.57 > 0.5$). This tendency can be observed in the box plots of Figure 14. In the case of merge scenarios with additional false negatives (Figure 14(a)), observe that the maximum whisker of the unstructured merge box plots is inferior to the median of the semistructured merge box plot. Besides, in terms of conflicts (Figure 14(b)), the 3rd quartile in the box plots of the unstructured strategy is inferior to the 1st quartile in the box plots of semistructured merge.

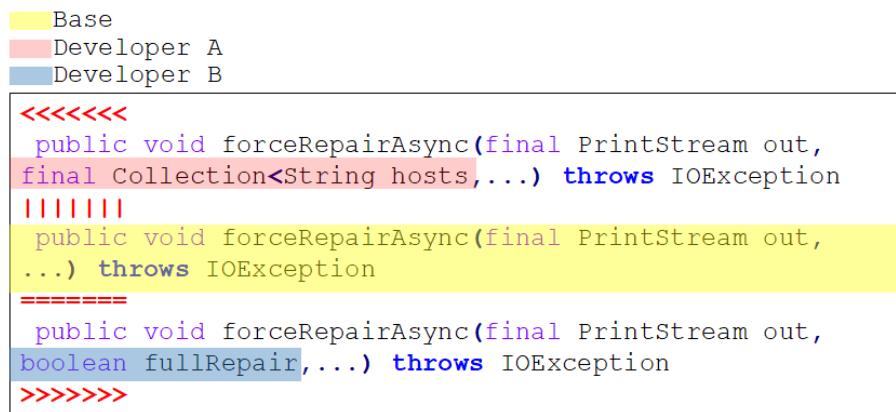
Figure 14 – Box plots describing the percentage, per project, of the additional false negatives in terms of merge scenarios and conflicts. Unstructured merge in red, semistructured in blue.



We were expecting a numerical advantage of unstructured merge due to the imprecision of our metric ($aFN(SS)$), but not at the observed level. When distinguishing among the kinds of semistructured merge additional false negatives (see Section 3.1.4), we found that only 0.46% of them are type ambiguity errors, 6.78% are due to initialization blocks, and 92.76% are due to unstructured merge accidental conflict detection. So, to understand

how high our upper bound could be, we manually analyzed 50 randomly selected possible additional false negatives of semistructured merge. We checked if they indeed represent missed conflicts. From the 50 analyzed cases, only 6 were confirmed false negatives. Among these, consider the conflict illustrated in Figure 15(a), where both developers added parameters to the same method; as the developers might not be expecting the extra parameter, the conflict is appropriate since this will likely affect the build. Semistructured merge is unable to detect this conflict because the method signature was changed; it assumes both developers deleted the original method and added two new methods with different signatures. Contrasting, Figure 15(b) illustrates a situation incorrectly classified as additional false negative by our metric. Developer A added a comment to the `getTable` declaration, while developer B added an access modifier. These changes clearly do not conflict. Accordingly, as they correspond to different parts of the node representing the `getTable` declaration, semistructured merge does not report conflict. Unstructured merge does report a false positive because the changes occurred in the same text area.

Figure 15 – Unstructured merge conflicts classified as semistructured merge additional false negatives.

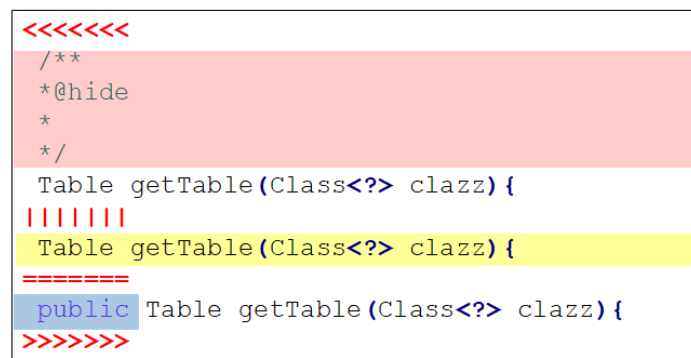


```

<<<<<<
public void forceRepairAsync(final PrintStream out,
final Collection<String> hosts,...) throws IOException
|||||||
public void forceRepairAsync(final PrintStream out,
...) throws IOException
=====
public void forceRepairAsync(final PrintStream out,
boolean fullRepair,...) throws IOException
>>>>>>

```

(a) False Negative



```

<<<<<<
/**
 *@hide
 *
 */
Table getTable(Class<?> clazz) {
|||||||
Table getTable(Class<?> clazz) {
=====
public Table getTable(Class<?> clazz) {
>>>>>>

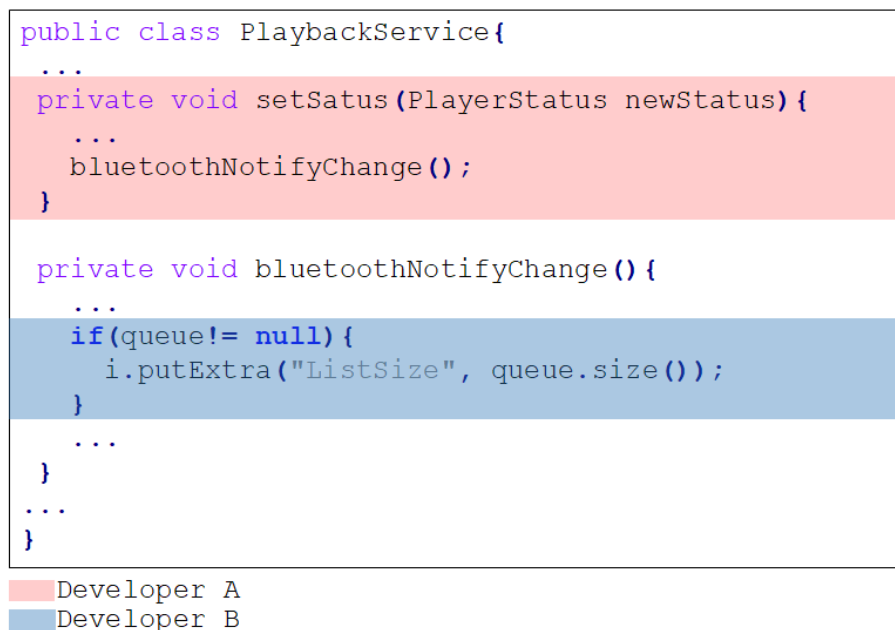
```

(b) False Positive

3.3.2.1 Additional False Negatives of Semistructured Merge are Harder To Detect and Resolve

When comparing false negatives, at first thought, the reasoning seems straightforward because the greater the number of false negatives, the greater the number of post-merge build and behavioral errors. Therefore, the weaker the correctness guarantees of the merging process. In that sense, the achieved results suggest that unstructured merge beats semistructured merge for most, but not all, scenarios and projects. However, we cannot ignore that some bugs are more critical than others, and that build problems can be automatically detected, while behavioral problems are often hard to detect. In particular, unstructured merge additional false negatives cause compilation errors, guiding the developers toward the location and cause of the problem. Conversely, semistructured merge additional false negatives might involve subtle errors. For instance, when one developer adds a new call to a method edited by the other developer, there is no compilation error, but there might be a behavioral issue. That is, the changes made by one developer might affect the behavior expected by the other. This situation is illustrated in Figure 16, showing code from the project ANTENNAPOD. The developer who added the `setStatus` method might not be expecting the extra notification added by the other developer on the `bluetoothNotifyChange` method (referenced by the first developer). We believe that, in such cases, the detection and resolution of the issue is likely more difficult and demands more effort. In situations like that, while semistructured always miss a conflict because changes correspond to different nodes, unstructured merge might *accidentally* detect a conflict depending on changes' area.

Figure 16 – Observed new element referencing edited one.



```

public class PlaybackService{
    ...
    private void setStatus (PlayerStatus newStatus){
        ...
        bluetoothNotifyChange();
    }

    private void bluetoothNotifyChange () {
        ...
        if(queue!= null){
            i.putExtra("ListSize", queue.size());
        }
        ...
    }
    ...
}

```

Developer A

Developer B

Summary: We found no evidence that semistructured merge has fewer false negatives than unstructured merge. Moreover, we argue that semistructured merge additional false negatives are harder to detect and resolve.

3.4 IMPROVING SEMISTRUCTURED MERGE

Even considering that our comparison process favors unstructured merge whenever we are not able to precisely classify a reported conflict, our findings about conflicts and false positives’ reduction are hardly sufficient to justify adoption of semistructured merge in practice. Practitioners might still be reluctant to adopt semistructured merge because of the risk of loss in part of the merge scenarios, and also because of the extra risk and complexity associated to its false negatives. In fact, if renaming is a common practice in a project, developers might have to deal with too many false positives renaming conflicts if they opt for semistructured merge. Similarly, if project changes often occur in the same text area, conflicts that would otherwise be detected by unstructured merge might escape the merging process. Our findings, nevertheless, shed light on how merge tools can be improved. They help us to better understand the technical justification that might prevent the adoption of semistructured merge tools in practice, and motivates us to propose an improved tool. So, we benefit from that and propose a merge tool that further combines both merge strategies to reduce the false positives and false negatives of semistructured merge.

3.4.1 Improvements

Our improved merge tool implements the algorithms underlying the scripts we used to detect false positives and false negatives in our empirical analysis (see Section 3.2.2).⁷ On top of a more efficient version of the *FSTMerge* tool (APEL et al., 2011) we implemented, we added a module (or *handler*) for semistructured merge additional false positives and three kinds of additional false negatives. After the merged tree is constructed, each handler updates the tree according to specific analyses based on information gathered during tree construction. We now detail these handlers.

3.4.1.1 Handling Renaming Conflicts

As explained in Section 3.1.2, semistructured merge additional false positives in relation to unstructured merge are due to renaming of elements. They occur because semistructured merge’s superimposition is not able to map the renamed element to their counterparts, and considers the renaming as a deletion instead. So, for instance, when one of the developers

⁷ Publicly available at <<https://github.com/guilhermejccavalcanti/jFSTMerge>>.

edits the body of an existing method declaration from the base revision, and the other developer modifies its signature, a conflict is reported.

The *renaming handler* first uses the *Levenshtein distance algorithm* (LEVENSHTEIN, 1966) to map renamed elements, with a 70% degree of textual similarity between the elements.⁸ This algorithm measures the difference between two strings sequences (in our case, we consider the textual representation of the elements). In practice, this means the minimum number of single-character edits (insertions, deletions or substitutions) required to change one string into the other. For instance, considering the example in Figure 6, there is a degree of similarity of 86% between the textual representation of the node representing the base version of the method `calculate`, and the method `sum` added by right. As this value is above our adopted 70% threshold, `sum` is selected as the renamed version of the method `calculate`. If more than one element satisfies our 70% degree of similarity, the algorithm picks the element with the greatest value. In case of tie, the algorithm randomly picks one of the tied.

Afterwards, the handler verifies if unstructured merge has reported a conflict with the original element's signature. If that is the case, the improved tool reports the renaming conflict, now filled with information of the renamed version gathered in the first step. In case no element satisfies our 70% degree of similarity, the conflict is reported without the information of the element with the new signature. A renaming conflict is characterized by the presence of the original element's signature in the conflict text. So, if unstructured merge does not report conflict with such signature, there is no guarantee that the renaming conflict is an additional false positive; thus the improved tool does not report the conflict. This is a major concern for the design of our improved tool: ensuring that, whenever possible, it is not worse than an unstructured merge tool, by invoking unstructured merge where the underlying algorithms are not accurate. For example, eliminating such a false positive could result in a false negative, but a common one to unstructured merge.

3.4.1.2 Handling Type Ambiguity Errors

To reduce semistructured merge additional false negatives, we implement three kinds of handlers. Beginning with the additional false negatives related to import declarations, as explained in Section 3.1.4, the issue arises because semistructured merge assumes that import declarations are always re-arrangeable. This might lead to a *type ambiguity error* when the import declarations involve members with the same name but from different packages.

So, the *type ambiguity error handler* uses compiler features to search for compilation problems related to import statements, avoiding all extra false negatives of this kind. The handler checks that by using Eclipse JDT. In particular, during semistructured merge's superimposition, we identify and store pairs of nodes representing introduced or changed

⁸ The degree of similarity adopted by the handlers was chosen on an *ad hoc* basis.

import declarations in the files being merged by semistructured merge. Afterwards, the handler compiles the files merged by semistructured merge, having the identified pair of import declarations, and searches for the compilation problems corresponding to type ambiguity error. In case the handler detects compilation problems, a conflict is reported with the involved import declarations. Note that compilation is a complex feature that impair performance and is not exactly going to work in any situation because of parsing errors. Nevertheless, this is not a major issue for our tool because (1) compilation is only necessary in three merge scenarios of our sample, and (2) when the tool cannot parse a code, it resort to unstructured merge.

The *type ambiguity error handler* also deals with import declarations that might cause behavioral errors instead of compilation problems. Such cases occur when one of the developers imports all members from a package, and the other developer imports a member with the same name of an existing member in the package imported by the first developer (see Section 3.1.4). The handler then checks, with a *grep*-based analysis, if the changes introduced by the developer who imported all members from the package contain the name of the member imported by the second developer. In case of a positive check, the handler reports a conflict involving the import declarations, but only if unstructured merge has reported a conflict with them too. Again, we guarantee that the improved tool is not worse than an unstructured merge tool: if unstructured merge has not reported conflicts with the import declarations, this would be a common false negative of both strategies.

3.4.1.3 Handling Matching of Initialization Blocks

Superimposition uses nodes' identifier to match them with their counterparts. So, for instance, a method declaration can be easily matched through its signature. However, not all elements have identifiers. As seen in Section 3.1.4, when semistructured merge attempts to merge initialization blocks, it cannot because these blocks do not have identifiers, so semistructured merge is unable to match them. This results in blocks being duplicated, even if developers have not changed them. The problem arises when developers do change initialization blocks in a conflicting manner: the conflict is missed by semistructured merge because they are not matched.

To deal with this kind of false negative, the *initialization blocks handler* uses the *Levenshtein distance algorithm* to match triples (from base, and the derived versions) of similar initialization nodes, using 80% of degree of similarity. The handler then invokes unstructured merge to integrate the matched elements, reporting the conflicts it reports. When there is only a single block, this is the match. In case there is no matching among initialization blocks, no conflict is reported. If more than one initialization block is matched, the handler takes the first one with the highest similarity.

Note that using unstructured merge as oracle for this handler was not possible. In par-

ticular, checking if unstructured merge conflicts involve initialization blocks is challenging, as initialization blocks have no identifiers. Thus, there is no precise way to determine if unstructured merge conflicts solely identify an initialization block. This way, trying to reduce false negatives, the improved tool could potentially create new false positives. We opted for this design because of the small risk and reduced impact involved.

3.4.1.4 Handling Accidental False Negatives

In Section 3.1.4 we explain that not all semistructured merge additional false negatives conform to a set of recurring syntactic patterns, they result from unstructured merge accidentally detecting conflicts that would otherwise escape if changes were performed in slightly different text areas. One of these situations occur when developers change or add, in the same text area, different but dependent declarations.

The *new element referencing edited one handler*, with a *grep*-based analysis, checks whether added elements textually reference edited ones. In such situations, there might be a harmful data or control flow between the elements, and, as there is no matching between these elements, a conflict is never reported by semistructured merge. If a reference is found, and unstructured merge also reports a conflict involving the elements, our tool also reports a conflict. This way we eliminate a possible false negative, making sure we are not adding an additional false positive in relation to unstructured merge.

As in this context false negatives might be more disruptive than false positives (BERRY, 2017), increasing the chances of detecting more false negatives at the expense of possible new false positives could be an option for the design of the new tool. However, aiming at reducing trade-offs and facilitating industrial adoption, we opted for a design that attempts to ensure that the new tool is not worse than unstructured merge with regard to false positives. This is why this handler only reports conflicts if unstructured merge reports a similar one.

In future versions, we should investigate the feasibility, benefits and drawbacks of incorporating static semantics information in the trees. For instance, with this kind of information, we could precisely determine that a deleted method by one developer was invoked by an edited method by the other developer (*call to undeclared method declaration*), and report a conflict. Such a feature possibly impairs parsing complexity and performance, so a detailed analysis is necessary to find out the tradeoff between the benefits and drawbacks.

3.4.2 Usability

The tool is universally applicable by simply calling unstructured merge for files it cannot process (invalid Java files or non-Java files). This way, the tool can be used wherever unstructured merge is used, including projects with multiple programming languages. Regarding Java support, the tool uses an updated annotated grammar to support Java 8.

Annotating other languages grammars, and implementing specific false positive and false negative handlers for these languages would make semistructured merge benefits more widely applicable.

We should indeed investigate how applicable to other kinds of languages, and to other parsers and grammar mechanisms, such as ANTLR, semistructured merge is (and, as a consequence, our tool). Also, whether the same implementation of the semistructured algorithm can deal with different languages. For instance, (TRINDADE et al., 2019) attempted to implement semistructured merge for JavaScript, which supports statements at the same syntactic level of commutative and associative elements such as function declarations. They found that current semistructured merge algorithms and frameworks are not directly applicable for scripting languages like JavaScript. By adapting the algorithms, they are still able to implement semistructured merge for JavaScript. However, the gains are much smaller than the ones observed for Java-like languages, suggesting that semistructured merge advantages might be limited for languages that allow both commutative and non-commutative declarations at the same syntactic level.

Our tool can be configured to resolve false positives due to code indentation: it compares elements content ignoring spacings. Finally, after installation, the tool is entirely integrated with git version control system (integration with others VCS is likely not hard too); whenever a user calls the `git merge` command, the tool is automatically invoked, generating results in the same format as default `git merge`. The tool can also be used standalone, likewise available *diff3* tools.

3.4.3 Gains

Based on the sample and results of our empirical evaluation (see Section 3.3), Table 1 summarizes how conflict numbers of the improved tool compare to unstructured merge and the original semistructured merge tool. Considering the aggregated scenarios of all projects, the improved tool reduces the number of reported conflicts by approximately 51% in relation to unstructured merge, and by 36% compared to the original semistructured tool; exploring another viewpoint, the table shows increasing rates. A similar reduction pattern, but with less intensity, can be observed for the number of merge scenarios with conflicts; the table shows the percentages in relation to the total number of scenarios followed by the increasing rates.

Table 1 – Comparing conflict numbers of unstructured, semistructured, and improved tools.

	Unstructured tool	Semistructured tool	Improved tool
Reported Conflicts	19,238 (206%)	14,544 (156%)	9,343 (100%)
Merge Scenarios with Conflicts	2,995 (8.8% / 155%)	2,420 (7.1% / 125%)	1,935 (5.7% / 100%)

With respect to false positives and false negatives, Table 2(a) summarizes the main result discussed in Section 3.3. Contrasting, Table 2(b) shows how the improved tool compares to unstructured merge. In our sample, the improved tool completely eliminates semistructured merge additional false positives. Although our *initialization blocks handler* might lead to false positives, as discussed in Section 3.5, we had no such case. Moreover, as the *new element referencing edited one handler* of the improved tool uses unstructured merge as oracle to reduce false negatives, it might actually have new false positives in common with unstructured merge. In our sample, this corresponds to at most 535 conflicts, in case all conflicts detected by that handler are inaccurate. So, the reported numbers of additional false positives of unstructured merge in relation to the original and improved tool (see both tables) differ in approximately 7%.

Table 2(b) also shows that the improved tool eliminates a few kinds of false negatives, leading to a reduction of approximately 23% in the number of additional false negatives in relation to the original semistructured tool. This way, the improved tool is superior to unstructured merge with respect to the overall number of additional false negatives: it misses at least 8% fewer false negatives than unstructured merge.⁹ Due to the quite conservative nature of our metric, and the results of our manual analysis (which suggests that the semistructured merge false negatives numbers might be around 12% of the reported numbers), we expect the advantage in favor of the improved tool to be much higher. Those advantages of the improved tool do not uniformly hold across projects, but most projects follow this pattern.

Table 2 – Comparing false positives and false negatives numbers of the improved and original tools with unstructured merge. Arrows indicate whether the number is underestimated (\uparrow , meaning the numbers should be bigger in practice) or overestimated (\downarrow).

(a)		
	Unstructured tool	Semistructured tool
Additional False Positives	7,958 \uparrow	5,201 \downarrow
Additional False Negatives	2,714 \uparrow	3,260 \downarrow

(b)		
	Unstructured tool	Improved tool
Additional False Positives	7,423 \uparrow	0
Additional False Negatives	2,714 \uparrow	2,489 \downarrow

Notice that only part of the false negatives is eliminated with the improved tool. The false negatives that were not eliminated are hard to detect because they do not follow a syntactic pattern. Our oracle for detecting them in the study relies on analyzing the merged code, which is appropriate for our retrospective analysis, but not for using

⁹ Remember that our metrics are approximations.

the tool in practice. So a handler could not rely on that. For instance, as explained in Section 3.2.2.4, when the resolution of an unstructured merge conflict does not keep all developers' changes, or adds new code, we consider the conflict as a semistructured merge false negative. However, this information is only available after the merge result has been committed. Finally, as the *renaming handler* uses unstructured merge as oracle to reduce false positives, it might lead to new common false negatives with unstructured merge. However, as our metric of unstructured merge false negatives does not include renaming conflicts, the corresponding numbers are the same in both tables.

Summary: In our sample, the improved semistructured tool, when compared to unstructured merge, reduces the number of reported conflicts, has no additional false positives, and has fewer false negatives.

3.4.4 Performance Evaluation

Although performance was not a priority for our design, we evaluated the improved tool performance on a random subsample of 1731 merge scenarios from 25 projects. This is a subset of the full sample described in Section 3.2.1, considering only scenarios that reported at least one conflict, regardless of the merge strategy. For each merge scenario, we invoked the original, improved and unstructured tools, 5 times each, measuring execution time. We conducted the evaluation on a desktop machine with Intel Core i5, 4 cores @4.0 GHz, 16 GB RAM and Windows 10 64 bits.

Taking the median of the measured times, the improved tool took approximately 24 minutes to merge the entire sample, compared to only 45 seconds of the unstructured merge tool. Note that our implementation could be optimized in a number of ways to reduce this large difference. For example, we explore no parallelization, and merge files sequentially. However, due to the handlers and complexity of the tree merging algorithm we use, we expect that an optimized tool would still be much slower than unstructured merge. It is though, much faster than the original semistructured merge tool, which took 123 minutes to merge the entire sample. We observed that the original tool has severe performance issues due to its prototype nature. In particular, the original tool creates a single representation (tree) in memory for all files in the merge scenario— including unchanged files, and files differing only by spacing. This leads to complex trees with expensive node matching, and slow tree merges. We address this issue by creating a tree per changed merged file, ignoring files that only had spacing related changes. We kept the tree merging algorithm, but the fewer and simpler trees substantially improve performance. These optimizations more than compensate the extra complexity associated to the false positives and false negatives detection handlers we implemented.

We also observed that, in practice, the performance difference between the improved and unstructured tool is often non prohibitive. In more than 80% of the scenarios, our tool took less than 1 second to merge the involved files. It took more than 5 seconds in only 2% of the scenarios, with a maximum of 67 seconds in a LUCENE-SOLR scenario that merges 303 files, resulting in 618 conflicts and 17,567 LOC of conflicting code. For the same scenario, unstructured merge took 6 seconds, resulting in 866 conflicts and 27,397 LOC of conflicting code. In our sample both the improved and unstructured tool spent, on average, less than 1 second per merge scenario ($0,83 \pm 2,47$ seconds with the improved tool, compared to $0,03 \pm 0,09$ seconds with unstructured merge, but $4,27 \pm 14,75$ seconds with the original tool).

Summary: The improved semistructured tool, when compared to unstructured merge, is not prohibitively slower in our sample.

3.5 THREATS TO VALIDITY

Our empirical analyses and evaluation naturally leave open a set of potential threats to validity explained in this section.

Our analysis of integration effort is based on the number of false positives reported by the merge strategies, the nature of renaming and ordering conflicts, and how contributions were integrated in the project repositories. Further analysis involving integrators would be important to reinforce our conclusions. Also, a more rigorous analysis based on conflict detection and resolution timing data, in the spirit of Berry (2017), could differently weight false positives and false negatives and better assess integration effort reduction. We also conducted a manual analysis to estimate the impact of false positives on integration effort. As explained in Section 3.3, the number of analyzed cases (50) is appropriate considering margins of error of 10% and 15%, respectively, for renaming and ordering conflicts. Reducing the margins of error would increase the accuracy of our conclusions, but would substantially increase the number of cases to manually analyze. Note, however, that the adopted margins are safe for the reached conclusions, which show large differences.

As our metrics are approximations, one can argue that we perhaps could not compare them properly, but the achieved results allowed us to make useful comparisons, especially in the case of false positives. The main issue is related to the semistructured merge additional false negatives metric; its upper bound is too high. We confirmed that by manually inspecting a small sample of merged code. As a consequence of the approximations we use, the obtained percentages should not be interpreted as the expected percentages of additional false positives and false negatives, but only as sufficient evidence to support our conclusions. In addition, having ground truth about integration conflicts (and therefore false positives and false negatives) would give a better idea of how relevant is the gains

of semistructured merge. In particular, there might be many false positives and false negatives common to both strategies, so the presented gains may not be representative in relation to the total (given by the ground truth). Also, the gains of our improved tool must be interpreted with caution, because they came from the same sample from which its improvements are based on. So a more accurate evaluation should consider a different sample. Still regarding our approximations, we check conflict resolution in projects repository to classify semistructured merge additional false negatives (see Section 3.2.2.4). We assume that the developer correctly analyzed the conflict and decided that there was no problem, so that would be an unstructured false positive, not a semistructured false negative. The problem is that the changes might still lead to semantic problems not perceived by the integrator, missing actual conflicts.

As described in Section 3.2.2.4, when textually similar initialization blocks conflict with unstructured merge, we consider the resulting number of conflicts as the number of semistructured merge additional false negatives related to initialization blocks. However, this might not be safe because the triple of initialization blocks that should be matched might involve elements with less than the adopted degree of similarity (80%). To check this was not a problem in our sample, we manually analyzed all files merged by unstructured merge having at least one reported conflict and edited initialization blocks. These necessary conditions for false negatives of this kind were satisfied only by 40 files in our sample. In this analysis, we checked if the reported conflicts involve the same initialization blocks matched by our similarity threshold. If the conflicts involve different initialization blocks, our metric could be missing actual conflicts. This only happened in a single file of the CASSANDRA project (`Cassandra.java`, an atypical file with more than 40K lines of codes and more than 50 conflicts). Nevertheless, further inspection of this file showed us that the involved initialization block was not edited, and that the corresponding conflict was actually a crosscutting conflict—the conflict was an unstructured merge additional false positive. In all other files, unstructured merge did not report conflicts involving edited initialization blocks other than those accused by our metric.

Different degrees of textual similarity was adopted not only in the empirical evaluation, but also in the design of our improved semistructured merge tool (see Section 3.4.1). Whereas the adopted thresholds values were based on an *ad hoc* analysis, we could address this systematically to find optimal values for the different situations in which we rely on textual similarity.

We analyze public Git repositories that might have suffered the effect of commands such as `rebase` and `cherry-pick`, which rewrite project history (BIRD et al., 2009). Consequently, depending on the development practices of each project, we may have lost merge scenarios where developers had to deal with merge conflicts, but that do not appear on Git history as merge commits. When those commands are used in a systematic way, they dramatically decrease the number of merge commits. Consequently, to analyze all merge scenarios, we

would need to have access to developers individual repositories. Therefore, our sample actually corresponds to part of the conflicts that actually happened in the analyzed projects. The impact of an increased sample on the results presented here is hard to predict, but we are not aware of factors that could make the missed conflicts different from the ones we analyzed.

Additionally, we had to discard Java files that could not be parsed by the semistructured tool used in the study. So one could argue that we bias the results in favor of the semistructured merge strategy because we actually miss the false positives and false negatives present in the discarded files. However, we found that this corresponds only to 0.16% of the total number of Java files in our sample. We believe this has insignificant impact on our results. We also discarded non-Java files from the analyzed projects. This corresponds to 1.73% of the total number of files in the sample. Per project, we discarded on average $15.33\% \pm 19.46\%$ files. Projects substantially differ in the overall number of files and non-Java files. For instance, JUNIT has 539 files, with only 0.90% of non-java files, whereas CLOJURE has 308 files, with 49.5% of non-java files. So, the results for projects such as CLOJURE could be different if one considers all kinds of files. Nevertheless, for both non-Java and invalid Java files, semistructured merge could be easily adapted to invoke unstructured merge, similarly as it does to method bodies. As a consequence, the strategies would behave identically and, therefore, present the same numbers for these files.

Finally, although the semistructured merge tool used in this study supports more languages, we restricted our sample to Java projects because our setup demands language dependent tool implementation and configuration. However, all the false positives and false negatives analyzed here are also likely to happen in projects written in other class based languages similar to Java. Besides, although our sample has a considerable degree of diversity (see Section 3.2.1), it would be important to systematically address diversity in further studies, including new dimensions such as programming languages. Finally, we only explored open source projects, but we are not aware of factors that could make conflicts present in projects of different nature unlike the ones we analyzed.

4 THE IMPACT OF STRUCTURE ON SOFTWARE MERGING: SEMISTRUCTURED VERSUS STRUCTURED MERGE

We found evidence that semistructured merge has significant advantages over unstructured merge in Chapter 3. Besides that, there is evidence that structured merge tools report significantly less conflicts than unstructured merge (APEL; LESSENICH; LENGAUER, 2012). However, it is unknown how semistructured merge compares with structured merge. (APEL; LESSENICH; LENGAUER, 2012) argue that structured tools are likely more precise than semistructured tools, and conjecture that a structured tool reports fewer conflicts than a semistructured tool. However, as discussed before, the reduction of reported conflicts alone is not enough to justify industrial adoption of a merge tool, as the reduction could have been obtained at the expense of missing actual conflicts between developers contributions. So, before deciding to replace state of practice unstructured tools by semistructured merge, we need to investigate whether structured merge is a better option than semistructured merge.

In fact, although one might expect only accuracy benefits from the extra structure exploited by structured merge, we have no guarantees that this is the case. In Chapter 3, we found evidence that the extra structure exploited by semistructured merge is not only beneficial: it helps to eliminate certain kinds of spurious conflicts (false positives) reported by unstructured merge, but it might introduce others that can only be solved by our solution that further combines semistructured and unstructured merge (see Section 3.4). Similarly, the extra structure helps semistructured merge to detect conflicts that are missed (false negatives) by unstructured merge, but it unfortunately comes with new kinds of false negatives. So, it is imperative to investigate whether the same applies when comparing semistructured and structured merge, as this is essential for deciding which kind of tool to use in practice.

So, to compare and better understand the differences between semistructured and structured merge, we run both strategies on more than 40,000 merge scenarios from more than 500 Java projects. In particular, we assess how often the two strategies report different results, and we identify false positives (conflicts incorrectly reported by one strategy but not by the other) and false negatives (conflicts correctly reported by one strategy but missed by the other). In particular, in this chapter we answer the following research questions:

- **RQ1:** *How many conflicts arise from the use of semistructured and structured merge?*
- **RQ2:** *How often do semistructured and structured merge differ with respect to the occurrence of conflicts?*
- **RQ3:** *Why do semistructured and structured merge differ?*

- **RQ4:** Which strategy reports fewer false positives?
- **RQ5:** Which strategy has fewer false negatives?
- **RQ6:** Does ignoring conflicts caused by changes to consecutive lines make the strategies more similar?

4.1 SEMISTRUCTURED AND STRUCTURED MERGE

As introduced in Section 2.3, the main difference between semistructured and structured merge is the exploited syntactic granularity. Merging programs for the structured strategy consists of traversing trees to find the differing nodes, much similar to semistructured merge (APEL; LESSENICH; LENGAUER, 2012). The difference, however, is that, while semistructured merge builds a syntax tree for the source code until the level of declarations, for instance, method declarations, representing statements and expressions in the syntax level of method body as plain text, structured merge builds a syntax tree for the entire source code.

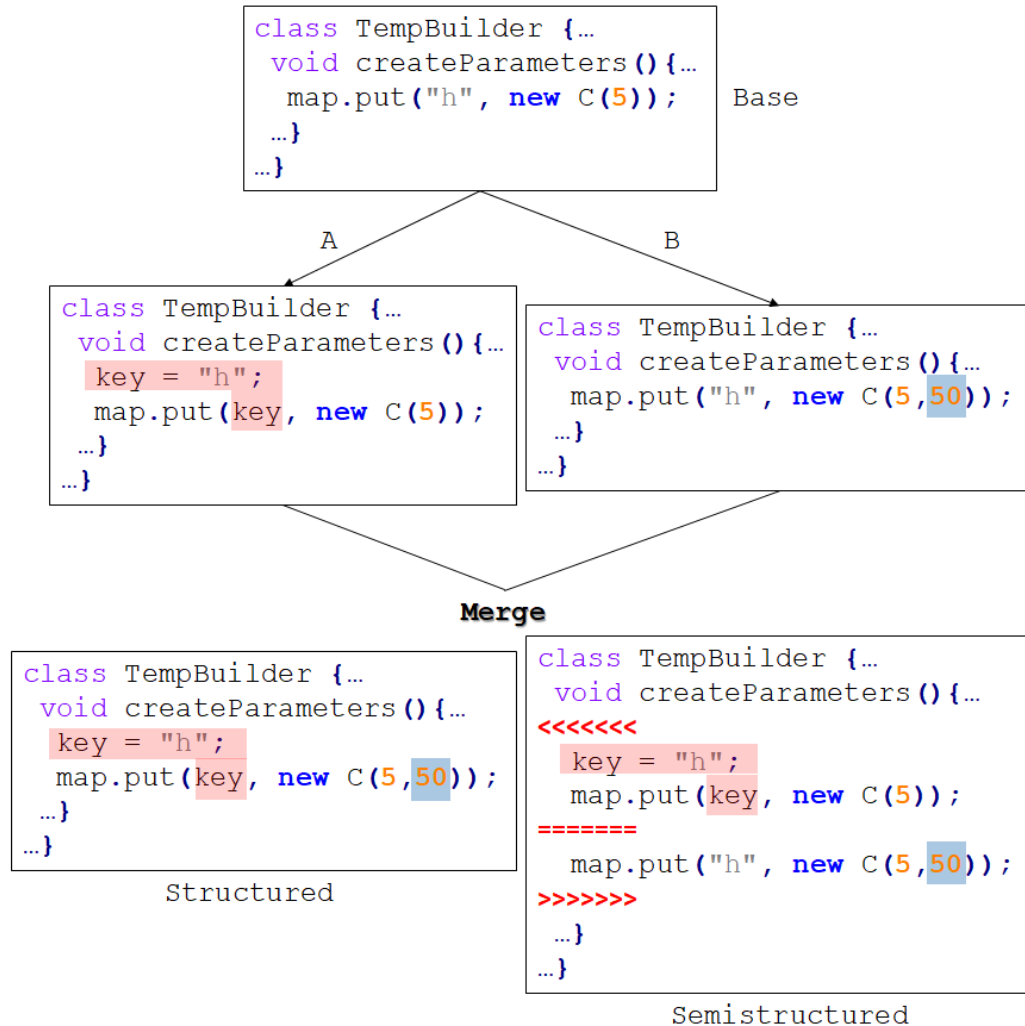
When comparing the input trees level-wise, a key point for structured merge is the distinction between ordered nodes (which must not be permuted) and unordered nodes (which can be permuted safely). For ordered nodes, the positions relative to the parent node are decisive: if they overlap, the nodes are flagged as conflicting. Whether unordered nodes are in conflict, depends on their type and name. The matching of nodes depends on their syntactic category. For instance, two method declarations are matched if their signatures are equal, again similar to semistructured merge. That similarity with respect to unordered nodes, between semistructured and structured merge, implies that structured merge shares semistructured merge false positives and false negatives described in Chapter 3. Besides, structured merge is also able to resolve unstructured merge ordering conflicts, and to merge duplicated declarations.

In summary, semistructured and structured merge differ only on how they represent the bodies of method, constructor, and field declarations. In a structured tool, such bodies are also represented as AST nodes. In a semistructured tool, they are represented as strings, and are merged by unstructured merge line-based algorithm. We illustrate how this difference impacts merging in Figure 17, which shows different versions of part of a method body.¹ The *base* version at the top shows a method call that adds a new key-value entry to a map. The *structurally merged* version at the bottom highlights, in red, the changes made by developer *A*, who simply refactored the code by extracting `key`. It also highlights, now in blue, the changes made by developer *B*, who added an extra argument to the constructor call. As the two developers changed different AST nodes from the *base* version, corresponding to different arguments of the method call, structured merge

¹ Based on method `createDefaultParametersToOptimized` merged in merge commit <<https://git.io/fjneH>> from our sample.

successfully integrates their changes. Contrasting, semistructured merge reports a conflict because the two developers changed the same line of code in the method body.

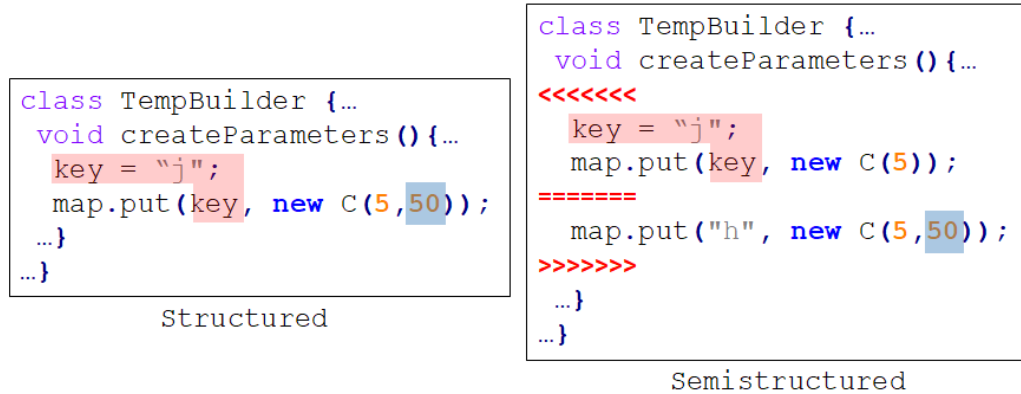
Figure 17 – Merging with Semistructured and Structured Merge (False Positive).



As explained in Section 2.3, to compare semistructured and structured merge, we could simply measure how often they are able to merge contributions as in the illustrated example. The preference would be for structured merge: the strategy that reports fewer conflicts. Given that merging contributions is the main goal of any merge tool, in principle that criterion could be satisfactory. However, in practice, merge tools go beyond that and detect other kinds of integration conflicts that do not preclude the generation of a valid program, but would lead to build or execution failures. For instance, consider the situation now illustrated on Figure 18, where developer *A*, besides extracting the key variable, also changed its value to "j". The merge tools would behave exactly as in the original example. In this case, however, the changes interfere (HORWITZ; PRINS; REPS, 1989), and the behavior expected by *A* (new key with old value) and *B* (old key with new value) will not be observed when running the integrated code. In this case, the preference

then would be for a semistructured tool, the tool that reports a conflict when integrating these changes.

Figure 18 – Merging with Semistructured and Structured Merge (True Positive).



Whereas the original example in Figure 17 illustrates semistructured merge reporting a *false positive* (incorrectly reported conflict), the modified example illustrates a structured merge *false negative* (missed conflict). This shows that our comparison criteria should go beyond comparing the number of reported conflicts. We should also consider the number of false positives and false negatives, that is, the possibility of missing or early detecting conflicts that could appear during building or execution. Such comparison should be based on the differences between the merge strategies. By definition, they differ only when merging the bodies of method, constructor, and field declarations.

4.2 RESEARCH QUESTIONS

To quantify the differences between semistructured and structured merge, and to help developers decide which strategy to use, we analyze merge scenarios from the development history of a number of software projects, while answering the following research questions.

RQ1: *How many conflicts arise when using semistructured and structured merge?*

To answer this question, we integrate the changes of each merge scenario with semistructured and structured merge. For the results of each strategy, we count the total number of conflicts, that is, the number of conflict markers in the files integrated by each strategy. We then count the number of conflicting merge scenarios, which means scenarios with, at least, one conflict, with semistructured or structured merge. As explained later in Section 4.3.2, to control for undesired variations on individual tools implementation, we have implemented a single configurable tool that, via command line options, selects a semistructured or structured merge strategy.

RQ2: *How often do semistructured and structured merge differ with respect to the occurrence of conflicts?*

We answer this question by measuring the number of merge scenarios having conflicts reported by only one of the strategies. In principle, the strategies could still differ when they both report conflicts for the same scenario, as the reported conflicts might be different. However, by definition, both strategies report the same conflicts occurring outside of method, constructor, and field declarations. We also observed that, when semistructured and structured merge reports conflicts on the same scenarios, the conflicts occurring inside such declarations are exactly the same or contain slightly different text among conflict markers, but are essentially the same conflict in the sense that they report the same issue. Similarly, we observed equivalent conflicts that are reported with a single marker by semistructured merge, but involve a number of markers in structured merge as illustrated later.

RQ3: *Why do semistructured and structured merge differ?*

We answer this question to better explain the differences quantitatively explored by the previous question. We manually inspect merge scenarios and the code merged with each strategy for a sample of scenarios that have conflicts reported by only one of the strategies, so we can understand the difference on the behavior of the two strategies that leads to diverging conflicts.

RQ4: *Which of the two strategies reports fewer false positives?*

A merge tool might report spurious conflicts in the sense that they do not represent a problem and could automatically be solved by a better tool. These are false positives, which lead to unnecessary integration effort and productivity loss, as developers have to manually resolve them. To capture true positives, as explained in Section 2.3, we rely on the notion of interference by Horwitz et al. (HORWITZ; PRINS; REPS, 1989), who state that two contributions (changes) to a base program interfere when the specifications they are individually supposed to satisfy are not jointly satisfied by the program that integrates them. This often happens when there is, in the integrated program, data or control flow between the contributions. We then say that two contributions to a base program are conflicting when there is not a valid program that integrates them and is free of interference.

As interference is not computable in our context (BERZINS, 1986; HORWITZ; PRINS; REPS, 1989), we rely on build and test information about the integrated code we analyze,

and, when necessary, resort to manual analysis. Again, we focus on scenarios that have conflicts reported by only one of the strategies; so only one of the strategies produced a clean merge. We attempt to build the clean merge and run its tests. If the build is successful and all tests pass, we manually analyze the clean merged code to make sure the changes do not interfere; passing all tests are a good approximation, but no guarantee that the changes do not interfere, as a project's test suite might not be strong enough, or even do not cover the integrated changes. If we find no interference in the clean merge, we count a scenario with false positive for the strategy that reported a conflict.

RQ5: *Which of the two strategies has fewer false negatives?*

A merge tool might also fail to detect a conflict (false negative). When this happens, the users would be simply postponing conflict detection to other integration phases such as building and testing, or even letting conflicts escape to operation. So, false negatives lead to build or behavioral errors, negatively impacting software quality and the correctness of the merging process. Similarly to RQ4, we rely on build and test information to identify false negatives. We attempt to build the clean merge and run its tests. If the build breaks or, at least, one test fails due to developers changes (when the base version and the integrated variants do not present build or test issues, but the merge result has issues, so the changes cause the problem), the strategy responsible for the clean merge has actually missed a conflict (false negative). Thus, we count a scenario with false negative for the strategy that yielded the clean merge.

It is important to emphasize that RQ4 and RQ5 consider only the differences between semistructured and structured merge strategies. Our interest here is to relatively compare both strategies— not to establish how accurate they are in relation to a general notion of conflict. So, we do not need to measure the occurrence of false positives and negatives when both strategies behave identically.

RQ6: *Does ignoring conflicts caused by changes to consecutive lines make the two strategies more similar?*

In the example of Figure 17, semistructured merge reports a conflict because developers *A* and *B* changed the same line in a method body. However, even if *A* had simply added a single line (even a comment like `//updating the map`) before the method call, semistructured merge would report a conflict. This happens because the invoked unstructured merge reports a conflict whenever it cannot find a line that separates developers changes. As in the example, structured merge would successfully integrate the changes. Assuming that changes to the same line are often less critical than changes to consecutive lines, it would be important to know whether a semistructured tool that

resolves consecutive lines conflicts would present closer results to a structured tool. So, to answer this question, we determine whether a semistructured merge conflict is due to changes in consecutive lines of code — that is, one of the developers changes line n and the other changes line $n + 1$. Then, for each merge scenario, we assess the number of reported conflicts by semistructured merge, and how many of these conflicts are in consecutive lines. Finally, answering this research question consists of revisiting previous research questions contrasting results with and without consecutive lines conflicts.

4.3 EMPIRICAL EVALUATION

To answer our questions and compute the related metrics, we adopt a two-step setup: mining and execution. In the mining step, we use tools that mine GitHub repositories of Java projects to collect merge scenarios— each scenario is composed by the three revisions involved in a three-way merging process associated with a merge commit, that is, a base commit and the two parents of the merge commit.

In the execution step, we merge the selected scenarios with both semistructured and structured merge. For each resulting merge free of conflicts, we use a build manager to build the merged version and execute its tests, as this might helps us to find false positives and false negatives. We now detail these steps. All the scripts and data used in this study are available in our online appendix (CAVALCANTI, 2019).

4.3.1 Mining Step

Regarding projects sampling, as our experiment relies both on the analysis of source code and build status information, for service popularity reasons (Zhao et al., 2017) we opt for GitHub projects that use Travis CI for continuous integration. Besides, as the merge tool used in the execution step is language dependent, we consider only Java projects. Similarly, as parsing Travis CI’s build log depends on the underlying build automation infrastructure, we analyze only Maven projects because we use its log report information for automatically filtering conflicts. Considering more languages, build systems, and CI services would demand significantly more implementation effort.

We start with the projects in the datasets of (MUNAIAH et al., 2017) and (BELLER; GOUSIOS; ZAIDMAN, 2017), which include numerous carefully selected open source projects that adopt continuous integration. From these datasets, we select Java projects that satisfy two criteria. First, the presence of Travis and Maven configuration files, which indicates that the project is configured to use the Travis CI service, and that the project uses the Maven build manager.² Second, the presence of, at least, one build process in the Travis CI service, and confirmation of its active status, which indicates the project has actually

² We check if the repository contains both Travis CI and Maven configuration files: `.travis.yml` and `pom.xml`.

used the service. It is important to note that we opt for a different sample of projects, instead of the ones in Chapter 3, because many of the projects in Chapter 3 do not satisfy the selection criteria described above.

After selecting the project sample, we execute a script that locally clones each project and retrieves its merge commit list—a merge commit represents a merge in the project history, and therefore can be used to derive a merge scenario (the parents of such a commit, together with their most recent common ancestor). As most projects adopted Travis CI only later in project history, we only consider merge commits dated after the first project build on Travis. For each scenario derived from these merge commits, we check the Travis CI status of the scenario’s three commits. If any of them has an *errored* (indicates a broken build) or *failed* status (indicates failure on tests), we discard the scenario, as we would not be able to confirm whether a problem in the merged version was caused by conflicting changes, they could well have been inherited from the parents, in this case.

As a result of the mining step, we obtained 43,509 merge scenarios from 508 selected Java projects. Although we have not systematically targeted representativeness or even diversity (NAGAPPAN; ZIMMERMANN; BIRD, 2013), we argue that our sample has a considerable degree of diversity concerning various dimensions. Our sample contains projects from different domains, such as APIs, platforms, and network protocols, varying in size and number of developers. For example, the TRUTH project has approximately 31 KLOC, while JACKSON DATABIND has more than 100 KLOC. Moreover, the WEB MAGIC project has 45 collaborators, while OKHTTP has 195. We provide a complete list of the analyzed projects in our online appendix (CAVALCANTI, 2019).

4.3.2 Execution Step

After collecting the sample projects and merge scenarios, we merge the selected scenarios with both semistructured and structured merge. To control for undesired variations, we have implemented a single configurable tool that, via command line options, selects semistructured or structured merge. This way we guarantee that structured merge behaves exactly as semistructured merge except for merging the body of method, constructor, and field declarations. The new implementation adapts and improves previous and independent implementations of a semistructured (see Section 3.4) and a structured merge tool.³

In particular, our tool is built on top of the semistructured tool. While a standard semistructured merge tool invokes unstructured merge to merge those declarations body, our configurable tool also allows structured merge to be invoked on declarations body. As explained in Section 4.1, by construction, semistructured and structured merge differ only when merging the bodies of method, constructor, and field declarations. Our tool ensures this behavior. Thus, when configured as semistructured merge, the tool invokes the standard and widespread *diff3* unstructured algorithm. When configured as structured

³ <https://github.com/se-passau/jdime>

merge, the tool invokes, to the best of our knowledge, the most mature and extensively evaluated structured implementation (APEL; LESSENICH; LENGAUER, 2012; LESSENICH; APEL; LENGAUER, 2015; LESSENICH et al., 2017; ZHU; HE, 2018; ZHU; HE; YU, 2019).

The tool takes as input the three revisions that compose a merge scenario and attempts to merge their files. For each merge scenario, the tool generates two merged versions: a semistructured version and a structured version. For each file in such versions, we count the number of reported conflicts. For the semistructured merge versions, we also count the number of conflicts that are due to changes in consecutive lines. To do so, we check whether the sets of changed lines in the variants are disjoint, and whether the numbers of the contribution lines in the conflict text are consecutive.⁴

With the number of conflicts in each merged version, we select scenarios having conflicts reported by only one of the strategies. The strategies could also differ by reporting different conflicts for the same scenario, as discussed earlier in Section 4.2. In our sample, however, we verified that whenever semistructured and structured merge report conflicts in the same scenario, these conflicts are in the same file. Even so, they could still report different conflicts in the same file. We have, in fact, observed such cases, but they actually are equivalent conflicts, reported by the strategies in different ways, using different sets of markers and associated conflicting text. So, we can consider them to be the same conflict, but with different textual representations derived from the difference in the exploited syntax granularity. This is illustrated in Figure 19, in a merge scenario from the NEO4J-FRAMEWORK project. In this example, both developers added different declarations for the same constructor. As this constructor is not declared in the base version, both strategies report conflicts. Structured merge reports a conflict for any two syntactic level differences between the versions, resulting in several small conflicts. Contrasting, semistructured merge reports a single conflict for the entire declaration.

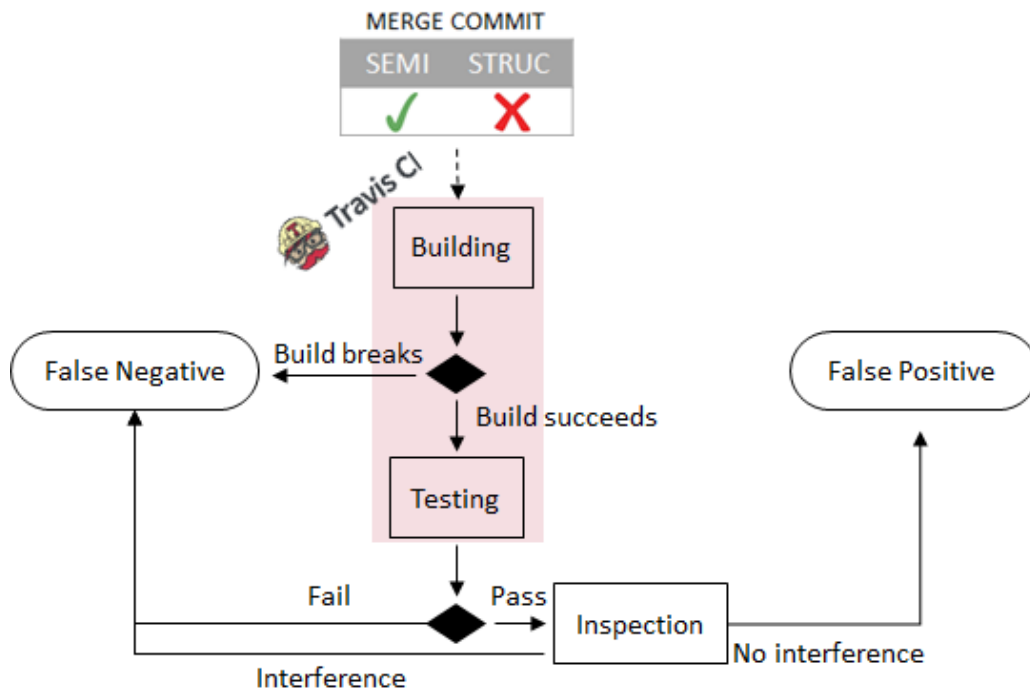
Figure 19 – Equivalent conflicts with different granularity.

Semistructured	Structured
<pre> <<<<<< public CrawlerRM(...) {... this.inSt = inSt; ...} ===== public CrawlerRM(...) {... this.reInSt = reInSt; ...} >>>>>> </pre>	<pre> public CrawlerRM(...) {... this. <<<<<< inSt ===== reInSt >>>>>> = <<<<<< inSt ===== reInSt >>>>>> ; ...} </pre>

⁴ We use GNU's diff command passing the base version and each variant separately.

Having identified scenarios for which the strategies differ, we then collect evidence of false positives and false negatives. We use Travis CI as our infrastructure for building and executing tests for each such scenario, as explained in Section 4.2 and illustrated in Figure 20. As Travis CI builds only the latest commit in the push command or pull request, not all commits in a project have an associated build status on Travis CI. The generated semistructured and structured merged versions certainly do not have a Travis CI build, as they are generated by our experiment. Because of that, we use a script that forces build creation for one of the merged versions. Basically, we create a project fork, activate it on Travis CI, and clone it locally. Then, every push to our remote fork creates a new build on Travis CI. So, for each scenario for which the strategies differ (by definition one of the merged versions is clean and the other is conflicting), we create a merge commit with the clean merged version, and push it to our remote fork to trigger a Travis CI build. Note that we are only able to build and test code without conflicts, as the conflicts markers invalidate program syntax.

Figure 20 – Building and testing merge commits. A green check mark indicates no conflict with one strategy, a red cross indicates conflict with the other strategy.



If the build status on Travis CI of the resulting merge commit is *errored*— when the build is broken— or *failed*— when the build is ok, but, at least, one of the tests failed— we consider that the corresponding merge scenario has a false negative from the strategy that did not report a conflict; therefore a true positive reported by the other strategy. However, it is possible that a build breaks or a test fails due to external configuration problems such as trying to download a dependency that is no longer available, or exceeding the time to execute tests. So, we filter these cases as they do not reflect issues caused by conflicting

contributions. To do so, we analyze, for each generated build, its Maven log report seeking for indicative message errors. Finally, since we have also filtered merge scenarios having problematic parents (see Section 4.3.1), if the new merge commit still has build or test issues, we can conclude that this is because developers changes interfere.

In case the resulting merge commit build status on Travis CI is *passed*, we are sure that the merged version has no build error, and all tests pass. Thus, this is a candidate false positive of the strategy that reported conflict. However, whereas this provides precise guarantees for build issues, the guarantees for test issues are as good as the project test suite. Even for projects with strong test suites, unexpected interference between merged contributions might be undetected by the existing tests. So, to complement test information, we manually inspect all conflicting files from all merged versions having potential false positives. In this manual analysis, two persons analyzed the first 5 conflicting files to consolidate the guidelines. Then, two other persons individually analyzed the remaining files. In case of divergence between individuals' classification for the same file, another person reviewed that file. In case of uncertainty regarding the contributions, a message was sent to the original committers to clarify the changes. We provide a sheet with the detailed analysis of all files in our online appendix.

During this manual analysis, we check the changes made by each developer, analysing whether they interfere, following the definition of interference in Section 2.3. If one of the developers does not change program semantics, such as when refactoring or simply changing comments, we consider that there is no interference. The corresponding merge scenario is then confirmed as having false positives. The same applies when the developers change unrelated state, or when they change assignments to unrelated local variables. Conversely, if both developers change program semantics, such as when modifying related state, or when changing assignments to the same variable, we consider that there is interference. We then conclude that the corresponding merge scenario has a false negative. As discussed in Section 4.1, the same applies to the variation of the example illustrated in Figure 17. For each merge scenario in which we find interference in the merged version, we add explanation and discuss a test case that fails in the base commit, passes in one of the parent commits, and fails in the merged version. This is further evidence that the changes made by the considered parent commit were affected by the changes of the other parent commit.

4.4 RESULTS

By executing the study design presented in the previous section, we analyze 43,509 merge scenarios from the development history of 508 Java projects. We compare semistructured and structured merge concerning a number of dimensions. In this section, we present the results, following the structure defined by our research questions. Detailed

results for the analyzed projects, including tables and plots, are available in our online appendix (CAVALCANTI, 2019).

4.4.1 How many conflicts arise when using semistructured and structured merge?

In our sample, we observed 4,732 conflicts when using semistructured merge, and 4,793 when using structured merge. This represents a reduction of 1.27% in the number of reported conflicts when using semistructured merge. Such result, at first, might be surprising to those that expect that more structure leads to conflict reduction. However, as pointed out in Section 4.3.2 and illustrated in Figure 19, structured merge might report more conflicts due to its structure-driven and fine-grained approach for dealing with declarations bodies, including expressions and statements. Structured merge leads to conflicts that respect the boundaries of the language syntax, but might result in many small conflicts that are reported as a single conflict by semistructured merge.

To control for the bias of conflict granularity, we consider also the number of merge scenarios with conflicts: 1,007 (2.31% of the scenarios) using semistructured merge, and 814 (1.87%) using structured merge. This time we observe a reduction of 19.17% in the number of scenarios with conflicts when using structured merge. In a per-project analysis, we observe similar results: $2.25 \pm 4.58\%$ (average \pm standard deviation) of conflicting scenarios with semistructured merge, and $1.8 \pm 3.92\%$ with structured merge.

Summary: Semistructured and structured merge report similar numbers of conflicts, but the number of merge scenarios with conflicts is reduced using structured merge (by about 19%). In general, conflicts are not frequent when using both strategies (in about 2% of the scenarios).

4.4.2 How often do semistructured and structured merge differ with respect to the occurrence of conflicts?

We found 223 (0.51%) scenarios with conflicts reported *only* by semistructured merge, and 30 (0.07%) scenarios have conflicts reported *only* by structured merge. So, the two strategies differ in 0.58% (253) of the scenarios in our sample. A per-project analysis gives a similar result: on average, the strategies differ on $0.52 \pm 2.06\%$ of the scenarios.

The reported percentages are particularly small because most scenarios are free of conflicts even when using less sophisticated strategies such as unstructured merge. In fact, most of them involve only changes to disjoint sets of files, and could not possibly discriminate between merge strategies because there is no chance of conflict.

So, it is important to consider the relative percentages of conflicting merge scenarios, which correspond to 2.28% of our sample scenarios. Overall, the strategies differ in 23.67% of the conflicting scenarios, with an average of $23.22 \pm 44.45\%$ in a per-project analysis.

The observed error bounds are explained by some projects having low rates of merge scenarios with conflicts. For instance, projects such as `CLOCKER`, `WIRE` and `LA4J` had only one conflicting merge scenario, and, for this single scenario, the strategies differ as a result of the reasons we explain on the next research question.

Summary: Semistructured and structured merge substantially differ in terms of reported number of conflicts when applied only to conflicting scenarios of our sample (they differ in about 24% of these scenarios).

4.4.3 Why do semistructured and structured merge differ?

To better explain the differences between the merge strategies, based on power and sample size estimation statistics, we manually analyzed a random sample of 111 merge scenarios that have conflicts reported by only one of the strategies. This includes 96 scenarios with conflicts reported only by semistructured merge, and 15 scenarios with conflicts reported only by structured merge. For each scenario, we analyzed developers changes, the code merged by one of the strategies, and the conflict reported by the other strategy. This way we associate characteristics of the integrated changes with details of the strategy that lead to conflicts.

Starting with scenarios with semistructured merge conflicts, and a structured clean merge, consider the example in Figure 21. Developer *A* added the `final` modifier to the `IOException` catch right after the `try` block. Meanwhile, developer *B* added a new catch to `ResourceNotFoundException`, also right after the `try` block. As no line separates these changes in two distinct areas of the text, semistructured merge—which invokes unstructured merge to integrate method bodies—reports the conflict illustrated in the figure. Developers then have to manually act and decide which catch should appear right after the `try` block. In contrast, structured merge detects that the changes affect different child nodes of a `try` node, and successfully integrates the changes by including the new child node (*B*’s contribution) and the existing changed node (*A*’s contribution). We observed the same kind of situation in every scenario that leads only to semistructured merge conflicts, including the motivating example illustrated earlier in this chapter.

Summary: Semistructured and structured merge differ when changes occur in overlapping text areas that correspond to different AST nodes.

Moving now to scenarios with structured merge conflicts, and a semistructured clean merge, consider the example in Figure 22(a). Developer *A* added a call to method `viewModel` to an existing method call chain. Developer *B* changed the argument of method `provided` in the same chain. Semistructured merge successfully integrates the changes because it

Figure 21 – Semistructured merge conflict from project GLACIERUPLOADER .

Structured

```
try{...
} catch (final ResourceNotFoundException e){...
} catch (final IOException e){...
}
```

Semistructured

```
try{...
<<<<<<
} catch (final IOException e){...
=====
} catch (final ResourceNotFoundException e){...
} catch (IOException e){...
}
>>>>>>
```

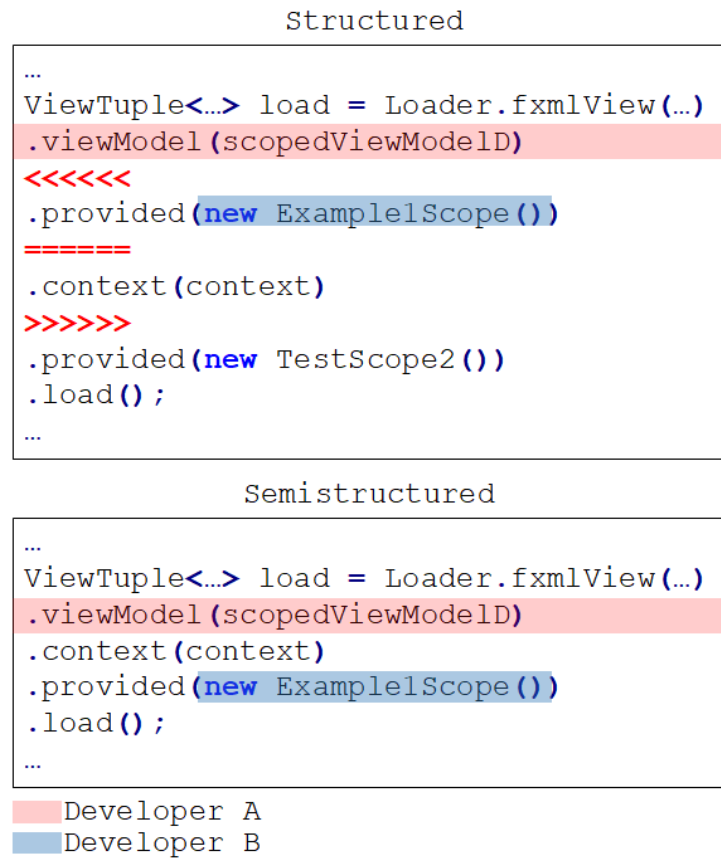
Developer A
 Developer B

detects they occur in non-overlapping text areas: the line that calls method `context` act as a separator between the areas. Structured merge reports a conflict because, by analyzing and matching the base AST with the developers ASTs (see Figure 22(b)), it incorrectly concludes that the left child of the second `MethodCall` node was changed by both developers. Indeed, as marked in red in the figure, the three nodes in this position are different. Developer *B* has not actually changed the call to `provided`, but changed the call to `context` position by adding a new method call to `viewModel`. As tree matching is top-down and mostly driven by `MethodCall` nodes in this case (APEL; LESSENICH; LENGAUER, 2012), structured merge is not able to correctly match the calls, and assumes that Developer *B* changed the call to `provided` by a call to `context`. That is why the reported conflict involves these two method calls; the second in the conflict text corresponds to a base node not changed by the developers (`context` call). The text does not refer to the AST node that actually caused the conflict (`viewModel` call).

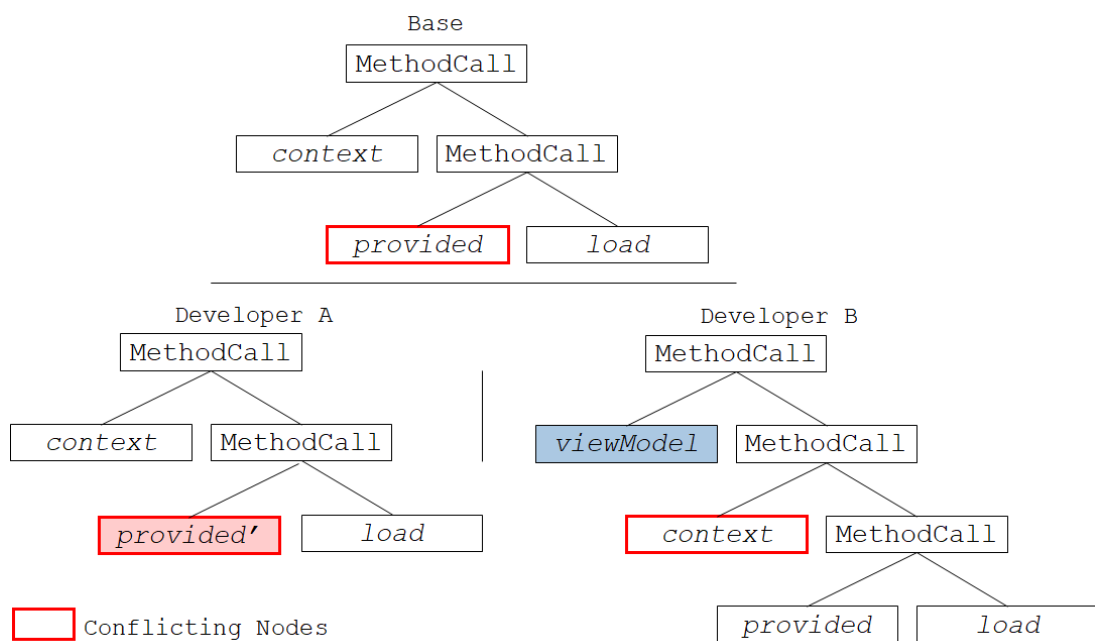
Structured merge makes a difference in a second kind of situation, as illustrated in Figure 23. In this example, developer *A* deletes an argument from the call to method `doInsertFinalNewLine` inside a `for` statement. Developer *B* converts the same `for` statement into a `for each` statement. Since these changes occur in non-overlapping text areas, semistructured merge successfully integrates the contributions. Structured merge reports a conflict because it is unable to match the new `for each` with the previous `for` statement because they are represented by nodes of different types. It correctly detects that the subtree of the `for` statement body was changed by one of the developers, but it incorrectly assumes that the whole `for` tree was deleted by the other developer. As a consequence, structured merge does not proceed merging the child nodes from these iteration statements, and reports a single conflict for the entire statements. Note that the changed method call

Figure 22 – Structured merge conflict from project MVVMFX .

(a) Code

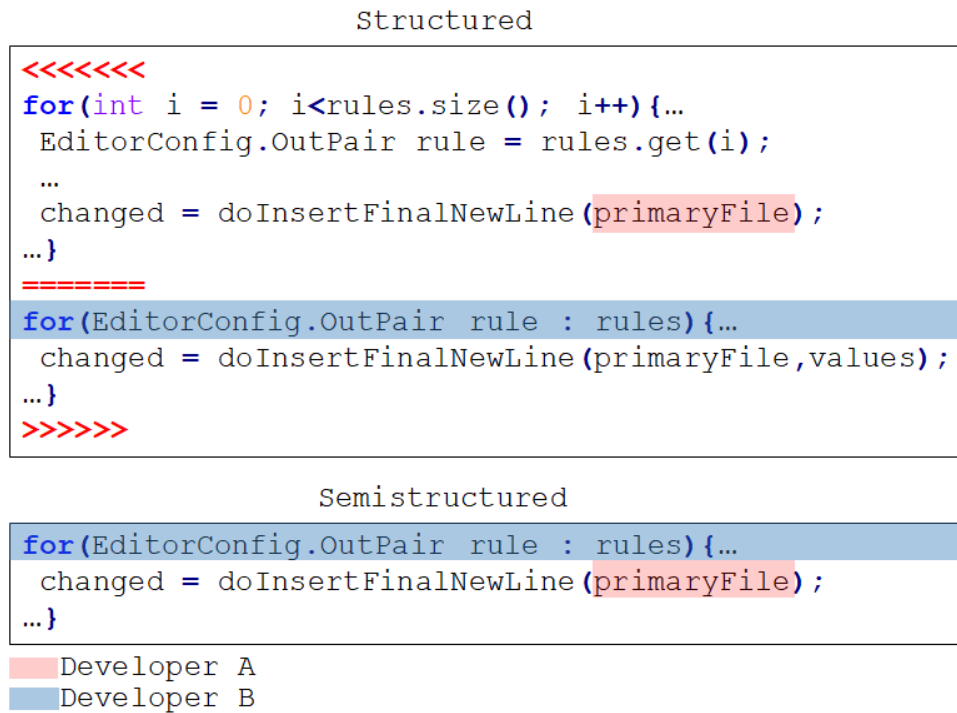


(b) AST



`doInsertFinalNewLine` is accidentally included in this deletion as it is not matched with the corresponding version in the `for each`.

Figure 23 – Structured merge conflict from project EDITORCONFIG-NETBEANS .



Summary: Semistructured and structured merge differ when changes occur in non-overlapping text areas that correspond to (a) the same node, and to (b) different but incorrectly matched nodes.

4.4.4 Which of the two strategies reports fewer false positives?

As explained in Section 4.3.2, we use Travis CI to build and test the resulting merged code of the 253 scenarios for which the strategies differ (one reports a conflict and the other cleanly merges the code). We found 44 scenarios with merged code that successfully builds and for which all tests pass; their Travis CI status is *passed*. Although this status provides precise guarantees that there are no build and test conflicts in these scenarios, there could still be other kinds of semantic conflicts, as unexpected interference between merged contributions might be missed by the existing tests. These 44 scenarios are then potential false positives of the strategy that reported a conflict, but we have to confirm this with a manual inspection of the merged code and the scenario contributions. Our goal is to identify possible interference between merged contributions. These scenarios were analyzed by two individuals separately. In 3 scenarios, there was disagreement between the

individuals, so the review of another individual was necessary. Besides, in only 1 scenario the contributions were not clear, so we asked contributions owner for clarification.

From the 44 potential scenarios with false positives, 39 are related to semistructured merge: they were successfully merged by structured merge and have a *passed* status in Travis CI. Conversely, only 5 scenarios are potential false positives of structured merge.

The manual analysis revealed that 36 of the 39 scenarios actually have semistructured merge false positives. Only 3 scenarios were actual true positives of semistructured merge, and, as a consequence, false negatives of structured merge. Although the build was successful, and none of the tests failed, for these three scenarios, we could still observe interference between the merged contributions. For instance, in a merge scenario from project SWAGGER-MAVEN-PLUGIN, both developers added elements to the same list. As a consequence, each developer expects different resulting lists, which are themselves different from the list that will be obtained by executing the merged code. None of this project's tests exercises these contributions, but it is not hard to come up with a test that passes in the developers versions but fails in the merged version, revealing a conflict. For example, suppose a test that checks whether the size of the list is $n+1$; if it passes in the developers' individuals versions, it will fail in the merged version, in which the size of the list will be $n+2$.

From the 5 scenarios having potential structured merge false positives, 4 of them were classified as actual false positives. Only 1 of the 5 scenarios was an actual true positive of structured merge, and a false negative of semistructured merge. The actual true positive is a scenario from project RESTY-GWT. In this scenario, one of the developers edited the condition and block of an existing `if` statement, while the other added another `if` statement after the previous `if` statement. Both `if` statements return different values based on the value of the same method parameter. However, the first developer's edited condition now satisfies both developers conditions, affecting the method result expected by the other developer, and no test of the mentioned project captures this interference. A test that captures this interference could be one, added by the second developer, that checks the value of the mentioned parameter, and then enters into his added `if` block. The test passes on second developer's version, and fails on the merged version because now it would enter on first developer's `if` block, returning a different value. This situation is illustrated in Figure 24. More specifically, suppose a test that receives an `Overlay` method returning a primitive type (`int`, for instance). The test then checks if result type's is `OverlayCallback`. The test passes on developers *B*'s version alone (in red in the figure), and fails on the merged version because of developer *A*'s changes (in blue).

4.4.5 Which of the two strategies has fewer false negatives?

We found 209 scenarios with merged code that either cannot be successfully built (Travis CI *errored* status) or can be properly built but, at least, one of the tests do

Figure 24 – Structured merge conflict from project RESTY-GWT .



not pass (Travis CI *failed* status). By performing a Travis CI log report analysis, we observe that most scenarios (169) *errored* and *failed* status are due to a number of reasons (Travis CI timeout, unavailable dependencies, etc.) unrelated to the contributions being merged, and that suggest these are older scenarios that would be hard to compile and build anyway — we name these as *undetermined* builds. So, we cannot automatically classify these *undetermined* builds as false negatives of the strategy that merged the code (the other having reported a conflict). Thus, we then focus on 40 scenarios with *errored* and *failed* status caused by the merged contributions (we confirm that by parsing Travis CI log messages and checking that they are compiler or test related), plus a manual inspection of 41 scenarios randomly selected from the *undetermined* builds, based on power and sample size estimation statistics, to check whether the merged contributions actually interfere. Since our sample does not include scenarios having broken or failing parents (see Section 4.3.1), if the resulting merged code presents build or test issues, we conclude this is due to interference between the merged contributions.

From the first group of 40 analyzed scenarios, we found only 4 scenarios having semistructured merge false negatives: 3 with *errored* status and 1 with *failed* status for the corresponding merge result produced by semistructured merge (structured merge having reported conflicts for these cases). In contrast, we found 36 scenarios having structured merge false negatives: 23 with *errored* status and 13 with *failed* status for the merge result produced by structured merge (semistructured merge having reported conflicts for these cases).

Although the merge strategies are somewhat different, we observed common causes for false negatives due to broken builds. For example, we found situations in clean merges from both strategies, in projects such as BLUEPRINTS and SINGULARITY, where one developer added a reference to a variable while the other developer deleted or renamed that variable. Consequently, the compiler could not build the file containing the reference to the removed or renamed element. We also observed situations, in projects such as NEO4J-RECO and VRAPTOR, where one developer changed the value passed as an argument, while the other developer changed the corresponding parameter’s type. After the merge, there is a compilation error reported due to the mismatch between expected and passed argument. It is important to emphasize that, although these merge strategies are not meant to detect build or test issues, they might detect, by accident, other kinds of conflicts that do not preclude them from generating a valid program, but would lead to build or execution failures (see Section 2.3).

Regarding test failures causing false negatives, the only failed scenario from semistructured merge was in project CLOSURE-COMPILER, where the developers changes are responsible to update the same list. Conversely, on failed scenarios from structured merge, we observed, for example, developers inadvertently changing the same connection creation in project JEDIS, or instantiating the same object with different constructors in project DSPACE.

From the manual inspection of the 41 *undetermined* builds, we classified 3 of them as having semistructured false negatives, and 6 of them as having structured merge false negatives. So, from the remaining inspected *undetermined* builds, 30 had actually semistructured merge false positives, and 2 of them had structured merge false positives. We summarize our findings for false positives and false negatives after all analyses on Table 3.

Table 3 – Numbers for merge scenarios with false positives and false negatives.

	Semistructured Merge	Structured Merge
False Positives	66	6
False Negatives	8	45

Summary: Semistructured merge reports more false positives (9 times more scenarios with false positives), and structured merge misses more conflicts (has more false negatives; 8 times more scenarios with missed conflicts).

4.4.6 Does ignoring conflicts caused by changes to consecutive lines make the two strategies more similar?

Our results shows that our metrics slightly drop if a semistructured merge tool could resolve conflicts due to changes in consecutive lines. In particular, the number of scenarios with semistructured merge conflicts is reduced by 3.38%. Besides, the number of scenarios in which semistructured and structured merge differ is reduced by 11.07%. Note that we only count consecutive lines conflicts, we actually do not resolve them. Thus, we do not have numbers for false positives and false negatives, which demands Travis CI builds of code without conflicts.

As we observed in projects such as QUICKML, SEJDA and SONARQUBE, this happens because changes to consecutive lines often correspond to changes to different AST nodes. In such situations, structured merge does not report conflicts. Thus, when semistructured merge is able to resolve consecutive lines conflicts, it might avoid conflicts due to changes to different AST nodes, similar to structured merge.

Summary: A semistructured merge tool that can resolve consecutive lines conflicts would present even closer number of scenarios with conflicts to structured merge, and fewer scenarios in which the two strategies differ.

4.5 DISCUSSION

Our results show that, overall, the two merge strategies rarely differ for the scenarios in our sample, as most of them are free of conflicts. Many of them change disjoint sets of files, having no chance of leading to conflicts, not mattering which merge strategy is adopted by the tool one uses. However, for scenarios that reflect more complicated merge situations, we do observe that the choice of the merge strategy makes a difference: considering scenarios with conflicts, the strategies differ in about 24% of the cases. This is, though, maybe surprisingly low given that most code and changes occur inside (method, constructor, etc.) declarations exploited by the significant extra structure considered by structured merge. In terms of conflicting scenarios with differing behavior, structure plays a similar role when moving from unstructured merge to semistructured merge (27%, see Section 3.3), and when moving from semistructured to structured merge (24%).

When the strategies differ, semistructured merge reports false positives in more merge scenarios than structured merge, whereas structured merge has more scenarios with false negatives than semistructured merge. The extent of the difference in the false positive and false negative rates are quite similar. Semistructured merge false positives are not hard to resolve: the fix essentially involves removing conflict markers. Analyzing the changes before removing the markers might be expensive, but certainly not as in unstructured merge (with its crosscutting conflicts as illustrated earlier in Figure 13), or as in structured merge (with its fine granularity conflicts, as illustrated in Figure 19). Contrasting, structured merge false negatives might be hard to detect and resolve. Most of the observed false negatives actually correspond to compilation and static analysis issues that escape the merging process but cannot escape the building phase. These are always detected and are often easy to resolve. However, a large part of the observed false negatives correspond to dynamic semantics issues that can easily escape testing and end up affecting users. These are hard to detect and, when detected, are often hard to solve. A more rigorous analysis based on conflict detection and resolution timing data, in the spirit of (BERRY, 2017), could differently weight false positives and false negatives and better assess the benefits of the strategies.

Based on our findings regarding false positives and false negatives, and given the observed modest difference between the merge strategies, we conclude that semistructured merge would be a better match for developers that are not overly concerned with false positives. This is reinforced by considering the observed performance overhead associated with structured merge, and the extra effort needed to develop structured tools (APEL et al., 2011). Together with our consecutive lines result, this discussion points to the development of a tool that adapts semistructured merge to report textual conflicts only when changes occur in the *same* lines (resolving conflicts caused by changes to *consecutive* lines). Such a tool could hit a sweet spot in the relation between structure and accuracy in non-semantic merge tools.

The derived observations from our study, especially the ones that explain when the strategies differ, shall help researchers and merge tool developers to further explore improvements to merge accuracy and the underlying tree matching algorithms. In the same line, our manual analysis of false positives reveal opportunities for making the tool avoid a number of false positives. For example, by detecting straightforward semantic preserving changes, we could avoid 42% of the semistructured false positives in our sample.

Combining the two merge strategies in a similar fashion as suggested by (APEL; LESSENICH; LENGAUER, 2012) seems also promising. One idea is to invoke structured merge, and when it does not detect conflicts, invoke semistructured merge and return its result, which would reduce the chances of false negatives. This is a conservative strategy, which considers the costs associated with false positives to be inferior than those associated with false negatives. Such a tool would eliminate structured merge false negatives, but

would still have semistructured merge false negatives. Conversely, in the best case, when structured merge does detect conflicts, it would present structured merge false positives; and, in the worst case, the tool would present semistructured merge false positives. A less conservative combination, in which semistructured is used as long as it does not detect conflicts, could also be explored by researchers. Actually, a tool implementing these combinations of strategies should allow users to decide which one is more suitable for their activities. As a drawback, such a combination of strategies has potentially the performance overhead of invoking two strategies, so a detailed investigation is necessary to check the tradeoff between performance and accuracy.

4.6 THREATS TO VALIDITY

We rely on manual analysis to identify interference between merged contributions, so there is a risk of misjudgment. To mitigate this threat, every scenario was analyzed separately by two individuals, and in case of disagreement, another individual acted as a mediator. We also asked the actual contributors for clarification of the changes when they were not clear, this was only necessary in one occasion.

We opted for a single merge tool that can be configured to work as semistructured and structured merge. This was necessary to ensure that we have a structured merge tool working as expected. This single tool is basically an extension of the semistructured merge tool able to invoke a structured merge tool on declarations' body. To the best of our knowledge, these tools are the most mature and evaluated available tools.

In addition, as we discard merge scenarios that we could not properly build on Travis, or that have broken or failed parents, we might have missed differences in the strategies behavior. We might have also missed that because we analyze only code integration scenarios that reach public repositories with merge commits; that is not the case, for example, of integrations with git rebase, or that were affected by git commands that rewrite history.

Finally, in this study we focus on open source Java projects hosted on GitHub, using Travis CI and Maven. Thus, generalization to other platforms and programming languages is limited. Such requirements were necessary because the merge tools are language specific, and to reduce the influence of confounds, increasing internal validity.

5 CONCLUSIONS

In this work, our main goal was to help developers deciding which kind of merge tool to use. In particular, conflicts are a recurring problem in the context of collaborative software development. As a consequence, one likely has to dedicate substantial effort to resolve them. To reduce such effort, unstructured, line-based merge tools, which are the state of practice, rely on purely textual analysis to detect and resolve conflicts. Structured merge tools are programming language specific and go beyond simple textual analysis by exploring the underlying syntactic structure and static semantics when integrating programs. Semistructured merge tools attempt to hit a sweet spot between unstructured and structured merge by partially exploring the syntactic structure and static semantics of the artifacts involved. For program elements whose structure is not exploited, like method bodies in Java, semistructured merge tools simply apply unstructured merge textual analysis. Previous studies provide evidence that semistructured and structured merge report less conflicts than unstructured merge. However, reduction on the number of reported conflicts alone is not enough to justify industrial adoption of a new merge tool due to the risk of missing conflicts (false negatives), or introducing false positives. Besides, it was unknown how semistructured merge compares with structured merge. Thus, to decide whether we should replace our merge tools, we needed to compare these merge strategies and understand their differences, strengths and weaknesses.

By reproducing more than 34,000 merges from 50 Java projects, we first investigated the relation between unstructured and semistructured merge with respect to the resulting occurrence of false positives and false negatives. In particular, our assumption was that false positives represent unnecessary integration effort, which decrease productivity, because developers have to resolve conflicts that actually do not represent interference between development tasks. Besides that, false negatives represent build or behavioral errors, negatively impacting software quality and correctness of the merging process. For most projects and merge scenarios, we observed that semistructured merge not only reduces the number of reported conflicts, but it also has fewer additional false positives when compared to unstructured merge. Furthermore, we find evidence that semistructured merge additional false positives are easier to analyze and resolve than those reported by unstructured merge. However, we found no evidence that semistructured merge has fewer additional false negatives than unstructured merge. We also argue that semistructured merge false negatives are harder to detect and resolve.

Driven by these findings, we proposed an improved semistructured merge tool that further combines unstructured and semistructured merge to reduce the false positives and false negatives of semistructured merge. We find evidence that the improved tool, when compared to unstructured merge in our sample, reduces the number of reported

conflicts by half, has no additional false positives, has at least 8% fewer false negatives, is not prohibitively slower, and presents no extra usability barriers in relation to state of the practice of merge tools. In practice, we expect the reduction in the number of false negatives to be much higher, given the conservative nature of our metrics. Although the improved tool has fewer false negatives, they might be harder to detect and resolve than unstructured merge false negatives. As we have no conflict detection and resolution timing data, we cannot, in a precise way, differently weight different kinds of false negatives and better assess the benefits of our tool. This would be needed to compare the tools in a more rigorous way. Similarly, we cannot precisely weight false positives and false negatives. This, however, is less critical in our case because, in the analyzed sample, the improved tool has no additional false positives in relation to unstructured merge.

Semistructured merge has shown significant advantages over unstructured merge. However, before deciding to replace state of practice unstructured tools by semistructured merge, we need to investigate whether structured merge is a better option than semistructured merge. So, we compared semistructured and structured merge by reproducing more than 43,000 merge scenarios from 508 Java projects. Our results show that users should not expect much difference when using a semistructured or a structured merge tool, especially when semistructured merge is able to resolve conflicts due to changes in consecutive lines of code. We also discuss that, when deciding which tool to use, a user should consider that semistructured merge reports more false positives, but structured merge misses more conflicts (false negatives). However, combining the two strategies seems promising as it is able to lessen disadvantages of both strategies.

Finally, our findings point to semistructured merge as a better replacement of unstructured tools for conservative developers, having significant gains with a closer behavior to unstructured tools than structured merge. Besides that, practitioners might be reluctant to adopt structured merge because of the observed performance overhead and its tendency to false negatives. So, when choosing between semistructured and structured merge, semistructured merge would be a better match for developers that are not overly concerned with semistructured extra false positives. Finally, tweaking semistructured merge, or even a combination with structured merge, might be the way for a sweet spot in the relation between structure and accuracy in non-semantic merge tools.

5.1 CONTRIBUTIONS

We summarize our contributions as follows:

- Show how frequently merge conflicts occur during the merging process by using unstructured, semistructured and structured merge tools;
- Derive a list of relative false positives and false negatives between unstructured and semistructured merge;

- Evaluate how frequently the relative false positives and false negatives occur during the merging process as approximations for integration effort and correctness;
- Propose and evaluate an improved semistructured merge tool that address kinds of the observed false positives and false negatives;
- Fill a gap in the literature by comparing semistructured and structured merge with respect to different dimensions;
- Evaluate how frequently builds break and tests fail after merging code with semistructured and structured merge as proxy for false positives and false negatives of these different merge strategies;
- We provide a replication package in our online appendix, allowing the replication of the conducted empirical studies.

5.2 FUTURE WORK

As the presented studies are part of a broader context, a set of related aspects will be left out of scope. Thus, the following topics are not directly addressed in this thesis, but we suggest them as future work:

- Further improvements of our semistructured merge tool to better detect false negatives, and resolve false positives. For instance, making it aware of refactoring related changes. Also, as mentioned in Section 4.5, it is worthwhile to evaluate a merge tool the combines semistructured and structured merge;
- Investigation and comparison with *semantic* merge: tools that incorporate semantics for merge generation to check whether merged contributions does not introduce new unwanted behaviors;
- Our results could benefit from replications analyzing other projects, including projects in centralized version control system such as SVN or CVS, different CI tools, and proprietary projects. Likewise, it would be interesting to have replications considering different programming languages, which would demand implementations of semistructured and structured merge for such languages. Also, different languages might lead to a different list of false positives and false negatives from those presented in Chapter 3. Alternatively, one could make a study to analyze our results on a per-project basis, understanding, for example, why some projects have more false positives than others and so on;
- During this work we explored only certain kinds of false positives and false negatives. However, it would also be important to investigate false positives and false negatives

in general, those common to both merge strategies, and how to improve a merge tool to detect and resolve them. It is important to identify interference and inconsistencies generated by dependencies between development tasks, and how the merge strategies deal with them. The focus is mainly on semantic interference and conflicts that are rarely detected and require more integration effort;

- We use false positives as proxies for integration effort. We believe that a more precise way to estimate the effort to resolve different types of conflicts would be to conduct controlled experiments where developers have to resolve conflicts while time and other metrics are being measured;
- In Chapter 4, we use existing tests as proxies for semantic conflicts (and false negatives). We could generate tests to expose more semantic conflicts as well. One possibility would be to attempt previous works (BöHME; OLIVEIRA; ROYCHOUDHURY, 2013; Shamshiri et al., 2013) strategies to generate test cases exercising developers' contributions;
- The setup adopted in Chapter 4 came from observed limitations of the setup adopted in Chapter 3. In particular, the differences between unstructured and semistructured merge allowed us to derive a catalog of false positives and false negatives, and to compare the strategies based on this catalog. However, this was not possible when comparing semistructured and structured merge, guiding us to the adopted setup based on builds and tests. One can, for sure, employ the second setup to compare unstructured and semistructured merge;
- We had performance comparison between semistructured and structured merge in mind. Indeed, in dry runs we observed that pure *JDime* (the adopted structured tool) was slower than semistructured merge, as expected. However, when we combined the tools in the single configurable tool we use in the experiment (see Section 4.3.2), semistructured merge became slower than structured merge. Therefore, there is a performance bug, and that is why we did not include a performance comparison. So, in a future work, we should provide a detailed performance comparison between these strategies.
- Throughout this work, we asked ourselves many complementary questions that would be interesting to further investigate such as the following:
 1. What is the nature of the task being independently developed that most frequently lead to conflicts, and how it leads to false positives and false negatives? Bug fixes, refactorings, adding new features, etc;

2. Considering different software architecture models, are there options that help to prevent conflicts, and consequently false positives and false negatives? Moreover, where are conflicts more frequently located?
3. What are the different factors, technical, and organizational, that impact the occurrences of conflicts?

5.3 RELATED WORK

This section presents related work in tools and strategies for conflict detection and resolution, and previous studies providing evidence on conflicts and their impact.

5.3.1 Strategies and Tools Assisting Conflict Detection and Resolution

A number of studies propose development tools and strategies to better support collaborative development environments. These tools try to both decrease integration effort and improve correctness during integration. For instance, Apel et al. (2011) propose and evaluate *FSTMerge*, the semistructured merge tool used as a basis in this work. We confirm the evidence presented by (Apel et al., 2011) and (CAVALCANTI; ACCIOLY; BORBA, 2015) that *FSTMerge* might reduce, but not for all projects and scenarios, the number of reported conflicts. However, these studies do not investigate whether the obtained reduction is achieved at the expense of extra false negatives, or new kinds of false positives that are harder to resolve. In fact, the set of conflicts reported by *FSTMerge* is not a subset of the conflicts reported by unstructured merge. Here we go further by analyzing the relation of additional false positives and false negatives between unstructured and semistructured merge in Chapter 3. We also propose an improved semistructured tool, which is essential for justifying industrial adoption of more advanced merge tools.

Our improved semistructured merge tool implements a basic syntactic-based renaming handler to detect renaming and to avoid false positives conflicts. In particular, it uses unstructured merge result whenever unstructured and semistructured merge differ. Nevertheless, a more advanced semantic-based refactoring detection module could further improve precision. (MALPOHL; HUNT; TICHY, 2003) attempt to automatically detect renamed identifiers across multiple files. When an identifier in one version is changed, then its references in other versions are actively updated. (DIG et al., 2008) present *MolhadoRef* to merge software in the presence of object-oriented refactoring at the API level. It records change operations (refactoring and edits) used to produce one version and replays them when merging versions, and is concerned with operations which have well-defined operations: inserting and deletion of packages, classes, method declarations and field declarations. *MolhadoRef* is built on top *Molhado*, a structure versioning framework that facilitates the construction of structure-oriented difference tools for various types of software artifacts (NGUYEN, 2006). *Gumtree* uses a two-phase strategy for AST matching (FALLERI et

al., 2014). In the first phase, it searches top-down for nodes whose names match and whose subtrees are isomorphic. In the second phase, it revisits unmatched nodes and searches for pairs which have a significant number of matching descendants. This way, renamed program elements and shifted code can be detected.

Structured and semantic merge strategies have also been proposed. Westfechtel (1991) and Buffenbarger (1995) have pioneered in proposing merge algorithms which incorporate context-free and context-sensitive structures. Their structured-oriented approaches are language-independent, as language features are represented in an abstract level. Later, a variety of approaches on structured diff and merge have been proposed. These include tools specific to Java (APIWATTANAPONG; ORSO; HARROLD, 2007) and C++ (GRASS, 1992).

Apel, Lessenich e Lengauer (2012) proposed *JDime*, the structured tool used in Chapter 4, capable of tuning the merging process on-line by switching between unstructured and structured merge, depending on the presence of conflicts. (LESSENICH et al., 2017) attempt to improve *JDime* by employing a syntax specific look-ahead to detect renamings and shifted code. They demonstrate that their solution can significantly improve matching precision in 28% while maintaining performance.

(ZHU; HE; YU, 2019) proposed *AUTOMERGE*, a structured merge tool, built on top of *JDime*, that matches nodes based on an adjustable so-called *quality function*. Their goal is to find a set of matching nodes that maximizes the *quality function*, preventing the matching of logically unrelated nodes, and, as consequence, false positives conflicts. They found that *AUTOMERGE* was able to reduce the number of reported conflicts by 63% when compared to original *JDime*, being only 17% slower. Besides, they found that about 99% of the results yielded by *AUTOMERGE* exactly correspond to original developers' result, compared to 93% from *JDime*. They, however, do not investigate whether the new matching leads to fewer false negatives, or introduces other kinds of false positives not reported by original *JDime*. Besides, considering the similarity to original developers' result as a ground truth might not be a safe decision, as they are based on state of practice unstructured tools, so developers' might have missed false negatives. Besides, most of merges are free of conflicts. Many of them change disjoint sets of files, having no chance of leading to conflicts, regardless of the adopted tool. Still adopting that criteria of similarity to original developers' result, we believe that semistructured merge tools would be superior in that sense as it presents a close behavior to unstructured merge.

We complement these prior studies by comparing semistructured and structured merge, not only in terms of reported conflicts, but also in terms of false positives and false negatives. We conclude that semistructured merge would be a better match for developers that are not overly concerned with false positives, especially when a semistructured merge tool resolve conflicts caused by changes to consecutive lines. We also suggest that a combination of these two strategies seems promising as it is able to reduce weakness of both strategies.

Semantic strategies have been for long not practical. (HORWITZ; PRINS; REPS, 1989)

were the first to propose an algorithm for merging program versions without semantic conflicts for a very simple assignment-based programming language. This original work was later extended to handle procedure calls (BINKLEY; HORWITZ; REPS, 1995) and to identify semantics-preserving transformations (YANG; HORWITZ; REPS, 1990). *SemanticDiff* (Jackson; Ladd, 1994) takes two versions of a C program and identifies differences between them by comparing the dependence relations between each procedure input and output to approximate observable behavior. (BERZINS, 1994) proposed a general approach by providing a language-independent definition of semantic merging with the use of a generalization of the use of traditional denotation semantics.

In recent effort towards semantic merge feasibility, (SOUSA; DILLIG; LAHIRI, 2018) proposed *SafeMerge*, a semantic tool that checks whether a merged program does not introduce new unwanted behavior. They achieve that by combining lightweight dependence analysis for shared program fragments and precise relational reasoning for the modifications. They found that the proposed approach can identify behavioral issues in problematic merges that are generated by unstructured tools. This tool needs as input a merged program besides the three versions present in a merge scenario, so it could be used in combination with a semistructured or structured merge tool— or even our suggested tool that further combines these two strategies (see Section 4.5)— to reduce their behavioral false negatives. However, *SafeMerge* only analyzes the class file associated with the modified method declarations, so it may suffer from both false positives and negatives too. In particular, their analysis results are only sound under the assumption that the external callees from other classes have not been modified.

To prevent conflicts, tools using different strategies have also been proposed. *Cassandra* (KASI; SARMA, 2013), for example, is a tool that analyzes task constraints to recommend an optimum order of tasks execution so that conflicts can be avoided. While the tasks are being developed, *Palantír* (Sarma; Redmiles; van der Hoek, 2012) is an awareness tool that informs developers of ongoing parallel changes, and *Crystal* (BRUN et al., 2011), proactively integrates commits from developer repositories with the purpose of warning them if their changes conflict. *WeCode* (aES; SILVA, 2012) continuously merges uncommitted and committed changes to detect conflicts on behalf of developers before they check-in their changes. *TIPMerge* (Costa et al., 2019) has an algorithm that recommends developers who are best suited to perform merges considering different metrics such as developers' experience in the project, their changes in the involved branches, and dependencies among modified files.

5.3.2 Evidence on Conflicts and Their Impact

Empirical studies provide evidence about the frequency and impact of conflicts, and their associated causes. They all conclude that conflicts are frequent. For example, (KASI; SARMA, 2013) and (BRUN et al., 2011) reproduce merge scenarios from different GitHub

projects with the purpose of measuring the frequency of merge scenarios that resulted in conflicts. These studies show average conflicting scenarios rates for merge conflicts of 14%, and 17% respectively. (ZIMMERMANN, 2007) conducted a similar analysis reproducing integrations from CVS projects instead. (PERRY; SIY; VOTTA, 2001) made an observational case study to analyze the effect of parallel changes on a large-scale industrial software system. They reported that, although 90% of the files could be merged without problems, the degree of parallel changes is high— merge conflicts involved between 2 to up to 16 parallel changes. Our work complements these studies bringing evidence of conflict frequency with the use of different merge strategies.

Other studies did not quantitatively measure the cost of resolving conflicts, but they reported, based on experimental observations, that resolving merge conflicts is not so trivial. It might take considerable time, and is an error-prone activity. For example, (SARMA; REDMILES; HOEK, 2012) reported that developers commonly rush to commit their tasks before others, so they would not have to deal with conflicts while pushing their changes to the shared repository. In addition, (BIRD; ZIMMERMANN, 2012) report that a frequent cause for integration errors are merge conflicts that were not resolved correctly. (McKee et al., 2017) conducted a series of interviews and surveys to understand developers perceptions of merge conflicts. They reported that if developers perceive a conflict as too complex or if they do not have much knowledge in the code area of the conflict, they might feel the need to alter their resolution strategy, such as reverting conflicting changes, and in some cases delaying the task of resolving conflicts. (ADAMS; MCINTOSH, 2016), and (HENDERSON, 2017) even report that companies have migrated to single-branched repositories to avoid difficult merges. By analyzing false positives and false negatives of different merge strategies, we were able to experience and to provide information on conflict resolution complexity.

There are also studies that analyze different technical and organizational aspects that might have an impact on the occurrence of conflicts, and characteristic of conflicts. (CATALDO; HERBSLEB, 2011) presented an empirical analysis of a large-scale project where they examined the impact that software architecture characteristics, and organizational factors have on the number of conflicts. They concluded that architecture related factors such as the nature and the quantity of component dependencies, as well as organizational factors such as the geographic dispersion of development teams, can lead to higher integration issues rates. (Menezes et al., 2018) analyze merge scenarios from open source Java projects to investigate the nature of merge conflicts in terms of what conflicts look like, what kinds of conflicts occur, how developers fix them, how conflicts relate to each other, and more. Based on their results, they argue that it is difficult to envision a single generic merge strategy that can automatically resolve all possible conflicts, because the diversity in conflicts is simply too large. Still, they believe it is possible to improve over the existing tools to better resolve conflicts, for instance, in the form of plug-ins that can automatically

handle specific kinds of conflict. (ACCIOLY; BORBA; CAVALCANTI, 2018) derive a catalog of conflict patterns expressed in terms of the structure of code changes that lead to merge conflicts. Their results show that most conflicts occur because developers independently edit the same or consecutive lines of the same method. However, the probability of creating a merge conflict is approximately the same when editing methods, class fields, and modifier lists. They noticed that most part of conflicting merge scenarios, and merge conflicts, involve more than two developers. Also, that copying and pasting pieces of code, or even entire files, across different repositories is a common practice and cause of conflicts. Similarly, (Yuzuki; Hata; Matsumoto, 2015) investigate how conflicts on method declarations are resolved on open source Java projects. They found that most part of them is resolved by adopting one of the versions, then discarding the other. All these findings about conflicts characteristics might be adapted by a merge tool as strategies for resolving conflicts.

Regarding the concept of integration effort, Prudêncio et al. (2012) suggest that it can be measured as the number of extra actions (additions, deletions or modifications) that a developer has to perform during code integration to conciliate the changes made in the contributions to be merged. Furthermore, Santos e Murta (2012) correlate the number of conflicts to that metric, suggesting that conflict reduction imply effort reduction. We opted for an additional qualitative analysis because this metric only estimates the fraction of the time taken by a developer to edit code, not taking into account the time that the developer took reasoning about how to resolve the conflict. This way, the editing time amounts to only part of the total integration effort time. Kasi e Sarma (2013) measure integration effort based on the number of days that the conflict persisted in a project repository. However, they assume that, during this period, the developers exclusively worked to resolve that conflict. As the precise fixing period might be hard to find, and we believe that is often not the case that developers exclusively work to resolve conflicts when they happen, we opted for a qualitative analysis.

Assessing how often integration results in build or test issues, which can be seen as a consequence of the false negatives in the merging process, partially motivated a couple of studies (BRUN et al., 2011; KASI; SARMA, 2013; ACCIOLY et al., 2018). Kasi e Sarma (2013), for instance, report integrations that result in build issues occurring in ranges between 2-15%, while Brun et al. (2011) describe both build and test issues ranging around 33%. Comparing merge strategies, however, is not the focus of these studies, they are actually based on traditional unstructured merge. Our work complements these studies in two manners. In Chapter 3, we bring evidence about the frequency of integration that had certain types of false positives and false negatives. Besides that, we have explored specific types of false negatives that cause build or test issues. In Chapter 4, we assess the frequency of build and test issues with different merge strategies, also considering a substantially larger sample, with more than 500 projects and 20 times more merge scenarios than the aggregated sample of these mentioned two studies.

Finally, a number of studies investigate the causes of errors in the build process (ZHANG; HASSAN, 2006; SEO et al., 2014; Kerzazi; Khomh; Adams, 2014; BELLER; GOUSIOS; ZAIDMAN, 2017; RAUSCH et al., 2017), but none of them investigate whether these errors are caused by conflicting contributions, and are therefore related to collaboration or coordination breakdowns. Although we consider a much larger sample than these previous studies, we did not find many build and test issues. Besides the fact that we use advanced merge tools, we believe there are two main reasons for the contrasting evidence. First, they do not consider parents commit build status. This way, the build might have been already broken or with failing tests before the merge. Second, previous studies perform build and tests locally, so some part of errored and failed builds might have been caused by external or configuration problems. In our study, we mitigate both threats since we analyze Travis CI log report to filter builds with errors caused by external problems, and we also check the merge commit parents status. This way we increase the confidence that the merge commit build problems are caused by conflicting contributions.

REFERENCES

- ACCIOLY, P.; BORBA, P.; CAVALCANTI, G. Understanding semi-structured merge conflict characteristics in open source java projects. *Empirical Software Engineering*, Springer, 2018.
- ACCIOLY, P.; BORBA, P.; SILVA, L.; CAVALCANTI, G. Analyzing conflict predictors in open source java projects. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. [S.l.]: ACM, 2018. (MSR'18).
- ADAMS, B.; MCINTOSH, S. Modern release engineering in a nutshell – why researchers should care. In: *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*. [S.l.]: IEEE, 2016. (SANER'16).
- aES, M. L. G.; SILVA, A. R. Improving early detection of software merge conflicts. In: *Proceedings of the 34th International Conference on Software Engineering*. [S.l.]: IEEE Press, 2012. (ICSE '12).
- APEL, S.; LENGAUER, C. Superimposition: A language-independent strategy to software composition. In: *Proceedings of the 7th International Conference on Software Composition*. [S.l.]: Springer-Verlag, 2008. (SC'08).
- APEL, S.; LESSENICH, O.; LENGAUER, C. Structured merge with auto-tuning: Balancing precision and performance. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.]: ACM, 2012. (ASE'12).
- APEL, S.; LIEBIG, J.; BRANDL, B.; LENGAUER, C.; KÄSTNER, C. Semistructured merge: Rethinking merge in revision control systems. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. [S.l.]: ACM, 2011. (ESEC/FSE'11).
- APIWATTANAPONG, T.; ORSO, A.; HARROLD, M. J. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, Kluwer Academic Publishers, 2007.
- BECK, K. *Extreme programming explained: embrace change*. [S.l.]: Addison-Wesley Professional, 2000.
- BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. Oops, my tests broke the build: An explorative analysis of travis ci with github. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. [S.l.]: IEEE, 2017. (MSR'17).
- BERRY, D. M. *Evaluation of Tools for Hairy Requirements Engineering and Software Engineering Tasks*. 2017. Disponível em: <https://cs.uwaterloo.ca/~dberry/FTP_SITE/tech.reports/EvalPaper.pdf>.
- BERZINS, V. On merging software extensions. *Acta Informatica*, Springer, 1986.
- BERZINS, V. Software merge: Semantics of combining changes to programs. *ACM Transactions on Programming Languages and Systems*, ACM, 1994.

- BINKLEY, D.; HORWITZ, S.; REPS, T. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, ACM, 1995.
- BIRD, C.; RIGBY, P. C.; BARR, E. T.; HAMILTON, D. J.; GERMAN, D. M.; DEVANBU, P. The promises and perils of mining git. In: *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*. [S.l.]: IEEE Computer Society, 2009. (MSR '09).
- BIRD, C.; ZIMMERMANN, T. Assessing the value of branches with what-if analysis. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. [S.l.]: ACM, 2012. (FSE'12).
- BÖHME, M.; OLIVEIRA, B. C. d. S.; ROYCHOUDHURY, A. Regression tests to expose change interaction errors. In: *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*. [S.l.]: ACM, 2013. (ESEC/FSE'13).
- BRINDESCU, C.; CODOBAN, M.; SHMARKATIUK, S.; DIG, D. How do centralized and distributed version control systems impact software changes? In: *Proceedings of the 36th International Conference on Software Engineering*. [S.l.]: ACM, 2014. (ICSE'14).
- BRUN, Y.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. Proactive detection of collaboration conflicts. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. [S.l.]: ACM, 2011. (ESEC/FSE'11).
- BUFFENBARGER, J. Syntactic software merging. In: *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*. [S.l.]: Springer-Verlag, 1995.
- CATALDO, M.; HERBSLEB, J. D. Factors leading to integration failures in global feature-oriented development: An empirical analysis. In: *Proceedings of the 33rd International Conference on Software Engineering*. [S.l.]: ACM, 2011. (ICSE'11).
- CAVALCANTI, G. *Online Appendix for Should We Replace Our Merge Tools?* 2019. Hosted on <<https://spgroup.github.io/s3m/>>.
- CAVALCANTI, G.; ACCIOLY, P.; BORBA, P. Assessing semistructured merge in version control systems: A replicated experiment. In: *Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement*. [S.l.]: ACM, 2015. (ESEM'15).
- CAVALCANTI, G.; BORBA, P.; ACCIOLY, P. Evaluating and improving semistructured merge. *Proceedings of the ACM Programing Language*, ACM, n. OOPSLA, 2017.
- CAVALCANTI, G.; BORBA, P.; SEIBT, G.; APEL, S. The impact of structure on software merging: Semistructured versus structured merge. In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.]: ACM, 2019. (ASE'19).
- CONRADI, R.; WESTFECHTEL, B. Version models for software configuration management. *ACM Computing Surveys*, ACM, 1998.

-
- Costa, C. d. S.; Figueiredo, J. J.; Pimentel, J. F.; Sarma, A.; Murta, L. G. P. Recommending participants for collaborative merge sessions. *IEEE Transactions on Software Engineering*, IEEE, 2019.
- CVS. 2019. URL: <http://www.nongnu.org/cvs/>.
- DIG, D.; MANZOOR, K.; JOHNSON, R. E.; NGUYEN, T. N. Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions of Software Engineering*, IEEE, 2008.
- ENG, J. Sample size estimation: how many individuals should be studied? *Radiology*, Radiological Society of North America, 2003.
- ESTUBLIER, J.; LEBLANG, D.; CLEMM, G.; CONRADI, R.; TICHY, W.; HOEK, A. van der; WIBORG-WEBER, D. Impact of the research community on the field of software configuration management: summary of an impact project report. *ACM SIGSOFT Software Engineering Notes*, ACM, 2002.
- FALLERI, J.-R.; MORANDAT, F.; BLANC, X.; MARTINEZ, M.; MONPERRUS, M. Fine-grained and accurate source code differencing. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. [S.l.]: ACM, 2014. (ASE'14).
- FOWLER, M. *Continuous Integration*. 2006. URL: <https://www.martinfowler.com/articles/continuousIntegration.html>.
- GIT. 2019. URL: <https://git-scm.com/>.
- GOUSIOS, G.; PINZGER, M.; DEURSEN, A. v. An exploratory study of the pull-based software development model. In: *Proceedings of the 36th International Conference on Software Engineering*. [S.l.]: ACM, 2014. (ICSE 2014).
- GRASS, J. E. Cdiff: A syntax directed differencer for c++ programs. In: *Proceedings of the USENIX C++ Conference*. [S.l.]: USENIX Association, 1992.
- HENDERSON, F. Software engineering at google. *CoRR*, arXiv, 2017.
- HORWITZ, S.; PRINS, J.; REPS, T. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, ACM, 1989.
- Jackson; Ladd. Semantic diff: a tool for summarizing the effects of modifications. In: *Proceedings 1994 International Conference on Software Maintenance*. [S.l.]: IEEE, 1994. (ICSM'94).
- JENKINS. 2019. URL: <https://jenkins-ci.org/>.
- KASI, B. K.; SARMA, A. Cassandra: Proactive conflict minimization through optimized task scheduling. In: *Proceedings of the 35th International Conference on Software Engineering*. [S.l.]: IEEE, 2013. (ICSE '13).
- Kerzazi, N.; Khomh, F.; Adams, B. Why do automated builds break? an empirical study. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. [S.l.]: IEEE, 2014.

- KHANNA, S.; KUNAL, K.; PIERCE, B. C. A formal investigation of diff3. In: *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*. [S.l.]: Springer-Verlag, 2007. (FSTTCS'07).
- LESSENICH, O.; APEL, S.; KÄSTNER, C.; SEIBT, G.; SIEGMUND, J. Renaming and shifted code in structured merging: Looking ahead for precision and performance. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. [S.l.]: IEEE, 2017. (ASE'17).
- LESSENICH, O.; APEL, S.; LENGAUER, C. Balancing precision and performance in structured merge. *Automated Software Engineering*, Springer, 2015.
- LEVENSHTAIN, V. I. Binary codes capable of correcting deletions, insertions, and reversals. In: *Soviet physics doklady*. [S.l.: s.n.], 1966.
- MALPOHL, G.; HUNT, J. J.; TICHY, W. F. Renaming detection. *Automated Software Engineering*, Kluwer Academic Publishers, 2003.
- McKee, S.; Nelson, N.; Sarma, A.; Dig, D. Software practitioner perspectives on merge conflicts and resolutions. In: *2017 IEEE International Conference on Software Maintenance and Evolution*. [S.l.]: IEEE, 2017. (ICSME'17).
- Menezes, G. G. L.; Murta, L. G. P.; Barros, M. O.; Van Der Hoek, A. On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github. *IEEE Transactions on Software Engineering*, IEEE, 2018.
- MENS, T. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, IEEE Press, 2002.
- MERCURIAL. 2019. URL: <https://www.mercurial-scm.org/>.
- MUNAIAH, N.; KROH, S.; CABREY, C.; NAGAPPAN, M. Curating github for engineered software projects. *Empirical Software Engineering*, Springer, 2017.
- NAGAPPAN, M.; ZIMMERMANN, T.; BIRD, C. Diversity in software engineering research. In: *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*. [S.l.]: ACM, 2013. (ESEC/FSE'13).
- NGUYEN, T. N. Object-oriented software configuration management. In: *Proceedings of the 22th International Conference on Software Maintenance*. [S.l.]: IEEE, 2006. (ICSM'06).
- O'SULLIVAN, B. Making sense of revision-control systems. *Communications of the ACM*, ACM, 2009.
- PERRY, D. E.; SIY, H. P.; VOTTA, L. G. Parallel changes in large-scale software development: An observational case study. *ACM Transactions on Software Engineering and Methodology*, ACM, 2001.
- PRUDÊNCIO, J. a. G.; MURTA, L.; WERNER, C.; CEPÊDA, R. To lock, or not to lock: That is the question. *Journal of Systems and Software*, Elsevier, 2012.
- RAUSCH, T.; HUMMER, W.; LEITNER, P.; SCHULTE, S. An empirical analysis of build failures in the continuous integration workflows of java-based open source software. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. [S.l.]: IEEE, 2017. (MSR'17).

- RIGBY, P.; BARR, E.; BIRD, C.; GERMAN, D.; DEVANBU, P. Collaboration and governance with distributed version control. *ACM Transactions on Software Engineering and Methodology*, v. 5, 1996.
- SANTOS, R. d. S.; MURTA, L. G. P. Evaluating the branch merging effort in version control systems. In: *Proceedings of the 26th Brazilian Symposium on Software Engineering*. [S.l.]: IEEE Computer Society, 2012. (SBES '12).
- SARMA, A.; REDMILES, D.; HOEK, A. van der. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, IEEE Press, 2012.
- Sarma, A.; Redmiles, D. F.; van der Hoek, A. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, IEEE Press, 2012.
- SEO, H.; SADOWSKI, C.; ELBAUM, S.; AFTANDILIAN, E.; BOWDIDGE, R. Programmers' build errors: a case study (at google). In: *Proceedings of the 36th International Conference on Software Engineering*. [S.l.]: ACM, 2014. (ICSE'14).
- Shamshiri, S.; Fraser, G.; Mcminn, P.; Orso, A. Search-based propagation of regression faults in automated regression testing. In: *Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops*. [S.l.]: IEEE, 2013.
- SOUSA, M.; DILLIG, I.; LAHIRI, S. K. Verified three-way program merge. *Proceedings of the ACM on Programming Languages (OOPSLA)*, ACM, 2018.
- SOUZA, C. R. B. de; REDMILES, D.; DOURISH, P. "breaking the code", moving between private and public work in collaborative software development. In: *Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work*. [S.l.]: ACM, 2003. (GROUP'03).
- SUBVERSION. 2019. URL: <https://subversion.apache.org/>.
- TICHY, W. F. Tools for software configuration management. *SCM*, 1988.
- TRAVIS. 2019. URL: <https://docs.travis-ci.com/>.
- TRINDADE, A.; BORBA, P.; CAVALCANTI, G.; SOARES, S. Semistructured merge in javascript systems. In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.]: ACM, 2019. (ASE'19).
- WESTFECHTEL, B. Structure-oriented merging of revisions of software documents. In: *Proceedings of the 3rd International Workshop on Software Configuration Management*. [S.l.]: ACM, 1991. (SCM '91).
- YANG, W.; HORWITZ, S.; REPS, T. A program integration algorithm that accommodates semantics-preserving transformations. *SIGSOFT Software Engineering Notes*, ACM, 1990.
- Yuzuki, R.; Hata, H.; Matsumoto, K. How we resolve conflict: an empirical study of method-level conflict resolution. In: *2015 IEEE 1st International Workshop on Software Analytics*. [S.l.]: IEEE, 2015. (SWAN'17).

-
- ZHANG, K.; HASSAN, A. E. Using decision trees to predict the certification result of a build. In: *Proceedings. 21st IEEE International Conference on Automated Software Engineering*. [S.l.]: IEEE, 2006.
- ZHAO, Y.; SEREBRENIK, A.; ZHOU, Y.; FILKOV, V.; VASILESCU, B. The impact of continuous integration on other software development practices: A large-scale empirical study. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. [S.l.]: IEEE, 2017. (ASE'17).
- Zhao, Y.; Serebrenik, A.; Zhou, Y.; Filkov, V.; Vasilescu, B. The impact of continuous integration on other software development practices: A large-scale empirical study. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. [S.l.]: IEEE, 2017. (ASE'17).
- ZHU, F.; HE, F. Conflict resolution for structured merge via version space algebra. *Proceedings of the ACM on Programming Languages (OOPSLA)*, ACM, 2018.
- ZHU, F.; HE, F.; YU, Q. Enhancing precision of structured merge by proper tree matching. In: *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. [S.l.]: IEEE, 2019. (ICSE'19).
- ZIMMERMANN, T. Mining workspace updates in cvs. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. [S.l.]: IEEE Computer Society, 2007. (MSR '07).