Pós-Graduação em Ciência da Computação

"COMPARING INTEGRATION EFFORT AND CORRECTNESS OF DIFFERENT MERGE APPROACHES IN VERSION CONTROL SYSTEMS"

Por

GUILHERME JOSÉ CARVALHO CAVALCANTI

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE/2016

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GUILHERME JOSÉ CARVALHO CAVALCANTI

“COMPARING INTEGRATION EFFORT AND CORRECTNESS OF DIFFERENT MERGE APPROACHES IN VERSION CONTROL SYSTEMS"

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR(A): Paulo Henrique Monteiro Borba

RECIFE/2016

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

C376c Cavalcanti, Guilherme José Carvalho
Comparing integration effort and correctness of different merge approaches in version control systems / Guilherme José Carvalho Cavalcanti. – 2016.
100 f.: il., fig., tab.

Orientador: Paulo Henrique Monteiro Borba.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2016.

Inclui referências e apêndices.

1. Engenharia de software. 2. Reuso de software. 3. Modularidade. I. Borba, Paulo Henrique Monteiro. (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2016-059

Guilherme José Carvalho Cavalcanti

Comparing Integration Effort and Correctness of Different Merge Approaches in Version Control Systems

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

Aprovado em: 29 /02/2016

BANCA EXAMINADORA

_____

Prof. Dr. Kiev Santos da Gama
Centro de Informática / UFPE

_____

Prof. Dr. Fernando Marques Figueira Filho
Departamento de Informática e Matemática Aplicada / UFRN

_____

Prof. Dr. Paulo Henrique Monteiro Borba
Centro de Informática / UFPE
(Orientador)

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. Em particular, Paola Accioly por fazer parte e ajudar no meu trabalho desde a minha graduação. Finalmente, agradeço a FACEPE por financiar minha pesquisa, e o CIn-UFPE por todo recurso que me ofereceu.

We are what we repeatedly do. Excellence, therefore, is not an act, but a habit.

—ARISTOTLE

Resumo

Durante a integração de contribuições de código resultantes das tarefas de desenvolvimento, frequentemente desenvolvedores têm que lidar com alterações conflitantes e dedicar considerável esforço para resolver conflitos. Enquanto as ferramentas de integração não-estruturadas tentam resolver automaticamente parte dos conflitos através de similaridade textual, ferramentas semiestruturadas tentam ir mais longe, explorando a estrutura sintática de parte dos artefatos envolvidos.

Para entender o impacto das abordagens de integração não-estruturada e semiestruturada sobre esforço de integração (Produtividade) e corretude do processo de integração (Qualidade), nós realizamos dois estudos empíricos. No primeiro, com o objetivo de aumentar o atual corpo de evidência e avaliar resultados para sistemas desenvolvidos usando um paradigma de controle de versão alternativo, nós replicamos um experimento para comparar a abordagem não-estruturada e semiestruturada de acordo com o número de conflitos reportados por ambas as abordagens. Nós usamos tanto a integração semiestruturada quanto a não-estruturada em uma amostra 2,5 vezes maior do que a do estudo original em relação ao número de projetos e 18 vezes maior em relação ao número de integrações realizadas, e comparamos a ocorrência de conflitos. Semelhante ao estudo original, observamos que a integração semiestruturada reduz o número de conflitos em 55% das integrações da nova amostra. Além disso, a redução de conflitos média observada de 62% nestas integrações é muito superior à observada anteriormente. Nós também trazemos nova evidência de que o uso da abordagem semiestruturada pode reduzir a ocorrência de integrações com conflitos pela metade.

Com o intuito de verificar a frequência de falsos positivos e falsos negativos originados do uso dessas abordagens, nós seguimos adiante e conduzimos um segundo estudo empírico. Nós comparamos as abordagens reproduzindo mais de 30.000 integrações de 50 projetos, coletando evidência sobre os conflitos reportados que não representam interferências entre as tarefas de desenvolvimento (falsos positivos), e interferências não reportadas como conflitos (falsos negativos). Em particular, a nossa suposição é de que falsos positivos denotam esforço desnecessário de integração porque os desenvolvedores têm que resolver conflitos que, na realidade, não representam interferências. Além disso, falsos negativos denotam problemas de *build* ou *bugs*, impactando negativamente a qualidade do software e corretude do processo de integração. Ao analisar esses fatores críticos, esperamos orientar os desenvolvedores em decidir qual abordagem deve ser usada na prática. Finalmente, nossos resultados mostram que a abordagem semiestruturada elimina uma parte significativa dos falsos positivos reportados pela abordagem não-estruturada, mas traz falsos positivos próprios. O número global de falsos positivos é reduzido com a integração semiestruturada, e nós argumentamos que os conflitos associados aos seus falsos positivos são mais fáceis de resolver quando comparados aos falsos

positivos reportados pela abordagem não-estruturada. Observamos, também, que mais interferências deixaram de ser detectadas pela abordagem não-estruturada, mas foram detectadas pela semiestruturada. No entanto, nós acreditamos que as interferências não detectadas pela abordagem semiestruturada são mais difíceis de detectar e resolver. Por fim, nosso estudo sugere como uma ferramenta de integração semiestruturada poderia ser melhorada para eliminar os falsos positivos e falsos negativos adicionados que possui em relação à não-estruturada.

Palavras-chave: Desenvolvimento colaborativo. Integração de software. Integração semiestruturada. Sistemas de controle de versões. Estudos empíricos

Abstract

During the integration of code contributions resulting from development tasks, one likely has to deal with conflicting changes and dedicate substantial effort to resolve conflicts. While unstructured merge tools try to automatically resolve part of the conflicts via textual similarity, semistructured tools try to go further by exploiting the syntactic structure of part of the artefacts involved.

To understand the impact of the unstructured and semistructured merge approaches on integration effort (Productivity) and correctness of the merging process (Quality), we conduct two empirical studies. In the first one, aiming at increasing the existing body of evidence and assessing results for systems developed under an alternative version control paradigm, we replicate an experiment to compare the unstructured and semistructured approaches with respect to the number of conflicts reported by both merge approaches. We used both semistructured and unstructured merge in a sample 2.5 times bigger than the original study regarding the number of projects and 18 times bigger regarding the number of performed merges, and we compared the occurrence of conflicts. Similar to the original study, we observed that semistructured merge reduces the number of conflicts in 55% of the performed merges of the new sample. Besides that, the observed average conflict reduction of 62% in these merges is far superior than what has been observed before. We also bring new evidence that the use of semistructured merge can reduce the occurrence of conflicting merges by half.

In order to verify the frequency of false positives and false negatives arising from the use of these merge approaches, we move forward and we conduct a second empirical study. We compare the unstructured and semistructured merge approaches by reproducing more than 30,000 merges from 50 projects, and collecting evidence about reported conflicts that do not represent interferences between development tasks (false positives), and interferences not reported as conflicts (false negatives). In particular, our assumption is that false positives amount to unnecessary integration effort because developers have to resolve conflicts that actually do not represent interferences. Besides that, false negatives amount to build issues or bugs, negatively impacting software quality and correctness of the merging process. By analyzing such critical factors we hope to guide developers on deciding which approach should be used in practice. Finally, our results show that semistructured merge eliminates a significant part of the false positives reported by unstructured merge, but brings false positives of its own. The overall number of false positives is reduced with semistructured merge, and we argue that the conflicts associated to its false positives are easier to resolve when comparing to the false positives reported by unstructured merge. We also observe that more interferences were missed by unstructured merge and reported by semistructured merge, but we argue that the semistructured merge ones are harder to detect and resolve than the other way around. Finally, our study suggests how a

semistructured merge tool could be improved to eliminate the extra false positives and negatives it has in relation to unstructured merge.

Keywords: Collaborative development. Software merging. Semistructured merge. Version control systems. Empirical studies

List of Figures

2.1 Centralized version control paradigm. 20
2.2 Distributed version control paradigm. 21
2.3 Merge conflict. 22
2.4 Ordering Conflict(Unstructured Merge). 25
2.5 Renaming Conflict(Semistructured Merge). 26
2.6 False Negatives arising from Semistructured Merge in relation to Unstructured Merge. 28
2.7 Duplicated declaration error (Unstructured Merge). 29

3.1 Replication study design. 31
3.2 Replication boxplots of sample projects (we have hidden outliers for a better visualization). 39
3.3 Renaming conflict example taken from project NServiceBus. 43

4.1 Experimental design. 48
4.2 Intercepting `FSTMerge` tool to find false negatives (FNa(UN)) *duplicated declaration error* candidates. 51
4.3 Intercepting `FSTMerge` tool to find false negatives (FNa(SS)) *type ambiguity error* candidates. 52
4.4 Intercepting `FSTMerge` tool to find false negatives (FNa(SS)) *new element referencing edited one* candidates. 53
4.5 Intercepting `FSTMerge` tool to find false positives (FPa(SS)) *renaming/deletion* conflict candidates. 54
4.6 Set of conflicts reported by unstructured and semistructured merge 55
4.7 Boxplots describing the percentage per project of added false positives in terms of merge scenarios and conflicts. 58
4.8 Boxplots describing the percentage per project of the added false negatives in terms of merge scenarios and conflicts. 60
4.9 Observed ordering conflicts. 62
4.10 Observed renaming conflicts. 64
4.11 Renaming conflict resolved by conciliating different changes. 65
4.12 Observed false negatives. 66
4.13 When nodes conflict with structured merge. 70
4.14 False positives added by unstructured/semistructured merge in relation to structured merge. 71
4.15 Edits to different parts of the same statement (Structured Merge). 72

B.1 Boxplots describing false positives added by unstructured/semistructured merge per project in terms of merge scenarios and conflicts in relation to Structured Merge. 100

List of Tables

3.1 Characteristics of the projects django and cassandra. 34
3.2 Number of merge scenarios where semistrucutured merge reported less, the same number, and more textual conflicts, conflicting lines of code and conflicting files than unstructured merge. 35
3.3 Replication results by projects. 37

A.1 Replication sample projects list and characteristics. 83
A.2 Replication results on merge scenarios where semistructured merge reduced the numbers. 84
A.3 Replication results on merge scenarios where unstructured merge reduced the numbers. 85

B.1 Java sample project list and characteristics. 87
B.2 False positives added by semistructured merge in terms of merge scenarios. . . 88
B.3 False positives added by semistructured merge in terms of conflicts. 89
B.4 False positives added by unstructured merge in terms of merge scenarios. . . . 90
B.5 False positives added by unstructured merge in terms of conflicts. 91
B.6 False negatives added by semistructured merge in terms of merge scenarios. . . 92
B.7 False negatives added by semistructured merge in terms of conflicts. 93
B.8 False negatives added by unstructured merge in terms of merge scenarios. . . . 94
B.9 False negatives added by unstructured merge in terms of conflicts. 95
B.10 Spacing conflicts in terms of merge scenarios. 96
B.11 Spacing conflicts in terms of conflicts. (Based on the number of reported conflicts of Semistructured Merge) 97
B.12 Consecutive lines conflicts in terms of merge scenarios. 98
B.13 Consecutive lines conflicts in terms of conflicts. (Based on the number of reported conflicts of Semistructured Merge) 99

Contents

1 Introduction **16**

2 Background **19**

2.1 Version Control Systems 19

2.2 Merging Software Artefacts 21

2.2.1 Unstructured merge tools 22

2.2.2 Semistructured merge tools 23

2.3 Checking Interference 24

2.3.1 False positives added by Unstructured Merge 25

2.3.2 False positives added by Semitructured Merge 26

2.3.3 False negatives added by Semistructured Merge 27

2.3.4 False negatives added by Unstructured Merge 27

3 Replication Study **30**

3.1 Replication Design 30

3.1.1 Mining Step 31

3.1.2 Execution Step 34

3.2 Evaluation Results 34

3.3 Discussion 40

3.3.1 Semistructured merge play to its strengths 40

3.3.2 Semistructured keeps or increases the number of conflicts 41

3.4 Threats to Validity 44

3.4.1 Construct Validity 44

3.4.2 Internal Validity 44

3.4.3 External Validity 44

4 Integration Effort and Correctness **46**

4.1 Empirical Evaluation 46

4.1.1 Experimental Setup 48

4.1.1.1 Mining Step 49

4.1.1.2 Execution and Analysis Steps 50

4.1.1.2.1 False Negatives Added by Unstructured Merge — `FNa(UN)` 51

4.1.1.2.2 Maximum Number of False Negatives Added by Semistructured Merge — `FNa(SS)` 51

4.1.1.2.3 Maximum Number of False Positives Added by Semistructured Merge — FPa(SS) . . . 54
4.1.1.2.4 Minimum Number of False Positives Added by Unstructured Merge — FPa(UN) . . . 55
4.2 Evaluation Results . . . 56
4.2.1 When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort? . . . 57
4.2.2 When compared to unstructured merge, does semistructured merge compromise integration correctness by missing more task interferences? . . . 58
4.3 Discussion . . . 60
4.3.1 Integration Effort . . . 60
4.3.2 Correctness . . . 65
4.3.3 Unstructured or Semistructured merge? . . . 67
4.4 Threats to Validity . . . 67
4.4.1 Construct Validity . . . 67
4.4.2 Internal Validity . . . 68
4.4.3 External Validity . . . 68
4.5 The Structured Merge Approach . . . 68

5 Conclusions **73**
5.1 Contributions . . . 74
5.2 Related Work . . . 74
5.3 Future Work . . . 76

References **77**

Appendix **81**

A Replication Study Appendix **82**

B Integration Effort and Correctnesses Study Appendix **86**

1

Introduction

In a collaborative development environment, developers often perform tasks in an independent way using individual copies of project files. As a result, when merging separate code changes from each task, one might have to deal with conflicting changes and dedicate substantial effort to resolve conflicts. These conflicts occur due to a number of reasons. For example, when different developers make changes to the same artefact without being aware of the others' changes — the so-called *direct*, *merge* or *textual* conflicts — or when there are concurrent modifications in different artefacts, leading to build or test failures — the *indirect* conflicts (BRUN et al., 2011; KASI; SARMA, 2013). Regardless of the conflict nature, they may hamper productivity, because detecting and resolving conflicts might be a tiresome and error prone activity. As a consequence, they delay the project while developers trace the cause and seek a solution (BIRD; ZIMMERMANN, 2012).

To learn about the occurrence of conflicts and their consequences, previous empirical studies answer questions concerning when developers detect conflicts, and how often conflicts occur. ZIMMERMANN (2007), for instance, by analysing four open source projects, describes that conflicts occurred in a range from 23% to 47% of all files integration. BRUN et al. (2011) and KASI and SARMA (2013), by analysing, respectively, nine and four projects hosted on GitHub, found that conflicts occurred in 16% and 13% of all merge scenarios (a set consisting of a common/ancestor and its derived revisions). They also found evidence of projects' merge scenarios free of merge conflicts resulting in build or semantic conflicts, more specifically, 31%of projects' merge scenarios in the sample of KASI and SARMA (2013) and 8% in the sample of BRUN et al. (2011).

Such evidence motivates and guides the design of tools that use different strategies to both decrease integration effort and guarantee integration correctness. For example, to reduce integration effort, the unstructured merge approach is purely text-based and resolves conflicts via textual similarity (KHANNA; KUNAL; PIERCE, 2007). On the other hand, a structured merge tool is tailored to a specific programming language and uses knowledge of the language's syntax to resolve conflicts (WESTFECHTEL, 1991; GRASS, 1992; MENS, 2002; APIWATTANAPONG; ORSO; HARROLD, 2007; APEL; LESSENICH; LENGAUER, 2012)).

Finally, the semistructured merge approach uses part of the structural information inherent to the software's artefacts to resolve conflicts, and when this information is not sufficient nor available, it applies the usual textual resolution from unstructured merge (APEL et al., 2011).

APEL et al. (2011), in a previous empirical study, found that the semistructured approach was promising if compared to the unstructured one. By studying 24 projects using Subversion (SUBVERSION, 2015), a Centralized Version Control System (CVCS), and analysing a total of 180 merge scenarios, they found that the semistructured approach was able to reduce the number of textual conflicts in 60% of their sample merge scenarios. In these scenarios, the observed average reduction was of 34%. They also found that, in 82% of their sample merge scenarios, the semistructured approach reduced the number of conflicting lines of code by, on average, 61%; and that, in 72% of their sample merge scenarios, semistructured merge reduced the number of conflicting files by, on average, 28%.

Given the importance of such tools for collaborative software development, to understand the impact of the unstructured and semistructured merge approaches on integration effort (Productivity) and correctness of the merging process (Quality), we conduct two empirical studies. In the first one, we further investigate APEL et al. (2011) hypothesis and replicate their study. To possibly expand external validity of the original study, we analyse different systems that use Git (GIT, 2015), a Distributed Version Control System (DVCS), since DVCSs have seen an increase in popularity compared to traditional CVCS, and offer extra information, such as merge tracking, that help us to better understand software development processes (BIRD et al., 2009; BRINDESCU et al., 2014). We then compare semistructured merge to the unstructured approach in 3266 merge scenarios from 60 projects, a sample 2.5 times bigger than the original study regarding the number of projects and 18 times bigger regarding the number of merge scenarios. Besides that, our methodology allows us to collect evidence about the occurrence of conflicting code integrations and, thus, compare with those from the studies of BRUN et al. (2011) and KASI and SARMA (2013). While this evidence rely on the use of an unstructured merge tool, here, we are able to bring new evidence about the occurrence of conflicting code integrations considering the use of a semistructured merge implementation.

Motivated by the achieved results on the first study, in order to verify the frequency of false positives and false negatives arising from the use of these merge approaches, we move forward and we conduct a second empirical study. More specifically, while APEL et al. (2011) compare these merge approaches with respect to the number of reported conflicts, we investigate which fraction of the detected conflicts are in fact relevant, that is, represent interferences between development tasks; and which fraction of the relevant conflicts (the existing interferences) they detect. In particular, such merge approaches lead to *false negatives* because they cannot detect certain kinds of interference between developers' tasks. Such false negatives might be harder to track, understand and resolve, leading to build or behavioral errors. Besides that, they lead to *false positives* when the detected conflicts do not represent interferences between developers' tasks, and resolving them is therefore an unnecessary effort. Thus, false positives decrease

productivity, and false negatives negatively impact software quality and the correctness of the merging process — factors that are critical to decide which approach should be used in practice. In this second study, we compare the unstructured and semistructured merge approaches by reproducing 34030 merge scenarios from 50 projects, while collecting evidence about which false positives conflicts reported by unstructured merge are not reported by semistructured merge, and which interferences are missed by semistructured merge and reported by unstructured merge (and vice versa). Initially, our objective was to compare integration effort and correctness also with the structured merge approach. However, we did not find sufficiently mature tools for this purpose (we discuss more details and insights about this later on this text). In particular, we investigate the following research questions:

- **RQ1:** *When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort?*
- **RQ2:** *When compared to unstructured merge, does semistructured merge compromise integration correctness by missing more task interferences?*

The remainder of this work is organized as follows:

- Chapter 2 reviews the main concepts used to understand this dissertation;
- Chapter 3 presents the replication study assessing semistructured merge in distributed version control systems, which was already published in CAVALCANTI; ACCIOLY; BORBA (2015);
- Chapter 4 presents the empirical study comparing integration effort and correctness of the unstructured and semistructured merge approaches;
- Chapter 5 draws our conclusions, summarizes the contributions of this research, and discusses related and future work.

2

Background

Here we explain the main concepts used in our work. Initially, we discuss the fundamentals of version control systems (VCSs) and its centralized and distributed paradigms in Section 2.1. In this context, we explain how software artefacts are merged and the characteristics of the unstructured and semistructured merge tools (Section 2.2). Finally, in Section 2.3, we describe the kinds of interferences between development tasks these merge approaches are able to detect or not.

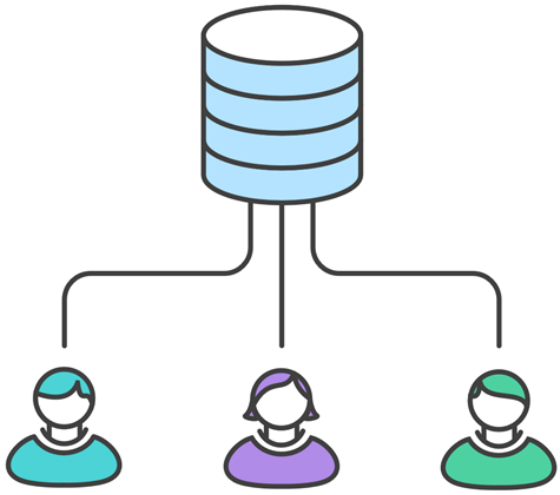
2.1 Version Control Systems

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 a developer to download and modify files in your local working area, which is periodically synchronized with the repository that contains the main version of the files. How the repositories are disposed and the developers access them determine the version control paradigm.

In a *Centralized Version Control System (CVCS)*, such as CVS (CVS, 2015) and Subversion (SUBVERSION, 2015), there is one central repository, which can accept code, and everyone synchronizes their work to it (see Figure 2.1). 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 back up can do so with no problems. The second developer

must merge in the first one's work before sending changes up, so as not to overwrite the first developer's changes.

Figure 2.1: Centralized version control paradigm.



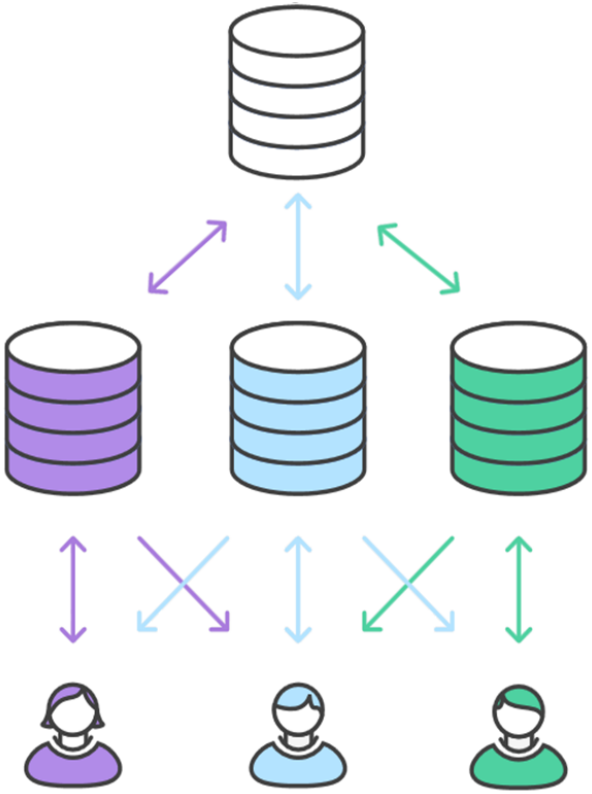
Source: the authors.

Conversely, in a *Distributed Version Control System (DVCS)*, such as Mercurial (MERCURIAL, 2015) and Git (GIT, 2015), there is no central location where the developers are working to (even if this still is a possibility). This paradigm 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 2.2). RIGBY et al. (1996) makes a interesting 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) between 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). At the end of 2012, GitHub (GITHUB, 2015), the most popular repository hosting service for Git projects, hosted over 4.6M repositories, compared to 300K repositories on SourceForge (SOURCEFORGE, 2015), the primary repository hosting service for Subversion (BRINDESCU et al., 2014). DVCS brings a whole set of novel capabilities. Using DVCSs, developers (i) can work in isolation on local copies of the repositories enabling them to work offline while still retaining full project history, (ii) they can cheaply create and merge branches, and (iii) they can commit individual changed lines in a file, as opposed to being forced to commit a whole file like in CVCSs.

Regardless of the VCS paradigm, they allow the creation of parallel versions of a software system. The problem, however, is not in creating parallel versions, but in figuring out how to merge them back into a single version (PERRY; SIY; VOTTA, 2001). While the VCS's

Figure 2.2: Distributed version control paradigm.



Source: the authors.

synchronization protocol allows rapid parallel development, it also allows developers to make conflicting changes inadvertently.

2.2 Merging Software Artefacts

In order to merge multiple source code artefacts, 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 other information from the VCS. Therefore, each difference between the two revisions leads to a conflict since it cannot be decided whether only one of the revisions introduced a change to the code or both. It also cannot be determined 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, and thus it has more information at its hands to decide where a change came from and whether it creates a conflict or not (PERRY; SIY; VOTTA, 2001; O'SULLIVAN, 2009).

Conflicts are frequent, persistent, and appear not only as overlapping textual edits — the *direct*, *merge* or *textual* conflicts — but also as subsequent build and test failures — the *indirect* conflicts (BRUN et al., 2011; GUIMARãES; SILVA, 2012). ZIMMERMANN (2007), for instance, by analysing four open source projects, describes that merge conflicts occurred in a

range from 23% to 47% of all files integration. BRUN et al. (2011) and KASI and SARMA (2013), by analysing, respectively, nine and four projects hosted on GitHub, found that merge conflicts occurred in 16% and 13% of all performed merges. They also found evidence of projects' merges free of merge conflicts resulting in build or behavioral conflicts, more specifically, 31%of projects' merges in the sample of KASI and SARMA (2013) and 8% in the sample of BRUN et al. (2011).

These conflicts emerge due to concurrent work, because developers are not aware of others' changes, and become more complex as changes grow without being integrated and as further developments are made. Conflicts are so annoying that some developers do not merge as frequently as desirable because of difficult merges (GRINTER, 1995), and developers rush their tasks to avoid being the ones responsible for the merge (SOUZA; REDMILES; DOURISH, 2003). 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.

2.2.1 Unstructured merge tools

While the VCSs have evolved over the years in several stages to cope with increasing demands, the tools that actually perform the merges have not evolved that much. When it comes to merges in VCSs, the state of the art is performing a textual, line-based merge (MENS, 2002). A popular example for an unstructured merge tool is GNU merge, which was developed to perform a three-way merge (DIFFUTILS, 2015). GNU merge works in the same way as rcsmerge, which was released as part of the Revision Control System (RCS) in 1982 and is today part of the GNU project (TICHY, 1985). Unstructured merge tools used by VCSs work and behave in a similar way as GNU merge does.

Basically, when merging files, an unstructured merge tool compares the modified files in relation to its common ancestor (the version from which the files have been derived), line by line, and detects the smallest sets of differing lines (*chunks*). As observed by KHANNA; KUNAL; PIERCE (2007), for each of the detected chunks, the algorithm checks whether there is an element common to all three revisions, separating the chunk's content into two distinct areas. If the developers modify the content in the same area, the algorithm reports a conflict. In the case a conflict was detected, it is displayed at the relevant place in the output as shown in Figure 2.3.

Figure 2.3: Merge conflict.

```
<<<<<<<  file1
conflicting lines in file1
=======
conflicting lines in file3
>>>>>>>  file3
```

Source: the authors.

The benefits of an unstructured 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 method is very effective. However, since this type of merge does not utilize knowledge about the structure of the input documents and the syntax of respective languages, it might miss conflicts, report spurious conflicts and is likely to produce syntactically incorrect output (HORWITZ; PRINS; REPS, 1989; BUFFENBARGER, 1995).

2.2.2 Semistructured merge tools

Semistructured merge, proposed by APEL et al. (2011), works on simplified parse trees, representing the program structure. Through an annotated grammar, it provides information about how nodes of certain types (methods, classes, etc.) and its subtrees can be merged. Such parse trees, include some but not all structural information of a program. Concerning Java, for example, classes, methods and fields are contained in the tree, whereas statements and expressions are hidden in the leaves of the tree in form of plain text (from there came the *semi* in the approach's name).

The idea is to use the structured information contained in the trees to merge revisions with less conflicts than unstructured merge, while not having to deal with the merging of separate statements on the syntax level inside of method body (which increases the complexity of the comparison algorithms, and affects the performance in full structured merges (APEL; LESSENICH; LENGAUER, 2012)). This trade-off allows semistructured merge to support a larger set of programming languages than full structured merges (because not the entire source code is parsed), and to provide superior conflict resolution in most cases, compared to conventional unstructured merge (APEL et al., 2011).

APEL et al. (2011) state that the ability of semistructured merge to resolve certain conflicts is based on the observation that the order of certain elements (classes, interfaces, methods, fields, and so on) does not matter — the so-called *ordering conflicts*. By abstracting the document structure as tree, the approach can identify ordered items. When semistructured merge doesn't know how to merge a software element, such as method body with statements (where the order *matters*), it represents the elements as plain text and uses (*calls*) conventional unstructured merge. It also allows special conflict handlers to be added, much like as plugins, to resolve specific cases of conflicts that the developers would like to resolve automatically. (Throughout the text we show examples of merged code by the two approaches.)

The merge algorithm itself is implemented via superimposition of the simplified parse trees (APEL; LENGAUER, 2008). With superimposition, two trees are composed by composing their corresponding nodes, starting from the root and proceeding recursively. Two nodes are composed to form a new node (1) when their parents (if there are parents) have been composed,

that is, they are on the same tree level, and (2) when they have the same name and type. The new node receives the name and type of the nodes that have been composed. If two nodes have been composed, the process of composition proceeds with their children. If a node has no counterpart to be composed with, it is added as separate child node to the composed parent node. This recurses until all leaves have been reached. Therefore, this approach allows to identify commutative and associative declarations, and implies that method body only have to be merged if the signature of two methods is identical. In this case, unstructured merge is launched to merge the method body.

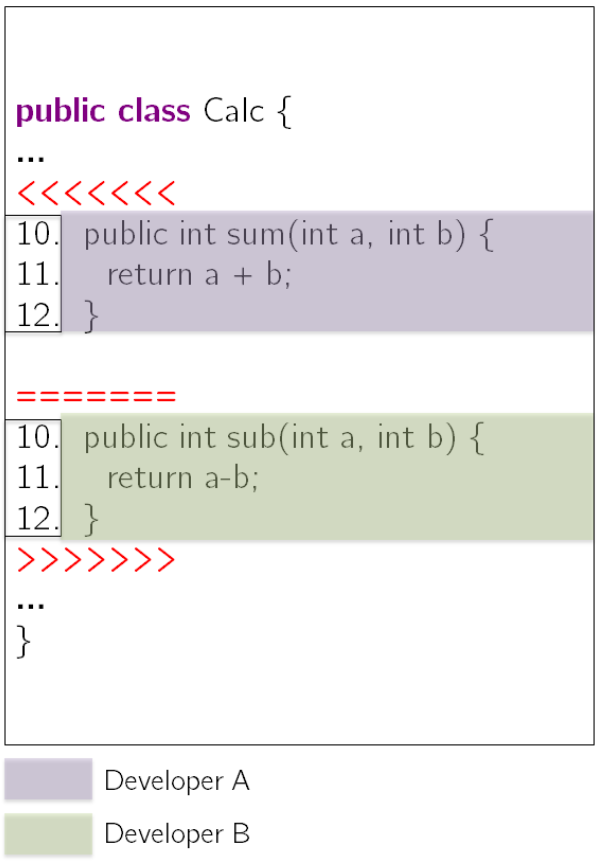
2.3 Checking Interference

Ideally, a merge tool should be able to detect interference between development tasks, reporting the interfering changes as conflicts, and automatically integrating the noninterfering ones (HORWITZ; PRINS; REPS, 1989; PERRY; SIY; VOTTA, 2001). As stated by PERRY; SIY; VOTTA (2001), the basic problem is that we have parallel changes made to a software system by multiple people, and these changes represent interactions that may interfere with each other. In particular, according to GOGUEN; MESEGUER (1982), one group of developers using a certain set of commands is noninfertering with another group if what the first group does with those commands has no effect on what the second group expects. In that sense, merge tools might report conflicts that do not represent interferences between development tasks (false positives), and they might also let interferences go on undetected (false negatives).

Despite being hard to establish ground truth on what are false positives and false negatives in the context of software merging, because, in particular, interference in this context is not computable (although it would be possible with the help of specialists for a small sample), it is still possible to compare different merge approaches with regard to the *added* occurrence of false positives and false negatives from one approach to another. When comparing unstructured merge with semistructured merge, it is necessary to look to where the approaches behave differently. So, we analysed the algorithms of both approaches to observe how and where they behave differently, and how such differences might lead to false positives and false negatives. In particular, as semistructured merge calls the unstructured approach inside methods body, in such cases, the false negatives and false positives of them are the same. Therefore, the differences in terms of false positives and false negatives between these merge approaches rely on how the approaches behave outside methods declarations.

Regarding the false negatives added by semistructured merge, they are due to changes occurring in the same code area (otherwise, unstructured merge would not report conflict) to different elements (otherwise, semistructured merge would match the elements) that can generate build or behavioral problems. The false negatives added by unstructured merge is just the opposite, that is, changes taking place in distinct code area of the same element (thus, semistructured merge could match the elements) that can cause build or behavioral problems.

Figure 2.4: Ordering Conflict(Unstructured Merge).



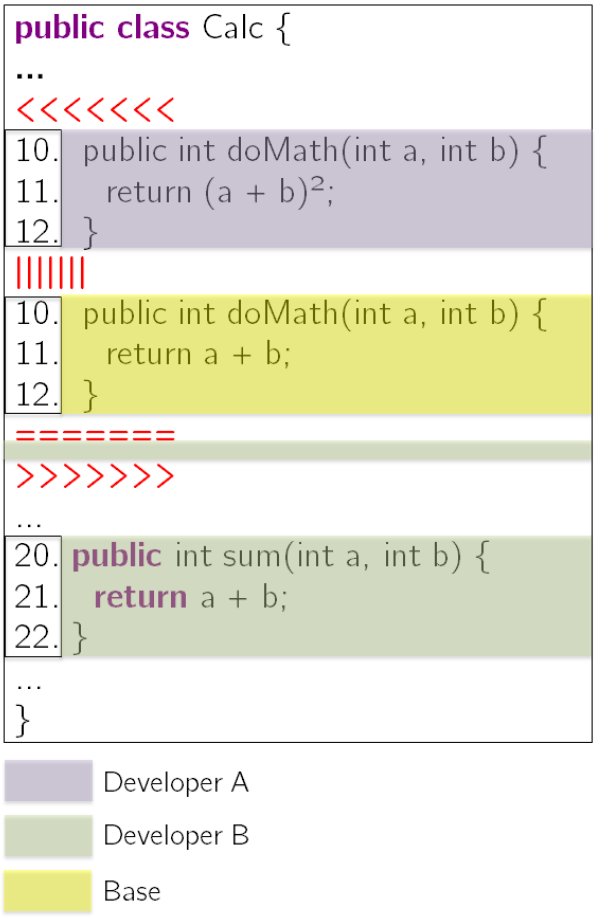
Source: the authors.

Similarly, the false positives added by unstructured merge are due to changes not representing interferences to different elements taking place in the same code area. Finally, the false positives added by semistructured merge are due to changes not representing interferences in different code area of the same element. Given this reasoning, in the following sections we describe the observed false positives and false negatives.

2.3.1 False positives added by Unstructured Merge

Beginning with the false positives added by unstructured merge, due to superimposition, semistructured merge can identify commutative and associative declarations. This is not the case for unstructured merge. As observed by APEL et al. (2011), one of the main weaknesses of unstructured merge is its inability to detect reordered declarations. In Java, for instance, a change in the order of methods and fields has no semantic impact on the program behavior, but using the unstructured approach to merge such changes will lead to conflicts (false positives) — the *ordering conflicts*. This situation is illustrated in Figure 2.4, where two different methods (*sum* and *sub*) were added in the same area of the text, leading to conflict by unstructured merge.

Figure 2.5: Renaming Conflict(Semistructured Merge).



Source: the authors.

2.3.2 False positives added by Semitructured Merge

Nevertheless, semistructured merge adds false positives too. More specifically, if a method is renamed or deleted in one revision, the merge algorithm is not able to map the renamed/deleted element to its previous version. This happens because the algorithm considers the identifier of the elements during the matching of the nodes of the parse trees. When the method is renamed or deleted, the algorithm can only identify two of the three versions expected by the three-way merge. In such cases, a conflict only occurs if the other revision changes the original method's body, because the three versions become different (the original, the body-changed and the renamed/deleted). In Figure 2.5, we illustrate this situation: one of the developers renamed the *doMath* method to *sum*, while the other developer kept the old signature, but edited its body. Besides, since semistructured merge cannot map *sum* to its previous version (*doMath*), the *sum* method does not appear in the conflict, that is, the *sum* method is not surrounded by the conflict markers.

Particularly, not all renaming or deletion conflicts are semistructured merge false positives and, indeed, represent interferences between development tasks. This happens, for instance, when there still are references (calls) to the original signature after the renaming/deletion. This

can occur when the developer who edits the method body adds a new reference to that method, indicating his intentions of using the method with the changes introduced by him, calling it with the original signature. This is now interfering with the renaming/deletion done in the other revision because the method with the original signature is no longer available. A renaming conflict can be considered false positive when the developer who did the renaming did not edit the method body. In such cases, the changes do not affect the expectations of the developers: the method will have the behavior desired by one developer, and will be called as wanted by the other developer. In turn, deletion conflicts are more restricted because the developer who edited the method has its contributions overwritten by the deletion, which is clearly an interference, unless the method was not previously used (*dead code*). Finally, it is important to note that if the renaming/deletion and the changes to the method's body correspond to different areas of the text, it will not lead to conflict by unstructured merge.

2.3.3 False negatives added by Semistructured Merge

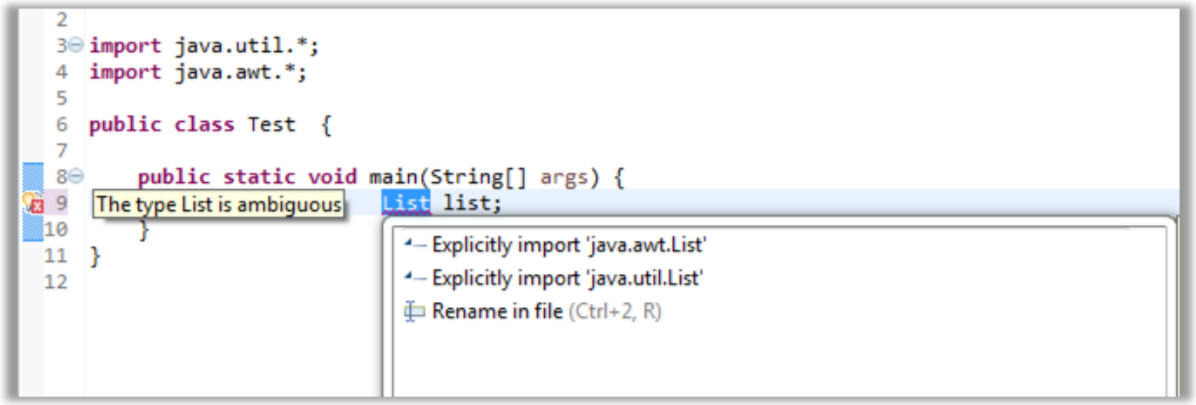
When it comes to semistructured merge false negatives not missed by unstructured merge, the issues emerge again from semistructured merge's matching process. The semistructured approach assumes that the order of import declarations does not matter, thus, it allows two developers to add import declarations in the same area of the program text. However, this might lead to *type ambiguity error* (see Figure 2.6 (a)) because the import declarations might involve members with the same name but from different packages. In the illustrated case, both imported packages have a `List` class. In that case, an unstructured tool would report a conflict because the import statements were added in the same or adjacent lines of the code.

Our analysis also indicated false negatives added by semistructured merge happening when one developer adds a new element referencing an existing one, and this existing element is edited by the other developer. As these changes correspond to different elements (technically, different nodes in the generated AST of semistructured merge), there is no matching between them, and, therefore, the approach does not report conflict. In such cases, the developer who added the new element might not be expecting the changes made to the element referenced by his element, possibly leading to behavioral errors. However, it is possible that these changes have occurred in the same area of the program text, leading to conflict by unstructured merge. In the example illustrated in Figure 2.6 (b), the new method *composed* is referencing the method *doMath* edited by the other developer.

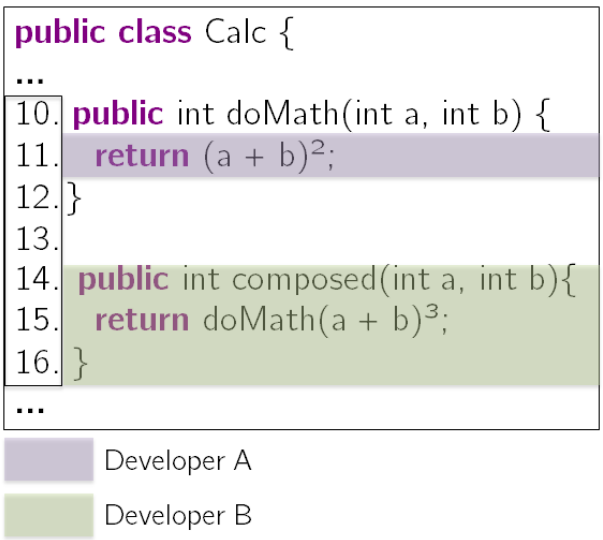
2.3.4 False negatives added by Unstructured Merge

Conversely, regarding unstructured merge false negatives in relation to semistructured merge, the problem is that the unstructured approach allows any kind of duplicated content in the same file as long as the changes occurs in different areas of the text. In terms of a programming language, unstructured merge raises no conflict when two developers add declarations with the

Figure 2.6: False Negatives arising from Semistructured Merge in relation to Unstructured Merge.



(a) Type Ambiguity Error

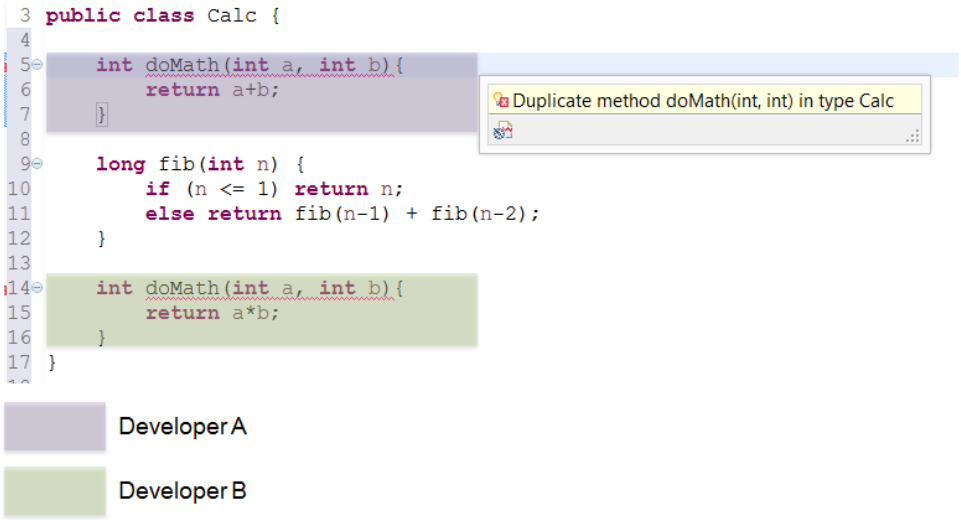


(b) Duplicated Declaration Error
Source: the authors.

same identifier in different areas of the code, leading to a *duplicated declaration error* by the compiler. In Figure 2.7, we illustrate this situation. Both developers added methods with the same signature but with different behaviors in different areas of the text, not leading to conflict by unstructured merge.

Exploiting the syntactic structure of the artefacts involved in code integration has shown reduction in the overall number of reported conflicts, mainly by eliminating obvious false positives of unstructured tools (WESTFECHTEL, 1991; GRASS, 1992; MENS, 2002; APIWATTANAPONG; ORSO; HARROLD, 2007; APEL et al., 2011; APEL; LESSENICH; LENGAUER, 2012). However, those studies do not further evaluate whether the observed reduction was obtained at the expense of introducing false negatives, or even other kinds of false positives that might be harder to resolve than the eliminated ones. This information is important because, in practice, false positives represent unnecessary integration effort, which decrease productivity,

Figure 2.7: Duplicated declaration error (Unstructured Merge).



Source: the authors.

because developers have to resolve conflicts that actually do not represent interferences. Besides that, false negatives represent build or behavioral errors, negatively impacting software quality and correctness of the merging process. Having the evidence of false positives and false negatives resulting from the merge approaches might be beneficial, because this way the developers could choose the merge approach to be used in terms of their impact on integration effort and correctness.

3

Replication Study

As we discussed in the previous chapter, semistructured merge is able to resolve conflicts, such as ordering conflicts, that unstructured merge cannot resolve. This way, we expect the semistructured approach to be able to decrease the occurrence of conflicts compared with the unstructured one. In this case, it is interesting to know how big is the reduction, and how frequently such conflicts occur in software projects. This is what led APEL et al. (2011) to conduct a empirical study to evaluate semistructured merge.

To further evaluate semistructured merge in a different context formed by different projects and version control paradigm, as a replicated study, we want to investigate the original study hypothesis.

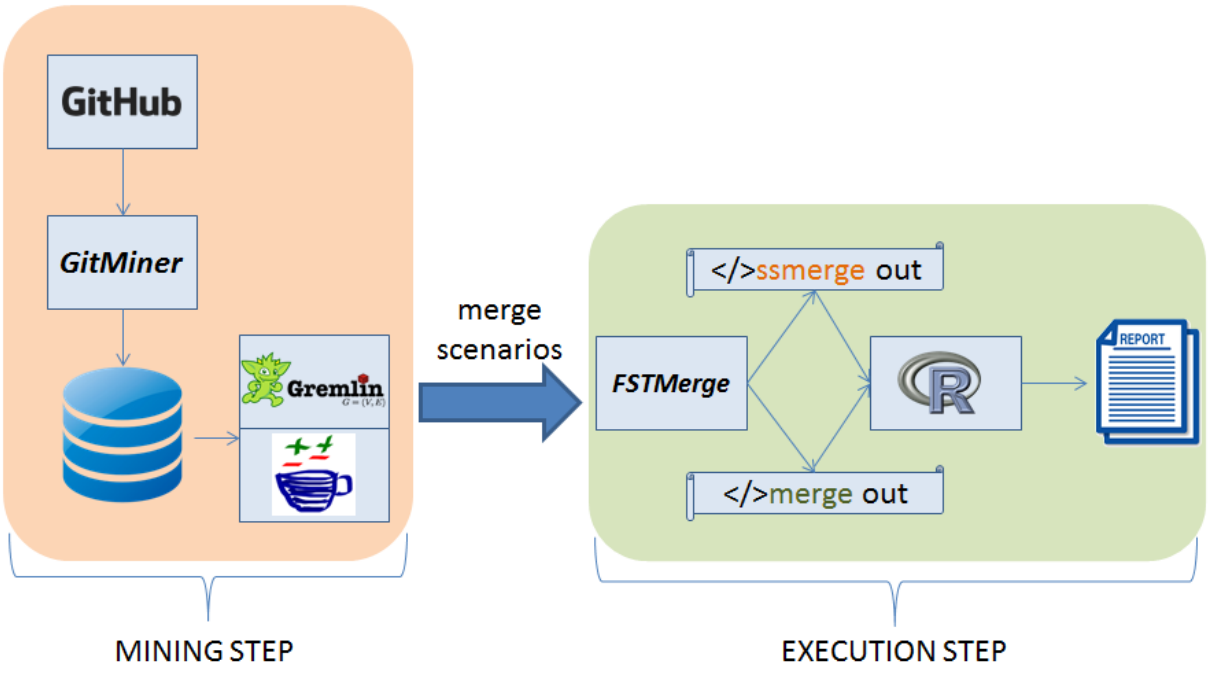
Original Hypothesis. *Many of the conflicts that occur in merging revisions are ordering conflicts, which can be resolved automatically with semistructured merge. An additional fraction of conflicts can be resolved with conflict handlers.*

Besides that, we want to go further and provide evidence of the occurrence of conflicting merge scenarios as done in the studies of BRUN et al. (2011) and KASI and SARMA (2013). While this evidence rely on the use of an unstructured merge tool, we aim to provide new evidence based on the use of the semistructured approach. By providing such evidence, the developers will have an idea of which merge approach leads to more conflicting code integrations.

3.1 Replication Design

During this replication, we designed a two-step study as illustrated in Figure 3.1. The study design is composed by a *mining* step, which is different from the original study because we are exploring DVCS repositories instead of CVCS ones; and by a *execution* step, which is similar to the original study, where we use the tool and scripts provided by the original authors. In particular, in the mining step, we select the sample projects written in Java, Python and C# — the languages supported by the semistructured merge tool used in this study. Besides that, we built tools that mine DVCS repositories to collect a number of merge scenarios from the selected projects. Subsequently, in the execution step, we use a prototype of the semistructured

Figure 3.1: Replication study design.



Source: the authors.

approach (the `FSTMerge` tool) in order to run the selected merge scenarios using both merge approaches. We also use R scripts to collect metrics on the number of conflict markers per file (*textual conflicts*), lines of code surrounded by these conflict markers (*conflicting lines of code*) and files with at least one conflict marker (*conflicting files*).

In addition, to assess the second part of the hypothesis, we evaluate the occurrence of conflicts that could be resolved with special conflict handlers — also called *semantic* conflicts in the original study.

Finally, to learn more about the nature of the studied merge approaches, we reviewed some of the merged revisions of the projects manually.

3.1.1 Mining Step

In the same way as the original study, we analyse the source code history of open source projects instead of developing our own case study, which could leave too much room for bias. How to select merge and conflicting scenarios is fundamental to the study since we want to expand the external validity of the original study.

For mining's purpose, to select the projects candidate, APEL et al. (2011) explore the SourceForge (SOURCEFORGE, 2015) open source software portal based on two criteria. First, the projects must be of reasonable but varying sizes. Second, either semistructured merge or unstructured merge must produce at least one conflict. To collect the merge scenarios, the authors had to deal with the shortcoming of Subversion has no proper mechanism allowing to identify the performed merges automatically. So, they analyse the projects' log to extract information

about the merges that developers actually performed, indicated by comments of the developers that point clearly to merges. They also considered merges that could have been performed or that are realistic considering the revision history, looking for patterns indicating sequence of multiple, alternating changes in different branches (for instance, *trunk-branch-trunk*), which indicates concurrent development and points to potential conflict scenarios (as long as the changes in different branches are not identical). Technically, they use Subversion (SUBVERSION, 2015) to browse the revision histories and to check out revisions. Finally, they selected a sample of 24 projects with 180 merge scenarios written in Java, Python and C#.

We decided to explore Git (GIT, 2015) and GitHub (GITHUB, 2015) because they offer extra information that help us to better understand projects' development processes, including merge tracking (BIRD et al., 2009; BRINDESCU et al., 2014), which allows us to conduct our mining step in an automatic and systematic way .

To select projects candidate, we first search for the top 100 projects with the highest number of stars on GitHub's advanced search page[1], which indicate the more popular projects and also suggests relevant project activity. After sorting the search result by the number of stars in descending order, we selected 60 projects varying on *number of commits* and *number of developers* (when we got around this number, we didn't observe significant variance in the chosen criteria). In particular, we chose these criteria because we do not want inactive projects, which could denote that they are toys or projects that do not reflect any current model of development. Besides, we believe that the greater the number of developers, the greater is the possibility of having conflicting code contributions resulting from development tasks, although only a portion of them may actually have worked on the project. Finally, we believe that the greater the number of commits, the greater is the possibility of finding a merge commit — a commit that represents a three-way merge. Regarding the sampling of merge scenarios, we are interested on those, from the entire histories of the projects, that had already shown conflicts on Git to provide the evidence of the occurrence of conflicting merge scenarios. Therefore, we used tools to mine GitHub and to reproduce Git merges.

We mine GitHub using the `GitMiner` tool (GITMINER, 2015). `GitMiner` receives a project or a particular GitHub user, connects to GitHub via GitHub's API and loads all the metadata available about the project or user. Finally, the metadata is stored in a Neo4j graph database (NEO4J, 2015). Such architecture is the most suitable for understanding individual dependencies between projects and its users activities, including their commits (RODRIGUEZ, 2010), which is fundamental in our mining step.

We then built a script to query the database to retrieve users' commits and to execute code integration using Git built-in merge tool, which is by default unstructured. Our script uses `Gremlin`, a graph traversal language (GREMLIN, 2015), and `JGit`, a Java library implementing the Git version control system (JGIT, 2015). The graph database represents commits as nodes with an `isMerge` attribute indicating whether the commit is a merge commit.

[1] `https://github.com/search/advanced`

In addition, each merge commit has two parents (here we call them *left* and *right* revisions) and a common ancestor (the *base* revision). Therefore, to identify merge scenarios, we query (1) the ID of all merge commits, checking which commits have the `isMerge` attribute with a true value, and (2) the ID of the revisions (base, left and right) that lead to the merge commit, and, thus, constitute the merge scenario. Finally, since we are interested on merge scenarios that had already shown conflicts, we use Git built-in merge tool to merge the revisions from the merge scenarios and we filter them by the conflicting ones, looking for those which have at least one file marked with conflict markers. For simplicity, from these merge scenarios, we only store the files with conflicts. This way, we run the study with the files and merge scenarios that had already shown conflicts on Git.

Following this procedure, we came to a sample of 4678 merge scenarios from 60 projects, written in Java, Python and C#. However, we had to discard 1412 merge scenarios because the semistructured implementation used in both studies does not support the files of these scenarios due to incompatibility between its current annotated grammar and the source code of the merge scenario. For example, there is an incompatibility with object initializers with named objects in C# and conditional assignment in Python, leading to parsing errors. As we will discuss in Section 3.4, this issue might threat internal validity because we could bias our results with scenarios only supported by the semistructured merge tool used in the study. We therefore run the study with 3266 merge scenarios, a number 18 times bigger than the original study (and 2.5 times bigger regarding the number of projects), which possibly expands the external validity of the original study. (The original study does not mention any incompatibility issue and discards.)

It is also important to note that, despite our assumptions underlying our project selection criteria, the development model adopted by the projects as well as Git internal mechanisms, such as `git rebase`, which allows the history of the repository to be rewritten, might decrease the occurrence of merge commits and conflicting merges commits. More specifically, we analyse projects that adopt a pull-based development, so we may have detected fewer conflicts that could exist if the project used a push-based development because in such cases the conflict may have been perceived by the integrator, who may have rejected the pull. We may also have detected fewer conflicts because merge commits that led to conflicts may have been erased from project's history with `git rebase`. For instance, as shown in Table 3.1, *django* is the project with the largest number of collaborators, a large number of commits, but proportionately few merge commits and conflicting merge commits. Its documentation makes explicit that they use a pull-based development model, with the frequent use of `git rebase` in the contribution process.[2] In turn, *cassandra* has substantially less collaborators than django, less commits, but nine times more merge commits and far more conflicting merge commits. This happens because this project has a patch-based contribution process, with no specific strategy to avoid conflicts.[3]

[2] `https://docs.djangoproject.com/en/1.7/internals/contributing/writing-code/working-with-git/`

[3] `http://wiki.apache.org/cassandra/HowToContribute`

All information on the sample is available in Appendix A.

Table 3.1: Characteristics of the projects django and cassandra.

Name	KLOC	Collaborators	Commits	Merge Commits	Conflicting Merge Commits	Language
cassandra	9	63	15427	4820	896	Java
django	12	717	19232	557	15	Python

3.1.2 Execution Step

After collecting the sample projects and merge scenarios in the previous step, we apply both unstructured and semistructured approaches to the merge scenarios and we analyse the number of textual conflicts, conflicting lines of code and conflicting files resulting from both unstructured and semistructured merge. Besides that, we also count the occurrence of semantic conflicts. We performed this step exactly like the original study, using the same tools and scripts. In particular, we use a virtual machine pre-configured provided by the original authors.

In the original study, the authors implemented a first prototype of a semistructured merge tool, called `FSTMerge`, which internally uses the Linux merge tool as unstructured merge tool.[4] `FSTMerge` is able to resolve ordering conflicts and can be extended with special conflict handlers. Currently, the tool has conflict handlers for 54 structural elements of Java, C#, and Python. Typically, the handlers are very simple and only flag a semantic conflict. So, they did not implement specific resolution strategies but serves just to count situations in which they can be applied, which is sufficient for the quantitative analysis. The tool takes as input the three revisions (base, left and right) that compose a merge scenario, obtained in the previous step, and applies both unstructured and semistructured merge. Finally, we use R scripts to account the results.

3.2 Evaluation Results

After performing the study described in the previous sections, we present the achieved results before discussing their implications in Section 3.3. Full results are available in Appendix A.

In the first row of Table 3.2, we summarize the number of merge scenarios according to the results presented by both semistructured and unstructured merge. More specifically, we show for how many merge scenarios semistructured merge reported less, the same number and more textual conflicts than unstructured merge. Besides that, in the second and third rows, we show

[4] `http://www.gnu.org/software/rcs/`

similar comparisons for conflicting lines of code and conflicting files. For example, 1804 in the first row and column indicates the number of code integrations in which the sum of conflicts markers resulting from semistructured merge is less than that from the unstructured merge. In a similar way, 581 in the second row and column indicates the number of code integrations in which the sum of lines of code surrounded by the conflict markers resulting from both approaches is identical. Finally, 9 in the third row and column indicates the number of code integrations in which the sum of files with at least one conflict marker resulting from the unstructured merge is less than that from semistructured merge.

Table 3.2: Number of merge scenarios where semistrucutured merge reported less, the same number, and more textual conflicts, conflicting lines of code and conflicting files than unstructured merge.

	Semistructured Reported Less	**Semistructured and Unstructured Reported the Same Number**	**Unstructured Reported Less**
Textual Conflicts	1804 (55.24%)	1179 (36.1%)	283 (8.66%)
Conflicting LOC	2323 (71.13%)	581 (17.79%)	362 (11.08%)
Conflicting Files	1566 (47.95%)	1691 (51.77%)	9 (0.28%)
Total Merge Scenarios			**3266**

Concerning textual conflicts, semistructured merge reduced the numbers in 55.24% of the sample merge scenarios. In these scenarios, we observed a reduction of, on average, 62.3%, with a maximum of 100% in a merge scenario from sympy, minimum of 2.7% in a merge scenario from OG-Platform and standard deviation of 23.57%. On the other hand, in 8.66% of the sample merge scenarios, unstructured merge reduced the number of textual conflicts compared to semistructured merge. In these scenarios, we observed a reduction of, on average, 40.49%, with a maximum of 100% in a merge scenario from nova, minimum of 0.99% in a merge scenario from jedis and standard deviation of 20.09%. In the remainder 36.1% of the sample merge scenarios, both approaches reported a similar number of textual conflicts.

In terms of conflicting lines of code, semistructured merge reduced the numbers in 71.13% of the sample merge scenarios. In these scenarios, we observed a reduction of, on average, 81.04%, with a maximum of 100% in the same merge scenario from sympy, minimum of 0.24% in another merge scenario from sympy and standard deviation of 13.52%. On the other hand, in 11.08% of the sample merge scenarios, unstructured merge reduced the number of conflicting lines of code compared to semistructured merge. In these scenarios, we observed a reduction of, on average, 43.62%, with a maximum of 100% in the same merge scenario from nova, minimum of 0.57% in a merge scenario from cassandra and standard deviation of 21.12%. In the remainder 17.79% of the sample merge scenarios, both approaches reported a similar number of conflicting lines of code.

Regarding conflicting files, semistructured merge reduced the numbers in 47.95% of the sample merge scenarios. In these scenarios, we observed a reduction of, on average, 65.51%, with a maximum of 100% in the same merge scenario from sympy, minimum of 2.33% in a

merge scenario from OpenRefine and standard deviation of 25.12%. On the other hand, only in 0.28% of the sample merge scenarios, unstructured merge reduced the number of conflicting files compared to semistructured merge. In these scenarios, we observed a reduction of, on average, 74%, with a maximum of 100% in the same merge scenario from nova, minimum of 25% in another merge scenario from cassandra and standard deviation of 27.92%. In the remainder 51.77% of the sample merge scenarios, both approaches reported a similar number of conflicting files.

Besides that, seeing that we filtered the merge scenarios by the conflicting ones (see Section 3.1.1 for more details), we are able to provide evidence about the occurrence of conflicting merge scenarios as it was done in the studies of BRUN et al. (2011) and KASI and SARMA (2013). That is, from the total of 69924 Merge Commits and 4678 Conflicting Merge Commits observed in our sample, we can determine that approximately 7% of the merge commits lead to conflicts. Similarly, BRUN et al. (2011) found that conflicts occurred, on average, in 16% of the merge scenarios, KASI and SARMA (2013) found that from 7% to 16% of the merge scenarios had conflicts. All these numbers rely on the use of an unstructured merge tool (in our case, we used the Git built-in merge tool). However, we found that semistructured merge reported occurrence of conflicts in 2362 of the analysed merge scenarios, which represents approximately 3% of the identified merge scenarios. In other words, if the semistructured approach had already been used in the studied projects, the occurrence of conflicting code integrations would have decayed from the previous 7% to remarkably 3% — an improvement of more than 50%. Note that due to the discard of merge scenarios because of the semistructured merge tool's compatibility issues, explained in Section 3.1.1, this numbers can be considered a lower bound.

In Table 3.3, we further detail what happened in the merge scenarios by showing results by projects. In this table, the arrows indicate whether semistructured merge decreased, kept or increased the numbers. For instance, projects such as monodevelop and nupic are cases where semistructured merge detects more textual conflicts than unstructured merge. In the next section, we explain that refactoring, such as renaming, and deletions is the reason for this increased number. On the other hand, projects such as Bukkit and Clojure are cases where semistructured performed similar to unstructured merge. In these cases, the explanation relies on the fact that semistructured merge calls the unstructured mechanism inside method body. In addition, from the total columns of Textual Conflicts, we observed that semistructured merge reported approximately 14,3K textual conflicts compared to approximately 18K from unstructured merge (a reduction of approximately 21%), which means that at least 3,7K textual conflicts are ordering conflicts. Likewise, we observe a substantial reduction by semistructured merge in the number of conflicting lines of code (1,25MLOC vs 283KLOC). Indeed, even in some projects where the semistructured approach reports at least the same number of textual conflicts, the number of conflicting lines of code is less than that from unstructured merge. It happens due to the semistructured merge structure-driven and fine-grained nature in which the conflicts respect boundaries of classes, methods, and other structural elements. Another substantial reduction is

Table 3.3: Replication results by projects.

Project	Unstructured			Semistructured			
	Textual Conf.	**Conf. Lines**	**Conf. Files**	**Textual Conf.**	**Conf. Lines**	**Conf. Files**	**Sem. Conf.**
astropy	28	901	5	13 ↓	378 ↓	3 ↓	2
atmosphere	96	3973	35	66 ↓	1275 ↓	26 ↓	7
Bukkit	27	1464	26	27 ↔	321 ↓	12 ↓	1
cassandra	2286	76816	1258	1576 ↓	31769 ↓	931 ↓	74
clojure	4	39	3	4 ↔	39 ↔	3 ↔	0
cloudify	150	4407	41	93 ↓	1544 ↓	30 ↓	6
CruiseControl.NET	1	8	1	1 ↔	8 ↔	1 ↔	0
cxf	1	219	1	1 ↔	16 ↓	1 ↔	1
django	10	548	9	1 ↓	22 ↓	1 ↓	4
dropwizard	16	479	10	7 ↓	86 ↓	6 ↓	2
Dynamo	336	56682	215	205 ↓	3339 ↓	84 ↓	78
edx-platform	380	7598	222	234 ↓	2620 ↓	114 ↓	55
EventStore	146	5206	99	75 ↓	861 ↓	35 ↓	2
flask	5	14	5	2 ↓	8 ↓	2 ↓	2
gradle	289	10784	206	159 ↓	1770 ↓	119 ↓	24
graylog2-server	104	3387	69	66 ↓	1022 ↓	38 ↓	9
infinispan	90	1433	46	71 ↓	702 ↓	33 ↓	1
ipython	57	4184	47	48 ↓	646 ↓	26 ↓	19
jedis	300	5613	103	276 ↓	2095 ↓	56 ↓	5
jsoup	5	76	3	1 ↓	9 ↓	1 ↓	0
junit	124	4062	75	57 ↓	539 ↓	40 ↓	36
kotlin	162	3880	86	88 ↓	1316 ↓	63 ↓	4
lucene-solr	1633	69035	723	1147 ↓	21678 ↓	485 ↓	53
matplotlib	263	7071	142	157 ↓	2300 ↓	81 ↓	34
mct	44	1344	24	33 ↓	522 ↓	19 ↓	1
mockito	76	992	32	6 ↓	72 ↓	4 ↓	0
monodevelop	911	207013	884	1096 ↑	32380 ↓	565 ↓	111
Nancy	22	536	22	17 ↓	175 ↓	16 ↓	0
netty	168	11922	92	110 ↓	3802 ↓	59 ↓	8
nova	1198	38243	750	871 ↓	10229 ↓	479 ↓	26
NRefactory	40	2156	34	10 ↓	129 ↓	10 ↓	6
NServiceBus	126	4164	109	88 ↓	1015 ↓	71 ↓	13
nupic	49	13896	38	60 ↑	2230 ↓	25 ↓	3
OG-Platform	3530	213733	2155	3266 ↓	51406 ↓	1316 ↓	406
OpenRefine	59	12408	54	106 ↑	3235 ↓	51 ↓	5
opensimulator	5	64	4	3 ↓	72 ↓	3 ↓	0
orientdb	340	15902	189	316 ↓	3948 ↓	111 ↓	68
pandas	28	380	14	18 ↓	348 ↓	6 ↓	5
Questor	78	14743	52	72 ↓	11167 ↓	49 ↓	3
ReactiveUI	34	281	32	2 ↓	27 ↓	2 ↓	0
realm-java	297	12449	190	301 ↑	4305 ↓	145 ↓	21
Rebus	12	341	12	3 ↓	31 ↓	3 ↓	0
requests	16	410	9	12 ↓	131 ↓	5 ↓	1
retrofit	35	671	17	13 ↓	216 ↓	7 ↓	2
roboguice	100	6277	63	105 ↑	1381 ↓	45 ↓	36
Rock	305	13448	170	128 ↓	2332 ↓	67 ↓	24
RxJava	49	2845	34	10 ↓	116 ↓	8 ↓	1
scrapy	5	20	5	0 ↓	0 ↓	0 ↓	11
SharpDevelop	2092	316419	1956	1701 ↓	49216 ↓	839 ↓	304
SignalR	2	14	2	1 ↓	8 ↓	1 ↓	0
slapos	90	5583	53	51 ↓	677 ↓	29 ↓	7
SparkleShare	31	640	11	30 ↓	258 ↓	10 ↓	0
sympy	785	39183	202	836 ↑	17899 ↓	109 ↓	8
taiga-back	9	257	3	1 ↓	5 ↓	1 ↓	1
testrunner	19	1465	10	15 ↓	315 ↓	7 ↓	0
tornado	16	121	16	2 ↓	16 ↓	2 ↓	16
Umbraco-CMS	557	20119	329	398 ↓	6924 ↓	212 ↓	23
voldemort	366	31424	142	257 ↓	4730 ↓	110 ↓	9
WowPacketParser	10	591	5	5 ↓	44 ↓	2 ↓	1
zamboni	4	38	4	2 ↓	4 ↓	2 ↓	0
Total	18021	1257971	11148	14320 ↓	283728 ↓	6581 ↓	1539

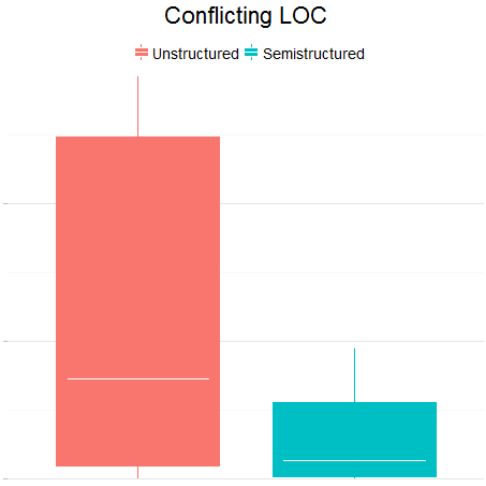
observed in the number of conflicting files (11148 vs 6581) too. Finally, similar as observed in the original study, the number of semantic conflicts found here is rather low compared to the numbers of ordering conflicts (1,5K versus 3,7K), especially when considering the quite high number of 54 elements that the semistructured merge tool handle with special conflict handlers.

In addition, discerning the significance of data by looking only at their values is not appropriate to extract the important characteristics of a dataset. For this reason, here we manage this shortcoming observed in the original study, and to better interpret data, we carried out a descriptive analysis to observe data tendency and distribution. The three boxplots in Figure 3.2 indicate that semistructured merge tends to have fewer textual conflicts, conflicting lines of code and conflicting files. The long upper whisker in the boxplots means that both approaches varied amongst the most positive quartile group, and very similar in the least positive quartile group. Note, for example, in the boxplots showing the distribution of textual conflicts and conflicting files according to the merge approach, that, despite the medians being close, the upper limit of the number of textual conflicts and conflicting files with the semistructured approach is close to the 3rd quartile when using unstructured merge. Besides that, regarding the number of conflicting lines, the overall results from semistructured merge are less than the median of the unstructured one, indicating a more significant reduction.

As a final remark, although semistructured merge has reduced the numbers, such reduction may have been insignificant if compared to the numbers presented by the unstructured mechanism. So, we need to know what is the probability of that relationship of reduction between the merge approaches, described in the achieved results, being due to random chance, and whether there is a good chance that we are right in finding that this relationship exists. Therefore, since each subject of our sample has two measurements (one with semistructured merge and other with unstructured merge) and deviates from normality for the three metrics, we performed a Wilcoxon Signed-Rank Test (WILCOXON; WILCOX, 1964) on the three metrics from both approaches and obtained a *p-value* of 2.414e-07, 4.21e-11 and 5.245e-11 regarding the number of textual conflicts, conflicting lines of code and conflicting files, respectively. Since these values are lower than 0.05, we cannot accept the null hypothesis of equality of the averages, and therefore there is evidence that semistructured merge has made a significant reduction in the numbers compared to unstructured merge. The original study did not describe hypothesis test, so we cannot compare our results to theirs.

Figure 3.2: Replication boxplots of sample projects (we have hidden outliers for a better visualization).

(a)



(b)

(c)
Source: the authors.

3.3 Discussion

In this section, we analyse the results presented in the previous section, comparing with those from the original study, and we provide additional discussion on their implications. Besides, we manually analysed selected samples of merged code in order to learn about the influence of the merge approach on the resulting code structure and to better understand our results.

3.3.1 Semistructured merge play to its strengths

In terms of merge scenarios, the number of merge scenarios in which semistructured merge reduced the metrics was lower in our study than in the original one. APEL et al. (2011) observed a reduction in the number of textual conflicts, conflicting lines of code and conflicting files in, respectively, 60%, 82% and 72% of their sample merge scenarios (compared to, respectively, 55.24%, 71.13% and 47.95% in our study). However, apart from our sample being substantially bigger than that from the original study (18 times regarding the number of merge scenarios and 2.5 times regarding the number of projects), which might explain that variance, the reductions observed in these scenarios were much more significant in our study. While they observed a reduction of 34%, 61% and 28% (with standard deviations of 21%, 22% and 12%) in the number of textual conflicts, conflicting lines of code and conflicting files, respectively, we observed a reduction of 62.3%, 81.04% and 65.51% (with standard deviations of 23.57%, 13.52% and 25.12%) in our replication. Also notable is the fact that, in the original study, unstructured merge performed better in that it reported less textual conflicts in 28% of their sample merge scenarios, compared to only 8.66% in ours (unfortunately, we do not have numbers regarding the other metrics to compare with). At least in terms of textual conflicts, it means that semistructured merge played to its strengths much more in this study than in the original study.

The observed numbers further confirm the original hypothesis that many of the conflicts that occur in merging revisions are ordering conflicts — more specifically, at least 21% (or 3,7K) of the reported textual conflicts — which can be resolved automatically with semistructured merge. We also found that an additional fraction of conflicts can be potentially resolved with language-specific conflict handlers. These findings reinforce the benefits of exploiting the syntactic structure of the artefacts involved in a code integration.

With respect to the positive implications of the results, PRUDêNCIO et al. (2012) suggests that the integration effort is the number of extra actions (additions, deletions or modifications on the artefacts) that the developer had to do during the integration to conciliate the changes made in revisions developed concurrently (here, the left and right revisions of a merge scenario). Furthermore, SANTOS and MURTA (2012) correlate the number of conflicts to that met-ric, suggesting that conflict reduction imply effort reduction. By following this reasoning, since semistructured merge reduced the number of conflicts compared to the unstructured approach, it would be also reducing the integration effort. Moreover, the substantial reduction made in

the conflicting lines of code could support this reasoning if we consider that the greater is the number of conflicting lines of code that the developer has to deal with, the greater is the effort to integrate the code contributions, because conflicting lines of code amounts to the size of the conflicts. The shortcoming of this metric is that it means only part of the time that the developer takes editing the code, it does not consider the time that the developer took reasoning about these activities. This way, this editing time can mean just one part of the total integration effort. A deeper analysis of integration effort (and correctness) is the focus of the next chapter of this work.

Finally, in the same way as the original study, we could observe that semistructured merge, due to its structure-driven and fine-grained nature, leads always to conflicts that respect boundaries of classes, methods, and other structural elements. This is not the case for unstructured merge, the conflicts are typically larger and often crosscut the syntactic program structure, which makes them harder to understand and resolve. It happens because the unstructured approach resolves conflicts via textual similarity, based on the area of the changes in the text, without any knowledge of the underlying language. Thus, it is common to find conflicts involving mismatched syntactic structures, such as a conflict involving the signature of a method and a loop statement simply because they were in the same area of the code, which hardly makes sense. Respecting structural boundaries (that is, aligning the merge with the program structure) might be beneficial, because, this way, developers could understand conflicts in terms of the underlying structure.

3.3.2 Semistructured keeps or increases the number of conflicts

In cases where the approaches reported a similar number of textual conflicts, it happened because the conflicts occurred inside method body. In this situations, the statements' order matters, and thus semistructured merge calls the unstructured one. We could see that in revisions from clojure, cassandra, flask and so on.

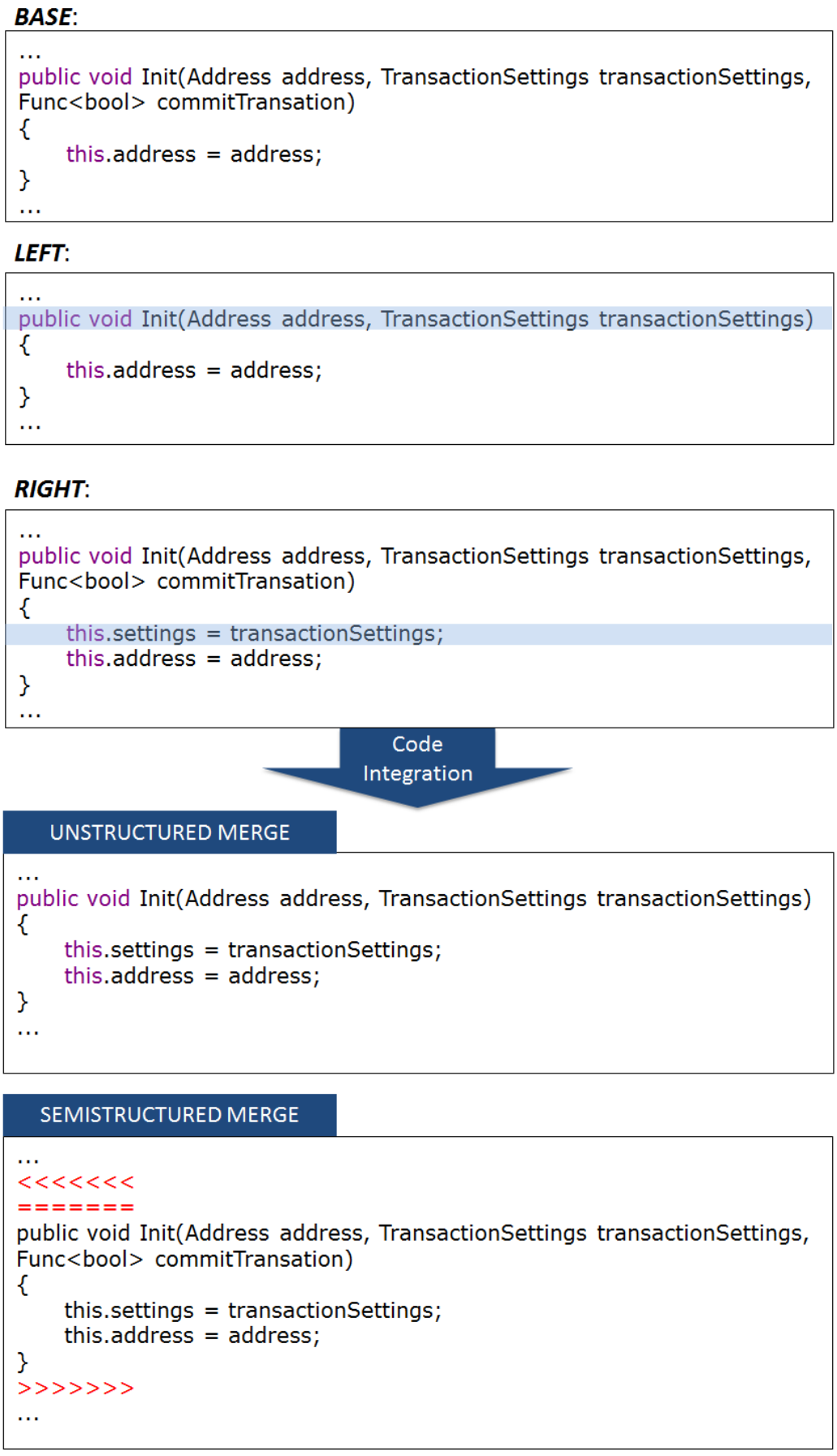
Finally, in cases where semistructured merge increases the number of textual conflicts or conflicting lines of code, APEL et al. (2011) found that refactorings, such as *renaming*, challenge semistructured merge due to the use of superimposition to merge revisions. If a program element is renamed in one revision, the merge algorithm is not aware of this fact and cannot map the renamed element to its previous version. This results in a situation in which there is, in one revision, an empty or non-existent element. We observed that this issue also applies to cases of deletions. Besides, if the other revision changes the original method body, when semistructured merge tries to integrate the methods, it will notify a conflict because the three versions are different (the original, the body-changed and the renamed/deleted). In Figure 3.3, we illustrate this situation using a snippet of code taken from a merge scenario of NServicebus. One of the developers (left) changed the signature of the *Init* method, while the other developer (right) kept the old signature, but added an extra assignment statement. Since the changes were made in

different areas of the text, unstructured merge did not report any conflict, however semistructured merge did report a conflict due to the modification in the method signature in the left revision and the modification in the method body in the right revision. It is important to notice that a method is identified by its name and the types of the formal parameters. If one of the two differ between two declarations, semistructured's merge current implementation cannot match them anymore.

The issue gets worse when, instead of a method, a directory is renamed or deleted. This happened, for instance, in merge scenarios from monodevelop, kotlin and graylog2-server. In these cases, unstructured merge reports a large conflict for each file into the directory because it cannot map the files of the directory to the corresponding files of the other revision and uses empty files instead. The same happens in semistructured merge, except that the conflicts are not reported per file but per method or constructor in the file. This results in more conflicts but the overall number of conflicting lines is smaller than in unstructured merge. The reason is that unstructured merge has a file level granularity, while semistructured merge has a structural element level granularity. Therefore, in these cases, unstructured merge flags entire files as conflicts, and semistructured merge flags only individual structural elements such as methods. A more recent study (SILVA et al., 2014) presents an approach for detecting differences, moves, and refactoring-related changes on source code through dynamic programming algorithms to find the *longest common subsequences* (HUNT; SZYMANSKI, 1977) between the files; one can improve a semistructured merge tool with a similar approach.

Whereas in the original study renaming is seen as an issue, that is, a false positive, which should not happen, if we consider the definition of *interference* given by GOGUEN; MESEGUER (1982), one group of developers using a certain set of commands is noninfertering with another group if what the first group does with those commands has no effect on what the second group expects, a fraction of the renaming/deletion conflicts can be seen as true positive. Figure 3.3 might also illustrates this example: the left developer might not be expecting for the extra assignment statement, and the right developer probably would call the *Init* method with three parameters, as in the original version. This might be an interference captured by semistructured merge but not by the unstructured one.

Figure 3.3: Renaming conflict example taken from project NServiceBus.



Source: the authors.

3.4 Threats to Validity

Since our study is a replication of the research made by APEL et al. (2011) it is natural that our study suffers from some of the same threats to validity. This holds particularly to the threats to construct and external validity. However, in our replication, we were able to improve internal and external validity.

3.4.1 Construct Validity

APEL et al. (2011) remarks that the output of semistructured merge in the presence of renaming is not satisfactory, but it still allows us to detect conflicts properly and to incorporate them in our data. A threat to the construct validity is that the number of conflicting lines of code may be estimated too low, because the renamed element is not considered. However, we believe that the occurrence of renaming-related conflicts would has little potential to impact the reduction of approximately 1MLOC we found here regarding the number of conflicting lines of code if the renamed version was considered.

3.4.2 Internal Validity

A potential threat to internal (and external) validity is also our approach to select conflicting merge scenarios. The problem is that some distributed development models make use of mechanisms, such as `git rebase`, that can rewrite the repository history, making impossible to identify merge commits of the old history. Otherwise, we could have more merge scenarios and more conflicts to analyse. This way, we have explored a lower bound of what really happened in projects' history. The impact of an increased sample on the results presented here is hard to predict. However, we still believe that we performed better than the original study in this aspect because we only analysed real merge scenarios from the sample projects instead of using "speculative" merge scenarios as happened in the original study.

Besides, one can argue that we bias the results for the reason that we discard merge scenarios not supported by the semistructured implementation used in the study. Nevertheless, we analysed source code carefully to understand this problem and we found that most of the unsupported content is language's constructions that happen inside method body, where semistructured merge works exactly like the unstructured one. Thus, it means that in such cases the results presented by both approaches would be almost the same.

3.4.3 External Validity

To increase external validity of the original study, we collected a substantial number of projects and merge scenarios written in different languages and of different domains, with a number 2.5 times bigger regarding to the number of projects and 18 times bigger regarding

the number of merge scenarios compared to the original study. However, our sample contains only open source projects hosted on GitHub. We believe that it would be also important to know how the approach performs on industrial systems. In particular, organizational factors might influence in how the tasks are assigned, and this might impact the occurrence of conflicts directly or indirectly. This way, if in such cases conflicts are uncommon, a more sophisticated approach, such as the semistructured one, might not be necessary.

4

Integration Effort and Correctness

In the previous chapters, we saw that semistructured merge is able to reduce the number of reported conflicts when compared to unstructured merge. For instance, APEL et al. (2011) showed that semistructured merge was able to reduce the number of conflicts by 34% compared to unstructured merge. In a replication of this study presented in the previous chapter, we found an even greater reduction of 62% in the number of conflicts, again in favor of semistructured merge.

However, the observed reduction might have been obtained at the expense of introducing false negatives, or even other kinds of false positives that might be harder to resolve than the eliminated by the use of the semistructured approach. So, we need further analysis because, in practice, false positives represent unnecessary integration effort, which decrease productivity, because developers have to resolve conflicts that actually do not represent interferences. On the other hand, false negatives represent build or behavioral errors, negatively impacting software quality and correctness of the merging process. This way, number of conflicts alone might not be a precise proxy for impact on productivity and quality. Having the evidence of false positives and false negatives resulting from the merge approaches might be beneficial, because this way the developers could choose the merge approach to be used in terms of their impact on integration effort and correctness.

4.1 Empirical Evaluation

Our evaluation aims to verify whether semistructured merge's reductions on the number of conflicts in relation to the unstructured merge approach actually lead to integration effort reduction with no negative impact on the correctness of the merging process. We do that reproducing merges from the development history of different projects hosted on GitHub, while collecting evidence about the occurrence of conflicts, false positives and false negatives described in Chapter 2. In particular, we investigate the following research questions:

- **RQ1:** *When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort?*
- **RQ2:** *When compared to unstructured merge, does semistructured merge compromise integration correctness by missing more task interferences?*

To answer **RQ1**, we compute the *maximum number of false positives added by semistructured merge* (spurious conflicts reported by semistructured merge and not reported by unstructured merge). We also compute the *minimum number of false positives added by unstructured merge* (spurious conflicts reported by unstructured merge and not reported by semistructured merge) metrics. Therefore, we are comparing the unstructured and semistructured merge approaches with respect to the false positives explained in Chapter 2. In particular, establishing ground truth for integration conflicts, or more generally interference between development tasks, is non-trivial and would significantly reduce the analysed sample. So, we actually analyse, both analytically and empirically, which conflicts reported by unstructured merge are not reported by semistructured merge, and vice versa. This way we can compute the number of false positives added and removed by one merge approach in comparison to the other. As the metrics name suggest, due to implementation limitations that we further detail later, they are proxies. Nevertheless, if we find that an upper bound value is lesser than a lower bound one, we can affirm that the real value underlying the *maximum* number is lesser than the real value underlying the *minimum* one.

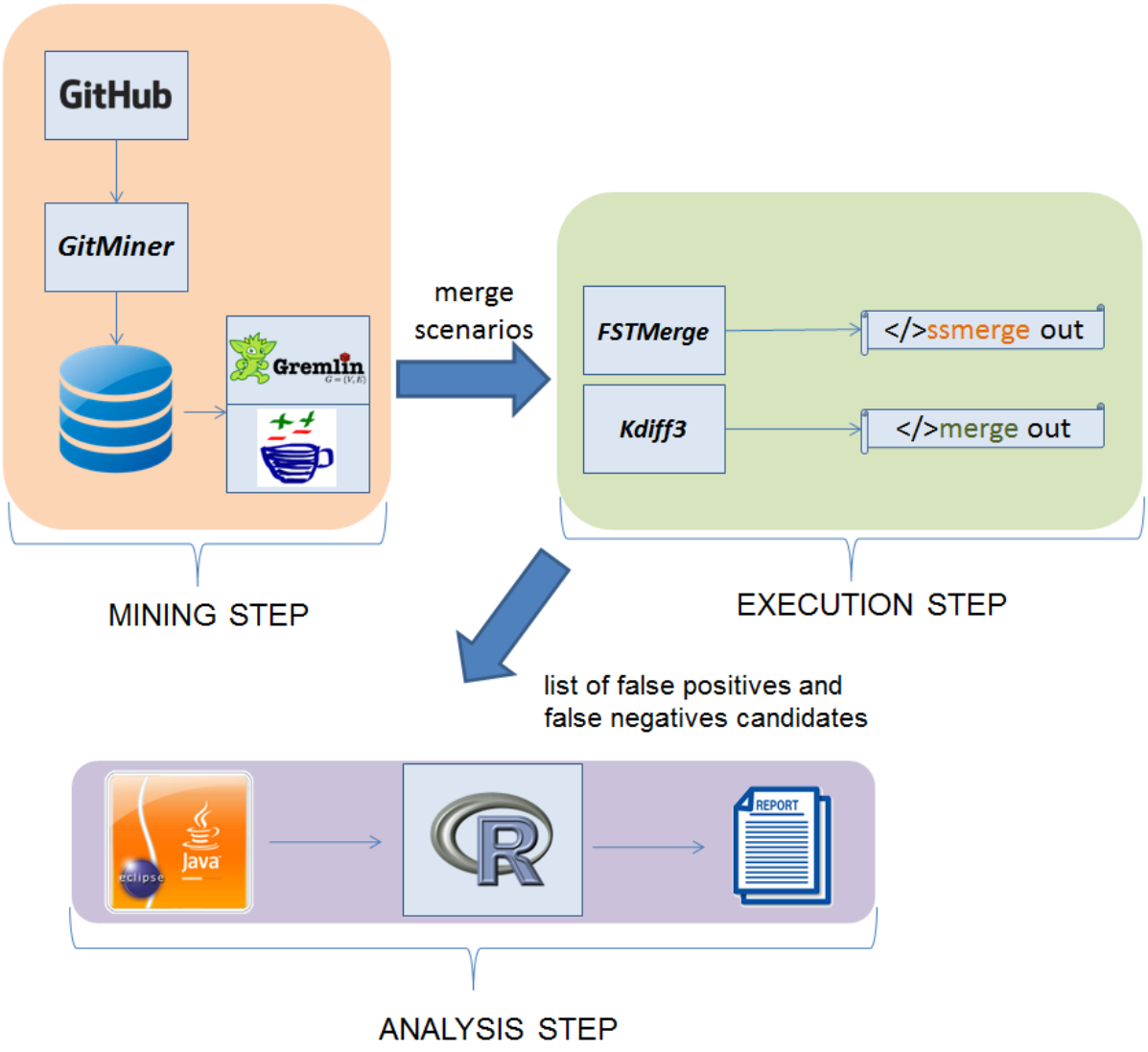
As different conflicts might demand different resolution effort, comparing conflict numbers might not be enough for understanding the impact on integration effort. However, if we find that the difference between the number of added false positives reported by the two approaches is substantial (which benefits from the comparison of maximum and minimum values), it might be possible to conclude impact on integration effort. Moreover, to understand the effort necessary to resolve different kinds of conflicts, we manually analyse a sample of the identified false positives to draw conclusions about the impact on integration effort. Our goal with this analysis is to try to verify if there is any possibility of the computed metrics be a proxy of integration effort.

For answering **RQ2**, we compute the *maximum number of false negatives added by semistructured merge* (conflicts missed by semistructured merge and correctly reported by unstructured merge), and the *number of false negatives added by unstructured merge* (conflicts missed by unstructured merge and correctly reported by semistructured merge) metrics, comparing the unstructured and semistructured merge approaches with respect to the false negatives explained in Chapter 2. Due to the mentioned difficulties on establishing ground truth to a large dataset, we are actually computing which false negatives of unstructured merge are detected by semistructured merge, and vice versa. Besides that, whereas we are able to compute the *number of false negatives added by unstructured merge* metric in a precise manner, the *maximum number of false negatives added by semistructured merge* metric is a proxy. However, it still is a valid comparison in case of finding that the upper bound value is lesser than the precise one.

4.1.1 Experimental Setup

To answer our research questions and compute the related metrics, we adapt and extend the setup of the replication study presented in Chapter 3 as illustrated in Figure 4.1. In the *mining* step, we use tools that mine Distributed VCS repositories to collect a number of merge scenarios. Subsequently, in the *execution* step, we use merge tools from both approaches in order to merge the selected merge scenarios and to find false positives and false negatives candidates. Afterwards, in the *analysis* step, we parse and compile the files to which the false positives and false negatives candidates belong to confirm their occurrence. Finally, we use R scripts to sum up the results. We explain the details in the following, explaining together the execution and analysis steps in the name of simplicity.

Figure 4.1: Experimental design.



Source: the authors.

4.1.1.1 Mining Step

This step is quite similar to the one described in Section 3.1.1. The difference relies on the selected projects and number of merge scenarios. We explain the details in the following.

To select projects candidate, we first search for the top 100 projects with the highest number of stars on GitHub's advanced search page, which might indicate considerable project relevancy and activity.[1] We restricted our sample to Java projects because the execution and analysis steps demand language dependent tool implementation and configuration. After sorting the search result by the number of stars in descending order, we selected 50 projects from that search result varying on the *number of commits* and *number of developers*, both extracted from repositories' pages on GitHub. In particular, we fixed the number of projects in 50 because when we got around this number, we didn't observe significant variance in the chosen criteria. We chose these criteria hoping that projects with more commits more likely contain more merge commits — commits that were the result of a `Git merge command` — and hoping that the greater the number of developers, the greater is the possibility of having conflicting code changes resulting from developers' tasks. Besides that, our 50 projects sample includes projects, such as cassandra, Junit and Voldemort, that were analysed in previous merging studies (BRUN et al., 2011; KASI; SARMA, 2013). We give a detailed list of the analysed projects in Appendix B.

Although we have not systematically targeted representativeness or even diversity (NAGAPPAN; ZIMMERMANN; BIRD, 2013), we believe that our sample projects have a considerable degree of diversity with respect to the already mentioned number of developers, but also source code size and domain. Our sample contains projects from different domains such as databases, search engines, and games. They also have varying sizes and number of developers. For example, retrofit, an HTTP client for Android and Java, has only 12 KLOC, while OG-Pratform, a solution for financial analytic, has approximately 2,035 KLOC. Moreover, mct has 13 collaborators, while dropwizard has 141.

After selecting the sample projects, we use the `GitMiner` tool (GITMINER, 2015) to mine GitHub repositories and collect merge scenarios from the entire histories of the selected projects. The tool converts a project version history into a graph database. We then implemented scripts that query the database to retrieve a list of all merge commit IDs and their parents IDs. Afterwards, we clone each project locally and, for each merge commit, we use the JGit API (JGIT, 2015) to checkout and copy the revisions involved in the merge scenario: the common/ancestor revision, and the two parents revisions of the merge commit (here we call *base*, *left*, and *right* revisions, respectively). In the replication study presented on Chapter 3, we filter the merge scenarios and files by those that had already shown textual conflicts on Git. In this study, we select all merge scenarios and Java files of these scenarios. This is necessary because, by ignoring a non-conflicting file or merge scenario with Git merge tool (which is unstructured by default), we might miss files and scenarios that could have conflicts with semistructured merge.

[1] `https://github.com/search/advanced`

As a result, we obtained 34030 merge scenarios from the 50 selected Java projects. Given that part of the execution and analysis steps are language dependent, we process only the Java files in these scenarios. This way, we measure only integration effort and correctness for the Java content, not for the scenario as a whole, which corresponds to, on average, 84.67% in relation to all projects' content. We also discard Java files containing elements not supported by the Java grammar used by the current version of `FSTMerge`, the semistructured merge tool we use in our study. For example, there is an incompatibility with Java annotations leading to semistructured parsing errors. We observed that these files correspond, on average, to only 0.16% in relation to the total number of Java files of our sample projects.

4.1.1.2 Execution and Analysis Steps

After collecting the sample projects and merge scenarios in the previous step, we use both unstructured and semistructured approaches to merge the selected scenarios and to find the false positives and false negatives described in Chapter 2.

In this step, we use the `FSTMerge` tool as our semistructured merge tool (APEL et al., 2011). Besides, for simplicity, we use the `Kdiff3` tool, which is one of the many standalone unstructured merge tools available (KDIFF3, 2015) (instead of using the virtual machine of the replication study). These tools take as input the three revisions (base, left and right) that compose a merge scenario 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, which would not be possible with a textual tool. Finally, when necessary, to confirm the occurrence of the false positives and false negatives, we use the features from the Eclipse JDT API, the base framework for many tools of the Eclipse IDE (JDT, 2015).

Additionally, in the study of APEL; LESSENICH; LENGAUER (2012), `FSTMerge`'s performance has shown to be an issue when merging systems with a large number of files. We also observed this issue in the replication study presented on Chapter 3. Since we use the tool in a large number of merge scenarios, we had to handle this issue, otherwise the study could become impracticable in terms of time. Originally, the tool creates a single representation (tree) in memory for all content (files) of the merge scenario. Therefore, the more files and more complex these files are, the more complex are the trees representing the merge scenario. As a consequence, such situations can lead to memory overflow and slower trees' merges. So, to handle this issue, we call the merge tools only in files that differ in the three revisions that compose a merge scenario, which is actually the case that matters in terms of software merging. This insight came from the observation that, in software merging, many of the files are not modified or are modified by only one of the developers (ALEXANDRU; GALL, 2015). In such cases the merge is trivial. So, when files are equals, or only one of them differ, we simply move the corresponding file to the resulting merged folder. This is a simple but useful optimization that could be applied to any merge tool.

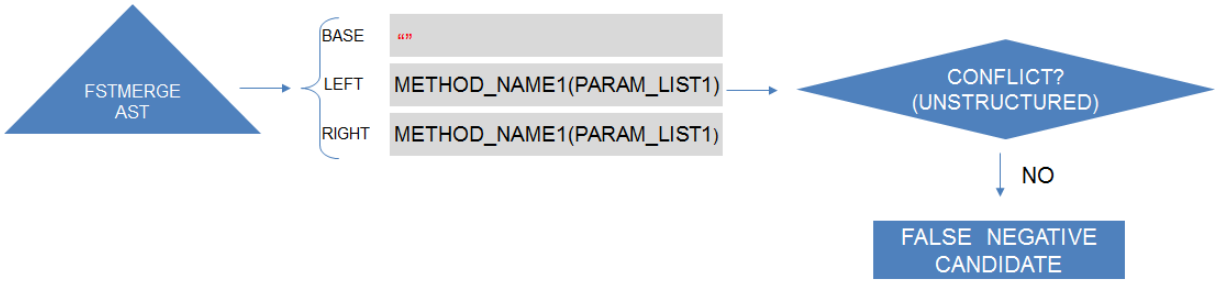
In the following sections we explain how we collect each metric.

4.1.1.2.1 False Negatives Added by Unstructured Merge — `FNa(UN)`

The unstructured merge false negatives that semistructured merge can detect occur when, in the changes introduced by the developers, there are elements, such as fields and methods, with the same name but in different areas of the program text, leading to the duplicated declaration error. This is not the case for semistructured merge because it is able to match the nodes representing the elements regardless the area in the program text.

To identify such situations, by intercepting `FSTMerge`'s execution, we look for triple of matched elements in which the base version of the element is empty and the left and right are not (see Figure 4.2). Such pattern indicates that there are two elements with the same name not present in the base revision of the code. Then, for each identified triple, we check if unstructured merge reports a conflict a containing the identifier of the possibly duplicated element (for example, a method signature). When we cannot find a conflict with such characteristic, unstructured merge was able to integrate the two versions of the element, or there might be a duplication. We check that by using Eclipse AST's compilation features to compile the file containing the observed duplication candidate merged by unstructured merge. Finally, we search for compilation problems corresponding to duplicated declaration error related to the observed candidate. In case of finding such compilation problem, we consider an occurrence of false negative.

Figure 4.2: Intercepting `FSTMerge` tool to find false negatives (FNa(UN)) *duplicated declaration error* candidates.



Source: the authors.

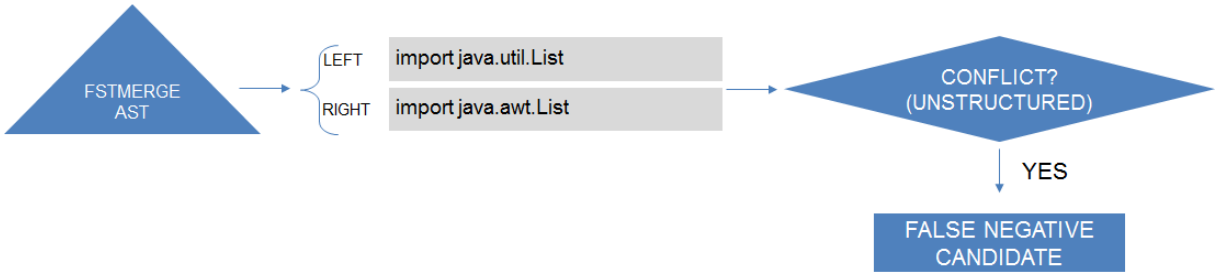
4.1.1.2.2 Maximum Number of False Negatives Added by Semistructured Merge — `FNa(SS)`

Semistructured merge reduces some of the false negatives of unstructured merge due to the detection of duplicated declarations, but it adds some too. In particular, semistructured merge is able to integrate elements introduced or modified by different developers in the same area of the text. As a consequence, the semistructured approach can lead to type ambiguity error when these elements are import statements, and they involve Java types with the same name. Similarly,

semistructured merge can lead to behavioral errors when an introduced element is referencing a modified one, and they are in the same area of the text. In what follows we further detail how identify these false negatives, and the reasons why this metric is an upper bound.

Beginning with the false negatives related to the import statements, they can occur in three situations, which we describe bellow, explaining together how we identify them. As illustrated in Figure 4.3, by intercepting `FSTMerge` execution, we identify pairs of nodes representing import statements introduced by different developers in the files being merged by semistructured merge. Afterwards, we check if the identified pairs are (1) involving members with the same name, as in `import java.util.List` and `import java.awt.List`. We also look if the identified pairs are (2) pairs of import statements of entire packages as in `import java.util.*` and `import java.awt.*` (as long as the packages have a member in common). Finally, whereas these previous cases lead to build problems, (3) we compute a case that can lead to behavioral errors instead (which is even worse). Such cases happen 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. For instance, one writes `import java.awt.*` and the other writes `import java.util.List`, where the package `java.awt.*` also has a member named `List`. These pairs of import statements can only be a false negative added by semistructured merge if they are added in the same area of the text, otherwise it would be a false negative of the two approaches. This way, we check in the corresponding file merged by unstructured merge if there is a conflict involving the pair of imports statements.

Figure 4.3: Intercepting `FSTMerge` tool to find false negatives (FNa(SS)) *type ambiguity error* candidates.



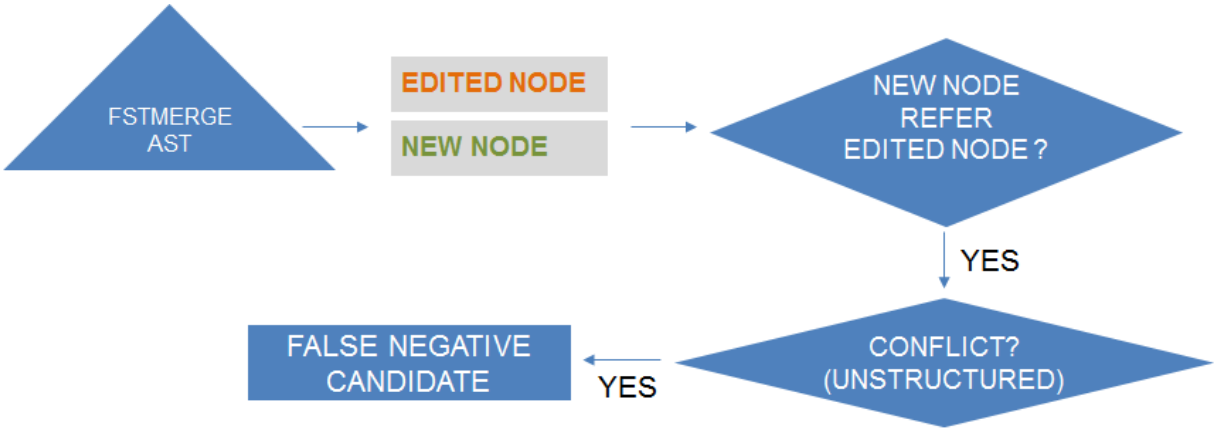
Source: the authors.

The first and second cases described in the previous paragraph only lead to type ambiguity errors when, in the presence of those pairs, the rest of the code refers to the imported elements using its abbreviated name as in `List list = List()`. We check that by using Eclipse AST's compilation features. We compile the files merged by semistructured merge and that have the a pair of import statements as just described, and we search and account the compilation problems corresponding to type ambiguity error. Regarding the third case, we approximate the verification by checking 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.

The other false negative added by semistructured merge happens when one developer adds a new element that references an existing one, and this existing element is edited by the other developer. In such cases, the developer who added the new element might not be expecting for the changes made to the element referenced by his element, possibly leading to behavioral errors. As these changes correspond to different elements (technically, different nodes in the generated AST of semistructured merge), the semistructured approach does not report conflict. However, it is possible that these changes have occurred in the same area of the program text, leading to conflict by unstructured merge.

By intercepting `FSTMerge` execution (see Figure 4.4), we identify all added and edited elements of the merge scenario. So, for each pair of added and edited elements, we check whether the added element references the edited one, looking for the identifier (the name of a field declaration, for instance) of the edited element in the added one. In case there is a reference, we verify if using unstructured merge to integrate this scenario reports a conflict involving both the added and the edited element. If there is a conflict, we consider an occurrence of false negative added by semistructured merge.

Figure 4.4: Intercepting `FSTMerge` tool to find false negatives (FNa(SS)) *new element referencing edited one* candidates.



Source: the authors.

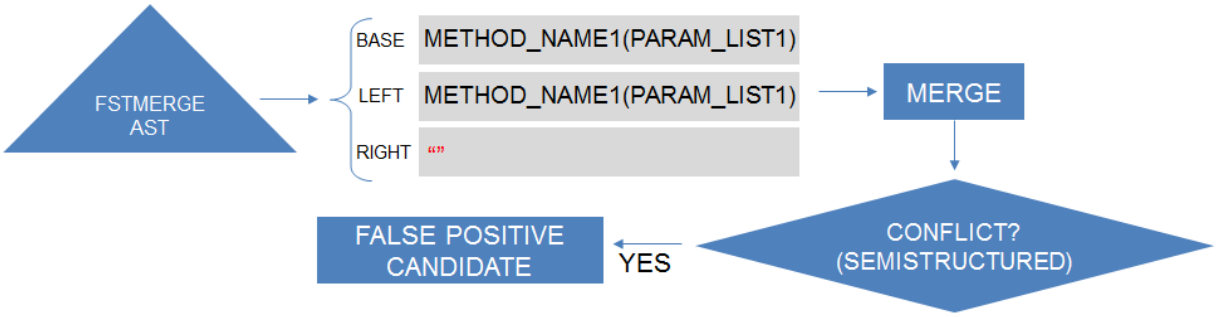
The two kinds of false negative added by semistructured merge are related to programs' semantic, and we do not verify semantics, either because this is, in general, not computable (WILHELM; WACHTER, 2008), or because its approximations, such as tests, might also be imprecise and could reduce the analysed sample. Thus, we are actually computing an upper bound of the false negatives added by semistructured merge metric. Besides, in both the new element referencing edited one false negative, and the third case of the import statements issue, we search for the identifier of the elements of interest textually, using the identifier of the elements as keyword. The problem is that the identifier might be ambiguous, that is, another element with the same identifier might exist, for instance. This way, we overestimate even more the occurrence of these false negatives.

4.1.1.2.3 Maximum Number of False Positives Added by Semistructured Merge — `FPa(SS)`

The semistructured merge added false positives in comparison to unstructured merge, as explained earlier in Section 2.3.2, usually happen when one of the developers edits the body of an existing method from the base revision, and the other developer deletes the method, or modifies its signature. In the following paragraphs we explain how we identify such false positives, and why this metric is an upper bound.

As programs' elements are nodes in the generated AST of the `FSTMerge` tool, to identify these false positives, we first observe conflicts detected by `FSTMerge` of triples of matched elements representing methods in which the left or right version of the element is empty, and the base version is not. Not having the left or right version of the element (represented by the empty string in Figure 4.5) indicates that the semistructured merge algorithm could not map the method to its previous version, either because the method was renamed, or because the method was deleted.

Figure 4.5: Intercepting `FSTMerge` tool to find false positives (FPa(SS)) *renaming/deletion* conflict candidates.



Source: the authors.

However, not all cases like that are false positives. There are situations in which these cases indeed represent interferences between development tasks and therefore should be considered true positives. As described in Section 2.3.2, this could occur, for instance, when the developers leave references to the renamed/deleted method. Thus, for each observed conflicting renamed/deleted method, we check whether there still are calls that refer to the original signature of the method. When there is no longer references to the original version, we assume that conflict as an added false positive of semistructured merge in relation to unstructured merge. We check that by using the parser and generated AST from the Eclipse JDT API. We parse all files of the project at the revision of the renamed/deleted method. We then look for nodes representing method calls and we check whether these calls point to the original signature of the renamed/deleted method of the conflict.

Finally, this metric is an upper bound because we look for references to methods via syntactic analysis and, when we do not find any reference to the method of the renaming/deletion conflict, we classify that conflict as false positive. However, our analysis only searches in the

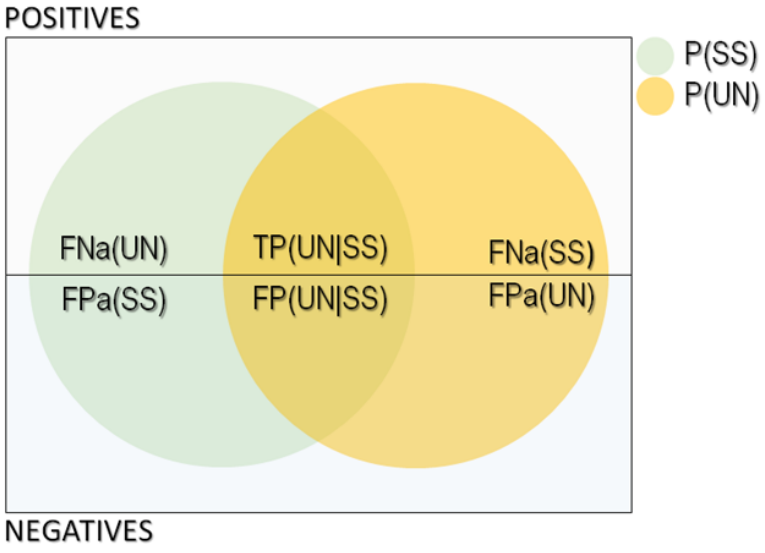
first level of references, otherwise the analysis could become computationally expensive and time-demanding, compromising the analysed sample size. So, although we believe that this is uncommon, it is possible that the referrer of the method of the renaming/deletion conflict is not being referenced too, and so on. Besides that, the version with the new signature can have its body edited too, which might interfere with the changes from the developer who kept the old signature. Finally, others renaming/deletion conflicts representing interferences, aside from these cases we identified, might exist. Therefore, we actually compute the *maximum* number of false positives added by semistructured merge metric.

4.1.1.2.4 Minimum Number of False Positives Added by Unstructured Merge — `FPa(UN)`

The false positives added by unstructured merge in comparison to semistructured merge are due to changes in the order of commutative and associative declarations — the ordering conflicts. We explain bellow how we compute such false positives, and why this metric is a lower bound. In particular, there are no specific patterns of ordering conflicts that would allow us to identify them systematically. Thus, we compute this metric in function of the other ones.

In the diagram of Figure 4.6, we illustrate the set of conflicts emerging from development tasks reported by unstructured and semistructured merge. Observe that each merge approach has its own set of reported conflicts. For instance, unstructured merge's set includes its own false positives (FPa(UN)), and false negatives added by the semistructured approach that unstructured merge is able to detect (FNa(SS)). The sets also includes true positives and false positives common to both approaches (TP(UN|SS) and FP(UN|SS)). Looking at these sets, we can infer how to calculate the false positives added by unstructured merge. In particular, from the diagram, note that:

Figure 4.6: Set of conflicts reported by unstructured and semistructured merge



Source: the authors.

$$P(UN) = FP(UN|SS) + TP(UN|SS) + FPa(UN) + FNa(SS)$$

From which follows that,

$$FPa(UN) = P(UN) - (FP(UN|SS) + TP(UN|SS)) - FNa(SS)$$

In order to `FPa(UN)` be a lower bound, we need an upper bound of `(FP(UN|SS) + TP(UN|SS))` because it is a subtractive factor. If we look at how `P(SS)` is composed, we see that:

$$P(SS) = FP(UN|SS) + TP(UN|SS) + FPa(SS) + FNa(UN)$$

Therefore,

$$FP(UN|SS) + TP(UN|SS) = P(SS) - FPa(SS) - FNa(UN)$$

Note that,

$$FP(UN|SS) + TP(UN|SS) + FPa(SS) = P(SS) - FNa(UN)$$

Thus, an upper bound of `(FP(UN|SS) + TP(UN|SS))` happens if `FPa(SS)` is at its minimum possible value (zero). Therefore, the upper bound of `(FP(UN|SS) + TP(UN|SS)))` is given by:

$$FP(UN|SS) + TP(UN|SS) \leqslant P(SS) - FNa(UN)$$

Finally, replacing `(FP(UN|SS) + TP(UN|SS))` in the formula of `FPa(UN)` given above, the minimum number of false positives added by unstructured merge becomes:

$$FPa(UN) \geqslant P(UN) - P(SS) + FNa(UN) - FNa(SS)$$

4.2 Evaluation Results

During this study, we analysed a total of 34030 merge scenarios from 50 Java projects. We identified 19238 conflicts by using unstructured merge, and 14544 using semistructured merge, representing a reduction of, approximately, 24% in the total number of conflicts (compared to a reduction of 21% in the replication study presented on Chapter 3). In this section, we present descriptive statistics structured according to our research questions. Full results are available in Appendix B.

In this study, our assumption is that the higher the number of false positives in these conflicts and merge scenarios, the greater the integration effort. Similarly, the higher the number

of interferences not detected by the merge approaches (false negatives), the weaker the correctness guarantees of the merging process. So, we compared the unstructured and semistructured merge approaches with regard to the occurrence of the false positives and false negatives described in the previous sections. In particular, we are interested in finding out the frequency of the false positives reported by unstructured merge not reported by semistructured merge, and vice-versa. We also want to know the frequency of the false negatives undetected by unstructured merge detected by semistructured merge, and vice versa.

4.2.1 When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort?

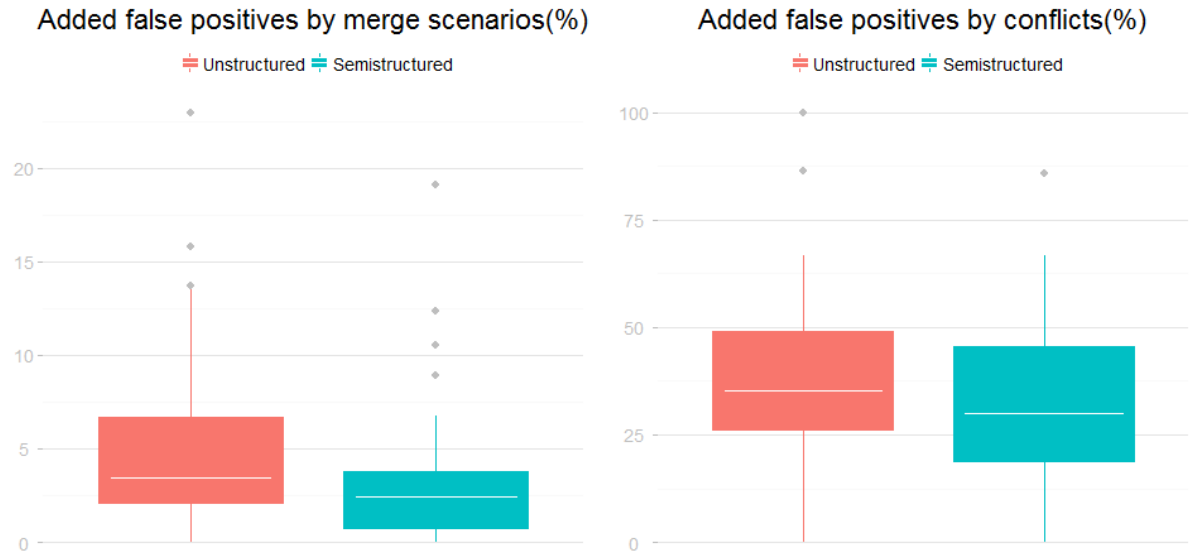
In order to answer RQ1, we need to compare the number of false positives added by each merge approach. In particular, our results show that, in our sample, when using an unstructured merge tool, on average, 5.35% of the merges scenarios have at least one of its added falses positives (FPa(UN)) (standard deviation of 4.85%). Moreover, on average, 38.11% of the reported conflicts are false positives (FPa(UN)) (standard deviation of 23.49%). This number is slightly bigger than the percentage of the reported conflicts that are false positives added by semistructured merge (FPa(SS)). We observed that, on average, 30.21% of the reported conflicts are false positives (FPa(SS)) when using semistructured merge (standard deviation of 20.68%). In addition, on average, 3.12% of the merge scenarios have at least one added false positive (FPa(SS))(standard deviation of 3.55%).

The bloxplots in Figure 4.7 (a) shows that semistructured merge indeed tends to have less merge scenarios with at least one of its added false positives (FPa(SS)). Observe that the 3rd quartile, which represents 75% of the projects of our sample, is inferior to the median of the merge scenarios that had at least one false positive added by unstructured merge (FPa(UN)). Besides that, the maximum whisker of the merge scenarios with at least one false positive (FPa(SS)) is inferior to the 3rd quartile of the merge scenarios with at least one false positive (FPa(UN)). The left side boxplot also points out Bukkit, cassandra and infinispan as outlier projects, with more than 12.5% of its merge scenarios having false positives (FPa(UN)). On the other hand, only the projects clojure and closure-compiler had no merge scenarios with false positives (FPa(UN)). Similarly, the right side boxplot shows that the outlier projects Bukkit, cassandra, lucene-solr and roboguice had more than 7.5% of the merge scenarios with false positives (FPa(SS)). Finally, only the projects closure-compiler, commafeed, commons, Essentials, jsoup and OpenRefine had no merge scenarios with false positives (FPa(SS)).

Conversely, the boxplots in Figure 4.7 (b) suggests that there is no significant difference between the number of reported conflicts accounted as added false positives by unstructured merge (FPa(UN)) and the number of reported conflicts accounted as added false positives by semistructured merge (FPa(SS)). Note that both the median, the 3rd quartile and the maximum whisker in the two bloxplots are close. In addition, the left side boxplot shows that the outlier

projects Conversations and mct had more than 85% of the reported conflicts accounted as false positives (FPa(UN)) added by unstructured merge, and in only the projects clojure and closure-compiler none of the reported conflicts are false positives (FPa(UN)). In turn, the right side boxplot shows that in the outlier project Equivalent-Exchange-3, more than 75% of the reported conflicts are false positives (FPa(SS)) added by semistructured merge, and projects such as OpenRefine and Jsoup had no false positives (FPa(SS)).

Figure 4.7: Boxplots describing the percentage per project of added false positives in terms of merge scenarios and conflicts.



(a) **(b)**

Source: the authors.

Given that our data are paired, come from the same sample, and deviates from normality, in other to check difference in the computed metrics, we conducted a Wilcoxon Signed-Rank Test (WILCOXON; WILCOX, 1964). Results of that analysis indicate that the frequency of merge scenarios that had false positives (FPa(SS))>0 is significantly lower than the frequency of merge scenarios that had false positives (FPa(UN))>0 (*p-value* equals to 1.536e-07, lower than 0.05). On the other hand, there is no significant difference between the percentage of reported conflicts that are accounted as false positives (FPa(SS)), and the percentage of reported conflicts that are accounted as false positives (FPa(UN)) (*p-value* equals to 0.1773, greater than 0.05).

4.2.2 When compared to unstructured merge, does semistructured merge compromise integration correctness by missing more task interferences?

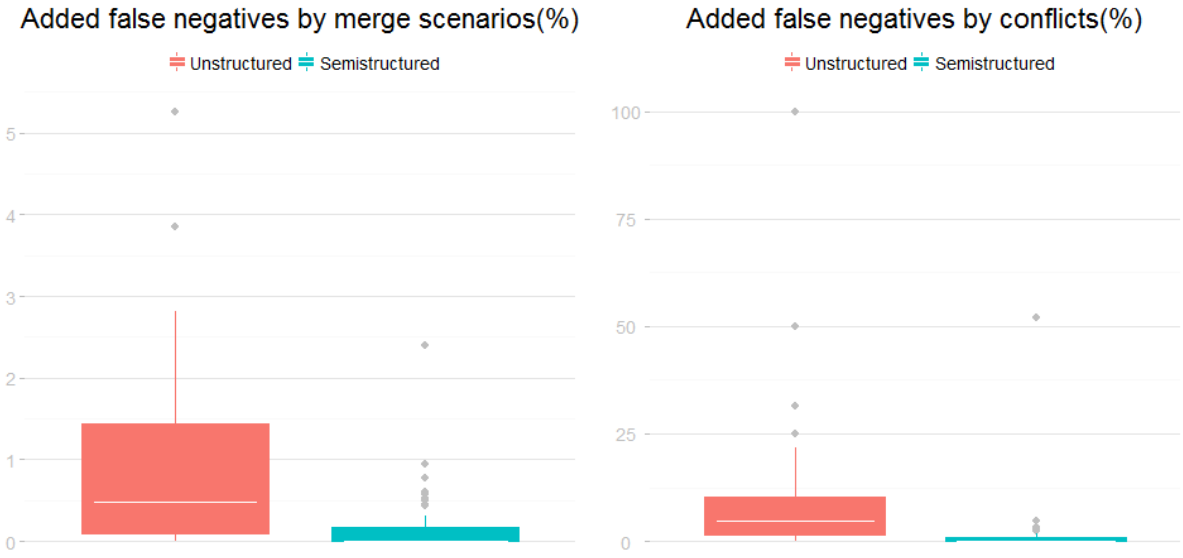
Moving on towards answering RQ2, we have to compare the number of false negatives added by each merge approach.

In terms of merge scenarios, giving the notion of how often the merging process had at least one added interference missed by one merge approach and detected by the other (false negatives), Figure 4.8 (a) shows the distribution of the percentage per project of merge scenarios having false negatives added by the merge approaches. On the left side boxplot, we observe that, in around 75% of the projects from our sample, less than 1.5% of the merge scenarios had false negatives added by unstructured merge (FNa(UN))>0, with an average of 0.88% and standard deviation of 1.07%. It also points out Bukkit and jitsi as outlier projects with more than 3.5% of its merge scenarios having false negatives (FNa(UN))>0. Similarly, the right side boxplot in Figure 4.8 (a) depicts the merge scenarios having false negatives added by semistructured merge (FNa(SS))>0. In this case, on average, 0.18% of the merge scenarios of our sample had false negatives (FNa(SS))>0, with standard deviation of 0.39%. Besides that, in around 75% of the projects from our sample, less than 0.5% of the merge scenarios had false negatives (FNa(SS))>0, and in outlier projects, such as lucene-solr and netty, more than 0.5% of the merge scenarios had false negatives (FNa(SS)).

In terms of conflicts, representing the percentage of added false negatives with respect to the total number of reported conflicts in the merged files, Figure 4.8 (b) shows the distribution of the percentage per project of the false negatives added by the merge approaches. On the left side boxplot, we observe that, in around 75% of the projects from our sample, the percentage of false negatives (FNa(UN)) was lower than 12.5%, with an average of 14.10% and standard deviation of 16.12%. In the outlier projects, such as closure-compiler and Essentials, the percentage of false negatives (FNa(UN)) was higher than 25%. Besides that, the right side boxplot in Figure 4.8 (b) depicts the distribution of the false negatives added by semistructured merge (FNa(SS)). In this case, the average percentage of false negatives (FNa(SS)) is 1.66%, with standard deviation of 7.32%. Additionally, in around 75% of the projects from our sample, the percentage of false negatives (FNa(SS)) with respect to the total number of reported conflicts was lower than 2%, and outlier projects, such as AntennaPod and cassandra, had the percentage of false negatives (FNa(SS)) higher than 6%. Finally, when distinguishing between the two kinds of false negatives (FNa(SS)) — the type ambiguity error and new element referencing edited one — we found that most of the false negatives (FNa(SS)) are related to new element referencing edited one. We discuss our thoughts about this in the next section.

Finally, a Wilcoxon Signed-Rank Test (WILCOXON; WILCOX, 1964) showed that there is significant difference between the frequency of merges scenarios that had false negatives added by unstructured merge (FNa(UN)) and falses negatives added by semistructured merge (FNa(SS)), and also between the percentage of FNa(UN) and FNa(SS) with respect to the total number of reported conflicts (*p-value* equals to, respectively, 1.056e-05 and 2.842e-06, both lower than 0.05).

Figure 4.8: Boxplots describing the percentage per project of the added false negatives in terms of merge scenarios and conflicts.



(a) **(b)**

Source: the authors.

4.3 Discussion

4.3.1 Integration Effort

As described in Section 4.1, our metrics are approximations. In particular, we computed an upper bound of the number of false positives added by semistructured merge, and a lower bound of the number of false positives added by unstructured merge. As we did not find significant statistical difference between them, our results actually mean that, on average, the maximum number of false positives added by semistructured merge is, at most, slightly lower than the minimum number of false positives added by unstructured merge in our sample. Therefore, the *real* number of false positives added by semistructured merge is also lower than the *real* number of false positives added by unstructured merge.

Although the indicative of reduction in the number of added false positives by semistructured merge suggests that this approach might reduce integration effort, a more accurate comparison would measure the effort required for analysing and discarding these false positives. This is needed because different conflicts might demand different effort to be analysed, discarded or resolved. Thus, to better understand the impact of these false positives on integration effort, we manually analysed a sample of false positives reported by both merge approaches. Actually, this analysis was very simple and not systematic, we randomly selected 20 merge scenarios from five projects, mixing outliers and average ones, and we selected one file, merged by the two approaches, from each of these scenarios. In some cases, analysing projects' commit history,

we also observed how the integrator resolved the identified false positives and if the integrator's decision prevailed in future versions of the code. It is important to note that our goal was not to find statistical evidence with this manual analysis, but to better understand the required integration effort in the perspective of software developers.

The analysis indicates false positives added by unstructured merge (the ordering conflicts) that seems easy to analyse and resolve, but also pointed out conflicts that suggest more effort. For instance, we observed conflicts caused by the introduction of simple declarations in the same text area, as illustrated in Figure 4.9 (a) in which the developers added different methods and attributes in the same area of the text. We believe that this kind of conflict can often be resolved with little effort because the integrator simple has to choose one of the declarations, or decide to keep all of them. We also identified another type of ordering conflict, which we call *crosscuting conflict*. This is a conflict that does not respect the boundaries of the syntactic structure, that is, a conflict mixing parts of different elements (the body of two different methods for instance). In Figure 4.9 (b) we illustrate two crosscuting conflicts observed in a merge scenario of project cassandra. Observe that parts of the two *getColumn* and the *validateMemtableSetting* methods are in conflict. Such conflicts are more difficult to understand and resolve because the conflict mixes parts of different syntactic structures. It is necessary to map these snippets of code to the corresponding syntactic structures, for instance, to which method the `for` and `if` statements belong. We believe that conflicts like this might demand the integrator to ask for help from the developers responsible for the changes, possibly demanding the effort of more than one developer. As such conflicts corresponds to different syntactic elements (technically different nodes in a parse tree), semistructured merge can resolve them automatically.

With respect to the semistructured merge's added false positives, which are a fraction of the renaming/deletion conflicts, as illustrated in Figure 4.10 (a), we found situations in which the integrator opted to resolve the conflict by choosing the changes made by only one of the developers. That is, he accepted the renaming/deletion done by one of the developers, or opted for the declaration with the original signature and edited body changed by the other developer. We also noticed that the integrator's decision prevailed in future versions of the code, which might indicate that this was an appropriate decision. This also suggests that the discarded changes might not be relevant, or that they are not critical for the developer responsible for them. Finally, as just one of the changes was kept, we believe that there was little integration effort to resolve such conflicts because there is no conciliation of the different changes, for instance. We also found false positives renaming/deletion conflicts in which a developer renamed/deleted the method, while the other developer changed the code indentation in the version with the original signature, without changing the body's content (Figure 4.10 (b)). In such cases, the solution of the conflict is trivial because two versions of the method are similar apart of the difference in the indentation. Indeed, in our small sample analysed manually, we did not observe any added false positive of semistructured merge that seems to be harder to resolve than a crosscuting conflict from unstructured merge. Finally, these observed conflicts not only can be easily resolved, but

Figure 4.9: Observed ordering conflicts.

```
<<<<<<<
 private final Multimap<ModelPath, ImmutableList<ModelPath>> usedMutators =
ArrayListMultimap.create();
=======
 private final Multimap<ModelPath, ModelMutation<?>> finalizers =
ArrayListMultimap.create();
>>>>>>>>
```

```
<<<<<<<
   public Map<String, String> pubsubNumSub(String... channels) {
    checkIsInMulti();
    client.pubsubNumSub(channels);
    return BuilderFactory.STRING_MAP.build(client.getBinaryMultiBulkReply());
   }
=======
    public String asking() {
        checkIsInMulti();
        client.asking();
        return client.getStatusCodeReply();
    }
>>>>>>>>
```

Developer A
Developer B

(a) Simple ordering conflicts

```
...
<<<<<<<
   public static void validateMemtableSettings(org.apache.cassandra.db.migration.avro.CfDef cf_def) throws
ConfigurationException
=======
   public ColumnDefinition getColumnDefinition(ByteBuffer name)
   {
      return column_metadata.get(name);
   }
   public ColumnDefinition getColumnDefinitionForIndex(String indexName)
>>>>>>>
<<<<<<<
      if (cf_def.memtable_flush_after_mins != null)
         DatabaseDescriptor.validateMemtableFlushPeriod(cf_def.memtable_flush_after_mins);
      if (cf_def.memtable_throughput_in_mb != null)
         DatabaseDescriptor.validateMemtableThroughput(cf_def.memtable_throughput_in_mb);
      if (cf_def.memtable_operations_in_millions != null)
         DatabaseDescriptor.validateMemtableOperations(cf_def.memtable_operations_in_millions);
   }
   public ColumnDefinition getColumnDefinition(ByteBuffer name)
   {
      return column_metadata.get(name);
   }
   public ColumnDefinition getColumnDefinitionForIndex(String indexName)
   {
      for (ColumnDefinition def : column_metadata.values())
=======
      for (ColumnDefinition def : column_metadata.values())
>>>>>>>
...
```

Developer A
Developer B

(b) Crosscuting conflicts
Source: the authors.

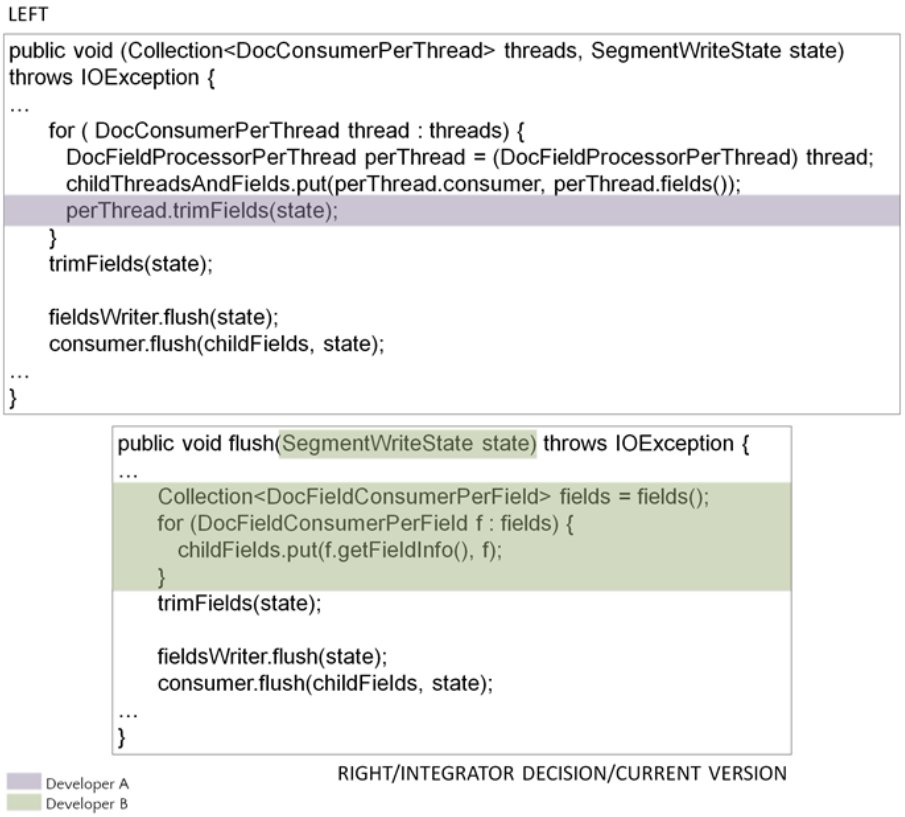
also can be resolved automatically.

To automatically resolve the false positives described in the previous paragraph, a semistructured merge tool could be extended with a mechanism that looks for references to the renamed/deleted element. When we did not find references to the renamed element, as described in Section 4.1.1.2, we classify the conflict as false positive. Alternatively, a safe, but "lazy", solution would be not to report the conflict and maintain both versions of the method. This last solution would lead in the worst case to *dead code*. In relation to the false positives due to code indentation, when the semistructured merge tool matches the nodes of the trees, it could compare the strings representing the body's content ignoring the spacings. Conversely, an improved unstructured merge tool would only be able to resolve its added false positives by behaving like semistructured merge.

On the other hand, in a few renaming conflicts of projects such as Bukkit and cassandra, we observed that the integrator had to conciliate the changes made by the developers suggesting significant resolution effort. This is illustrated in Figure 4.11, where the integrator merged the changes in the body of the method with the original signature with the body of the renamed version (which also had its body edited). Besides, that decision prevailed in future versions of the code. However, as both changes had to be conciliated to cope with the needs of the developers, it actually suggests that there was an interference between the developers' tasks, and the renaming conflict was a true positive instead. Thus, our numbers of false positives added by semistructured merge are more overestimated than what we expected in Section 4.1.1.2 because they include others kinds of conflicts representing interference between developers' tasks.

In our sample, semistructured merge reduced the overall number of reported conflicts, and there is also an indicative that it added less false positives when compared to unstructured merge. Furthermore, we did not observe false positives added by semistructured merge that seems to require more effort to be resolved than the unstructured merge ones. Finally, we believe that semistructured merge could be extended to eliminate the its observed false positives without much difficulty.

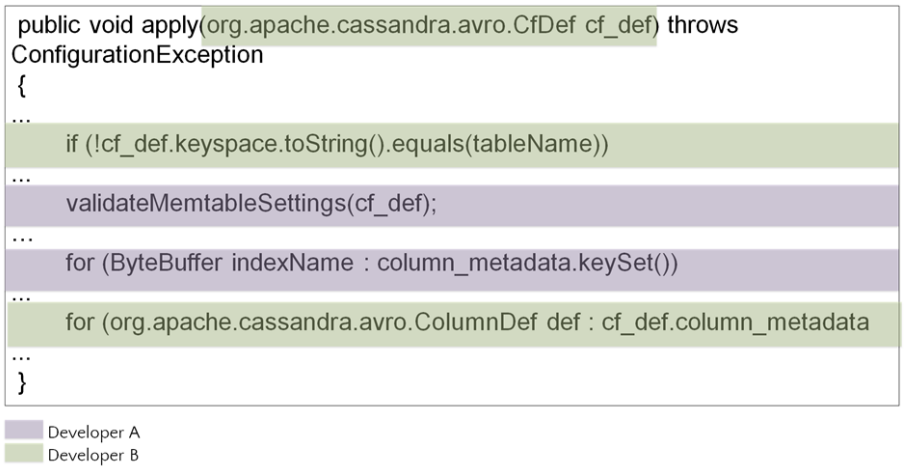
Figure 4.10: Observed renaming conflicts.



(a) Renaming conflict resolved by discarding one's changes.

(b) Indentation renaming conflict
Source: the authors.

Figure 4.11: Renaming conflict resolved by conciliating different changes.



Source: the authors.

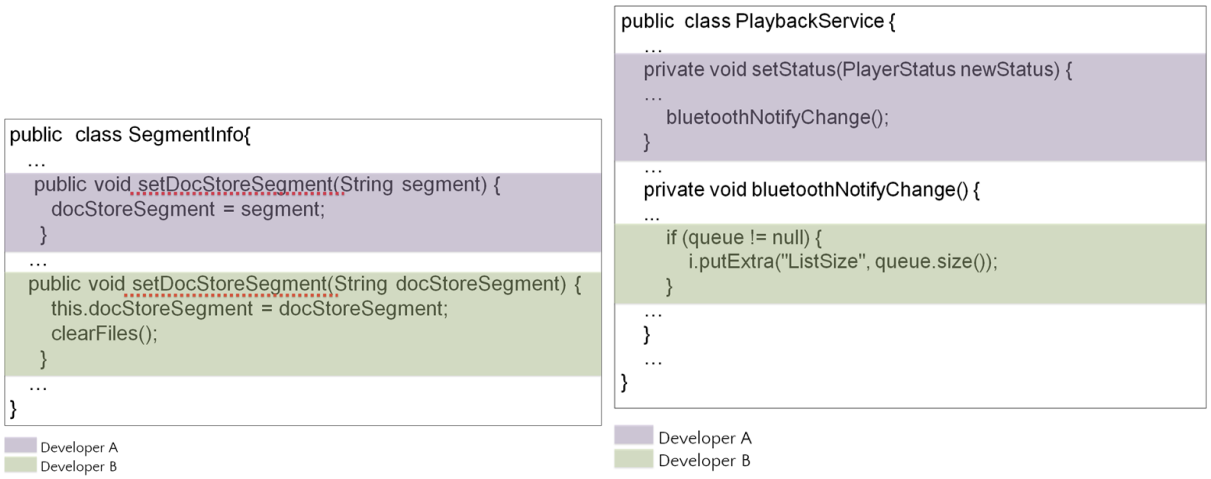
4.3.2 Correctness

When comparing false negatives not detected by unstructured merge but detected by semistructured merge, and vice versa, the reasoning is straightforward because the greater the number of false negatives, the greater the number of build or behavioral errors, and therefore the weaker the correctness guarantees of the merging process. As explained in Section 4.1, the computed number of false positives added by semistructured merge metric is an approximation (upper bound), whereas the unstructured one is precise. So, the achieved results actually mean that the exact number of false negatives added by unstructured merge is almost 6 times bigger than the maximum number of false negatives added by semistructured merge.

These results show that semistructured merge, as is currently implemented in the `FSTMerge` tool used in this study, adds less false negatives than unstructured merge. However, the false negatives added by unstructured merge cause compilation errors (the duplicated declaration error), illustrated by the underline in Figure 4.12 (a), guiding the developers toward the location and cause of the problem. On the other hand, both kinds of added false negatives by semistructured merge include "discret" problems. When one developer adds a new element referencing an edited existing one, there is no compilation error, but possibly there is a behaviorial issue. In particular, the changes made by one developer might affect the behavior expected by the other. This situation is illustrated in Figure 4.12 (b), 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). The same happens in the third case (package vs. member) of the import statements issue (see Section 4.1.1.2.2). We believe that in such cases the detection and resolution of the issue is likely more difficult and demands substantially more effort.

Whereas an improved unstructured merge would only be able to detect its added false negatives by applying the techniques used by the semistructured approach, we believe that the

Figure 4.12: Observed false negatives.



(a) **(b)**

Source: the authors.

`FSTMerge` tool could be extended to detect its extras false negatives without much difficulty. For example, the tool could include some compiler features to search for compilation problems related to the false negatives to detect them, similar to what we did during the experiment (see Section 4.1.1.2). In the case of the false negatives that might cause behavioral errors, the tool could also look at the developers contributions as a whole to try to infer interference between them. In particular, in the third case of the import statements issue, the tool could verify if the changes introduced by the developer who imported the package contain the name of the member imported by the second developer. Besides that, in the cases of new artefact referencing edited one, the tool could check whether the added element references the edited one, looking for the identifier of the edited element in the added one. Given that such analyses are not precise, they could also add false positives, but this addition would not be significant as suggested by the achieved results.

Finally, we got surprised with the small number of false negatives added by semistructured merge related to the import statements in our sample. However, the strategies we use to identify these situations in the setup described in Section 4.1.1.2.3 can give clues in what happened. One possibility is that there was no conflict by unstructured merge related to the imports, that is, it was a false negative of the two merge approaches. Additionally, even in cases of having conflicts by unstructured merge, there was no compilation error related to the imports, either because the developers used the members using their full name, or because they simply did not use the imported members. Finally, in the third case of the import statements issue (package vs. member), which might cause behavioral errors, the changes introduced by the developer who imported the package might not refer the member imported by the second developer. Even so, a deeper analysis might be necessary.

Although the number of merges leading to false negatives using unstructured merge and semistructured merge are almost insignificant in our sample, unstructured merge added more false negatives than the semistructured approach. However, we believe that the semistructured merge's added false negatives are harder to detect and resolve, but a semistructured tool could be fixed to detect them without much difficult.

4.3.3 Unstructured or Semistructured merge?

All the criteria we used to classify an approach as better than the other — the number of reported conflicts in the replication study of Chapter 3, or the number of false positives and false negatives in this second study — indicate the semistructured approach as the best.

However, we wouldn't say it's an easy choice. For instance, depending on the development phase of the project, if there are too many refactorings, the developers will have to deal with too many renaming conflicts. If the code coupling is high, in which different elements tend to depend from each other, there may be a higher incidence of the semistructured merge behavioral false negatives, which are more difficult to detect and resolve.

Nevertheless, if it were an improved semistructured tool, with simple improvements such as we suggested, the semistructured approach could be chosen without regrets.

4.4 Threats to Validity

Our empirical analyses and evaluations naturally leave open a set of potential threats to validity, which we explain in this section.

4.4.1 Construct Validity

In this study, we are measuring integration effort mainly based on the number of false positives reported by the different merge approaches. However, different conflicts may require different efforts to be resolved. For instance, taking into account the conflicts mentioned in Section 4.3.1, it is not hard to believe that a single crosscuting conflict requires more effort to be resolved than several indentation renaming/deletion conflicts. Nonetheless, in addition to the quantitative comparison, we have manually analysed sample of false positive to better understand their impact on integration effort. Although this analysis was very simple, involving a small sample, it still allowed us to better understand the required integration effort.

Another threat to construct validity is our metrics because, as they are approximations, one can argue that we perhaps could not compare them properly. However, the achieved results allowed us to make the comparisons accordingly. For instance, considering the false negatives in which we compared an upper bound with a precise value, as the achieved upper bound was

lower than the precise value, is straightforward that the real value underlying the upper bound was also lesser than the precise value.

4.4.2 Internal Validity

A potential threat to the internal (and external) validity of our study is our approach to collect merge scenarios. As we analyse public Git repositories, we might be missing merge scenarios and conflicts resolved in private repositories and erased by mechanisms such as rebasing. Therefore, our sample actually corresponds to part of the conflicts that actually happened in the analysed 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 erased conflicts different from the ones we analysed.

One could also argue that we bias the results in favor of the semistructured merge approach because we discard files not supported by the semistructured implementation used in the study (in particular, due to Java annotations). We actually miss the false positives and false negatives present in these files. However, this correspond only to 0.16% of the total Java files. Therefore, we believe that such issue has insignificant impact on our results. We also discarded projects' non Java content, which corresponds to, on average, 15.33% of all projects' content. Nevertheless, in both cases, as semistructured merge uses the unstructured one internally to merge methods body, it could be extended to merge these files. As a consequence, the approaches would have similar behavior.

4.4.3 External Validity

Although the semistructured merge tool used in this study actually supports more programming languages than just Java (more specifically, the tool also supports Python and C#), we restricted our sample to Java projects because our setup demanded language dependent tool implementation and configuration. However, all the false positives and false negatives analysed here are also likely to happen in projects written in other languages. Besides, in spite of this limitation, as discussed in Section 4.1.1.1, we believe that our sample projects has a considerable degree of diversity with respect to number of developers, source code size and domain.

4.5 The Structured Merge Approach

The merge approaches are not restricted to the already discussed unstructured and semistructured merge approaches. There are also structured merge tools (WESTFECHTEL, 1991; GRASS, 1992; APIWATTANAPONG; ORSO; HARROLD, 2007; APEL; LESSENICH; LENGAUER, 2012), or even semantic merge (BERZINS, 1994; JACKSON; LADD, 1994; BINKLEY; HORWITZ; REPS, 1995). In the last case, they are promising, but still limited to

academic research and too immature to be used in real-world software projects (MENS, 2002; APEL; LESSENICH; LENGAUER, 2012).

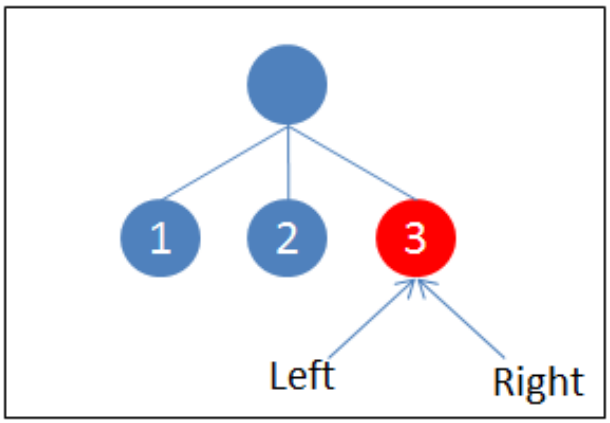
Initially, our goal was to compare integration effort and correctness across merge approaches likely to be used in practice, in particular, the unstructured, semistructured and structured approaches. As structured approach, we opted for the implementation of APEL; LESSENICH; LENGAUER (2012). That is a quite recent implementation, which is still being improved, and has been evaluated in real-world software projects. In addition, due to our experience with the semistructured merge of the same authors (APEL et al., 2011), we had support from them. However, when we tried to use the tool in our context, it did not work properly (too many crashes, for example) and we could not continue the study. In this section, we discuss our findings.

Comparing two programs for structured merge means traversing trees/graphs and finding the differing nodes, much similar to semistructured merge (APEL; LESSENICH; LENGAUER, 2012). The difference, however, is that, while semistructured merge parses the source code until the level of method declarations, representing statements and expressions on the syntax level of method body as plain text, the structured merge approach parses the entire source code.

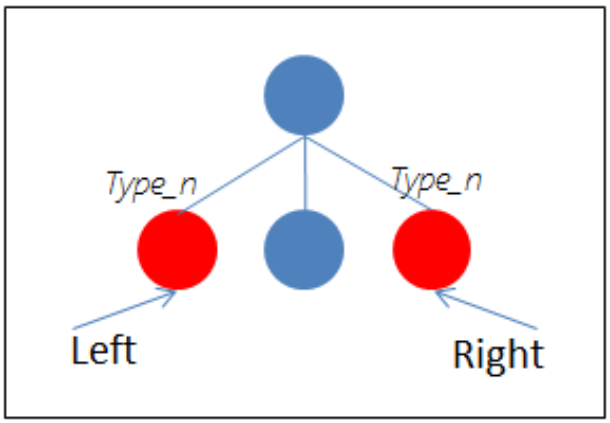
A key point for structured merge is the distinction between ordered nodes (which must not be permuted) and unordered nodes (which can be permuted safely), comparing the input trees level-wise. For ordered nodes, the positions relative to the parent node are decisive: if they overlap, the nodes are flagged as conflicting (see Figure 4.13 (a)). Whether unordered nodes are in conflict, depends on their type and name (see Figure 4.13 (b)). The matching of nodes depends on their syntactic category. For instance, two method declarations are considered equal if their signatures are equal, again similar to semistructured merge. As immediate consequence, structured merge also has the false positives and false negatives added by semistructured merge (the renaming/deletion conflicts, the type ambiguity error, and new element referencing edited one) because all of them happen due to the matching process, which is similar in the two approaches for unordered nodes. Nevertheless, it is also able to detect duplicated declarations. Besides that, since the approach is able to differentiate between ordered and unordered nodes, the approach is also able to resolve the ordering conflicts (the added false positives of unstructured merge). That, however, is where the similarities among the approaches end. To identify the false positives and false negatives added by structured merge, and those that the approach is able to resolve and detect, we must look to where the approach behaves differently. In particular, this happens inside method body because, in such cases, semistructured and unstructured merge work likewise, merging the method body content textually, whereas the structured approach does it structurally.

One of the false positives added by unstructured/semistructured merge in relation to structured merge we identified happens due to code spacing/indentation — what we call *spacing* conflicts (see Figure 4.14 (a)). (Re)formatting of code produces conflicts when textual merge is used, because the position of brackets and the indentation style (for instance, tabs or spaces)

Figure 4.13: When nodes conflict with structured merge.



(a) Ordered Nodes

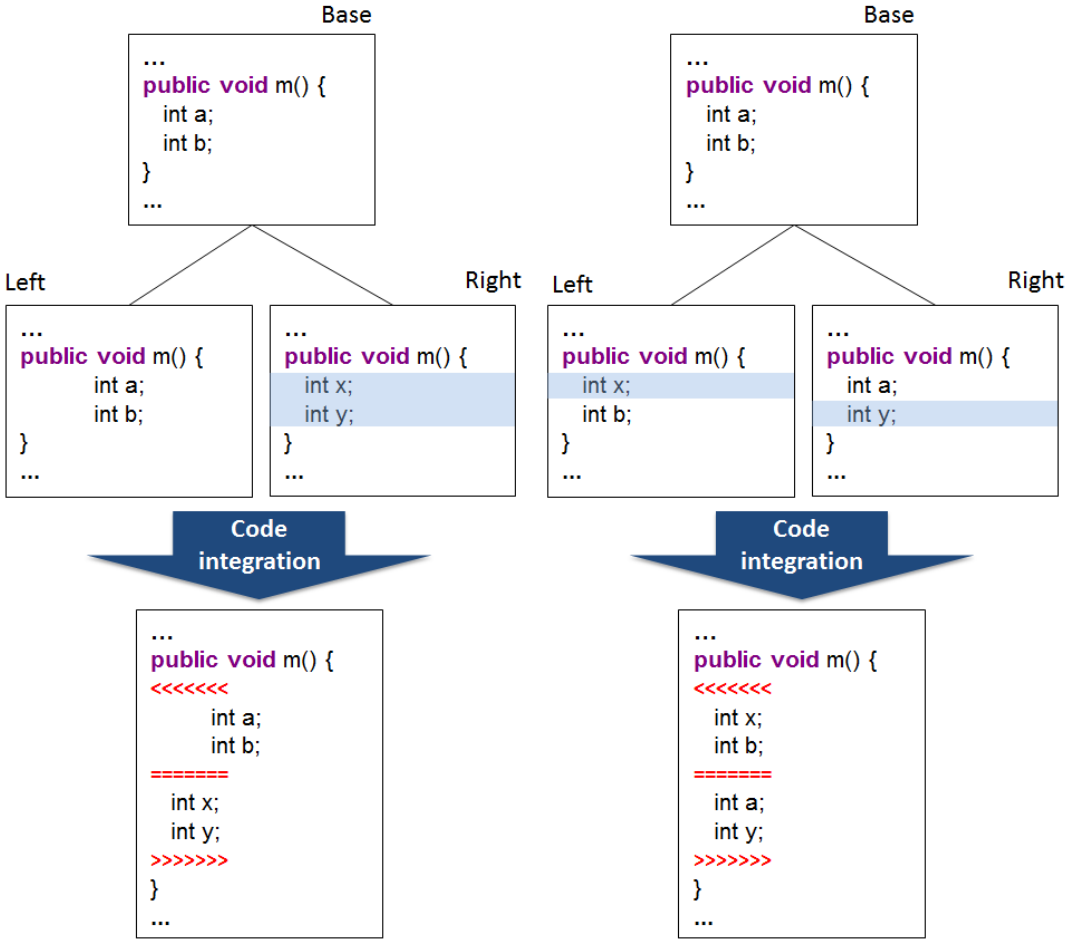


(b) Unordered Nodes
Source: the authors.

might be different according to the settings of the developer's editor. Besides, reformatting is not an uncommon practice, and, as a consequence, many unnecessary conflicts are created by textual merges. This is not the case for structured merge, in which, during its parsing process, the code formatting is lost and no longer relevant. Another kind of false positive added by unstructured/semistructured merge, as illustrated in Figure 4.14 (b), is due to changes of consecutive lines of code — the *consecutive lines* conflict. As there is no common element separating the text, the entire content is considered conflicting. On the other hand, theses conflicts are properly addressed by structured merge because the content is treated structurally. So far, we have not identified false positives added by structured merge, or others false positives added by unstructured/semistructured merge (if both exist).

Regarding the added false negatives, the problems emerges from the high degree of granularity of structured merge. Whereas a statement typically corresponds to a single line in the text, so if the developers edit the same statement, textual merge notify a conflict. With structured merge, each part of a statement corresponds to different nodes. Besides that, the position of these parts of a statement matters (they are ordered nodes). So, as different changes to the same statement corresponds to different positions, structured merge does not notify conflict

Figure 4.14: False positives added by unstructured/semistructured merge in relation to structured merge.



(a) Spacing conflict **(b)** Consecutive lines conflict

Source: the authors.

(because the positions do not overlap). However, theses changes possibly will cause behavioral errors. We call this false negative as *edits to different parts of the same statement*. In Figure 4.15 we illustrate this situation with different edits to the same `for` statement, and how structured merge interprets the situation. We have not identified others false negatives relative to the unstructured/semistructured and structured merge approaches yet.

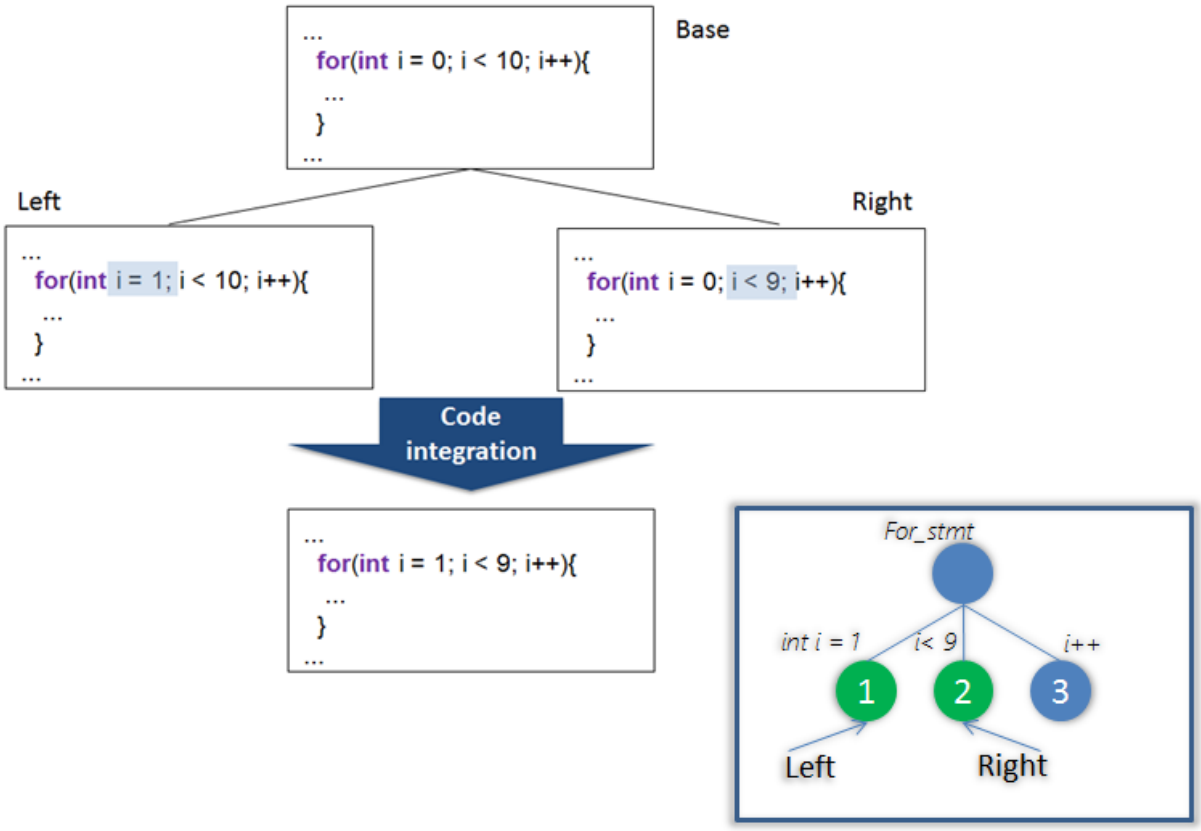
In a pilot experiment, by intercepting `FSTMerge` tool execution, we were able to assess the frequency of the spacing and consecutive lines conflicts. We did that each time nodes representing methods were being merged with semistructured merge. Regarding the consecutive lines conflicts, we identify such conflicts using string comparison to check if only consecutive lines were edited. On the other hand, by comparing left and right revisions to the base revision, ignoring tabs, spaces and line breaks. If at least one of the revisions (left and right) is equal to the base, we classify this conflict as a spacing conflict. We performed this pilot on the same sample described in Section 4.1.1.2. More specifically, 34030 merge scenarios of 50 java projects.

We found that 1.46% of the merge scenarios had at least one spacing conflict (standard

deviation of 1.63%). Besides that, 10.87% of the reported conflicts are spacing conflicts (standard deviation of 10.44%). Similarly, we found that 2.53% of the merge scenarios had at least one consecutive lines conflict (standard deviation of 2.84%), and that 15.56% of the reported conflicts are consecutive lines conflicts (standard deviation of 19.25%). (Detailed results are available on Appendix B.)

In summary, up to now, we observed that structured merge adds the same false positives and false negatives of semistructured merge. The approach is also able to resolve the ordering conflicts of unstructured merge. In addition, it is able to resolve unstructured/semistructured merge spacing and consecutive lines conflicts. However, the approach adds false negatives regarding editions to different parts of the same statement.

Figure 4.15: Edits to different parts of the same statement (Structured Merge).



Source: the authors.

5

Conclusions

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, while unstructured merge tools try to automatically resolve part of the conflicts via textual similarity, structured and semistructured merge tools try to go further by exploiting the syntactic structure of the elements involved.

To understand the impact of the unstructured and semistructured merge approaches on integration effort (Productivity) and correctness of the merging process (Quality), we conducted two empirical studies. In the first one, with a replicated experiment of APEL et al. (2011), we evaluated the semistructured approach in a sample 2.5 times bigger than the original study regarding the number of projects and 18 times bigger regarding the number of merge scenarios. In proportions far greater than the original study, we observed a significant decrease in the number of textual conflicts, conflicting lines of code and conflicting files by using semistructured merge compared to the unstructured merge. We also observed that an additional fraction of conflicts could be potentially resolved with language-specific conflict handlers. Finally, we state that if semistructured merge was used in place of the unstructured one, the occurrence of conflicting merge scenarios would drop by half.

In order to verify the frequency of false positives and false negatives arising from the use of these merge approaches, we conducted a second empirical study. By reproducing 34030 merges from 50 Java projects, we compared the relation between these merge approaches 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 interferences between development tasks. Besides that, false negatives represent build or behavioral errors, negatively impacting software quality and correctness of the merging process.

We observed that semistructured merge not only reduced the number of conflicts, but also added less false positives when compared to unstructured merge. Furthermore, we did not observe false positives added by semistructured merge that require more effort to be resolved than the unstructured merge ones. Finally, we found that semistructured merge added less false

negatives than unstructured merge, but we argue that they are harder to detect and resolve than the other way around. Additionally, our study suggested that a semistructured merge tool could be easily extended to resolve its added false positives and to detect its added false negatives.

5.1 Contributions

This work makes the main following contributions:

- Replicates an empirical study assessing semistructured merge in distributed version control systems;
- Shows how frequently merge conflicts occur during the merge process by using both semistructured and unstructured merge;
- Derives a list of false positives and false negatives from one merge approach in relation to the other;
- Evaluates how frequently the added false positives and false negatives occur during the merge process as approximations for integration effort and correctness;
- Describes how a semistructured merge tool could be improved to address the observed false positives and false negatives.

5.2 Related Work

A number of studies propose development tools and strategies to better support collaborative development environments. Those tools use different strategies to both decrease integration effort, and improve correctness during tasks integration. For instance, APEL et al. (2011) propose and evaluate the `FSTMerge` tool, a semistructured merge tool. In this work, we show evidence that `FSTMerge` indeed reduces the number of reported conflicts. We also conduct a deeper analysis regarding its false positives and false negatives. Conversely, structured and semantic merge approaches have been proposed. WESTFECHTEL (1991) and BUFFENBARGER (1995) propose tools that incorporate structural information such as the context-free and context-sensitive syntax in the merging process. Researchers also propose a wide variety of structural comparison and merge tools including tools specific to Java (APIWATTANAPONG; ORSO; HARROLD, 2007) and C++ (GRASS, 1992). APEL; LESSENICH; LENGAUER (2012) also propose a tool that tune the merging process on-line by switching between unstructured and structured merge, depending on the presence of conflicts. Some tools even consult additionally semantic information of the language (BINKLEY; HORWITZ; REPS, 1995). Finally, NIU; EASTERBROOK; SABETZADEH (2005) propose a merge approach that is both structured and independent of the language. Other approaches require that the documents to be merged

come with a formal semantics (BERZINS, 1994; JACKSON; LADD, 1994). However, while these different techniques are mainly worried about the number of reported conflicts, we focus on falses positives and false negatives added by different merge approaches. Moreover, other studies offer solutions to detect conflicts before code integration. For instance, Palantir (SARMA; REDMILES; HOEK, 2012) informs developers of ongoing parallel changes, and Crystal (BRUN et al., 2011) proactively integrates commits from developers repositories with the purpose of warning them if their changes conflict. Our study found a pattern of renaming conflict in which the integrator had to conciliate changes made in the different versions. We believe that these conflicts might benefit from such awareness tools.

Regarding the concept of integration effort, PRUDêNCIO et al. (2012) suggests hat it can be measured as the number of extra actions (additions, deletions or modifications on the artefacts) that the developer had to do during the code integration to conciliate the changes made in revisions developed concurrently. Furthermore, SANTOS and MURTA (2012) correlate the number of conflicts to that metric, suggesting that conflict reduction imply effort reduction. We opted for a qualitative analysis because this metric measures only the fraction of the time taken by the developer to edit the code, not taking into account the time that the developer took reasoning about how to resolve the conflict. This way, this editing time amounts to one part (peharps, the minor part) of the total integration effort time. On the other hand, KASI and SARMA (2013) measure integration effort based on the number of days that the conflict persisted in the project's repository. However, they assume that, during this period, the developers exclusively worked to resolve that conflict. As this information 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. In our study we have analysed conflicts that do not represent interferences between developers' tasks (false positives), and, as such, resolving them is an unnecessary integration effort. So, implementing mechanisms to automate the resolution of such conflicts might decrease the integration effort.

Finally, there are studies describing empirical evaluations that provide evidence about the frequency and impact of conflicts, and their associated causes. For example, KASI and SARMA (2013), and BRUN et al. (2011) reproduce merge scenarios from different GitHub systems with the purpose of measuring the frequency of merge scenarios that resulted in conflicts. Moreover, ZIMMERMANN (2007) does a similar analysis, but with a different metric, since the author reproduces files integration from projects on CVS. All of these work conclude that conflicts indeed occur frequently. Our work complements these studies by bringing evidence about the frequency of integrations that had conflicts when using a semistructured merge tool. Besides that, we bring evidence about the frequency of integrations that had certain types of false positives and false negatives, meaning how often integrations demands unnecessary integration effort and had interferences undetected by the unstructured and semistructured merge tools. In addition, KASI and SARMA (2013), and BRUN et al. (2011) also study the frequency of merge scenarios that had build or test failures, which can be seen as a consequence of the false negatives in the

merging process. In this respect, we have explored specific types of falses negatives that cause build or test failures.

5.3 Future Work

As discussed in previous chapters, initially our objective was to compare integration effort and correctness also with the structured merge approach. However, we did not find sufficiently mature tools for this purpose. Despite this limitation, we still were able to discuss some insights about this on Section 4.5. So, as immediate future work, we plan to conduct a complete comparative study involving the unstructured, semistructured and structured merge approaches.

In addition, during this study we explored certain kinds os false positives and false negatives. In particular, as establishing ground truth for integration conflicts, or more generally interference between development tasks, is non-trivial and would significantly reduce the analysed sample, we have actually analysed, both analytically and empirically, which conflicts reported by unstructured merge are not reported by semistructured merge, and vice versa. However, it would also be important to know false positives and false negatives in general, those common to both merge approaches. We want to identify interferences and inconsistencies generated by dependencies between the development tasks and how the current merge approaches deal with them. The focus is mainly on semantic interference and conflicts that are rarely detected and require more integration effort. By having this knowledge, it would be possible to improve an existing merge approach, or propose a new one. Additionally, experiments to measure required effort for analysing, resolving or even discard conflicts, and also exploring others language than just Java might be beneficial.

References

ALEXANDRU, C.; GALL, H. Rapid Multi-Purpose, Multi-Commit Code Analysis. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 37. **Proceedings...** [S.l.: s.n.], 2015. (ICSE '15).

APEL, S. et al. Semistructured Merge: rethinking merge in revision control systems. In: ACM SIGSOFT SYMPOSIUM AND THE 13TH EUROPEAN CONFERENCE ON FOUNDATIONS OF SOFTWARE ENGINEERING, 19. **Proceedings...** ACM, 2011. (ESEC/FSE '11).

APEL, S.; LENGAUER, C. Superimposition: a language-independent approach to software composition. In: INTERNATIONAL CONFERENCE ON SOFTWARE COMPOSITION, 7. **Proceedings...** Springer-Verlag, 2008. (SC' 08).

APEL, S.; LESSENICH, O.; LENGAUER, C. Structured Merge with Auto-tuning: balancing precision and performance. In: IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 27. **Proceedings...** ACM, 2012. (ASE 2012).

APIWATTANAPONG, T.; ORSO, A.; HARROLD, M. J. JDiff: a differencing technique and tool for object-oriented programs. **Automated Software Engineering**, [S.l.], 2007.

BERZINS, V. Software Merge: semantics of combining changes to programs. **ACM Trans. Program. Lang. Syst.**, [S.l.], 1994.

BINKLEY, D.; HORWITZ, S.; REPS, T. Program Integration for Languages with Procedure Calls. **ACM Transactions on Software Engineering and Methodology**, [S.l.], 1995.

BIRD, C. et al. The Promises and Perils of Mining Git. In: IEEE INTERNATIONAL WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, 6. **Proceedings...** IEEE Computer Society, 2009. (MSR '09).

BIRD, C.; ZIMMERMANN, T. Assessing the Value of Branches with What-if Analysis. In: ACM SIGSOFT 20TH INTERNATIONAL SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING. **Proceedings...** ACM, 2012. (FSE '12).

BRINDESCU, C. et al. How Do Centralized and Distributed Version Control Systems Impact Software Changes? In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 36. **Proceedings...** ACM, 2014. (ICSE 2014).

BRUN, Y. et al. Proactive Detection of Collaboration Conflicts. In: ACM SIGSOFT SYMPOSIUM AND THE 13TH EUROPEAN CONFERENCE ON FOUNDATIONS OF SOFTWARE ENGINEERING, 19. **Proceedings...** 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.

CAVALCANTI, G.; ACCIOLY, P.; BORBA, P. Assessing Semistructured Merge in Version Control Systems: a replicated experiment. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, 9. **Proceedings...** ACM, 2015. (ESEM '15).

CONRADI, R.; WESTFECHTEL, B. Version models for software configuration management. **ACM Computing Surveys (CSUR)**, [S.l.], v.30, n.2, p.232–282, 1998.

CVS. http://www.nongnu.org/cvs/.

DIFFUTILS. http://www.gnu.org/software/diffutils/.

ESTUBLIER, J. et al. Impact of the research community on the field of software configuration management: summary of an impact project report. **ACM SIGSOFT Software Engineering Notes**, [S.l.], v.27, n.5, p.31–39, 2002.

GIT. https://git-scm.com/.

GITHUB. https://github.com/.

GITMINER. URL: https://github.com/pridkett/gitminer.

GOGUEN, J. A.; MESEGUER, J. Security policies and security models. In: IEEE SYMPOSIUM ON SECURITY AND PRIVACY. **Anais...** [S.l.: s.n.], 1982.

GOUSIOS, G.; PINZGER, M.; DEURSEN, A. v. An Exploratory Study of the Pull-based Software Development Model. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 36. **Proceedings...** ACM, 2014. (ICSE 2014).

GRASS, J. E. Cdiff: a syntax directed differencer for c++ programs. In: USENIX C++ CONFERENCE. **Proceedings...** USENIX Association, 1992.

GREMLIN. URL: https://github.com/tinkerpop/gremlin/.

GRINTER, R. E. Using a Configuration Management Tool to Coordinate Software Development. In: CONFERENCE ON ORGANIZATIONAL COMPUTING SYSTEMS. **Proceedings...** ACM, 1995. (COCS '95).

GUIMARãES, M. L.; SILVA, A. R. Improving Early Detection of Software Merge Conflicts. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 34. **Proceedings...** IEEE Press, 2012. (ICSE '12).

HORWITZ, S.; PRINS, J.; REPS, T. Integrating Noninterfering Versions of Programs. **ACM Transactions on Programming Languages and Systems**, [S.l.], 1989.

HUNT, J. W.; SZYMANSKI, T. G. A Fast Algorithm for Computing Longest Common Subsequences. **Communications of the ACM**, [S.l.], 1977.

JACKSON, D.; LADD, D. A. Semantic Diff: a tool for summarizing the effects of modifications. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE. **Proceedings...** IEEE Computer Society, 1994. (ICSM '94).

JDT, E. http://www.eclipse.org/jdt/.

JGIT. URL: http://www.eclipse.org/jgit/.

KASI, B. K.; SARMA, A. Cassandra: proactive conflict minimization through optimized task scheduling. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 35. **Proceedings...** IEEE Press, 2013. (ICSE '13).

KDIFF3. http://kdiff3.sourceforge.net/.

KHANNA, S.; KUNAL, K.; PIERCE, B. C. A Formal Investigation of Diff3. In: INTERNATIONAL CONFERENCE ON FOUNDATIONS OF SOFTWARE TECHNOLOGY AND THEORETICAL COMPUTER SCIENCE, 27. **Proceedings...** Springer-Verlag, 2007. (FSTTCS' 07).

MENS, T. A State-of-the-Art Survey on Software Merging. **IEEE Transactions on Software Engineering**, [S.l.], 2002.

MERCURIAL. https://www.mercurial-scm.org/.

NAGAPPAN, M.; ZIMMERMANN, T.; BIRD, C. Diversity in Software Engineering Research. In: JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, 9. **Proceedings...** ACM, 2013. (ESEC/FSE '13).

NEO4J. URL: http://www.neo4j.org/.

NIU, N.; EASTERBROOK, S.; SABETZADEH, M. A category-theoretic approach to syntactic software merging. In: SOFTWARE MAINTENANCE, 2005. ICSM'05. PROCEEDINGS OF THE 21ST IEEE INTERNATIONAL CONFERENCE ON. **Anais...** IEEE, 2005. (ICSM '05).

O'SULLIVAN, B. Making Sense of Revision-control Systems. **Communications of the ACM**, [S.l.], 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**, [S.l.], 2001.

PRUDêNCIO, J. a. G. et al. To Lock, or Not to Lock: that is the question. **Journal of Systems and Software**, [S.l.], 2012.

RIGBY, P. et al. Collaboration and Governance with Distributed Version Control. **ACM Transactions on Software Engineering and Methodology**, [S.l.], v.5, 1996.

RODRIGUEZ, M. A. Graph databases: trends in the web of data. **KRDB Trends in the Web of Data School-Brixen/Bressanone**, [S.l.], 2010.

SANTOS, R. d. S.; MURTA, L. G. P. Evaluating the Branch Merging Effort in Version Control Systems. In: BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, 26. **Proceedings...** 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**, [S.l.], 2012.

SILVA, F. F. et al. Towards a Difference Detection Algorithm Aware of Refactoring-Related Changes. In: INTERNATIONAL CONFERENCE ON AVAILABILITY, RELIABILITY AND SECURITY, 9. **Proceedings...** IEEE Computer Society, 2014. (ARES '14).

SOURCEFORGE. http://sourceforge.net/.

SOUZA, C. R. B. de; REDMILES, D.; DOURISH, P. "Breaking the Code", Moving Between Private and Public Work in Collaborative Software Development. In: INTERNATIONAL ACM SIGGROUP CONFERENCE ON SUPPORTING GROUP WORK, 2003. **Proceedings...** ACM, 2003. (GROUP '03).

SUBVERSION. https://subversion.apache.org/.

TICHY, W. F. RCS—a system for version control. **Software: Practice and Experience**, [S.l.], v.15, n.7, p.637–654, 1985.

TICHY, W. F. Tools for Software Configuration Management. **SCM**, [S.l.], v.30, p.1–20, 1988.

WESTFECHTEL, B. Structure-oriented Merging of Revisions of Software Documents. In: INTERNATIONAL WORKSHOP ON SOFTWARE CONFIGURATION MANAGEMENT, 3. **Proceedings...** ACM, 1991. (SCM '91).

WILCOXON, F.; WILCOX, R. A. **Some rapid approximate statistical procedures**. [S.l.]: Lederle Laboratories, 1964.

WILHELM, R.; WACHTER, B. Abstract interpretation with applications to timing validation. In: COMPUTER AIDED VERIFICATION. **Anais...** [S.l.: s.n.], 2008.

ZIMMERMANN, T. Mining Workspace Updates in CVS. In: FOURTH INTERNATIONAL WORKSHOP ON MINING SOFTWARE REPOSITORIES. **Proceedings...** IEEE Computer Society, 2007. (MSR '07).

Appendix

A

Replication Study Appendix

In this appendix we provide replication study of Chapter 3 sample projects characteristics and evaluation results. We also provide online links to source code, repository, and data mentioned in this work.

- Mining scripts: `http://tinyurl.com/h33rczl`
- Sample dataset: `http://tinyurl.com/horttx7`
- Semistructured merge tool and evaluation scripts: `http://tinyurl.com/hxedzjk`

Table A.1: Replication sample projects list and characteristics.

Name	KLOC	Collaborators	Commits	Merge Commits	Conflicting Merge Commits	Analysed Conflicting Merge Commits	Language
astropy	244	95	10532	1791	18	5	Python
atmosphere	84	81	5091	244	10	9	Java
Bukkit	67	106	1509	19	4	4	Java
cassandra	173	63	15427	4820	896	700	Java
clojure	95	101	2667	37	2	2	Java
cloudify	168	25	11048	213	14	14	Java
CruiseControl.NET	108	44	3746	197	3	1	C#
cxf	1.206	25	9824	71	1	1	Java
django	12	717	19232	557	15	7	Python
dropwizard	55	141	2502	256	8	7	Java
Dynamo	247	37	14362	3582	342	101	C#
edx-platform	67	181	27051	7448	212	120	Python
EventStore	269	35	3795	781	52	29	C#
flask	28	236	2082	288	13	5	Python
gradle	855	132	22418	555	77	77	Java
graylog2-server	142	35	2947	214	31	25	Java
infinispan	724	76	6792	23	5	5	Java
ipython	85	343	18366	3178	56	37	Python
jedis	34	71	962	192	40	40	Java
jsoup	33	33	754	42	5	3	Java
junit	47	89	1973	351	39	34	Java
kotlin	1.171	88	18762	504	39	36	Java
lucene-solr	1.751	23	13422	211	74	72	Java
matplotlib	180	275	13239	2041	135	62	Python
mct	217	13	975	199	13	13	Java
mockito	68	37	2363	38	4	4	Java
monodevelop	28	159	32583	1204	60	29	C#
Nancy	294	190	3979	670	27	15	C#
netty	295	132	6348	175	10	10	Java
nova	157	552	31320	13166	664	488	Python
NRefactory	62	29	4320	387	30	13	C#
NServiceBus	10	69	7488	742	70	39	C#
nupic	174	49	3499	976	23	22	Python
OG-Platform	2.035	33	26700	5278	445	432	Java
OpenRefine	526	21	1997	70	7	7	Java
opensimulator	191	62	21516	1582	15	4	C#
orientdb	435	60	8744	818	72	65	Java
pandas	342	314	10934	2278	44	12	Python
Questor	135	27	3316	256	33	21	C#
ReactiveUI	86	61	3072	210	22	7	C#
realm-java	86	18	2924	782	79	79	Java
Rebus	3.538	23	1382	118	10	7	C#
requests	31	324	3711	771	23	6	Python
retrofit	12	59	817	280	4	4	Java
roboguice	217	16	1049	78	14	14	Java
Rock	762	15	8612	1225	123	69	C#
RxJava	105	97	3230	419	25	25	Java
scrapy	56	124	4408	362	10	6	Python
SharpDevelop	176	66	14677	721	170	129	C#
SignalR	238	59	4363	346	12	2	C#
slapos	192	18	7375	643	32	31	Python
SparkleShare	64	122	4033	274	9	9	C#
sympy	298	316	20837	2936	184	77	Python
taiga-back	34	19	1474	95	3	2	Python
testrunner	58	39	7413	32	4	4	Python
tornado	42	158	2473	302	23	10	Python
Umbraco-CMS	110	124	5870	2032	258	159	C#
voldemort	283	54	3832	457	51	49	Java
WowPacketParser	6	45	3695	138	7	3	C#
zamboni	100	142	22386	2249	7	4	Python
Total	19.308	6698	522218	69924	4678	3266	

Table A.2: Replication results on merge scenarios where semistructured merge reduced the numbers.

Project	Unstructured			Semistructured			Reduction		
	Textual conf.	**Conf. Lines**	**Conf. Files**	**Textual conf.**	**Conf. Lines**	**Conf. Files**	**Textual conf.**	**Conf. Lines**	**Conf. Files**
CruiseControl.NET	-	-	-	-	-	-	-	-	-
Dynamo	246	56287	159	66	2808	27	73.17%	95.01%	83.02%
EventStore	88	4904	70	13	481	6	85.23%	90.19%	91.43%
monodevelop	403	206905	862	334	32210	543	17.12%	84.43%	37.01%
Nancy	9	525	9	3	154	3	66.67%	70.67%	66.67%
NRefactory	35	2124	27	5	97	3	85.71%	95.43%	88.89%
NServiceBus	89	4098	86	44	915	48	50.56%	77.67%	44.19%
opensimulator	3	20	1	1	5	0	66.67%	75.00%	100.00%
Questor	50	14634	19	40	11058	16	20.00%	24.44%	15.79%
ReactiveUI	33	281	31	1	27	1	96.97%	90.39%	96.77%
Rebus	11	341	11	2	31	2	81.82%	90.91%	81.82%
Rock	235	12861	124	53	1687	21	77.45%	86.88%	83.06%
SharpDevelop	1520	316147	1891	866	48903	774	43.03%	84.53%	59.07%
SignalR	1	6	1	0	0	0	100.00%	100.00%	100.00%
SparkleShare	5	615	1	1	227	0	80.00%	63.09%	100.00%
Umbraco-CMS	393	18692	214	199	4766	97	49.36%	74.50%	54.67%
WowPacketParser	9	587	3	4	40	0	55.56%	93.19%	100.00%
astropy	25	665	3	7	133	1	72.00%	80.00%	66.67%
django	9	548	8	0	22	0	100.00%	95.99%	100.00%
edx-platform	318	7535	172	159	2493	64	50.00%	66.91%	62.79%
flask	3	6	3	0	0	0	100.00%	100.00%	100.00%
ipython	40	4162	25	11	613	3	72.50%	85.27%	88.00%
matplotlib	227	6770	114	119	1922	53	47.58%	71.61%	53.51%
nova	776	36690	421	335	6743	146	56.83%	81.62%	65.32%
nupic	31	13840	18	10	2143	5	67.74%	84.52%	72.22%
pandas	24	131	9	12	19	0	50.00%	85.50%	100.00%
requests	10	358	6	5	45	2	50.00%	87.43%	66.67%
scrapy	5	20	5	0	0	0	100.00%	100.00%	100.00%
slapos	81	5567	44	39	661	20	51.85%	88.13%	54.55%
sympy	109	38934	171	28	16104	78	74.31%	58.64%	54.39%
taiga-back	9	257	3	1	5	1	88.89%	98.05%	66.67%
testrunner	14	1465	8	10	315	5	28.57%	78.50%	37.50%
tornado	15	110	15	1	5	1	93.33%	95.45%	93.33%
zamboni	2	36	2	0	2	0	100.00%	94.44%	100.00%
atmosphere	93	3901	26	63	976	17	32.26%	74.98%	34.62%
Bukkit	9	1464	26	2	321	12	77.78%	78.07%	53.85%
cassandra	1643	69395	616	852	16459	287	48.14%	76.28%	53.41%
clojure	-	-	-	-	-	-	-	-	-
cloudify	138	4296	28	78	1287	17	43.48%	70.04%	39.29%
cxf	-	219	-	-	16	-	-	92.69%	-
dropwizard	11	472	5	1	55	1	90.91%	88.35%	80.00%
gradle	212	10583	154	72	1451	67	66.04%	86.29%	56.49%
graylog2-server	79	3040	58	28	560	27	64.56%	81.58%	53.45%
infinispan	89	1428	45	70	697	32	21.35%	51.19%	28.89%
jedis	115	5167	66	61	1571	19	46.96%	69.60%	71.21%
jsoup	5	76	2	1	9	0	80.00%	88.16%	100.00%
junit	111	4028	55	44	505	20	60.36%	87.46%	63.64%
kotlin	143	3466	55	67	749	32	53.15%	78.39%	41.82%
lucene-solr	1526	67849	669	1024	19484	431	32.90%	71.28%	35.58%
mct	21	1314	11	9	478	6	57.14%	63.62%	45.45%
mockito	76	992	31	6	72	3	92.11%	92.74%	90.32%
netty	158	11900	83	100	3780	50	36.71%	68.24%	39.76%
OG-Platform	2328	210562	1805	1304	46479	966	43.99%	77.93%	46.48%
OpenRefine	13	12189	51	11	2995	48	15.38%	75.43%	5.88%
orientdb	225	15640	157	109	3640	79	51.56%	76.73%	49.68%
realm-java	179	12069	106	115	3678	61	35.75%	69.53%	42.45%
retrofit	31	668	16	8	213	6	74.19%	68.11%	62.50%
roboguice	34	6237	49	20	1316	31	41.18%	78.90%	36.73%
RxJava	46	2822	30	6	76	4	86.96%	97.31%	86.67%
voldemort	297	30477	100	163	3418	68	45.12%	88.78%	32.00%
Average							**62.30%**	**81.04%**	**65.51%**
Median							60.36%	83.03%	63.64%
Standard Deviation							23.57%	13.52%	25.12%

Table A.3: Replication results on merge scenarios where unstructured merge reduced the numbers.

Project	Unstructured			Semistructured			Reduction		
	Textual Conf.	**Conf. Lines**	**Conf. Files**	**Textual Conf.**	**Conf. Lines**	**Conf. Files**	**Textual Conf.**	**Conf. Lines**	**Conf. Files**
CruiseControl.NET	-	-	-	-	-	-	-	-	-
Dynamo	22	295	1	71	431	2	69.01%	31.55%	50.00%
EventStore	12	124	-	16	202	-	25.00%	38.61%	-
monodevelop	493	36	-	747	98	-	34.00%	63.27%	-
Nancy	3	4	-	4	14	-	25.00%	71.43%	-
NRefactory	-	-	-	-	-	-	-	-	-
NServiceBus	22	43	-	29	77	-	24.14%	44.16%	-
opensimulator	-	43	-	-	66	-	-	34.85%	-
Questor	10	-	-	14	-	-	28.57%	-	-
ReactiveUI	-	-	-	-	-	-	-	-	-
Rebus	-	-	-	-	-	-	-	-	-
Rock	6	298	-	11	356	-	45.45%	16.29%	-
SharpDevelop	517	44	-	780	85	-	33.72%	48.24%	-
SignalR	-	-	-	-	-	-	-	-	-
SparkleShare	20	18	-	23	24	-	13.04%	25%	-
Umbraco-CMS	71	1163	-	106	1894	-	33.02%	38.60%	-
WowPacketParser	-	-	-	-	-	-	-	-	-
astropy	1	236	-	4	245	-	75.00%	3.67%	-
django	-	-	-	-	-	-	-	-	-
edx-platform	21	37	-	34	101	-	38.24%	63.37%	-
flask	-	-	-	-	-	-	-	-	-
ipython	6	18	0	26	29	1	76.92%	37.93%	100.00%
matplotlib	11	232	-	13	309	-	15.38%	24.92%	-
nova	112	1357	1	226	3290	5	50.44%	58.75%	80.00%
nupic	9	56	-	41	87	-	78.05%	35.63%	-
pandas	0	241	0	2	321	1	100.00%	24.92%	100.00%
requests	4	52	-	5	86	-	20.00%	39.53%	-
scrapy	-	-	-	-	-	-	-	-	-
slapos	3	-	-	6	-	-	50.00%	-	-
sympy	645	197	-	777	1743	-	16.99%	88.70%	-
taiga-back	-	-	-	-	-	-	-	-	-
testrunner	-	-	-	-	-	-	-	-	-
tornado	-	-	-	-	-	-	-	-	-
zamboni	-	-	-	-	-	-	-	-	-
atmosphere	-	72	-	-	299	-	-	75.92%	-
Bukkit	18	-	-	25	-	-	28%	-	-
cassandra	93	5395	3	174	13284	5	46.55%	59.39%	40.00%
clojure	-	-	-	-	-	-	-	-	-
cloudify	3	101	-	6	247	-	50.00%	59.11%	-
cxf	-	-	-	-	-	-	-	-	-
dropwizard	2	4	-	3	28	-	33.33%	85.71%	-
gradle	22	126	-	32	244	-	31.25%	48.36%	-
graylog2-server	20	290	-	33	405	-	39.39%	28.40%	-
infinispan	-	-	-	-	-	-	-	-	-
jedis	153	438	-	183	516	-	16.39%	15.12%	-
jsoup	-	-	-	-	-	-	-	-	-
junit	-	-	-	-	-	-	-	-	-
kotlin	4	362	-	6	515	-	33.33%	29.71%	-
lucene-solr	16	1030	-	32	2038	-	50.00%	49.46%	-
mct	16	15	-	17	29	-	5.88%	48.28%	-
mockito	-	-	-	-	-	-	-	-	-
netty	-	-	-	-	-	-	-	-	-
OG-Platform	915	2146	-	1675	3902	-	45.37%	45.00%	-
OpenRefine	43	199	-	92	220	-	53.26%	9.55%	-
orientdb	93	216	-	185	262	-	49.73%	17.56%	-
realm-java	75	218	-	143	465	-	47.55%	53.12%	-
retrofit	3	-	-	4	-	-	25.00%	-	-
roboguice	35	40	-	54	65	-	35.19%	38.46%	-
RxJava	1	4	-	2	21	-	50.00%	80.95%	-
voldemort	29	622	-	54	987	-	46.30%	36.98%	-
Average							**40.49%**	**43.62%**	**74%**
Median							36.71%	39.07%	80.00%
Standard Deviation							20.10%	21.13%	27.93%

B

Integration Effort and Correctnesses Study Appendix

In this appendix we provide integration effort and correctness study of Chapter 4 sample projects characteristics and evaluation results. We also provide online links to source code of the implementations mentioned in this work.

- Mining scripts: `http://tinyurl.com/zszeftr`
- Semistructured merge tool and execution scripts: `http://tinyurl.com/j8jp4dt`
- Analysis scripts: `http://tinyurl.com/gmxmyl3`

Table B.1: Java sample project list and characteristics.

Name	URL	KLOC	Collaborators	Commits	Merge Commits
Activiti	https://github.com/Activiti/Activiti	387	112	5541	786
AndEngine	https://github.com/nicolasgramlich/AndEngine	40.7	20	1800	115
andlytics	https://github.com/AndlyticsProject/andlytics	48.5	27	1388	560
AntennaPod	https://github.com/AntennaPod/AntennaPod	54.9	52	2368	519
antlr4	https://github.com/antlr/antlr4	11.4	48	4350	656
atmosphere	https://github.com/Atmosphere/atmosphere	84	81	5091	244
BroadleafCommerce	https://github.com/BroadleafCommerce/BroadleafCommerce	219	43	9292	898
Bukkit	https://github.com/Bukkit/Bukkit	67	106	1509	19
cassandra	https://github.com/apache/cassandra	173	63	15427	3360
cgeo	https://github.com/cgeo/cgeo	53	77	8532	1890
clojure	https://github.com/clojure/clojure	95	101	2667	37
closure-compiler	https://github.com/google/closure-compiler	467	156	5880	233
cloudify	https://github.com/CloudifySource/cloudify	168	25	11048	213
commafeed	https://github.com/Athou/commafeed	20.2	65	2420	241
commons	https://github.com/twitter/commons	67.5	82	1618	208
Conversations	https://github.com/siacs/Conversations	39,2	60	2738	481
cxf	https://github.com/apache/cxf	1.206	25	9824	71
deeplearning4j	https://github.com/deeplearning4j/deeplearning4j	265	57	3161	731
dropwizard	https://github.com/dropwizard/dropwizard	55	141	2502	256
Equivalent-Exchange-3	https://github.com/pahimar/Equivalent-Exchange-3	28.8	97	1955	387
Essentials	https://github.com/essentials/Essentials	68.7	69	4328	572
gradle	https://github.com/gradle/gradle	855	132	22418	554
graylog2-server	https://github.com/Graylog2/graylog2-server	142	35	2947	212
groovy-core	https://github.com/groovy/groovy-core	224.3	137	12377	678
infinispan	https://github.com/danberindei/infinispan	724	76	6792	23
jedis	https://github.com/xetorthio/jedis	34	71	962	192
jenkins	https://github.com/jenkinsci/jenkins	996	366	21170	2008
jitsi	https://github.com/jitsi/jitsi	381	37	12182	78
jsoup	https://github.com/jhy/jsoup	33	33	754	42
junit	https://github.com/junit-team/junit	47	89	1973	350
k-9	https://github.com/k9mail/k-9	103	135	5743	426
kotlin	https://github.com/JetBrains/kotlin	1.171	88	18762	499
lucene-solr	https://github.com/apache/lucene-solr	1.751	23	13422	209
mct	https://github.com/nasa/mct	217	13	975	199
mockito	https://github.com/mockito/mockito	68	37	2363	38
netty	https://github.com/netty/netty	295	132	6348	175
OG-Platform	https://github.com/OpenGamma/OG-Platform	2.035	33	26700	4522
okhttp	https://github.com/square/okhttp	43.7	91	2003	1038
OpenRefine	https://github.com/OpenRefine/OpenRefine	526	21	1997	70
OpenTripPlanner	https://github.com/opentripplanner/OpenTripPlanner	124	69	8367	683
orientdb	https://github.com/orientechnologies/orientdb	435	60	8744	817
Osmand	https://github.com/osmandapp/Osmand	646	302	22939	3864
realm-java	https://github.com/realm/realm-java	86	18	2924	782
retrofit	https://github.com/square/retrofit	12	59	817	280
roboguice	https://github.com/roboguice/roboguice	217	16	1049	73
robolectric	https://github.com/robolectric/robolectric	74.4	246	5041	1463
rstudio	https://github.com/rstudio/rstudio	494	34	13592	549
RxJava	https://github.com/ReactiveX/RxJava	105	97	3230	418
Spout	https://github.com/SpoutDev/Spout	70.5	66	5912	854
voldemort	https://github.com/voldemort/voldemort	283	54	3832	457
Total		15.058		339774	34030

Table B.2: False positives added by semistructured merge in terms of merge scenarios.

Project	Merge Scenarios	Merge Scenarios with Renaming or Deletion Conflicts	(%)
Activiti	786	10	1.27
AndEngine	115	4	3.48
andlytics	560	5	0.89
AntennaPod	519	11	2.12
antlr4	656	21	3.2
atmosphere	244	6	2.46
BroadleafCommerce	898	57	6.35
Bukkit	19	2	10.53
cassandra	3360	299	8.9
cgeo	1890	46	2.43
clojure	37	1	2.7
closure-compiler	233	0	0
cloudify	213	5	2.35
commafeed	241	0	0
commons	208	0	0
Conversations	481	1	0.21
cxf	71	2	2.82
deeplearning4j	731	13	1.78
dropwizard	256	3	1.17
Equivalent-Exchange-3	387	1	0.26
Essentials	572	0	0
gradle	554	33	5.96
graylog2-server	212	11	5.19
groovy-core	678	22	3.24
infinispan	23	1	4.35
jedis	192	13	6.77
jenkins	2008	19	0.95
jitsi	78	2	2.56
jsoup	42	0	0
junit	350	7	2
k-9	426	18	4.23
kotlin	499	16	3.21
lucene-solr	209	40	19.14
mct	199	2	1.01
mockito	38	1	2.63
netty	175	5	2.86
OG-Platform	4522	176	3.89
okhttp	1038	2	0.19
OpenRefine	70	0	0
OpenTripPlanner	683	45	6.59
orientdb	817	23	2.82
Osmand	3864	14	0.36
realm-java	782	25	3.2
retrofit	280	2	0.71
roboguice	73	9	12.33
rstudio	1463	6	0.41
rundeck	549	2	0.36
RxJava	418	6	1.44
Spout	854	14	1.64
voldemort	457	23	5.03
Total	**34030**	**1024**	**3.01**
Mean			**3.12**
Standard Deviation			**3.55**

Table B.3: False positives added by semistructured merge in terms of conflicts.

Project	Conflicts	Renaming or Deletion Conflicts	(%)
Activiti	123	56	45.53
AndEngine	37	4	10.81
andlytics	49	16	32.65
AntennaPod	152	84	55.26
antlr4	115	58	50.43
atmosphere	89	10	11.24
BroadleafCommerce	516	158	30.62
Bukkit	12	8	66.67
cassandra	4191	788	18.8
cgeo	235	88	37.45
clojure	5	1	20
closure-compiler	1	0	0
cloudify	154	31	20.13
commafeed	1	0	0
commons	0	0	0
Conversations	25	1	4
cxf	45	9	20
deeplearning4j	100	25	25
dropwizard	9	3	33.33
Equivalent-Exchange-3	78	67	85.9
Essentials	20	0	0
gradle	159	67	42.14
graylog2-server	78	37	47.44
groovy-core	193	124	64.25
infinispan	71	18	25.35
jedis	280	59	21.07
jenkins	178	35	19.66
jitsi	21	7	33.33
jsoup	1	0	0
junit	58	14	24.14
k-9	188	98	52.13
kotlin	112	28	25
lucene-solr	1171	278	23.74
mct	40	2	5
mockito	6	1	16.67
netty	111	72	64.86
OG-Platform	3572	2224	62.26
okhttp	9	3	33.33
OpenRefine	138	0	0
OpenTripPlanner	513	226	44.05
orientdb	445	134	30.11
Osmand	216	30	13.89
realm-java	303	138	45.54
retrofit	13	5	38.46
roboguice	105	56	53.33
rstudio	72	9	12.5
rundeck	16	10	62.5
RxJava	31	10	32.26
Spout	139	42	30.22
voldemort	348	67	19.25
Total	**14544**	**5201**	**35.76**
Mean			**30.21**
Standard Deviation			**20.68**

Table B.4: False positives added by unstructured merge in terms of merge scenarios.

Project	Merge Scenarios	Merge Scenarios with Ordering Conflicts	(%)
Activiti	786	22	2.8
AndEngine	115	4	3.48
andlytics	560	9	1.61
AntennaPod	519	17	3.28
antlr4	656	21	3.2
atmosphere	244	7	2.87
BroadleafCommerce	898	123	13.7
Bukkit	19	3	15.79
cassandra	3360	455	13.54
cgeo	1890	77	4.07
clojure	37	0	0
closure-compiler	233	0	0
cloudify	213	7	3.29
commafeed	241	1	0.41
commons	208	1	0.48
Conversations	481	7	1.46
cxf	71	2	2.82
deeplearning4j	731	46	6.29
dropwizard	256	5	1.95
Equivalent-Exchange-3	387	1	0.26
Essentials	572	2	0.35
gradle	554	51	9.21
graylog2-server	212	21	9.91
groovy-core	678	32	4.72
infinispan	23	3	13.04
jedis	192	23	11.98
jenkins	2008	66	3.29
jitsi	78	2	2.56
jsoup	42	4	9.52
junit	350	22	6.29
k-9	426	27	6.34
kotlin	499	26	5.21
lucene-solr	209	48	22.97
mct	199	5	2.51
mockito	38	3	7.89
netty	175	7	4
OG-Platform	4522	257	5.68
okhttp	1038	2	0.19
OpenRefine	70	2	2.86
OpenTripPlanner	683	82	12.01
orientdb	817	44	5.39
Osmand	3864	51	1.32
realm-java	782	49	6.27
retrofit	280	2	0.71
roboguice	73	9	12.33
rstudio	1463	42	2.87
rundeck	549	5	0.91
RxJava	418	23	5.5
Spout	854	29	3.4
voldemort	457	31	6.78
Total	**34030**	**1778**	**5.22**
Mean			**5.35**
Standard Deviation			**4.85**

Table B.5: False positives added by unstructured merge in terms of conflicts.

Project	Conflicts	Ordering Conflicts	(%)
Activiti	166	59	35.54
AndEngine	39	23	58.97
andlytics	82	29	35.37
AntennaPod	142	0	0
antlr4	127	45	35.43
atmosphere	118	50	42.37
BroadleafCommerce	851	299	35.14
Bukkit	12	3	25
cassandra	6796	1496	22.01
cgeo	304	136	44.74
clojure	4	4	100
closure-compiler	0	0	0
cloudify	163	69	42.33
commafeed	2	1	50
commons	1	0	0
Conversations	21	21	100
cxf	77	24	31.17
deeplearning4j	253	69	27.27
dropwizard	19	6	31.58
Equivalent-Exchange-3	68	11	16.18
Essentials	23	0	0
gradle	288	71	24.65
graylog2-server	145	30	20.69
groovy-core	215	56	26.05
infinispan	90	53	58.89
jedis	293	155	52.9
jenkins	429	121	28.21
jitsi	29	10	34.48
jsoup	7	1	14.29
junit	134	39	29.1
k-9	266	73	27.44
kotlin	171	70	40.94
lucene-solr	1677	833	49.67
mct	44	38	86.36
mockito	57	5	8.77
netty	167	31	18.56
OG-Platform	3066	1235	40.28
okhttp	6	4	66.67
OpenRefine	59	59	100
OpenTripPlanner	649	241	37.13
orientdb	509	240	47.15
Osmand	311	180	57.88
realm-java	322	126	39.13
retrofit	35	6	17.14
roboguice	100	49	49
rstudio	145	58	40
rundeck	18	5	27.78
RxJava	76	21	27.63
Spout	224	91	40.62
voldemort	438	268	61.19
Total	**19238**	**6514**	**33.86**
Mean			**38.11**
Standard Deviation			**23.49**

Table B.6: False negatives added by semistructured merge in terms of merge scenarios.

Project	Merge Scenarios	Merge Scenarios with Type Ambiguity Error	Merge Scenarios with New Artefact Referencing Edited One	(%)
Activiti	786	0	0	0
AndEngine	115	0	0	0
andlytics	560	0	0	0
AntennaPod	519	0	4	0.77
antlr4	656	0	1	0.15
atmosphere	244	0	0	0
BroadleafCommerce	898	0	4	0.45
Bukkit	19	0	0	0
cassandra	3360	2	8	0.3
cgeo	1890	0	0	0
clojure	37	0	0	0
closure-compiler	233	0	0	0
cloudify	213	0	0	0
commafeed	241	0	0	0
commons	208	0	0	0
Conversations	481	0	0	0
cxf	71	0	0	0
deeplearning4j	731	0	0	0
dropwizard	256	0	0	0
Equivalent-Exchange-3	387	0	0	0
Essentials	572	0	0	0
gradle	554	0	1	0.18
graylog2-server	212	0	2	0.94
groovy-core	678	0	1	0.15
infinispan	23	0	0	0
jedis	192	0	1	0.52
jenkins	2008	0	2	0.1
jitsi	78	0	0	0
jsoup	42	0	0	0
junit	350	0	0	0
k-9	426	0	0	0
kotlin	499	0	0	0
lucene-solr	209	0	5	2.39
mct	199	0	1	0.5
mockito	38	0	0	0
netty	175	0	1	0.57
OG-Platform	4522	1	13	0.31
okhttp	1038	0	0	0
OpenRefine	70	0	0	0
OpenTripPlanner	683	0	3	0.44
orientdb	817	0	5	0.61
Osmand	3864	0	1	0.03
realm-java	782	0	0	0
retrofit	280	0	0	0
roboguice	73	0	0	0
rstudio	1463	0	0	0
rundeck	549	0	0	0
RxJava	418	0	0	0
Spout	854	0	1	0.12
voldemort	457	0	2	0.44
Total	**34030**	**3**	**56**	**0.17**
Mean				**0.18**
Standard Deviation				**0.39**

Table B.7: False negatives added by semistructured merge in terms of conflicts.

Project	Conflicts	Type Ambiguity Errors	New Artefact Referencing Edited One	(%)
Activiti	123	0	0	0
AndEngine	37	0	0	0
andlytics	49	0	0	0
AntennaPod	152	0	166	52.2
antlr4	115	0	2	1.71
atmosphere	89	0	0	0
BroadleafCommerce	516	0	18	3.37
Bukkit	12	0	0	0
cassandra	4191	14	194	4.75
cgeo	235	0	0	0
clojure	5	0	0	0
closure-compiler	1	0	0	0
cloudify	154	0	0	0
commafeed	1	0	0	0
commons	0	0	0	0
Conversations	25	0	0	0
cxf	45	0	0	0
deeplearning4j	100	0	0	0
dropwizard	9	0	0	0
Equivalent-Exchange-3	78	0	0	0
Essentials	20	0	0	0
gradle	159	0	1	0.62
graylog2-server	78	0	2	2.5
groovy-core	193	0	1	0.52
infinispan	71	0	0	0
jedis	280	0	1	0.36
jenkins	178	0	2	1.11
jitsi	21	0	0	0
jsoup	1	0	0	0
junit	58	0	0	0
k-9	188	0	0	0
kotlin	112	0	0	0
lucene-solr	1171	0	10	0.85
mct	40	0	2	4.76
mockito	6	0	0	0
netty	111	0	1	0.89
OG-Platform	3572	1	111	3.04
okhttp	9	0	0	0
OpenRefine	138	0	0	0
OpenTripPlanner	513	0	5	0.97
orientdb	445	0	8	1.77
Osmand	216	0	1	0.46
realm-java	303	0	0	0
retrofit	13	0	0	0
roboguice	105	0	0	0
rstudio	72	0	0	0
rundeck	16	0	0	0
RxJava	31	0	0	0
Spout	139	0	1	0.71
voldemort	348	0	9	2.52
Total	**14544**	**15**	**535**	**3.78**
Mean				**1.66**
Standard Deviation				**7.32**

Table B.8: False negatives added by unstructured merge in terms of merge scenarios.

Project	Merge Scenarios	Merge Scenarios with Duplicated Declaration Errors	(%)
Activiti	786	3	0.38
AndEngine	115	3	2.61
andlytics	560	2	0.36
AntennaPod	519	1	0.19
antlr4	656	4	0.61
atmosphere	244	4	1.64
BroadleafCommerce	898	14	1.56
Bukkit	19	1	5.26
cassandra	3360	58	1.73
cgeo	1890	6	0.32
clojure	37	0	0
closure-compiler	233	1	0.43
cloudify	213	5	2.35
commafeed	241	0	0
commons	208	0	0
Conversations	481	0	0
cxf	71	2	2.82
deeplearning4j	731	3	0.41
dropwizard	256	0	0
Equivalent-Exchange-3	387	0	0
Essentials	572	3	0.52
gradle	554	8	1.44
graylog2-server	212	2	0.94
groovy-core	678	4	0.59
infinispan	23	0	0
jedis	192	4	2.08
jenkins	2008	16	0.8
jitsi	78	3	3.85
jsoup	42	0	0
junit	350	4	1.14
k-9	426	3	0.7
kotlin	499	2	0.4
lucene-solr	209	3	1.44
mct	199	0	0
mockito	38	0	0
netty	175	1	0.57
OG-Platform	4522	31	0.69
okhttp	1038	1	0.1
OpenRefine	70	1	1.43
OpenTripPlanner	683	9	1.32
orientdb	817	15	1.84
Osmand	3864	4	0.1
realm-java	782	14	1.79
retrofit	280	1	0.36
roboguice	73	0	0
rstudio	1463	2	0.14
rundeck	549	1	0.18
RxJava	418	0	0
Spout	854	3	0.35
voldemort	457	3	0.66
Total	**34030**	**245**	**0.72**
Mean			**0.88**
Standard Deviation			**1.07**

Table B.9: False negatives added by unstructured merge in terms of conflicts.

Project	Conflicts	Duplicated Declaration Errors	(%)
Activiti	166	8	4.6
AndEngine	39	10	20.41
andlytics	82	4	4.65
AntennaPod	142	2	1.39
antlr4	127	12	8.63
atmosphere	118	29	19.73
BroadleafCommerce	851	59	6.48
Bukkit	12	1	7.69
cassandra	6796	1907	21.91
cgeo	304	11	3.49
clojure	4	0	0
closure-compiler	0	4	100
cloudify	163	54	24.88
commafeed	2	0	0
commons	1	0	0
Conversations	21	0	0
cxf	77	12	13.48
deeplearning4j	253	6	2.32
dropwizard	19	0	0
Equivalent-Exchange-3	68	0	0
Essentials	23	23	50
gradle	288	21	6.8
graylog2-server	145	11	7.05
groovy-core	215	13	5.7
infinispan	90	0	0
jedis	293	66	18.38
jenkins	429	22	4.88
jitsi	29	4	12.12
jsoup	7	0	0
junit	134	5	3.6
k-9	266	17	6.01
kotlin	171	14	7.57
lucene-solr	1677	60	3.45
mct	44	0	0
mockito	57	0	0
netty	167	8	4.57
OG-Platform	3066	113	3.55
okhttp	6	2	25
OpenRefine	59	27	31.4
OpenTripPlanner	649	46	6.62
orientdb	509	71	12.24
Osmand	311	6	1.89
realm-java	322	39	10.8
retrofit	35	2	5.41
roboguice	100	0	0
rstudio	145	5	3.33
rundeck	18	1	5.26
RxJava	76	0	0
Spout	224	6	2.61
voldemort	438	13	2.88
Total	**19238**	**2714**	**14.10**
Mean			**9.62**
Standard Deviation			**16.12**

Table B.10: Spacing conflicts in terms of merge scenarios.

Project	Merge Scenarios	Merge Scenarios with Spacing Conflicts	(%)
Activiti	786	8	1.02
AndEngine	115	1	0.87
andlytics	560	6	1.07
AntennaPod	519	7	1.35
antlr4	656	4	0.61
atmosphere	244	4	1.64
BroadleafCommerce	898	23	2.56
Bukkit	19	1	5.26
cassandra	3360	55	1.64
cgeo	1890	5	0.26
clojure	37	0	0
closure-compiler	233	0	0
cloudify	213	5	2.35
commafeed	241	0	0
commons	208	0	0
Conversations	481	2	0.42
cxf	71	2	2.82
deeplearning4j	731	7	0.96
dropwizard	256	0	0
Equivalent-Exchange-3	387	1	0.26
Essentials	572	4	0.7
gradle	554	2	0.36
graylog2-server	212	2	0.94
groovy-core	678	13	1.92
infinispan	23	0	0
jedis	192	13	6.77
jenkins	2008	5	0.25
jitsi	78	0	0
jsoup	42	0	0
junit	350	6	1.71
k-9	426	8	1.88
kotlin	499	4	0.8
lucene-solr	209	13	6.22
mct	199	3	1.51
mockito	38	1	2.63
netty	175	2	1.14
OG-Platform	4522	76	1.68
okhttp	1038	0	0
OpenRefine	70	3	4.29
OpenTripPlanner	683	20	2.93
orientdb	817	10	1.22
Osmand	3864	10	0.26
realm-java	782	17	2.17
retrofit	280	1	0.36
roboguice	73	3	4.11
rstudio	1463	7	0.48
rundeck	549	1	0.18
RxJava	418	1	0.24
Spout	854	6	0.7
voldemort	457	20	4.38
Total	**34030**	**382**	**1.12**
Mean			**1.46**
Standard Deviation			**1.63**

Table B.11: Spacing conflicts in terms of conflicts. (Based on the number of reported conflicts of Semistructured Merge)

Project	Conflicts	Spacing Conflicts	(%)
Activiti	123	12	9.76
AndEngine	37	2	5.41
andlytics	49	12	24.49
AntennaPod	152	44	28.95
antlr4	115	12	10.43
atmosphere	89	6	6.74
BroadleafCommerce	516	66	12.79
Bukkit	12	5	41.67
cassandra	4191	172	4.1
cgeo	235	7	2.98
clojure	5	0	0
closure-compiler	1	0	0
cloudify	154	50	32.47
commafeed	1	0	0
commons	0	0	0
Conversations	25	4	16
cxf	45	2	4.44
deeplearning4j	100	18	18
dropwizard	9	0	0
Equivalent-Exchange-3	78	2	2.56
Essentials	20	6	30
gradle	159	2	1.26
graylog2-server	78	4	5.13
groovy-core	193	44	22.8
infinispan	71	0	0
jedis	280	83	29.64
jenkins	178	6	3.37
jitsi	21	0	0
jsoup	1	0	0
junit	58	9	15.52
k-9	188	42	22.34
kotlin	112	7	6.25
lucene-solr	1171	45	3.84
mct	40	4	10
mockito	6	1	16.67
netty	111	2	1.8
OG-Platform	3572	259	7.25
okhttp	9	0	0
OpenRefine	138	39	28.26
OpenTripPlanner	513	30	5.85
orientdb	445	32	7.19
Osmand	216	35	16.2
realm-java	303	35	11.55
retrofit	13	1	7.69
roboguice	105	6	5.71
rstudio	72	7	9.72
rundeck	16	2	12.5
RxJava	31	1	3.23
Spout	139	37	26.62
voldemort	348	43	12.36
Total	**14544**	**1196**	**8.22**
Mean			**10.87**
Standard Deviation			**10.44**

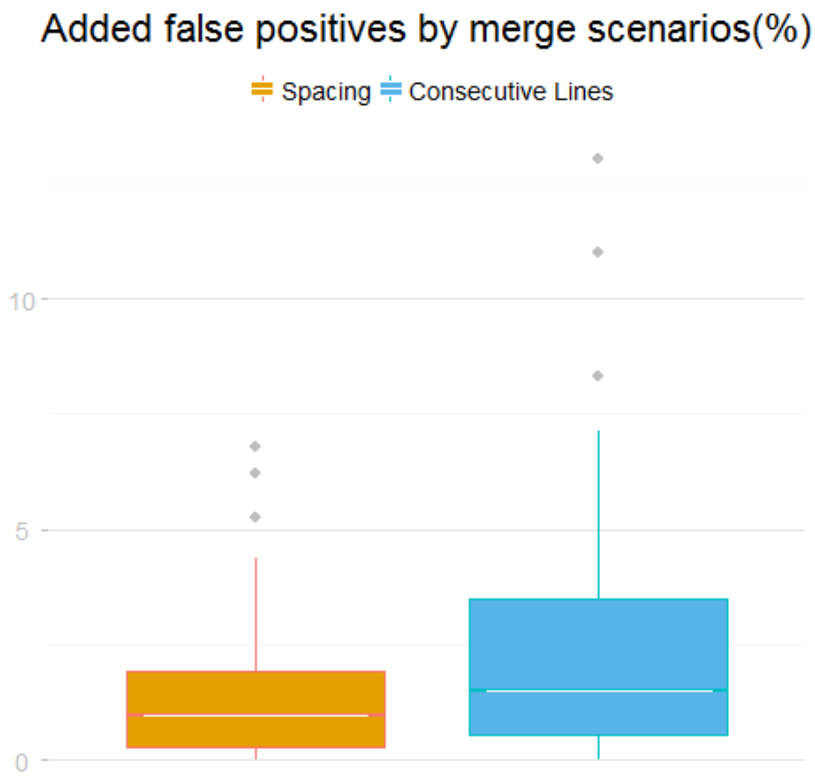
Table B.12: Consecutive lines conflicts in terms of merge scenarios.

Project	Merge Scenarios	Merge Scenarios with Consecutive Lines Conflicts	(%)
Activiti	786	9	1.15
AndEngine	115	1	0.87
andlytics	560	5	0.89
AntennaPod	519	9	1.73
antlr4	656	7	1.07
atmosphere	244	2	0.82
BroadleafCommerce	898	42	4.68
Bukkit	19	0	0
cassandra	3360	240	7.14
cgeo	1890	31	1.64
clojure	37	2	5.41
closure-compiler	233	0	0
cloudify	213	5	2.35
commafeed	241	0	0
commons	208	0	0
Conversations	481	2	0.42
cxf	71	0	0
deeplearning4j	731	10	1.37
dropwizard	256	1	0.39
Equivalent-Exchange-3	387	0	0
Essentials	572	4	0.7
gradle	554	19	3.43
graylog2-server	212	9	4.25
groovy-core	678	8	1.18
infinispan	23	3	13.04
jedis	192	16	8.33
jenkins	2008	20	1
jitsi	78	1	1.28
jsoup	42	1	2.38
junit	350	10	2.86
k-9	426	10	2.35
kotlin	499	8	1.6
lucene-solr	209	23	11
mct	199	7	3.52
mockito	38	2	5.26
netty	175	3	1.71
OG-Platform	4522	120	2.65
okhttp	1038	0	0
OpenRefine	70	4	5.71
OpenTripPlanner	683	28	4.1
orientdb	817	28	3.43
Osmand	3864	18	0.47
realm-java	782	25	3.2
retrofit	280	1	0.36
roboguice	73	3	4.11
rstudio	1463	10	0.68
rundeck	549	0	0
RxJava	418	0	0
Spout	854	12	1.41
voldemort	457	31	6.78
Total	**34030**	**790**	**2.32**
Mean			**2.53**
Standard Deviation			**2.84**

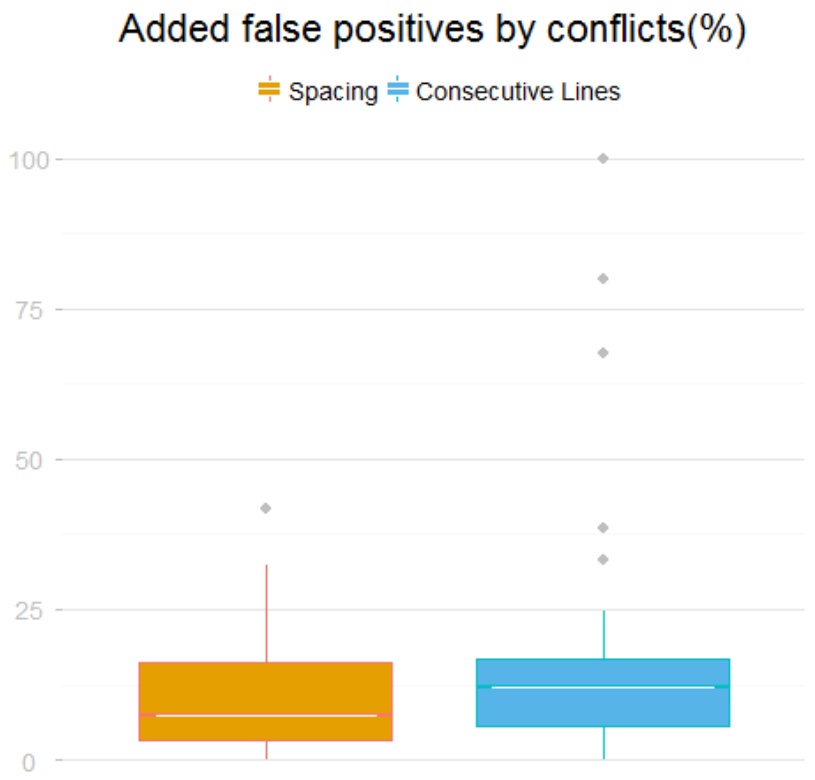
Table B.13: Consecutive lines conflicts in terms of conflicts. (Based on the number of reported conflicts of Semistructured Merge)

Project	Conflicts	Consecutive Lines Conflicts	(%)
Activiti	123	15	12.2
AndEngine	37	1	2.7
andlytics	49	9	18.37
AntennaPod	152	24	15.79
antlr4	115	15	13.04
atmosphere	89	7	7.87
BroadleafCommerce	516	84	16.28
Bukkit	12	0	0
cassandra	4191	451	10.76
cgeo	235	56	23.83
clojure	5	4	80
closure-compiler	1	0	0
cloudify	154	35	22.73
commafeed	1	0	0
commons	0	0	0
Conversations	25	2	8
cxf	45	0	0
deeplearning4j	100	15	15
dropwizard	9	1	11.11
Equivalent-Exchange-3	78	0	0
Essentials	20	4	20
gradle	159	22	13.84
graylog2-server	78	12	15.38
groovy-core	193	11	5.7
infinispan	71	5	7.04
jedis	280	69	24.64
jenkins	178	29	16.29
jitsi	21	1	4.76
jsoup	1	1	100
junit	58	14	24.14
k-9	188	15	7.98
kotlin	112	13	11.61
lucene-solr	1171	54	4.61
mct	40	27	67.5
mockito	6	2	33.33
netty	111	7	6.31
OG-Platform	3572	228	6.38
okhttp	9	0	0
OpenRefine	138	13	9.42
OpenTripPlanner	513	67	13.06
orientdb	445	54	12.13
Osmand	216	29	13.43
realm-java	303	43	14.19
retrofit	13	5	38.46
roboguice	105	3	2.86
rstudio	72	12	16.67
rundeck	16	0	0
RxJava	31	0	0
Spout	139	19	13.67
voldemort	348	58	16.67
Total	**14544**	**1536**	**10.56**
Mean			**15.56**
Standard Deviation			**19.25**

Figure B.1: Boxplots describing false positives added by unstructured/semistructured merge per project in terms of merge scenarios and conflicts in relation to Structured Merge.



(a)



(b)
Source: the authors.