UNIVERSIDADE FEDERAL FLUMINENSE

GLEIPH GHIOTTO LIMA DE MENEZES

ON THE NATURE OF SOFTWARE MERGE CONFLICTS

Niterói

2016

UNIVERSIDADE FEDERAL FLUMINENSE

GLEIPH GHIOTTO LIMA DE MENEZES

ON THE NATURE OF SOFTWARE MERGE CONFLICTS

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Doctor of Science. Area: Systems and Information Engineering.

Advisor: Leonardo Gresta Paulino Murta

Co-advisor: Márcio de Oliveira Barros

Niterói

2016

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

M543 Menezes, Gleiph Ghiotto Lima de
On the nature of software merge conflicts / Gleiph Ghiotto Lima de Menezes. – Niterói, RJ : [s.n.], 2016.
129 f.

Tese (Doutorado em Computação) - Universidade Federal Fluminense, 2016.
Orientadores: Leonardo Gresta Paulino Murta, Márcio de Oliveira Barros.

1. Engenharia de software. 2. Desenvolvimento de software. 3. Otimização (Computação). 4. Linguagem de programação. I. Título.

CDD 005.1

ON THE NATURE OF SOFTWARE MERGE CONFLICTS

Gleiph Ghiotto Lima de Menezes

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Doctor of Science. Area: Systems and Information Engineering.

Approved in December of 2016.

_____

Prof. D.Sc. Leonardo Gresta Paulino Murta – Advisor – UFF

_____

Prof. D.Sc. Márcio de Oliveira Barros – Co-advisor – UNIRIO

_____

Prof. D.Sc. Viviane Torres da Silva – UFF

_____

Prof. D.Sc. Esteban Walter Gonzalez Clua – UFF

_____

Prof. D.Sc. Claúdia Maria Lima Werner – UFRJ

_____

Prof. Ph.D. Paulo Henrique Monteiro Borba – UFPE

_____

Prof. Ph.D. André van der Hoek – UCI

Niterói

2016

“Live as if you were to die tomorrow. Learn as if you were to live forever.”

(Mahatma Gandhi)

“Mistakes are the portals of discovery”

(James Joyce)

ACKNOWLEDGEMENT

I would like to thank my advisor Leonardo Murta for all dedication and enthusiasm he always shows during the Master and the Doctorate. He is the professional example that I would like to follow.

I would like to thank my co-advisor Márcio Barros for all the effort during the attempts of applying Search-based Software Engineering in our research and during the reviews of papers and the Thesis. He is another awesome example to follow.

I would like to thank André for all the support that he gave my while I was in the USA and when I came back. In addition, I feel blessed for receiving contributions from such a very important person in the Software Engineering community.

I would like to thank my family, especially my parents, for all the support they gave me in the whole life, especially when they "understood" I needed to be offline for some time.

I would like to thank all my friends at UFF that helped me when I was having troubles and became part of my family.

I would like to thank my roommates for understanding the moments I need to concentrate and for all the times we spent together having fun.

Finally, I would like to thank CAPES for the financial support during the doctorate.

RESUMO

Quando diversos desenvolvedores alteram um software em paralelo, em algum momento as alterações concorrentes precisam ser reintegradas (i.e., sofrer *merge*) ao software em desenvolvimento. Diversas técnicas de *merge* foram propostas, mas nenhuma delas resolve de forma automática todos os casos possíveis. Na verdade, de 10 a 20% dos casos de merge falham por conta de conflitos. Nestes casos, os desenvolvedores devem intervir manualmente para resolvê-los. Atualmente poucas informações sobre a natureza dos conflitos de *merge* são conhecidas. Há diversas questões de pesquisa em aberto, como, por exemplo: (1) qual a quantidade de regiões em conflito por *merges* que falharam? (2) qual a quantidade de linhas de código nas regiões em conflito? (3) quantos constructos da linguagem existem em cada região em conflito? (4) quais os padrões entre constructos da linguagem podem ser observados nas regiões em conflito? (5) qual a distribuição de decisões dos desenvolveres na resolução das regiões em conflito? (6) qual a dificuldade de resolução dos tipos de conflitos encontrados? (7) quais padrões entre constructos da linguagem e decisões dos desenvolvedores podem ser observados nas regiões em conflito? (8) qual a correlação entre a quantidade de desenvolvedores e o número de *commits*, *merges* e *merges* que falharam? Para responder estas perguntas, nesta Tese realizamos um estudo quantitativo e qualitativo detalhado da natureza de *merges* por meio da: (1) análise dos conflitos de merge de milhares de projetos *open source*; (2) coleta e classificação das resoluções que os desenvolvedores utilizaram para tratar conflitos; e (3) análise das relações entre vários aspectos dos conflitos de merge e a estratégia de resolução. As respostas a essas questões de pesquisa deram origem a recomendações que podem apoiar desenvolvedores de ferramentas de *merge* a tratar alguns tipos de conflitos automaticamente ou aprimorar o processo de merge. Finalmente, como uma prova de conceito, uma das recomendações identificadas durante as análises foi implementada e avaliada, apresentando resultados promissores.

Palavras-chave: Merge de Software, Conflitos de Merge, Resolução de Merge

ABSTRACT

When multiple developers change a software system in parallel, these concurrent changes need to be merged to appear together in the software being developed. Numerous merge techniques have been proposed to support this task, but none of them can fully automate the merge process. Indeed, it has been reported that some 10 to 20% of all merge attempts result in a merge conflict, meaning that a developer has to intervene to manually complete the merge. To date, we have little insight into the nature of these merge conflicts. Multiple research questions are still open, such as: (1) what is the distribution in number of conflicting chunks for merge failures? (2) What is the distribution in size of conflicting chunks, as measured in lines of code (LOC)? (3) What is the distribution in language constructs involved in conflicting chunks? (4) What, if any, patterns exist in the language constructs of failed merges involving multiple conflicting chunks? (5) What is the distribution of developer decisions? (6) What is the distribution in difficulty level of kinds of conflicting chunks? (7) What, if any, patterns exist between the language constructs of conflicting chunks and developers' decisions? (8) What is the correlation between the number of developers and the number of commits, merges, and failed merges? This thesis contributes with an in-depth quantitative and qualitative study of merge conflicts by: (1) dissecting the merge conflicts found in the histories of thousands of open source projects, (2) collecting and classifying the manual resolution strategies that developers used to address these merge conflicts, and (3) analyzing the relationships between various aspects of the merge conflicts and the chosen resolution strategies. Our findings give rise to three primary recommendations for future merge techniques, that – when implemented – can help automatically resolve certain types of conflicts and provide the developer with tool-based assistance to more easily resolve other types of conflicts. Finally, as a proof of concept, we implemented one of our recommendations in a tool. The evaluation of this proof-of-concept tool showed promising results.

Keywords: Software Merge, Merge Conflict, Merge Resolution.

LIST OF FIGURES

LIST OF TABLES

ACRONYMS LIST

AST – Abstract Syntax Tree

CB – Combination

CC – Concatenation

CR – Cost Ration

LOC – Line of Code

NC – New Code

NN – None

VCS – Version Control System

V1 – Version 1

V2 – Version 2

SUMMARY

CHAPTER 1 – INTRODUCTION

1.1 MOTIVATION

According to Bersoff (1980), “no matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle”. Based on this statement, one can conclude that the software evolution process, that is, the process through which change are (systematically or not) introduced into a software project, is close to the daily business of software developers and users. Such evolution is required to attend the needs of users who work in and interact with a dynamic world, in which requirements change all the time due to changes in laws, rules, preferences, technologies, and so on (Lehman et al., 1997).

In this context arises Software Configuration Management, a Software Engineering discipline that offers support for software evolution (Dart, 1991). This discipline is supported by systems, such as the Version Control System (VCS), that manage changes in software artifacts (*e.g.*, source code, documentation, and so on) and enable controlled access to a repository in which these artifacts and their change history are stored. VCS allows software developers to obtain a set of artifacts (*i.e.*, a configuration), edit them, and share their contributions with other developers.

The use of branches in configuration management repositories is essential to software development, as it enables concurrent work (Berczuk et al., 2003). Branches are independent development lines that allow software artifacts to evolve in an isolated way, in parallel with the main line of development (*mainline*). By making changes to a branch, a developer can implement a feature or make a bug fix in isolation from changes made by other developers. However, once the changes are finished, they must be integrated with the project mainline (or another branch) to be available to other developers and, eventually, to the users. An approach to do so is to apply a merge tool, which implements some underlying merge technique to automate as much as possible the task of combining the changes made on the branch with the code in the mainline (Mens, 2002).

Many merge techniques have been developed over the years. These techniques differ considerably in what kind of information they use to compare two versions of an artifact and identify conflicting changes. A conflicting change happens when developers edit the source code of an artifact and generate versions that cannot be integrated automatically. A very

common example of conflicting change happens when two developers edit the same lines of code of a given Java class. A significant number of merge techniques rely on lines of code as the basis for combining the changes made in a branch with the code in the mainline; these techniques are called *unstructured merge techniques* (Hunt, J. W.; McIlroy, 1976; Miller; Myers, 1985; Myers, 1986; Berlage; Genau, 1993; Oster et al., 2006; Chacon, 2009). Other techniques, termed *structured merge techniques*, rely on syntax (Westfechtel, 1991; Binkley et al., 1995; Buffenbarger, 1995; Hunt, J. J.; Tichy, 2002; Shen; Sun, 2004, 2005; Apiwattanapong et al., 2007) and semantics (Berzins, 1994; Jackson; Ladd, 1994). Hybrid approaches that mix aspects of both unstructured and structured techniques, termed *semi-structured merge techniques*, have been explored as well (Apel et al., 2011, 2012; Leßenich et al., 2014).

Other techniques were developed to identify conflicts that can end up in a failed merge or to identify the failed merges as soon as possible. Failed merges are the result of conflicting changes and take place when the merge approach fails in an attempt to integrate two different versions of a given artifact. The so-called *awareness approaches* work while developers are coding, enabling the identification of conflicts that can generate a failed merge. Some examples of awareness approaches are Palantir (Sarma; van der Hoek, 2002; Sarma et al., 2003, 2012), Crystal (Brun et al., 2011a, b), and WeCode (Guimarães; Silva, 2012). On the other hand, *continuous integration approaches* try to identify failed merges as soon as possible and report the conflict to developers or integrators that, in theory, will resolve these conflicts and share the resolution with other developers.

Despite all of these advances, it is well known that, in practice, merge tools are not perfect – they cannot account for every possible concurrent changes that different developers may make. When a merge fails, the developer has to step in, analyze the respective changes and the conflict reported by the merge tool, and resolve the conflict manually. Software merge is known as a difficult task, one that developers wish to avoid as much as possible (Berlin; Rooney, 2006; Duvall et al., 2007). Still, it has been reported that 10 to 20% of all merges fail (Mens, 2002; Brun et al., 2011b), with some projects experiencing rates of over 50% (Kasi; Sarma, 2013).

There are few studies about how merges take place and how they are resolved (Brun et al., 2011b; Kasi; Sarma, 2013; Leßenich et al., 2014; Cavalcanti et al., 2015; Yuzuki et al., 2015; Santos; Kulesza, 2016). To the best of our knowledge, among these few studies, just one (Yuzuki et al., 2015) tried to understand where conflicts take place by analyzing a set of conflicts that occurred in *method declarations* of ten Java projects. However, this study brings

shallow evidences due to the restricted number of projects. Moreover, they did not search for patterns that can be explored to support merge resolution nor investigated how conflicts are usually resolved. Due to the lack of information about the nature of software merge conflicts, there is little evidence about the direction that developers of merge tools should follow and what features actually provide concrete benefits to the users.

1.2 METHODOLOGY

To spur the development of new merge techniques, reducing the failure rate for merge, we posit that it is necessary to take a deep dive into merge conflicts. How do they look like in detail? How do developers resolve them? Are there relationships between the nature of certain merge conflicts and the resolution strategies used by developers? If we were to have answers to these questions, perhaps it could be possible to identify new heuristics, build support for certain types of merge conflicts, or at least address repetitive situations that are seen all the time in practice. It may of course also be possible that nearly every single conflict is unique in its own right, and that it is thus highly unlikely that we might ever be able to develop better merge tools. Regardless of the answer, knowing it is important to future research.

The research work reported in this Thesis comprises three stages. First, we performed a manual analysis of five open source projects mainly written in Java to characterize the failed merges that occurred in these projects. This detailed manual analysis of each failed merge case allowed us to determine the most important aspects that should be addressed to characterize merges resulting in conflict. It also allowed us to discover the information that could be relevant to find patterns and strategies that improve the quality of software merge (that is, to build techniques and tools that might automatically resolve some of those failed merges without requiring the intervention of a developer).

For each failed merge, we collected its conflicting chunks (as a failed merge might be the result of conflicts in multiple, disjoint parts of the artifacts), the size of each chunk in lines of code, the programming language constructs contained in the chunks, and the way developers chose to resolve each chunks. We then looked for patterns among various aspects of the conflicting chunks (e.g., size, language constructs) and the resolutions chosen by the developers. We also guided our analyses by source code examples that illustrate the patterns and heuristics proposed for each case.

Next, we performed a broader, automatic analysis of software merge involving more than 2,700 open source projects written in Java and about 25 thousand failed merges, summing up 175,805 conflict chunks. While the manual analysis was helpful to understand

the object under analysis, the automatic analysis allowed the generalization of the results to a class of software projects (open source software written in Java), reducing the external threat to validity imposed by the limited number of projects analyzed manually. In addition, by running an automatic analysis we also mitigated the internal threat to validity resulting from the hard and error prone process of collecting information manually, performed by a single researcher.

Finally, as a proof of concept of one of our heuristics, we implemented an approach that helps developers to deal with failed merges comprising more than one conflicting chunk by ordering these chunks according to their dependencies. This approach demonstrates that the information collected in the manual and automatic analysis can lead to the development of improved merge approaches. The ordering approach is composed of two algorithms that handle dependencies extracted from chunks and order them in the sequence that should facilitate the work of the developers assigned to resolve a failed merge. For example, the resolution of former chunks can help on the resolution of later chunks.

1.3 GOALS

The main goal of this Thesis is to present the results of the analyses about the nature of failed merges. These analyses are guided by the following questions:

1. What is the distribution in number of conflicting chunks for merge failures?
2. What is the distribution in size of conflicting chunks, as measured in lines of code (LOC)?
3. What is the distribution in language constructs involved in conflicting chunks?
4. What, if any, patterns exist in the language constructs of failed merges involving multiple conflicting chunks?
5. What is the distribution of developer decisions?
6. What is the distribution in difficulty level of kinds of conflicting chunks?
7. What, if any, patterns exist between the language constructs of conflicting chunks and developers' decisions?
8. What is the correlation between the number of developers and the number of commits, merges, and failed merges?

By answering these questions, we identified some opportunities that can lead to improvements for automatic or semi-automatic (by providing guidance to developers) merge resolution. As a secondary contribution of this Thesis, we implemented a proof-of-concept

merge resolution tool that orders the chunks of a failed merge exploiting their dependencies. We also provide an initial validation for this tool by comparing its results with those provided by traditional VCS.

1.4 MAIN CONTRIBUTIONS

After running the manual and automatic analysis, we could collect evidences that (1) conflicting chunks generally contain all of the necessary information to resolve them – 87% of the chunks do not need extra code beyond the one in the versions that generated the conflict; (2) the resolution order of conflicting chunks matter – 46% of the failed merges analyzed manually have dependences between pair of chunks that can be explored in order to ease further resolutions; and (3) past choice of how conflicting chunks were resolved can inform future choices – information of how a specific kind of conflict was previously resolved can suggest future resolutions. Moreover, our proof-of-concept tool showed that ordering chunks can improve *assistance* in 48.39% of the failed merges and 35.48% of the failed merges have lower *noise* among conflicting chunks when compared to the traditional approaches in VCS.

1.5 ORGANIZATION

The Thesis is organized in five chapters including this Introduction. Chapter 2 presents some background information about VCS, along with related work about merge analysis.

Chapter 3 presents the manual and automatic analyses performed to characterize failed merges. It establishes a terminology that serves as a basis to understand the subsequent analyses and their results. Then, the analyses and their respective research questions are introduced. Next, we describe which projects were selected for manual analysis, which were selected for automatic analysis, as well as the detailed processes through which data were collected. Finally, the results of the analysis are presented and discussed.

Chapter 4 presents the proof-of-concept tool to order conflicting chunks comprised in failed merges. To do so, we present some concepts as dependencies and dependency graph. Then, we present the algorithms used to order the chunks. Finally, we show a case study and evaluate the approaches in a set of 31 failed merges.

Finally, Chapter 5 concludes this Thesis showing the contributions of this work, limitations, and future work.

CHAPTER 2 – SOFTWARE MERGE

2.1 INTRODUCTION

Developers create branches in VCS to allow parallel and isolated work during software development. Thus, developers can perform corrective tasks, adaptive tasks, or other kinds of maintenance activities without generating side effects in the tasks performed simultaneously by other developers. Nonetheless, once a comprehensive set of tasks is concluded, developers have to combine their changes with those performed by other developers. The combination of changes is called *merge* and is thoroughly discussed in Section 2.4. Several developers may also work in the same branch, implying a lower degree of isolation. In this case, each change performed by a developer leads to the immediate need of a merge. This situation looks simpler than the previous, given that each merge integrates the contents affected by a single change, and not two or more independent sequences of changes made over time and probably involving a larger number of software artifacts.

Merge tools can support the combination of changes performed by two or more developers (Mens, 2002). During the evolution of a software product, some changes will be automatically combined by current merge approaches, while others will demand the support of developers playing the role of integrators. Usually, merge cases are resolved automatically when developers change different parts of the software artifacts. On the other hand, there are situations in which traditional merge approaches are unable to avoid conflicts (*i.e.*, conflicts take place when developers change the same structure in parallel, such as, the same line in a source code file) and the work of an integrator is required. In the former case, the merge tool may miss some existing conflict (false negative). In the latter case, the merge tool may indicate a conflict that could have been automatically resolved (false positive).

Software merge is a common problem, faced on a daily basis by developers. In this context, merge tools are an important resource to perform this task automatically, becoming essential to deliver products to the user in due time (Shen; Sun, 2004). Consequently, software merge is considered a hot topic by researchers that intend to reduce the cost of software integration through automated tasks. Research on software merge encompasses merge of models (Murta et al., 2008; Koegel et al., 2010), images (Silva Junior et al., 2012), and text files (Mens, 2002; Shen; Sun, 2004, 2005). However, we still do not have a clear vision of how software merge happens in practice and the very nature of software merge

conflicts. This vision would provide valuable information to the development of better merge approaches in the future.

This chapter is divided into six sections, including this introduction. Section 2.2 presents an overview of VCS, explores some concepts, and introduces how repositories of different kinds of VCS can be organized. Section 2.3 explores the policies adopted by VCS. Section 2.4 shows the different kinds of merge (*i.e.*, physical, syntactical, and semantic) highlighting the main characteristics of them and giving some examples. Section 2.5 introduces researches comparing different merge approaches, assigning developers to perform the merge, and so on. Finally, we conclude this chapter in Section 2.6 presenting some final remarks.

2.2 VERSION CONTROL SYSTEMS AND THEIR REPOSITORIES

A VCS allows the storage, retrieval, comparison, and comprehension of software evolution. Among these tasks, the storage of artifacts is done in a repository. Every time an artifact or a set of them is stored, a new version is created (Berczuk et al., 2003). Artifact retrieval is performed through requests of one or a set of artifacts (*i.e.*, a configuration) to the repository. Comparison is performed using VCS commands that allow identifying the differences between stored versions, the so-called *deltas*. Finally, software evolution can be analyzed through queries based on the software development history, using VCS commands to recover information such as who changed a given artifact, which lines were changed (for file-based VCS), when the changes occurred, and the goal of these changes. It is important to highlight that repository management is always performed automatically, avoiding manual tasks that might be counterproductive and error prone. For instance, when developers are not using VCS repositories, they may overwrite artifacts during development tasks.

VCSs can be classified either as centralized (CVCS) or distributed (DVCS), according to their architecture. CVCS uses a single remote repository (*i.e.*, a central repository). Some examples of CVCS are CVS (Fogel; Bar, 2001), Perforce (Wingerd, 2005), and Subversion (Collins-Sussman et al., 2008). Figure 1 presents the traditional work cycle of a CVCS, with its three main commands: *checkout*, *commit*, and *update*. The *checkout* command is responsible for obtaining artifacts from the repository to a workspace[1]. The workspace is a

[1] Workspaces keep software developers isolated from changes performed by other developers until an update or a pull is performed to sync the workspace with other repositories.

copy, on the developer's machine, of the artifacts recovered from the repository. After executing the *checkout* command, the developer can change the artifacts in the workspace and then execute the *commit* command. At this moment, the VCS integrates the edited contents in the workspace with the contents of the repository. If some change was performed in the central repository meanwhile, the developer needs to update the contents of the workspace before *committing* the integrated content. The *update* command synchronizes the changes performed in the remote repository with those performed in the local workspace. After the *update*, the workspace will have all the changes performed in the remote repository. Consequently, there may happen conflicts during this integration and these conflicts must be resolved by the developer before executing the *commit*.

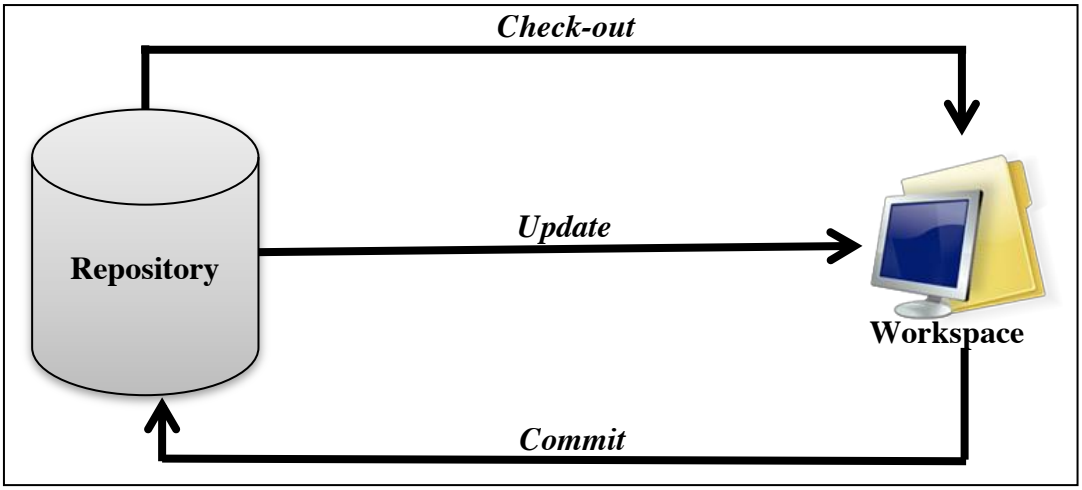

Figure 1. Work cycle in CVCS.

Differently from CVCS, DVCS includes support for many repositories in its architecture. The work cycle of a DVCS is roughly the same as of a CVCS, regarding their *checkout* and *commit* commands. However, the first difference is the *clone* command, which copies the whole remote repository to the developer's workstation (Chacon, 2009). During the execution of a *clone* operation, the developer's workstation receives the whole repository instead of just a specific configuration in the workspace as in a CVCS. Figure 2 shows a hypothetic example of a DVCS architecture, which is comprised of three repositories: one bare repository (*i.e.*, repository without workspace) that is the reference for other two standard repositories (*i.e.*, repositories attached to a workspace). It is worth mentioning that DVCS allows developers to share (dashed arrows in the figure) artifact with other repositories beside the reference repository.

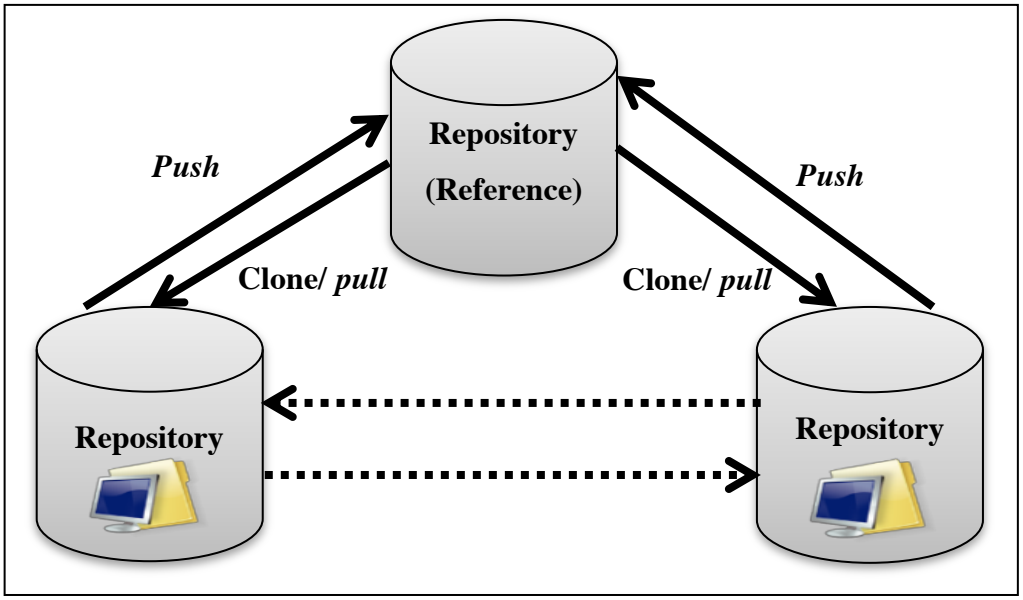

Figure 2. Work cycle in DVCS.

A DVCS also has commands to synchronize the content of different repositories. As shown in Figure 2, the *clone* command copies the whole repository, including the software development artifacts and their history of changes. Then, the developer can act as in a CVCS, changing artifacts and committing these changes to the local repository, until he/she decides to share the local contributions with other developers. Then, the developer runs the *push* command, which sends all the local changes to a remote repository. Nevertheless, the developer may execute the *pull* command to bring all changes registered in a remote repository to the local repository. Thus, as in the *update* for CVCS, conflicts can happen during execution of a *pull* for DVCS.

2.3 CONCURRENCY CONTROL

Most well-known VCSs support two policies of concurrency control (Sarma et al., 2003; Prudêncio et al., 2012): pessimistic and optimistic. In the former, a lock is used to serialize the access to software artifacts. Consequently, an artifact cannot be edited in parallel (*i.e.*, just one developer can edit the artifact in a specific time), avoiding concurrent changes by two or more developers. Therefore, low-level conflicts are avoided. Although the pessimistic policy avoids developers having the obligation of resolving low-level conflicts, high-level conflicts may still occur, as artifacts have dependencies to other artifacts. Moreover, it might lead to unnecessary serialization, as two or more developers may want to change different parts of an artifact in parallel, without generating conflicts. In addition, according to Mens (2002), the pessimistic policy is not well suited for projects having a large development team.

On the other hand, the optimistic policy allows two or more developers to edit the same artifact in parallel. If two developers change the same artifact, one developer may have to merge the changes performed by both due to conflicts (*e.g.*, changes performed over the same artifact, which cannot be automatically combined). Additionally, in some cases, the developer in charge of merging artifacts must reach an agreement with the other developer to decide on the resolution of the conflict. This occurs because the changes can have diverging intentions that are not in the domain of knowledge of a single developer.

Pessimistic concurrency control policies cannot be applied for DVCS due to its distributed nature. Consequently, conflicts can arise during the synchronization of repositories. Hence, the use of merge tools in the context of DVCS is even more important, since each clone represents the creation of an implicit branch. Moreover, developers can be geographically distributed, making the communication harder and the failed merges more challenging to resolve.

Different VCS adopt different concurrent control policies. For instance, RCS (Tichy, 1985) and Visual SourceSafe (Serban, 2007) opted for the pessimistic policy. Subversion (Collins-Sussman et al., 2008) implements both policies and the developers can choose between pessimistic or optimistic. Finally, CVS (Cederqvist, 2003), Mercurial (O'Sullivan, 2009), and Git (Chacon, 2009) use only the optimistic policy.

2.4 MERGE

Merge support is necessary during all the development life cycle of large scale systems (Perry et al., 1998; Mens, 2002). This is enforced by Walrad and Strom (2002), who highlight the need of branches to support the development and maintenance of software systems. These branches need to be integrated to the mainline and, consequently, merge arises as one of the main resources for delivering software systems with high quality and within the schedule constraints imposed to software projects (Shen; Sun, 2004).

Software merge can be defined as an operation that combines the changes performed in different development lines (Mens, 2002). Furthermore, merge algorithms can be classified considering how the differences between development lines are obtained (*i.e.,* during the edition of artifacts or after) and the resources used (*e.g.*, text file structures, language elements, or application domain), among others. These algorithms are also classified as unstructured (*i.e.*, the physical merge), structured (*i.e.*, the syntactical and semantic merges), and semi-structured algorithms (*i.e.*, a combination between unstructured and structured merge algorithms). The aforementioned characteristics are discussed in the next subsections.

2.4.1 PHYSICAL MERGE

Physical merge (Hunt, J. W.; McIlroy, 1976; Miller; Myers, 1985; Myers, 1986; Berlage; Genau, 1993; Chacon, 2009), or textual merge, represents the first category of merge algorithms implemented in VCS (Shen; Sun, 2004). In this kind of merge, the artifacts are combined considering only the structures present in text files (Mens, 2002). Hence, the differences between two files are identified considering characters, words, or lines. Consequently, physical merge arises as a software merge approach that is able to integrate textual files regardless of their type (*e.g.*, Java, XML, C, and so on), but with limited applicability on binary files.

Physical merge algorithms can vary in granularity and this choice affects the quality and computational cost of the merge. For instance, the finest and indivisible grain in text files is the character, and it can lead to a precise representation about what was really changed by the developers in a given artifact. However, it can also lead to inefficient results in practice (Mens, 2002), due to the amount of detail involved in the comparison. On the other hand, adopting the entire text file as an indivisible grain would generate a huge amount of conflicts if two developers change a given file in parallel, even if the changes were made in different parts of the file. Therefore, the line grain is used in the traditional merge algorithms to represent the indivisible unit of software artifacts.

Merge algorithms may also be classified as two-way or three-way. Two-way merge can be used to combine software artifacts without a common ancestor. This algorithm identifies the differences between two artifacts and combines them. However, it does not have the exact information about the changes that were performed in the artifact. For example, if a line was added in one file or removed from the other. Figure 3 shows an example of such situation. In this example, a two-way merge can generate four different results. The merge algorithm cannot know if line two was changed from "*double meters;*" to "*double meters = 0;*" or from "*double meters = 0;*" to "*double meters;*", or even if both were added concomitantly. Conversely, it is also not possible to decide if line four was changed from "*double kilometers =0;*" to "*double kilometers*" or vice-versa. Under those circumstances, merge results can be any combination illustrated in Figure 3.

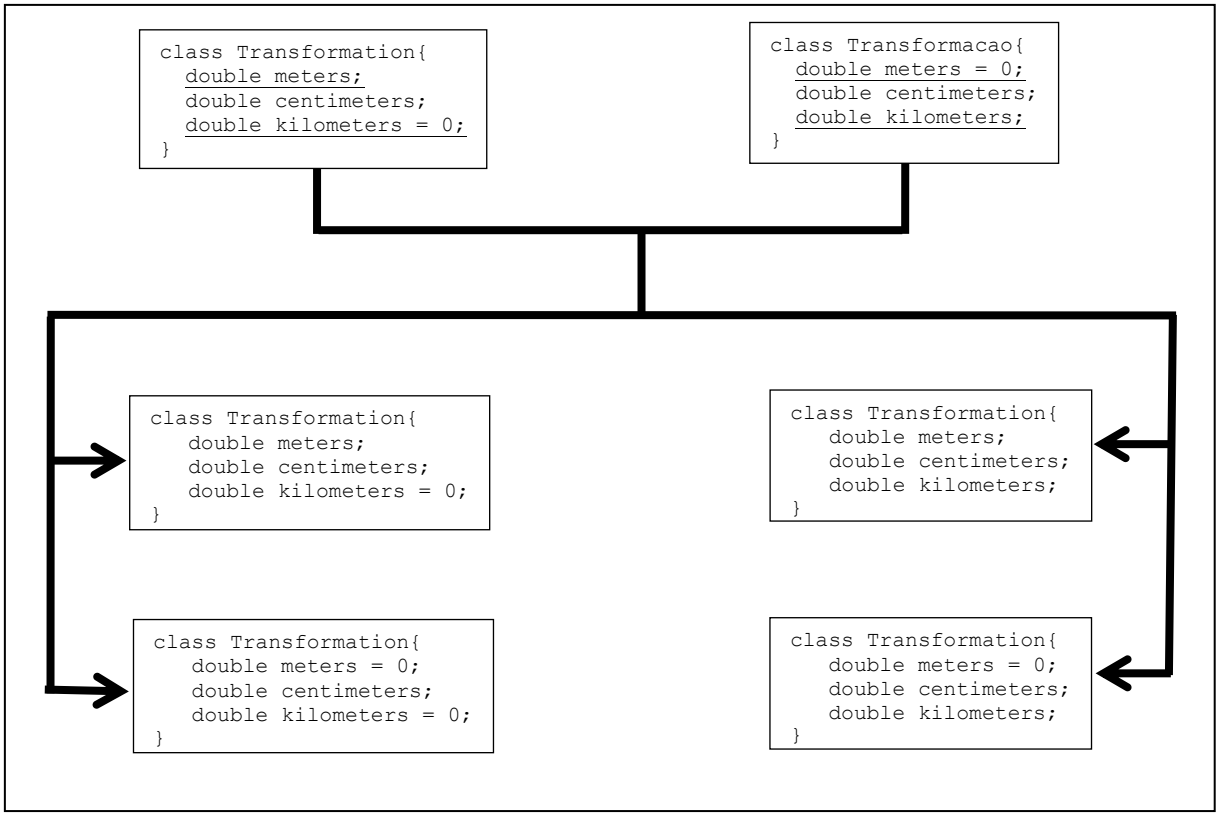

Figure 3. Usage example of two-way merge.

Therefore, two-way merge is unable to clearly identify the intention of the developer during his/her changes in the software artifact. To solve this limitation, three-way merge uses a common ancestor of the versions generated by the developers. The ancestor is the reference used to decide if a line was added or removed, following a simple heuristic: if a line does not exist in the common ancestor and exists in the version generated by the developer, then this line was added; on the other hand, if a line exists in the common ancestor but no longer exists in the developer's version, then it was removed.

Figure 4 shows an example of three-way merge. In this example, the same changes shown in Figure 3 were performed, but this scenario has a common ancestor. Thus, the merge algorithm realizes that the contents of line 4 was changed from "*double kilometers;*" to "*double kilometers = 0;*" on the left-hand side. While, on the right-hand side, line 2 was changed from "*double meters;*" to "*double meters = 0;*". Consequently, the common ancestor helps on identifying the intention of the changes.

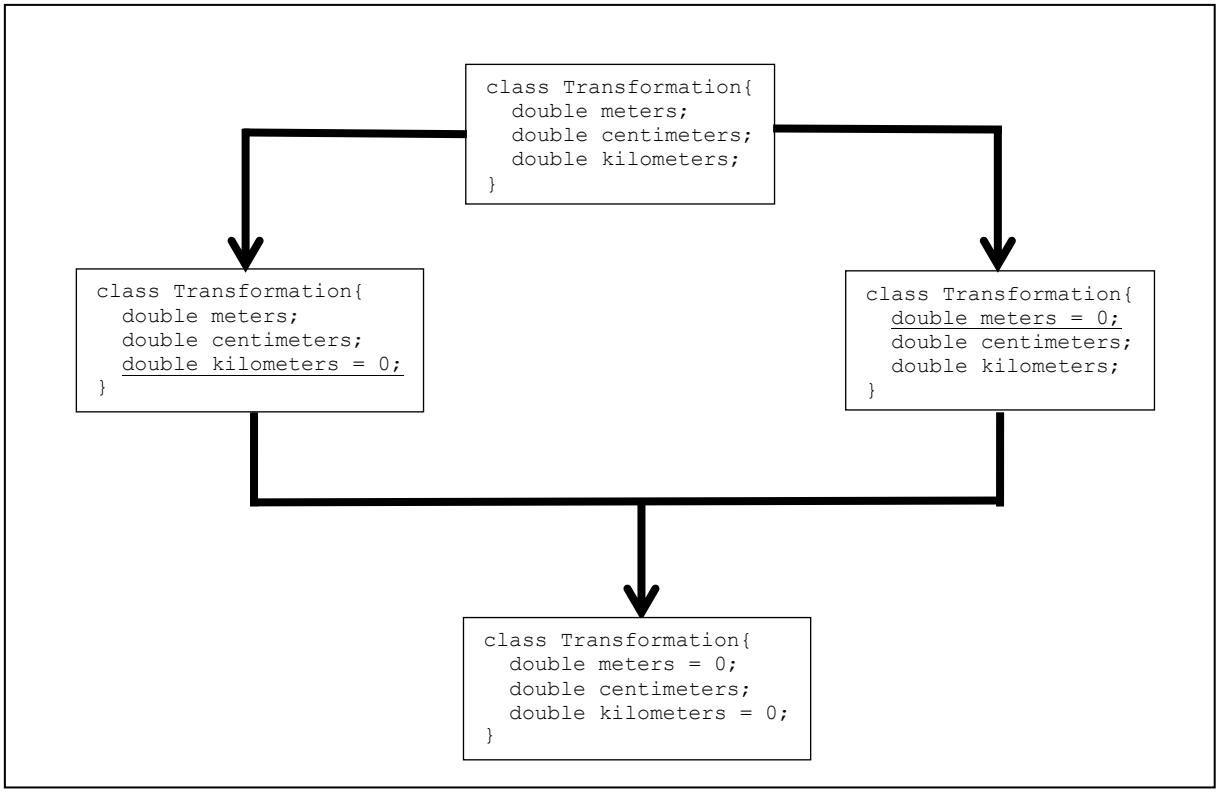

Figure 4. Usage example of three-way merge.

During the physical merge, conflicts may be identified. Figure 4 illustrates a merge scenario on which no low-level conflict took place because the changes were made in different regions of the software artifact. However, conflicts can occur when developers edit the same region of an artifact. A conflict leads to a decision problem about which version content should be taken as a resolution: the left-hand side, the right-hand side, a combination of some lines from each version, the concatenation of both versions, or a new code? This decision must be delegated to a developer who knows the application domain, or to specialized tools that can help developers in conflict resolution.

Figure 5 shows a scenario in which conflict occurs during the merge. In this scenario, line three, which had the content "*double centimeters;*", has now two possible resolutions: based on the change of the left-hand side, it should be "*double centimeters = 0;*", while considering the change in the right-hand side, it should be "*double centimeters = 0.0;*". This scenario presents a conflict because the merge algorithm does not know which change to pick, given that the line was changed in parallel to different contents. Therefore, the algorithm shows both alternatives and asks the developers for a resolution.

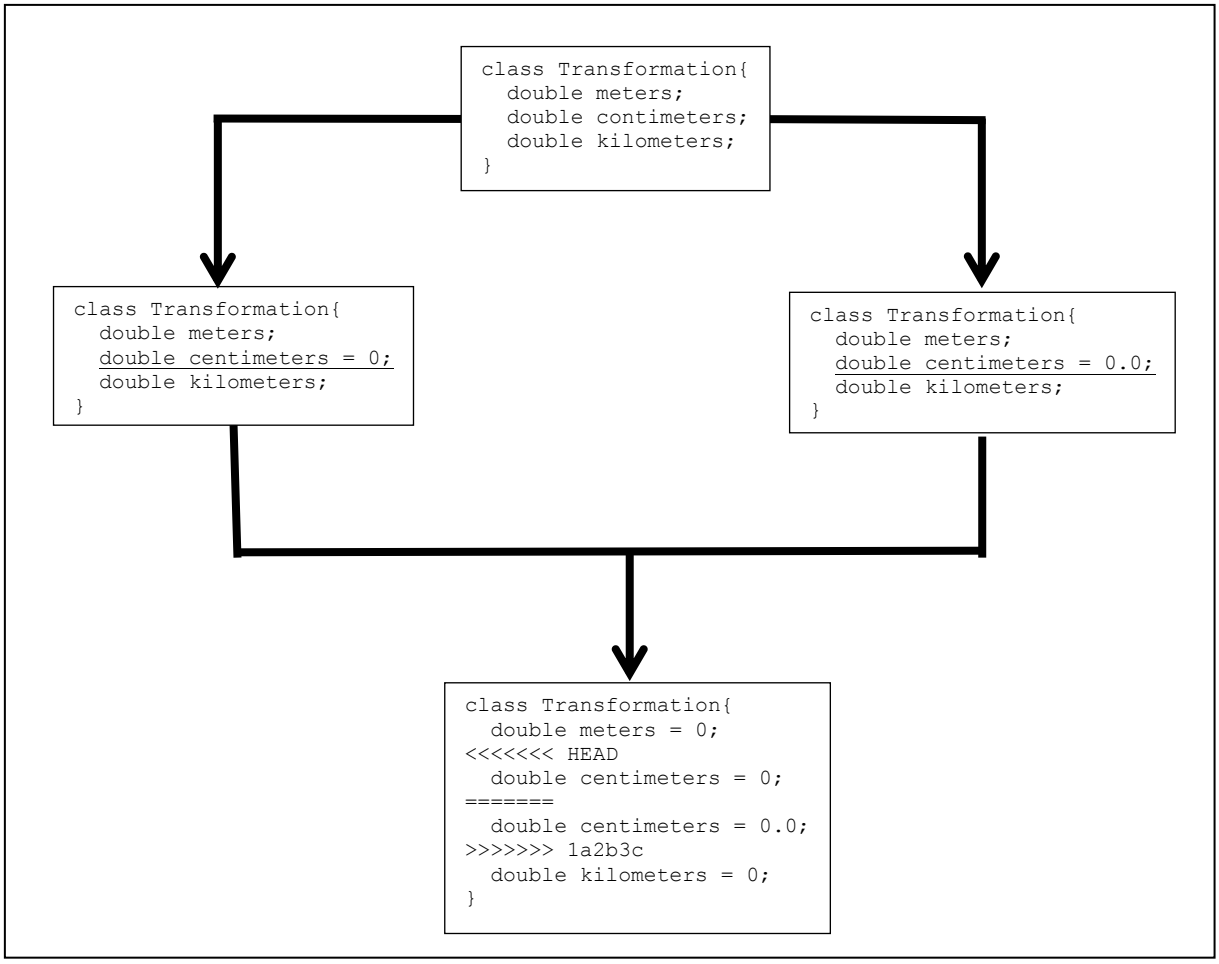

Figure 5. Conflicting scenario using three-way merge.

When a merge attempt fails, a conflict may manifest itself in multiple parts of the artifacts. That is, it is possible – and frequently happens – that, when artifacts are in conflict, the conflict exhibits itself in several regions across the artifacts. We term each pair of those regions that is in conflict a *conflicting chunk*.

Figure 5 illustrates a conflicting chunk using Git notation (Chacon, 2009), the DVCS that hosts all projects in this Thesis, that was created for didactical goals. It uses three marks: the beginning mark, represented by "<<<<<<<", which is followed by the version in which changes are to be integrated (in this case HEAD, the last version of the branch on which the user is working; this is the version in the workspace of the developer in Git); the separator, "=======", which demarks the code that differs between the two versions in conflict; and the ending mark, ">>>>>>>", which is followed by the version from which the changes are to be taken, in this case 1a2b3c. Although marks differ per merge tool, all merge tools provide conceptually equivalent results.

Two-way and three-way merge approaches are state-based. State-based algorithms derive the changes through comparisons among different versions of the same artifact. For

instance, three-way merge uses the *diff* algorithm to verify how the artifact evolution took place from one version to another. A sequence of addition and removal operations is obtained by using heuristics that classify changes based on the chosen granularity (*e.g.*, lines of source code). Nevertheless, this approach has some limitations to identify the developer's intention.

Besides two-way and three-way merge, another state-based approach is called differential synchronization (Fraser, 2009). This approach performs the synchronization and combination of software artifacts in real time. It differs from the traditional merge algorithms because it neglects the isolation generated by workspaces. Differential synchronization proposes that merge should be performed in both directions to maintain the artifact in the workspace and in the central repository as similar as possible at all times. However, there are situations in software development scenarios that require the isolation, such as when a developer explicitly asks for isolation using branches.

Operation-based algorithms are used to deal with the limitations in capturing the real intention of developers. This kind of algorithm obtains information about the changes while developers edit the source-code and stores these changes in an operation history (Berlage; Genau, 1993). The operations stored can be as complex as necessary to represent the real intention of developers. An example of an operation-based approach for software merge was proposed by Shen and Sun (2004) and uses addition and removal operations to represent the developer's intention. Figure 6 presents an overview of how this approach acts in the example used along with this section. The operation in the left-hand side is represented by the notation Ad("static ", 2, 10), which means an insertion of the content "static " in line 2 and position 10. In the right-hand side, we can observe the operation Ad(" = 0", 2, 16). Thus, the result is obtained by applying the following sequence of operations: first the operation Ad(" = 0", 2, 16) is applied in the line 2 position 16 of the common ancestor; then, operation Ad("static ", 2, 10) is applied in the line 2 position 10 of the resulting file, leading to the artifact presented in Figure 6. Note that if the opposite order of application were chosen, the operation Ad(" = 0", 2, 16) should be changed to keep the context. Then, after applying Ad("static ", 2, 10), the displacement of 6 positions in the line should be considered in the operation Ad(" = 0", 2, 16), which should be transformed to Ad(" = 0", 2, 22).

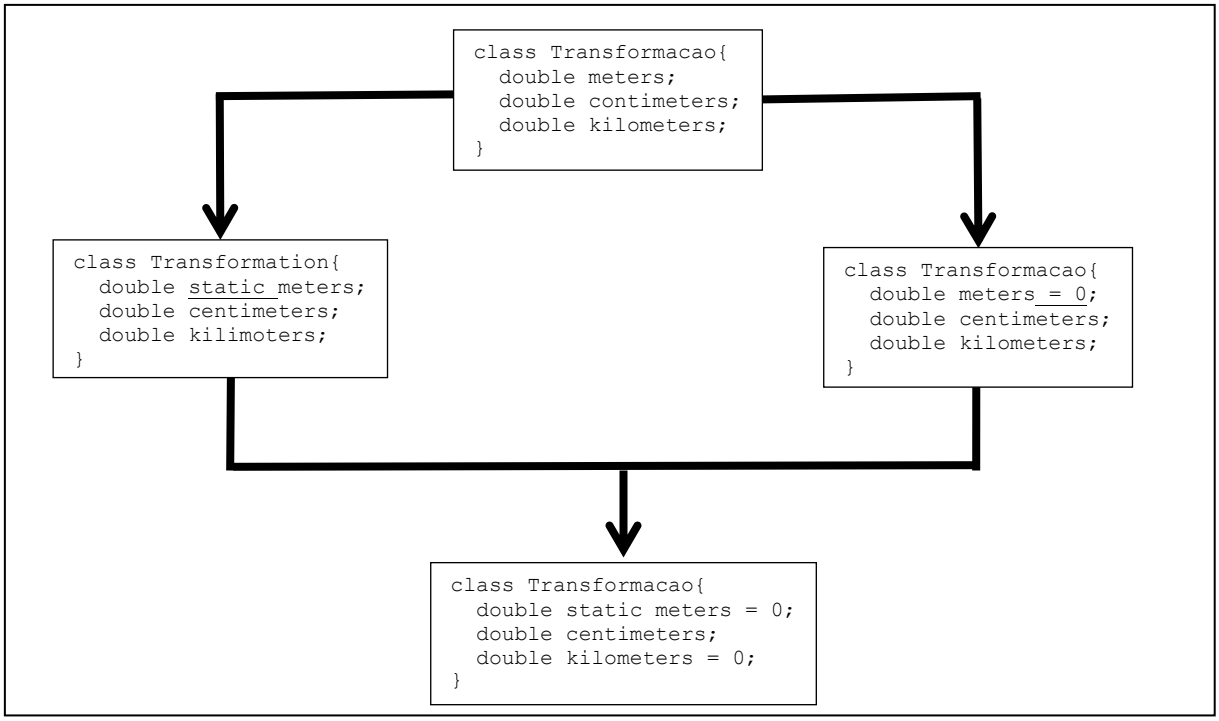

Figure 6. Operation-based merge example.

This example shows that operation-based approaches using the character grain describe the developer's intention in a better way than state-based approaches using the line grain. Some examples of operation-based approaches are: GINA (Berlage; Genau, 1993), an approach that allows optimistic policy differently from previous ones (Ellis; Gibbs, 1989; Rhyne; Wolf, 1992); IceCube (Kermarrec et al., 2001), an approach that tries to reorder operations to avoid conflicts; approaches based on operation transformation (Ellis; Gibbs, 1989; Ressel et al., 1996; Suleiman et al., 1997; Vidot et al., 2000; Molli et al., 2003; Cart; Ferrie, 2007); and approaches based on commutative replicated data type (Oster et al., 2006; Preguica et al., 2009; Weiss et al., 2009; Roh et al., 2011), which was created to work with commutative operations (*i.e.*, operations that can be applied in different orders without affecting the final result).

However, this kind of merge approach needs to be integrated into the development environment, which makes operation-based merge intrusive because developers need to install plug-ins in their environments to map the operations during development and generate the operation log history. This limitation can make the operation-based approach unrealistic for big and complex software projects, which generally have developers geographically distributed and, additionally, each developer or development team has preferences about the environment used to do their work.

Physical merge represents a category of algorithm largely used in practice. One of the biggest users of physical merge is the VCS, which uses line grain due to efficiency, scalability, and precision (Mens, 2002). The usage of the line grain has the drawback of not being able to combine changes performed in parallel over the same line. Nevertheless, according to studies performed in industry, this fact does not represent a big problem since 90% of the changed files are combined in an automatic way using the line grain. However, according to Mens (2002), the 10% of the merge cases that do not have automatic resolution should use syntactical and/or semantic information of software artifacts to be resolved.

2.4.2 SYNTACTICAL MERGE

Syntactical merge represents a category of merge approaches that is more powerful than physical merge (Mens, 2002) due to its ability to take into account the structure of the software, which is related to the programming language or notation in which the artifacts are represented. Moreover, this kind of approach is not influenced by changes that do not have syntactical relevance, such as adding whitespaces and tabs to the source code in programming languages like Java and C++. Some examples of syntactical merge approaches are described by Westfechtel (1991), Binkley *et al*. (1995), Buffenbarger (1995), Hunt *et al*. (2002), Shen and Sun (2005), and Apiwattanapong *et al*. (2007).

Syntactical merge can be divided into two categories: tree-based and graph-based. Consequently, syntactical merge requires an initial step that consists in transforming the content of the artifacts into an Abstract Syntax Tree (AST) or a graph. Thus, the comparison of programs using this kind of approach consists in walking on the tree or graph and looking for different nodes. In other words, after the transformation of the source code in its tree or graph representation, characteristics strictly related to its textual representation are not identified as conflicts, unless these characteristics are part of the grammar of the language.

Figure 7 shows a syntactical merge example that presents the *Transformation* class with two attributes: *meters* and *centimeters*. The initial step transforms the source code into the structure of an AST representing the class contents in a higher-level abstraction. Next, the AST that represent the software artifacts are compared. On the left-hand side of Figure 7, whitespaces were added before the attributes *meters* and *centimeters* to organize the source code. In parallel, on the right-hand side, the attributes were initialized with zero. Considering those differences, a syntactical merge algorithm can combine the changes because just the assignments to zero have syntactic impact in the code. It is important to highlight that the formatting characteristics of the merge result will depend on the AST writer. Therefore,

developers have no guarantee that the formatting characteristics of the source code will be preserved after the merge.

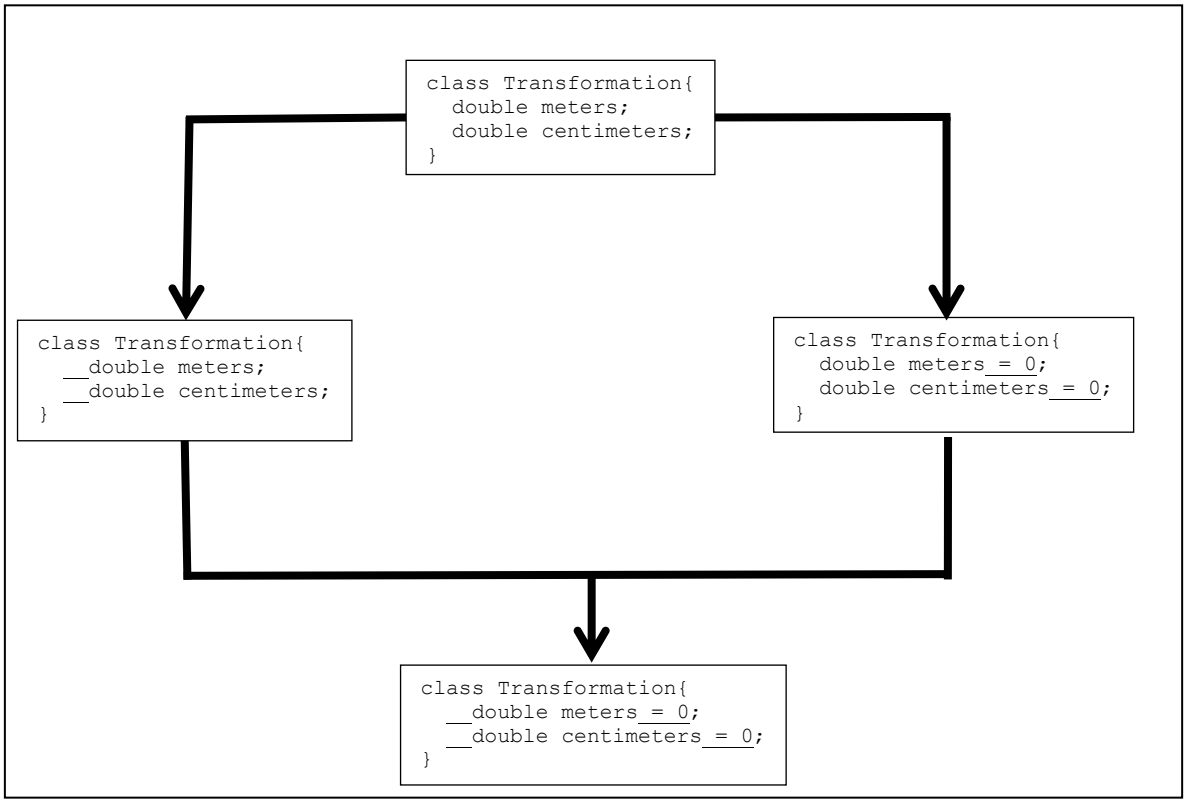

Figure 7. An example of syntactical merge.

Although syntactical merge can resolve merge cases in which physical merge would lead to conflicts, it still has some disadvantages. The main drawback is that syntactical merge is language dependent, that is, if the merge algorithm was developed for the Java language, it can only merge files written in this language. On the other hand, physical merge is language independent because it only relies on structures that all text files have (*e.g.*, lines, words, and characters). Thus, if an approach intends to resolve merges for several languages, it needs to provide many translators to build the AST or graph representations from the source code and also create the textual representation of the source code from the AST. Another disadvantage of syntactical merge is its low efficiency, because depending on the desired precision, the problem of identifying differences between trees or graphs can be NP-complete (Zhang; Jiang, 1994). Finally, another drawback is the loss of textual formatting since the code must be translated to a tree and translated back to text afterwards.

Westfechtel (1991) proposed a merge technique that uses a context-free grammar and Binkley *et al*. (1995) proposed an approach that takes into account the behavior of procedure calls while performing merges. These two techniques are a step toward structured merge, but they fail in simple situations such as renaming a variable or method. Hunt *et al*. (2002)

introduced an extensible language-aware merge technique that deals with renaming and non-local conflicts (*i.e.*, conflicts that can be in different files or entities), but cannot identify differences in the behavior caused by dynamic binding.

Shen and Sun (2005) worked on an approach that does the repositioning of operations and syntactical verification to realize if the merge result has conflicts. This approach is implemented in a VCS called FORCE (Shen; Sun, 2005) and takes into account operations to add and remove textual structures. Both operations, adding and removing, are described in function of the line and position of each edition as shown in the example of Figure 6. The authors show some examples where conflicts can be identified, but the approach still presents limitations regarding which conflicts should be reported. For instance, consider the piece of code "*a = 1;*" that is edited by two developers. Assume that the first developer changes the previous version to "*a = a + 1;*" and the second edits the same piece of code to "*a = 1 + a;*". Since FORCE is operation-based, it identifies that the changes were performed in different areas and the final result will be "*a = a + 1 + a;*", which does not represent the intention of the developers, although the result is correct from a syntactical point-of-view.

Finally, Apiwattanapong *et al*. (2007) proposed an approach to differentiate Java files that takes into consideration the constructs of that programming language. Given two programs, the algorithm identifies the matching language constructs (*e.g.*, *classes*, *methods*, *and so on*) and, for each pair of constructs, it shows the similarities and differences between them. The *diff* represents an initial step towards merge and, consequently, can be treated as a related approach in this topic. However, as any syntactical merge, it has limitations related to the number of programming languages that it can deal with – the current implementation works only with programs written in Java.

2.4.3 SEMANTIC MERGE

Semantic merge represents a more powerful category of merge algorithms when compared to physical and syntactical merge. This category can identify conflicts that physical and syntactical merges cannot because it is related to semantic aspects of the programming language and the behavior of the software system. Figure 8 presents a scenario with the *Transformation* class that has the method *transform*, which was designed to convert a value represented in centimeters to meters while preserving the original signal. Again, consider a scenario in which two developers change the same artifact in parallel. The first developer performs the change represented in the left-hand side of Figure 8 to convert values from

centimeters to kilometers. In parallel, the second developer performed the change represented in the right-hand side, ensuring that the output of the method will always be positive.

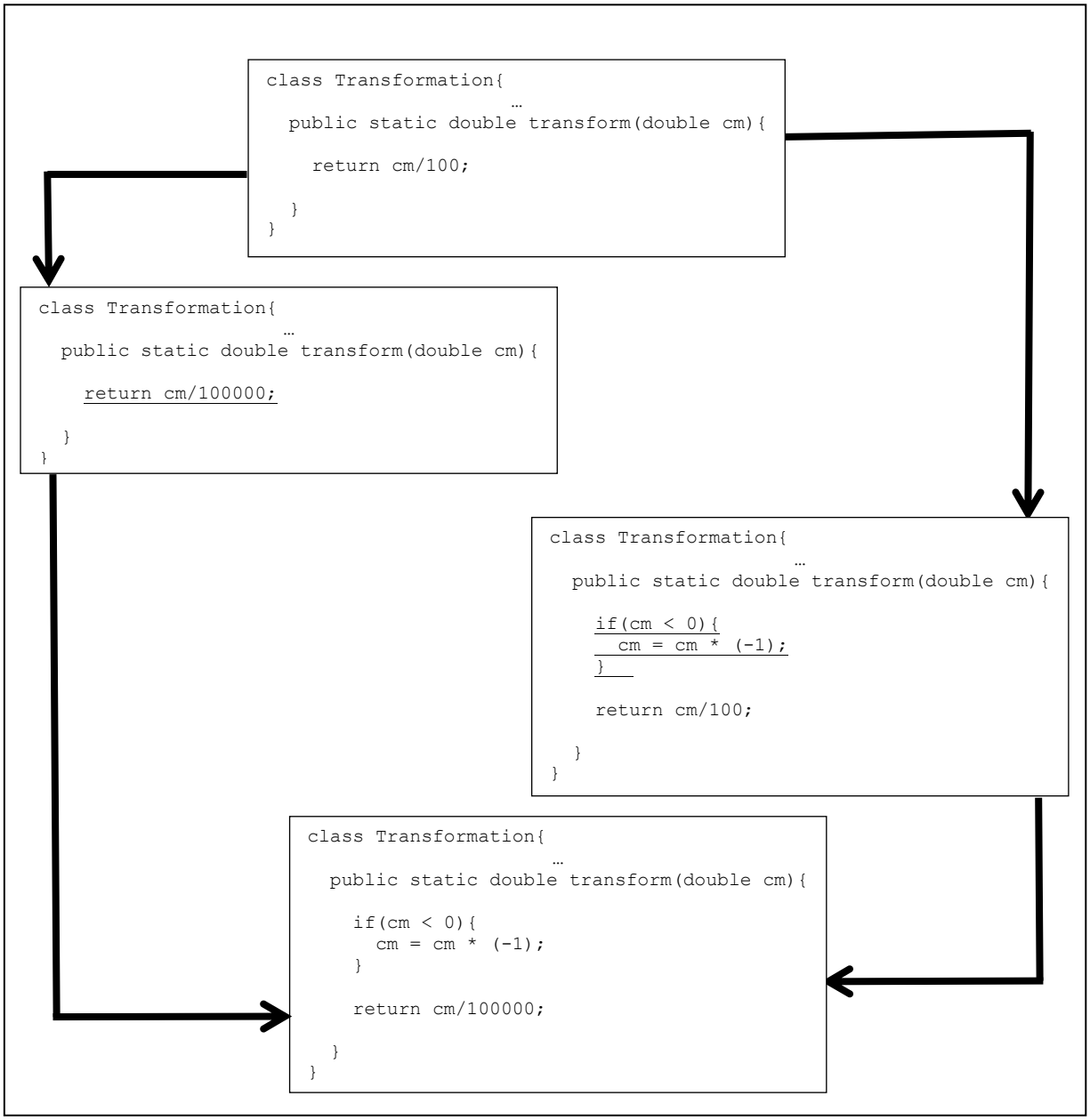

Figure 8. An example of semantic conflict.

These changes generate neither physical (*i.e.*, there are no changes in the same area) nor syntactical (*i.e.*, the final result is respecting the Java language grammar) conflicts. However, they generate a semantic conflict, that is, the expected behavior of the program is not preserved (*i.e.*, when the signal of *cm* is negative, the signal will be changed, which disagrees with the initially established rules).

Semantic conflicts can be classified as static or behavioral (Mens, 2002). Static semantic conflicts can be extracted from the language semantics. For instance, a static semantic conflict happens when a variable is renamed to two different names by two

developers in parallel. Static conflicts can be identified through complex techniques, such as the construction of program dependency graphs (Horwitz; Reps, 1992) and program slicing (Weiser, 1981). In contrast, behavioral semantic conflicts are captured when the changes performed in parallel result in an artifact that does not have the behavior designed by the developer, as presented in Figure 8. This kind of conflict cannot be extracted from AST or graphs representing the program. Behavioral semantic conflicts resolutions can use test cases or OCL restrictions to guarantee that the changes being merged do not affect the expected behavior of the software.

Semantic merge can identify conflicts that physical and syntactical merges cannot (Mens, 2002). Nevertheless, it also has some disadvantages and limitations. For instance, semantic techniques are domain dependent (*i.e.*, they depend on the relations extracted from the programming language and on the domain of the application) and the application domain cannot be extracted from an AST. Therefore, this kind of merge approach still demands research to be useful in practice.

Few researchers are currently interested in semantic merge. Bersinz (1994) proposed a language-independent semantic model to combine software artifacts that uses Boolean algebra and establish condition under which a set of changes can be merged independently (*i.e.*, non-conflicting situations). On the other hand, the authors also show situations in which changes cannot be merged. On their turn, Jackson and Ladd (1994) proposed a diff approach that receives procedures as input and generates a report summarizing the differences between them. This approach does not actually merge the artifacts, but show the semantic differences between them.

2.4.4 SEMI-STRUCTURED MERGE

The merge techniques proposed in the previous subsections present tradeoffs among precision, generality, and performance. Physical merge is the most general technique, but it is not precise because it works over textual structures with no knowledge about how these structures connect to each other and can be interpreted. Moreover, physical merge shows high performance when working in the line grain, but loses performance when working with characters. On the other hand, syntactical and semantic merges are more precise, but they demand more computational resources than physical merge. In addition, they work over specific file types, being less general.

In an attempt to reconcile precision, generality, and performance, researchers have explored semi-structured merge techniques that combine aspects of both unstructured and

structured merge techniques (Apel et al., 2011, 2012; Leßenich et al., 2014). Apel et al. (2011), for instance, sought to build a merge tool that can address different programming languages. To this end, they parameterize the merge tool using multiple programming language grammars, falling back to unstructured merge when an unknown language is found. This offered a reconciliation of precision and generality, but suffered in performance. Hence, this topic was treated in subsequent work by using auto-tuning (Apel et al., 2012; Leßenich et al., 2014). This technique uses unstructured merge by default and, in cases in which a certain merge case leads to conflicts, structured merge takes place. With auto-tuning, the authors reduced the time spent in structured merge, applying unstructured when it can solve the problem and relying on structured merge only when necessary. As a consequence, this approach is more exposed to false negative conflicts than structured merge.

2.5 MERGE ANALYSIS

Some work studied how frequent are the different types of conflicts (Brun et al., 2010, 2011a; Kasi; Sarma, 2013; Santos; Kulesza, 2016), the effort of integrating software artifacts when using specific merge approaches (Leßenich et al., 2014; Cavalcanti et al., 2015), how to assign developers to resolve merge conflicts (Costa, C.; Figueiredo; Ghiotto; et al., 2014; Costa, C.; Figueiredo; Murta, 2014), and how conflicting merges are usually resolved (Yuzuki et al., 2015).

Brun et al. (2011a) analyzed the frequency of physical, syntactical, or semantic conflicts in the history of the following software projects: Git, Perl5, Voldemort, Gallery3, Insoshi, jQuery, MaNGOS, Rails, and Samba. They observed that physical conflicts occur in 16% of the merges, while syntactical and semantic conflicts happen in 1.4% and 6.4% of the merges, respectively. This represents around 20% of merge failures. In contrast, Kasi and Sarma (2013) made a similar analysis on four software projects (Perl, Storm, Jenkins, and Voldemort) that exhibited 40%, 44%, 34%, and 54% merge failures, respectively. Both works show that merge conflicts are frequent and approaches to deal with them are welcome.

Santos and Kulesza (2016) observed the amount and types of merge conflicts that happen when evolving and merging the SIGAA project[2]. They analyzed the following kinds of merge conflicts: (1) direct conflicts, which represent a pair of code changes applied to the

[2] SIGAA is a web information system designed to automate business processes for universities focusing on different and complementary aspects, such as academic, administration, planning and management.

same code element (*e.g.*, *attributes*, *methods*) by both the source (*i.e.*, the system that has the changes) and target systems (*i.e.*, the system that will receive the changes) when source and target evolved from the same source; (2) indirect conflicts, which happen when code changes applied to the source system are located in the call graph of code changes in the target system; and (3) pseudo-conflicts, which happen when the source and the target systems modify the same class or interface, but different and independent code elements (*attributes* or *methods*). Based on these conflict types and the number of development issues (*i.e.*, issues created during development as the ones related to new requirements, bug fix, and so on) they concluded that 65.18% of the issues do not have conflict, 5.6% have direct conflicts, 18,83% have indirect conflicts, and 10.34% have pseudo conflicts. Moreover, 5.6% of the detected conflicts are direct, 68.14% are indirect, and 22.20% are pseudo-conflicts. These results show that about 35% of all issues involve conflicts and few of these are direct conflicts, which are easy to find. Thus, approaches that identify conflicts and/or help developers to resolve them are needed.

Another group of work analyzes how semi-structured merge improves unstructured merge (Leßenich et al., 2014; Cavalcanti et al., 2015). In the first work (Leßenich et al., 2014), the authors analyzed 50 projects and 434 merge cases concluding that, in almost all projects, structured merge reports fewer and smaller (*i.e.*, less lines of code) conflicts than unstructured merge. In addition, structured merge is substantially slower, but the results are improved with auto-tuning. In a second study (Cavalcanti et al., 2015) 60 projects and 3,266 merge cases were analyzed (7.5 times more merge cases than in the former study). The authors also concluded that the number of unstructured conflicts is higher than those reported by semi-structured merge approaches. They suggest that using semi-structured merge can decrease the manual integration effort without compromising the correctness of the software.

Other researchers characterized the problem of assigning developers to merge branches (Costa, C.; Figueiredo; Ghiotto; et al., 2014; Costa, C.; Figueiredo; Murta, 2014) and proposed a tool to help on this task (Costa, Catarina et al., 2016b, a). They could observe that some merge cases have up to 50 developers changing files in each branch and that, in most cases, there are few intersections of developers across branches. Moreover, they run a survey with 164 developers and could observe that most developers try to commit early to avoid merge conflicts and, when merge conflicts arise, they contact other developers to resolve conflicts together. They also proposed an approach that aims at analyzing the developers who changed key files in the branch histories and in the history before branching to rank them and indicate who are more capable to resolve the conflicts.

Finally, Yuzuki *et al*. (2015) analyzed the characteristics of conflicts in *method declarations* over the following Java projects: James, JRobin, Maven Plugins, and seven Eclipse extensions (Ajdt, Gmp graphiti, Jumula core, Paho mqtt java, Scout sdk, Stardust ui web, and UML2). They observed how conflicts are generated and concluded that 44% of the conflicts were due to concurrent changes (*e.g*., changes in the same part of the *method* made by 2 or more developers), while 48% were due to removing *methods*, and 8% were caused by renames. They also observed how the conflicts are resolved classifying in 1-way (*i.e*., selecting one of the versions in conflict) or other (*i.e*., other solution including some content from the versions in conflict or adding new code). In this analysis, they concluded that 99% of the conflicts inside methods were resolved by choosing one of the conflicting versions, which is a pattern that repeats in eight projects.

2.6 FINAL REMARKS

In this chapter, we discussed the relevance of software merge and presented some few works that shed some light on the nature of merge conflicts. For instance, some works show the number of merges that presents physical, syntactical, or semantic conflicts during the merge attempt. Other works discuss how people react to a failed merge when it happens and who should be in charge of resolving conflicts. Finally, some work started to step over the ground of failed merges by analyzing specific types of failed merge and how such type of failed merge is usually resolved.

Considering the aforementioned discussion, we can conclude that there is a gap in the literature that needs to be filled: the absence of studies that analyze the characteristics of scenarios involving software merge, diving into the source code and providing evidences that would help us to have a deeper understanding of the nature of failed merges. Which are the characteristics of failed merges? How are the conflicts resolved? Are there patterns in the conflicts? To the best of our knowledge, the only work that discusses this subject was conducted by Yuzuki *et al*. (2015). However, this study considers only 10 Java projects and focused on a very specific language construct: *method declaration*. As a consequence, deepest studies that collect more information about a larger number of software projects might be useful and important to guide developers with concrete evidences to improve merge techniques.

CHAPTER 3 – ANALYSES OVER SOFTWARE MERGE CONFLICTS

3.1 INTRODUCTION

Although merge is a technique largely used in VCS to integrate versions that evolve in parallel, there is no study to understand the nature of failed merges. To contribute in the understanding of how the failed merges look like and build knowledge to support the development of new merge techniques that reduce the failure rate, we present an analysis performed manually to examine the history of five open source projects, finding the merges that conflicted while these projects were developed.

For each conflicting merge, we collected the following data: (1) the number of conflicting chunks (as a merge conflict might be the result of conflicts in multiple, disjoint parts of the involved artifacts), (2) the size of each of these chunks in lines of code, (3) the programming language constructs contained in each chunk, and (4) the way developers chose to resolve different conflicting chunks (as it is possible that they used different strategies to resolve different parts of the same merge conflict). Having this in hand, we then looked for patterns between various aspects of the conflicting chunks (*e.g.*, size, subset of language constructs) and the kinds of resolutions chosen by the developers (*i.e.*, wholesale adopt one of the versions, concatenate both version in either order, combine code from both versions without writing new code, and mix existing code from one or both versions with newly written code). Finally, we confirmed whether the patterns found in the manual analyses were due to the selected projects or might be found in similar ones through automatic analyses that were performed over a set of almost three thousand open source projects.

The remainder of this chapter is organized as follows. Section 3.2 describes the terminologies used in this chapter. Section 3.3 introduces the analyses that we did in this chapter. Section 3.4 describes how we performed the manual analyses and Section 3.5 presents the steps we followed to have the automated results. Section 3.6 presents our results, characterizing merge conflicts, resolution strategies, and the relationships between them. Section 3.7 discusses the implications of our findings. Section 3.8 covers threats to validity. Section 3.9 concludes the analyses presenting the main findings.

3.2 TERMINOLOGY

As discussed in Section 2.4.1, failed merges can be comprised of a multitude of conflicting chunks, as the one presented in Figure 9. However, this representation in sometimes difficult do understand. To ease comprehension, we adopt the side-by-side representation depicted in Figure 10. We refer to the code on the left-hand side of Figure 10 as *version 1* (before the separator in Figure 10, representing the changes made in the workspace of the developer) and the code on the right-hand-side of Figure 10 as *version 2* (after the separator in Figure 10, representing the version from which the changes are to be integrated into the workspace; often this will be the main branch).

```
public RuleStopState stopState;
<<<<<<< HEAD
  public boolean isPrecedenceRule;
=======

  @Override
  public int getStateType() {
    return RULE_START;
  }
>>>>>>> b80ad5052d1b693be6e5c0a2b
}
```

Figure 9. Conflicting chunk of merge b14ca5 from ANTLR4.

```
public RuleStopState stopState;
```

version 1	*version 2*

`public Boolean isPrecedenceRule;`	`@Override`
	`public int getStateType() {`
	`  return RULE START;`
	`}`

```
}
```

Figure 10. Simplified side-by-side representation for the conflicting chunk of Figure 9.

A specific goal of the Thesis is to dive into the nature of conflicting chunks. Additionally, the literature discussed in the Chapter 2 motivates us to go toward a syntactical analysis, programming language-dependent approach as a way to improve the current merge approaches. Thus, we asked questions such as what kinds of language constructs (e.g., *for*, *while*, *if*, *class*, *variable*, *method*) are part of conflicting chunks and how often, or which patterns may exist in the language constructs that appear together in conflicting chunks. As discussed in the following, we selected five Java projects to analyze, which is why we used the Java language specification to create the list of language constructs for our analysis, including statements (e.g., *for*, *while*, *if*), definitions (e.g., *class*, *variable*, *method*), invocations, and so on. We decided to also include annotations (e.g., @Override, @NotNull), as they can change the semantics of method or variable declarations. To use the conflict in Figure 10 as an example, the conflicting chunk contains the following language constructs: *variable* in *version 1*, and *annotation*, *method declaration*, *variable*, and *return* in *version 2*.

Another concept we use in our analysis is the *kind of conflict*, which allows classifying different conflicting chunks. We define the kind of conflict as the *concatenation* of all of the unique language constructs (in alphabetical order) that are present in a conflicting chunk. To assign a type of conflict for a given conflicting chunk, then, we take the language constructs from both *version 1* and *version 2*, sort them, remove duplicates, and concatenate the remaining ones. In Figure 10, the kind of conflict is "*annotation, method declaration, variable*".

When developers face a failed merge, they have to resolve the conflicting chunk(s). Exactly how they do that is what we term a *developer decision*. We study developer decisions on a conflicting chunk basis, and identify five different ways of resolving the conflict: (1) adopt the code of *version 1*, (2) adopt the code of *version 2*, (3) concatenate both versions, in either order, (4) combine select code from both versions without writing any new code, and (5) mix existing code with newly written code. We identify these choices in the remainder of the Thesis as: *version 1* (*V1*), *version 2* (*V2*), *concatenation* (*CC*), *combination* (*CB*), and *new code* (*NC*). For the automatic analysis, a new choice was added to represent the case in which *none* (NN) of the versions nor new code was used in the merge resolution performed by the developer. In such a case, the solution is empty.

Finally, as part of our analysis we want to quantify the difficulty that a particular kind of conflict poses. For this, the size of the code in the conflicting chunks is obviously important, but another indication is provided by the choices a developer makes in resolving a kind of conflict. A kind of conflict that is always resolved with *new code* (i.e., type *NC*) is presumably more difficult than a kind of conflict that can be resolved by always choosing *version 1* or *version 2*, for example. To provide a (relatively crude, but as we shall see effective) basis for comparison, we distinguish between *cheap chunks* (conflicting chunks that are resolved through choosing *version 1*, *version 2*, or *concatenation*) and *expensive chunks* (resolved through *combination* or *new code*). The motivation is that the latter two types of conflict require a developer to engage in depth with the conflicting chunks, and generally involve more time and effort.

3.3 ANALYSES

The focus of our work is on understanding merge conflicts in detail, together with the resolutions that developers frequently use to address these conflicts. The more we understand about failed merges, more opportunities might be found to design new merge tools that leverage on the lessons learned. To build this understanding, we identified seven incremental

analyses that we manually performed on the conflicts in the history of five different Java projects (see below for the projects we selected). Afterwards, the same analyses were automatically performed in a much larger set of projects, adding one new analysis (RQ8). Each analysis adds detail to the previous analyses either by performing a more fine-grained study of earlier results or by correlating findings from previous analyses with factors that could explain them. We describe each analysis briefly here, and detail them further in Section 3.5 when we discuss our results.

RQ1. What is the distribution in number of conflicting chunks for merge failures?

The number of conflicting chunks involved in a merge failure influences the difficulty of resolving the overall conflict. The more chunks, the more places developers need to examine and crosscheck while resolving a conflict. This is equally true for tools: it is likely to be more difficult to develop effective merge tools that can consider the full complexity of a multitude of chunks being in conflict as compared to just one or a few. Our first analysis, then, focuses on understanding the number of conflicting chunks that appear in merge failures.

RQ2. What is the distribution in size of conflicting chunks, as measured in lines of code (LOC)?

When it comes to the anticipated difficulty of resolving a particular merge conflict, complementary to the number of conflicting chunks is the size of those conflicting chunks: the higher the LOC, the more code has to be inspected and worked with to resolve that conflicting chunk – whether by a developer or a merge tool. Hence, our second analysis focuses on assessing the distribution of LOC in conflicting chunks.

RQ3. What is the distribution in language constructs involved in conflicting chunks?

As already alluded to, the kind of conflict can similarly influence the difficulty of resolving it. As one example, suppose the only language construct involved in a conflicting chunk is *import*. Concatenating the two *import* statements, like semi-structured merge (Apel et al., 2011) does, will likely resolve the conflict in most cases, and is an operation that could easily be performed by a tool (perhaps with some checks whether both *imports* are truly needed in the final merged result after all conflicting chunks have been processed). Whether or not the creation of such heuristics is a viable direction for new merge tools depends on the kind of conflicts and the frequency with which they appear. Our third analysis, then, focuses on uncovering the language construct involved in conflicting chunks.

RQ4. What, if any, patterns exist in the language constructs of failed merges involving multiple conflicting chunks?

When a failed merge involves multiple conflicting chunks, it may be that certain dependencies exist that are indicative of certain heuristics that could help resolve the conflict. For instance, if one conflicting chunk involves an *import* and another conflicting chunk a *method invocation*, resolving the *method invocation* conflict first could well help in resolving the *import* conflict. Understanding which patterns exist across chunks is the focus of our fourth analysis.

RQ5. What is the distribution of developer decisions?

From all the different decisions that developers can make, only one demands addition of *new code* (*NC*). If the majority of decisions does not involve new code, a sensible first step forward is to develop a new merge tool that presents five options (*version 1*, *version 2*, *concatenation*, *combination*, and *none*) and assists the developer in choosing from among them (and, in the case of *combination*, in selecting and organizing the desired lines of code from each of the two versions). Conversely, if most conflicting chunks are *NC*, this approach would not help as much. Understanding the resolutions that developers choose, then, should give us a first indication of the space of possible merge tools that should be designed next, and is the focus of our fifth analysis.

RQ6. What is the distribution in difficulty level of kinds of conflicting chunks?

While the first five research questions examine properties of conflicting chunks, our next analysis assesses the chunks through the lens of the kinds of conflicts they contain and what the chosen developer decisions reveal about their respective difficulty levels. This analysis provides a first look at the distribution of developer decisions as related to difficulty levels. It might be, for instance, that nearly all kinds of conflicting chunks require *NC* developer decisions, or that very few ever involve *CC* decisions. This, in turn, begins to provide preliminary evidence as to what kind of tool support may be required.

RQ7. What, if any, patterns exist between the language constructs of conflicting chunks and developers' decisions?

Our seventh analysis takes a closer look at the relationship between the kinds of conflicting chunks and developer decisions by examining whether the presence of certain

language constructs or combinations thereof might explain the difficulty level of resolutions. That is, rather than examining the kind of conflict in its entirely, we look at individual and smaller combinations of language constructs to examine whether some of them can predict certain developer decisions. If this is the case, one could imagine heuristics that encapsulate these patterns in the support offered by new merge tools. It might also be that there is an absence of predictive power, in which case each failed merge needs to be treated uniquely.

RQ8. What is the correlation between the number of developers and the number of commits, merges, and failed merges?

There is some folklore concerning merge. For instance, it is accepted that the number of commits, merges, and conflicting merges in a project is proportional to the number of developers involved in the project. However, to the best of our knowledge no experimental analysis has verified this "common sense". Thus, our final analysis uses correlation to evaluate whether there are strong relations between the number of developers and the number of commits, merges, and failed merges in each project.

3.4 THE DESIGN OF THE MANUAL ANALYSES

In this section, we introduce the process that was followed to extract the information reported in the manual analyses. Section 3.4.1 presents constraints that guided the selection of the open source projects that were manually analyzed and shows information related to the usage of VCS by the developers working on these projects. Section 3.4.2 describes the steps that were manually performed to collect the information required to perform the proposed analyses, such as developer decisions and language constructs.

3.4.1 SELECTION OF OPEN SOURCE PROJECTS

We engaged in a careful process of selecting suitable open source projects as the basis for our analysis. After inspecting and considering a number of systems to build an initial understanding of merge conflicts, we articulated the following constraints for selecting potential systems: (1) they all should have the same main programming language (in our case we chose Java, as it is one of the most popular programming languages (Cass, 2015)), since this would allow us to study similarities and differences among systems, (2) the system had to have more than 1,000 commits, in order to identify systems that had seen serious development effort, and (3) it had to have more than 10 developers, in order to increase our chances of identifying meaningful conflicts.

Out of the resulting set of systems, we selected a corpus of five projects in the manual analyses: *ANTLR4*, a parser for programming languages; *Lombok*, a project that helps in writing succinct boilerplate code through annotations; *MCT*, a NASA-developed real-time monitoring platform; *Twitter4J*, an API for accessing Twitter; and *Voldemort*, a distributed key-value storage system. Table 1 presents key statistics related to the history of these projects: the total number of commits over the history of the project (#Commits); the number of merges (#Merges); the number of developers who performed at least one commit (#Dvl); the number of failed merges (#FM); and the total number of conflicting chunks (#CC).

Table 1. Key statistics of the selected projects.

Project	#Commits	#Merges	#Dvl	#FM	#CC
ANTLR4	2,870	352	14	27	86
Lombok	1,636	106	13	22	69
MCT	1,013	206	16	17	52
Twitter4J	1,938	211	84	38	98
Voldemort	4,275	480	54	65	401
Total	11,732	1,355	181	169	706
Java total	-	-	-	147	616

We manually examined all 1,355 merges performed in these projects, out of which 169 (12.5% – in line with the 10 - 20% range reported elsewhere (Mens, 2002; Brun et al., 2011b)) were in conflict, with a total of 706 conflicting chunks. Because all of these projects had some auxiliary files not written in Java, some merges pertained to those auxiliary files. We performed our analysis on the 147 failed merges and 616 conflicting chunks that were in the Java files only.

3.4.2 DATA COLLECTION

To construct Table 1 and perform the detailed analysis presented in Section 3.6, we manually examined all the Java merges of all five projects, recording the language constructs involved in the conflicting chunks and the developer decision that was applied to each of the conflicting chunks. Although this was a time-consuming task, we felt it was preferred to do a manual analysis rather than an automated analysis, for several reasons. First, we felt it would help us understand the issues in much greater detail, shaping future automated analyses, but not putting the proverbial “cart before the horse”. Second, observations from our manual inspections fueled the formulation of the analyses we described in Section 3.3, as engaging with the conflicts at a very detailed level helped us begin to understand what sorts of

phenomena were present in the data we were collecting. Finally, performing a manual analysis helped us identify particularly enlightening examples, with some of them presented in this Thesis. Overall, we are able to provide a more qualitative and deeper exploration due to the experiences gained in manual extraction and analysis of the data.

To extract the data, we replayed each merge that belongs to a specific project by performing the following steps: (1) identifying the parents of the merged version, (2) redoing the merge of the parents, and (3) observing if the merge failed or not. For each failed merge, we analyzed each conflicting chunk (as captured in the format shown in Figure 10) to record the language constructs that were part of it, cataloguing if the language construct belonged to *version 1* or *version 2*. By doing this manually, we were able to add interpretation where it was needed, for instance by ignoring whitespace as a source of conflicts. More importantly, however, we could address issues such as the one illustrated by Figure 11. From the merge result (as shown at the bottom with *new code* added by a developer), we can deduce that the situation was the one in which a pair of developers had each added a method, only one of which was needed, but in somewhat adjusted form. Thus, the conflict does not concern *method signature*, *return statement*, *method invocation*, and *variable*, as a naïve approach would have documented, but just two *method declarations*: *adjustSeekIndex* and *reset*. Thus, this conflict was recorded as a *method declaration* conflict.

```
@Override
```

version 1	*version 2*

`protected int adjustSeekIndex(int i) {`
`  return skipOffTokenChannels(i);`
	`public void reset() {`
	`  super.reset();`
	`  p = nextTokenOnChannel(p, channel);`

```
        }
```

merge resolution

```
@Override
protected int adjustSeekIndex(int i) {
  return nextTokenOnChannel(i, channel);
}
```

Figure 11. Conflicting chunk of merge 18f535 and its *new code* resolution in ANTLR4.

This situation arose multiple times, in various forms involving different language constructs. In each of the cases it turned out that the outermost language construct (typically a language declaration, but sometimes also a *for* or an *if*) was the governing concern regarding the conflict and its resolution. We therefore documented the outermost language constructs involved in each of the conflicting chunks we found.

We also examined each conflicting chunk and its subsequent merge result (as checked in to the repository) to understand how the developer ultimately resolved each conflicting chunk that originally resulted in the failed merge. By studying the merged source code and the original conflicting chunk, we determined whether the chosen solution was one of the

versions (*V1* or *V2*), a *concatenation* of the versions *(CC,* consisting of *version 1* followed by *version 2* or vice versa), the *combination* of code from both versions without newly written code (*CB*), or a merge involving additional *new code* (*NC*). For example, Figure 11 shows a resolution in which the developer chose only a few lines across both versions, and changed a line from "*return skipOffTokenChannels(i);*" to "*return nextTokenOnChannel(i, channel);*". Thus, the developer decision was *NC*.

It is worth mentioning that we extracted developers' decisions from the commit immediately after the failed merge. However, to account for situations in which developers postponed the resolution (they, for instance, might comment on a conflict and delegate its resolution to another developer, or they could simply override the other developer's changes for now to resolve the conflict at a later time), we did not only analyze the immediate next commit, but also any commits up to one month after the original commit. If such a later commit changed the code of the originally conflicting merge, we examined whether it represented a postponed resolution by checking whether any code from the original merge conflict was now included. This occurred in few cases; six times to be precise.

3.5 THE DESIGN OF THE AUTOMATIC ANALYSES

To determine whether the answers found by the manual analysis for the questions proposed in Section 3.3 were due to specificities of the selected projects or could be generalized to, in the least, the larger population of open source Java projects, we needed information from a large number of software projects. We chose to collect this data from GitHub[3], which allows creating free repositories in the cloud. Free repositories are usually public and data about the software projects residing in them can be collected both by cloning the repositories or using an API.

The process of collecting information about merges comprises four steps: (1) extract GitHub metadata, (2) select projects, (3) analyze project repositories, and (4) analyze data extracted from repositories.

Cloning repositories can be very costly due to the size of the files that participate in the history of the project. In order to reduce the overall computational cost of our analysis, we first used the API provided by GitHub that enables us to collect limited project metadata and select a sample of relevant projects based on this information. Some metadata we extracted in this first step are the last date/time the project was updated, the number of developers per

[3] https://github.com

project, and the size of source code (measured in bytes) written in each language used in the project. As a result of this step, we extracted metadata from 1,997,541 projects.

Next, we selected all active Java projects from this sample. A project was classified as active if it was updated at least once since January, 2015. A project was classified as Java if the amount of code written in Java was greater than the amount of code written in any other programming languages. The selected projects have on average 92.79% of Java source code with standard deviation of 13.29%, showing that the projects are mainly coded in Java. After filtering the projects in our sample according to these criteria, we selected 13,576 projects.

Then, we cloned the repositories of these projects and extracted fine-grained information about merge cases. Figure 12 shows an overview of the steps performed to obtain such information. After cloning the repository, we selected all the merge commits and classified them as failed or successful merge. To do so, we replayed each merge and checked if conflicts arose, as in the manual analyses, by automatically executing the following steps: (1) identifying the parents of the merged version, (2) redoing the merge of the parents, and (3) observing if the merge failed or not. When the merge does not fail, we classify it as a non-conflicting merge (*i.e.*, the cases on which merges are resolved automatically) or a fast-forward (*i.e.*, the cases on which the merge just updates the head pointer of the mainline) and the analysis of this merge is concluded. Otherwise, we collect complementary information about the failed merges.

Then, we selected the projects that have at least one failed merge and are not forks of other projects in our dataset. We took this decision because we are interested in projects that have failed merges and we do not intend to count the same merge twice or more, as forks may share part of the history. Thus, selecting just the main repositories, we avoid taking the replicated commits that are copied when a developer forks a given project. After applying this criterion, our sample was reduced to 3,796 projects. Finally, we discarded projects that do not have conflicts in Java files. After applying this criterion, we selected a corpus composed by 2,731 projects to perform the automatic analysis and answer the research questions. These projects have together 25,328 failed merges and 175,805 conflicting chunks.

As explained in Section 2.4.1, failed merges have one or more conflicting chunks distributed in the source code files. Thus, when a failed merge is found, the analysis identifies all files containing conflicting chunks. To identify these files, we ran a Git command that lists all files in conflict and parsed these files to find the conflicting chunks based on the source code marks that delimit the conflict. For each conflicting chunk, we identified how developers resolved it and which language constructs were involved in its source code. The developer

decision is identified by comparing the final decision of the developer assigned to resolve the conflict with one of the decisions listed in Section 3.2. This comparison was not sensitive to formatting characters such as blank spaces, which would end up in a *new code* decision whenever a developer changes just indentation, for instance. If none of these reflect the final resolution, a *new code* resolution is identified. To extract the language constructs, we parsed the files of *version* 1 and *version* 2 and collected the outermost constructs inside the area of these versions, using a similar procedure to that used in the manual analysis and discussed in Section 3.4.2. At the end of the analysis of each commit, we stored the following data: the merge status (*i.e.*, conflicting, non-conflicting, or fast-forward), the files involved in the conflict, the conflicting chunks, and the developer decision for each chunk.

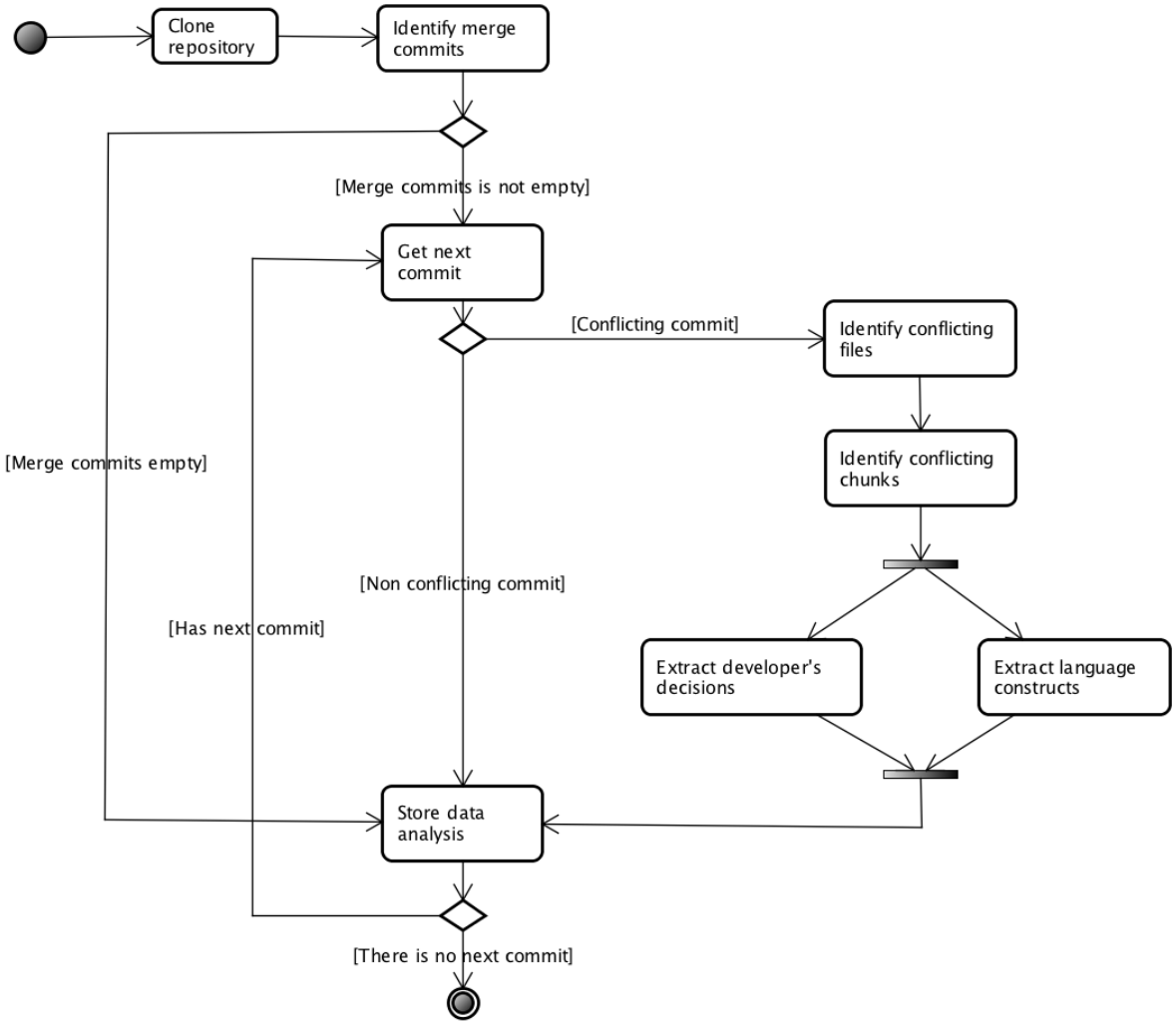

Figure 12. Repository analysis activities.

As previously discussed, our focus is to analyze projects coded in Java. However, our current implementation allows us to extract some data like LOC, developer decisions, and

number of conflicting chunks, independently of the programming language of the project. In other words, the research questions RQ1, RQ2, RQ5, and RQ8, which are not related to language constructs, can be answered for any project under Git. On the other hand, the research questions RQ3, RQ4, RQ6, and RQ7 can be answered only for Java projects. The current version of our implementation can collect language constructs of Java projects. Consequently, new parsers must be implemented to analyze projects coded in other programming languages.

It is also important to highlight that we do not have problems with crossing method boundaries during language constructs extraction. Some conflicting chunks do not have well-formed source code, not allowing the parser to extract the AST directly from the chunk source code. For example, only one part of *method declaration* may be comprised in a chunk. To deal with this problem, we parse the source code of the versions that generated the merge and then we calculate the relative area of the source code in *version 1* and *2* to collect the language constructs that are inside this area.

Finally, we did not measure the time spent to collect the information from the projects we analyzed. However, the process described in Figure 12 for the 13,576 projects took about 4 months. This time is mainly related to collecting syntactical information, as most part of the computational cost is concentrated in the AST extraction.

3.6 RESULTS

This section presents our results, organized per each of the analyses outlined in Section 3.3. Aiming at facilitating the identification of the results related to the manual and automatic analyses, we use **manual analyses** to mark the begin of the manual results and **automatic analyses** to highlight the begin of the automatic results.

RQ1. What is the distribution in number of conflicting chunks for merge failures?

Considering the **manual analyses,** Figure 13 shows the number of failed merges involving different numbers of conflicting chunks for the five projects of our corpus. Most failed merges involved just 4 or fewer conflicting chunks (111 out of 147, 76%) and more than half involved merely 1 or 2 conflicting chunks (87 merges, 59%). Such low numbers provide initial hope that opportunities may exist for newly designed merge tools. This, of course, depends on the nature of the conflicting chunks, as fewer chunks do not necessarily mean less complicated resolutions. We return to this topic in subsequent analyses.

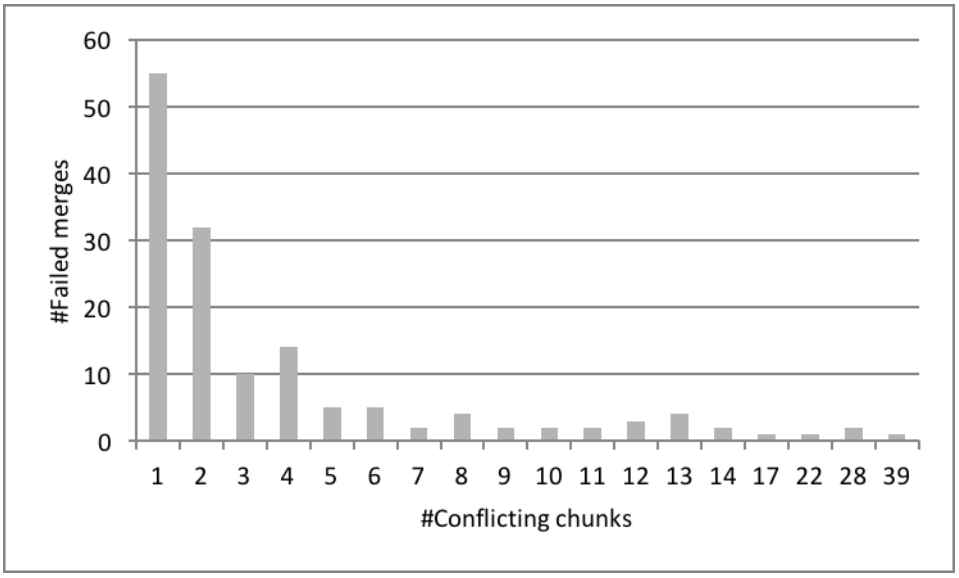

Figure 13. Histogram of conflicting chunks.

Four failed merges involved more than 20 conflicting chunks each, with 1 of them involving as many as 39. Manually inspecting these four cases revealed that they are very complex merges: no simple heuristic would be able to address them. Instead, all four involved *NC* as the resolution strategy for at least some of the conflicting chunks, combining code from both versions with additional *new code*.

Finally, we observe that a non-trivial number of failing merges exist with a number of conflicting chunks from 5 to 17. Overall, a quite even distribution exists, with none of the projects we studied standing out: conflicts from each of the five projects appeared across the spectrum. We manually examined the conflicts and, as we shall discuss later in more detail, the conflicts ranged from simple to resolve to being very difficult to resolve, with some amenable to automation and some not.

Considering the **automatic analyses,** Figure 14 shows that the number of failed merges having a given number of conflicting chunks in the larger sample of projects in our corpus obeys almost the same distribution of Figure 13. To improve visualization, we limited the chart to failed merges having up to 11 chunks. As well as in the manual analysis, most failed merges have few conflicting chunks: 40% of the failed merges have a single conflicting chunk and 90% of these merges have up to 10 chunks. The remaining 10% of the failed merges have up to 10,315 chunks. However, a deeper analysis considering the difficulty of each failed merge, as presented in the manual analysis, was not possible since this analysis is subjective and we increased our corpus from 147 to 25,328 failed merges, being hard to analyze by hand.

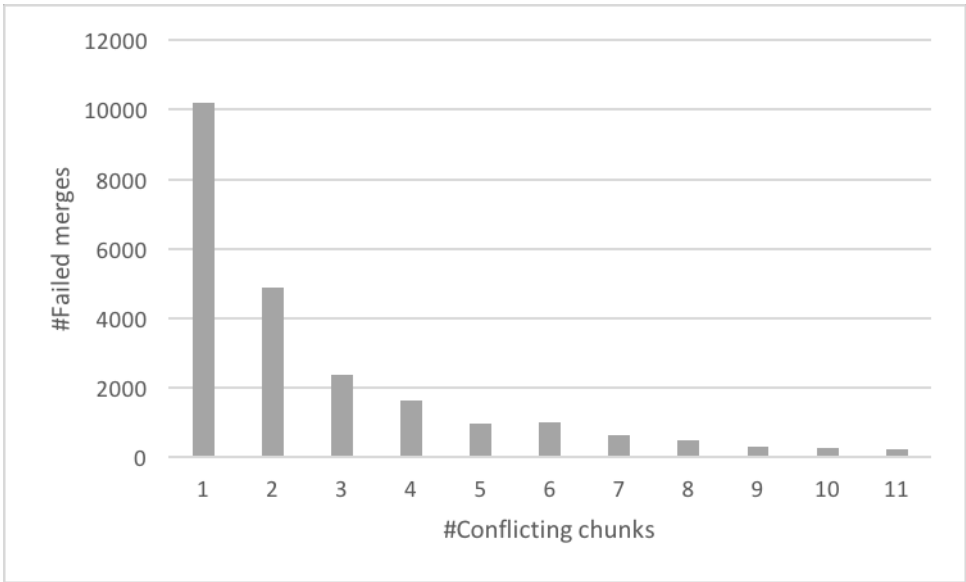

Figure 14. Distribution of conflicting chunks by failed merges.

The results of manual and automatic analyses give the same message: (1) most failed merges comprise few conflicting chunks and (2) around 60% of the failed merges have more than one chunk. The first conclusion can motivate the creation of automatic approaches that deal with only one chunk, which can decrease the challenges when compared to the ones with more chunks.

RQ2. What is the distribution in size of conflicting chunks, as measured in lines of code (LOC)?

Considering the **manual analyses,** Figure 15 shows the relationship between the number of lines of code in *version 1* and the number of lines of code in *version 2* for each conflicting chunk of the five projects comprising our corpus (with the bottom left rectangle of the top figure blown-up in the bottom figure). First, we note that the number of lines of code that are in conflict in *version 1* varies from 0 to 313, and in *version 2* from 0 to 270. 95.94% of the conflicting chunks, however, have less than 50 LOC in both *version 1* and *version 2*. This means that developers have at most 100 LOC in total to examine when they resolve these conflicting chunks. When we focus on the chunks with at most 5 LOC in each version, this still accounts for 51.46% of the cases, meaning that in over half of the conflicting chunks, developers have to look at merely 10 LOC total. Note that in a number of cases one of the versions in the conflicting chunk has zero LOC; this is the result of a change that in its entirety consists of the removal of lines of code.

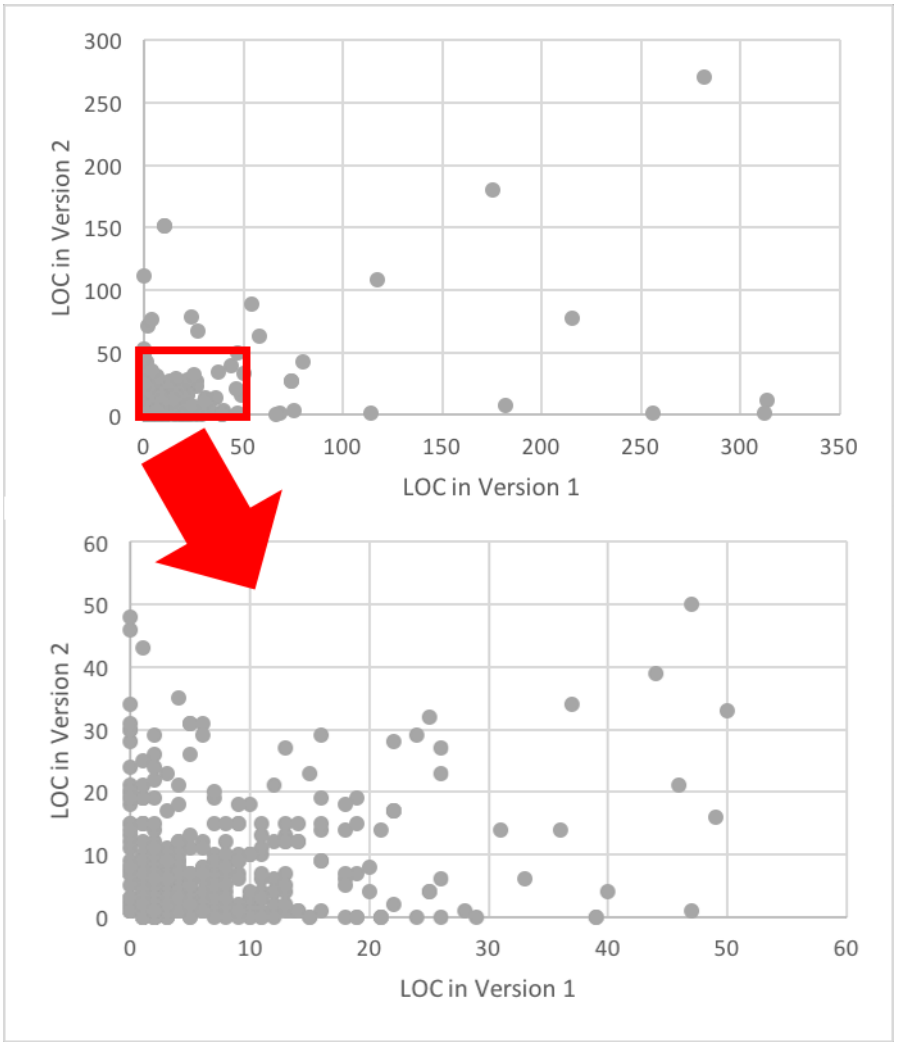

Figure 15. Relationship between LOC in version 1 and 2.

Our second observation is that it is relatively rare for the two versions of a conflicting chunk to both having high numbers of LOC (see Table 2). It is more common for 1 to be large (*i.e.*, having more than 50 LOC), and the other to be relatively small. Just 6 out of 616 chunks in our sample have more than 50 LOC in both versions. Perhaps unsurprisingly, manual examination of these 'unbalanced' conflicting chunks shows that the version with the high number of LOC often involves language constructs that encompass other constructs, such as *class declaration* and *if statement*. These kinds of statements often cover larger, wholesale changes to a piece of code.

Table 2. Distribution of chunks by LOC interval.

Project	LOCV1 ≤ 50 LOCV2 ≤ 50	LOCV1 ≤ 50 LOCV2 > 50	LOCV1 > 50 LOCV2 ≤ 50	LOCV1 > 50 LOCV2 > 50
ANTLR4	73	0	0	1
Lombok	38	3	3	2
MCT	57	0	1	1
Twitter4J	88	2	2	1
Voldemort	335	3	4	1
Total	591	8	10	6

Table 3 provides an alternative view, examining the median (Med), average (Avg), and standard deviation (Std) of the conflicting changes in *version 1* and *version 2*, per project. In four of the five projects, *version 1* represents the larger average change, except for MCT, in which *version 2* is on average larger. Overall, we observe that the average is consistently larger than the median, and the standard deviation is far larger than the average. This implies that some extremely large and rare (thus, insensitive to the median) chunks are found for all projects, as observed in Table 2.

Table 3. Average Size of Conflicting Chunks.

Project	Version 1			Version 2		
	Med	Avg	Std	Med	Avg	Std
ANTLR4	3	6.20	9.56	2	5.97	11.68
Lombok	4.5	6.73	17.26	7	4.85	9.31
MCT	2	20.57	50.18	2	26.72	54.28
Twitter4J	3	14.58	39.99	4	9.08	13.52
Voldemort	2	7.77	25.41	3	7.40	12.38

MCT also stands out in terms of the average size of its conflicting chunks: they are much larger than the other systems. This is perhaps not surprising, given the high average number of lines of code added per check-in for MCT: 467.74. Still, as compared to the 4 other systems (ANTLR4 427.74; Twitter4J 205.79; Voldemort 197.94; Lombok 104.97), this does not seem to entirely explain the difference, especially with respect to ANTLR4, which has the smallest average size per conflicting chunk yet the second-largest average number of added lines of code per check-in.

Considering the **automatic analyses**, Figure 16 shows the distribution of the number of LOC in conflict in *version* 1 (horizontal axis) and in *version* 2 (vertical axis) for the

conflicting chunks identified for the sample of projects. We observe that there exist huge conflicting chunks (with up to 26 thousand LOC in conflict, 12 thousand LOC in *version* 1 and 14 thousand LOC in *version* 2), but these are rare. Moreover, the chart in Figure 16.a shows that the number of conflicting chunks having more than 2,000 LOC in each version represents only 0.05% of the conflicting chunks. To have a better view of this distribution, we focus on smaller conflicts in other charts.

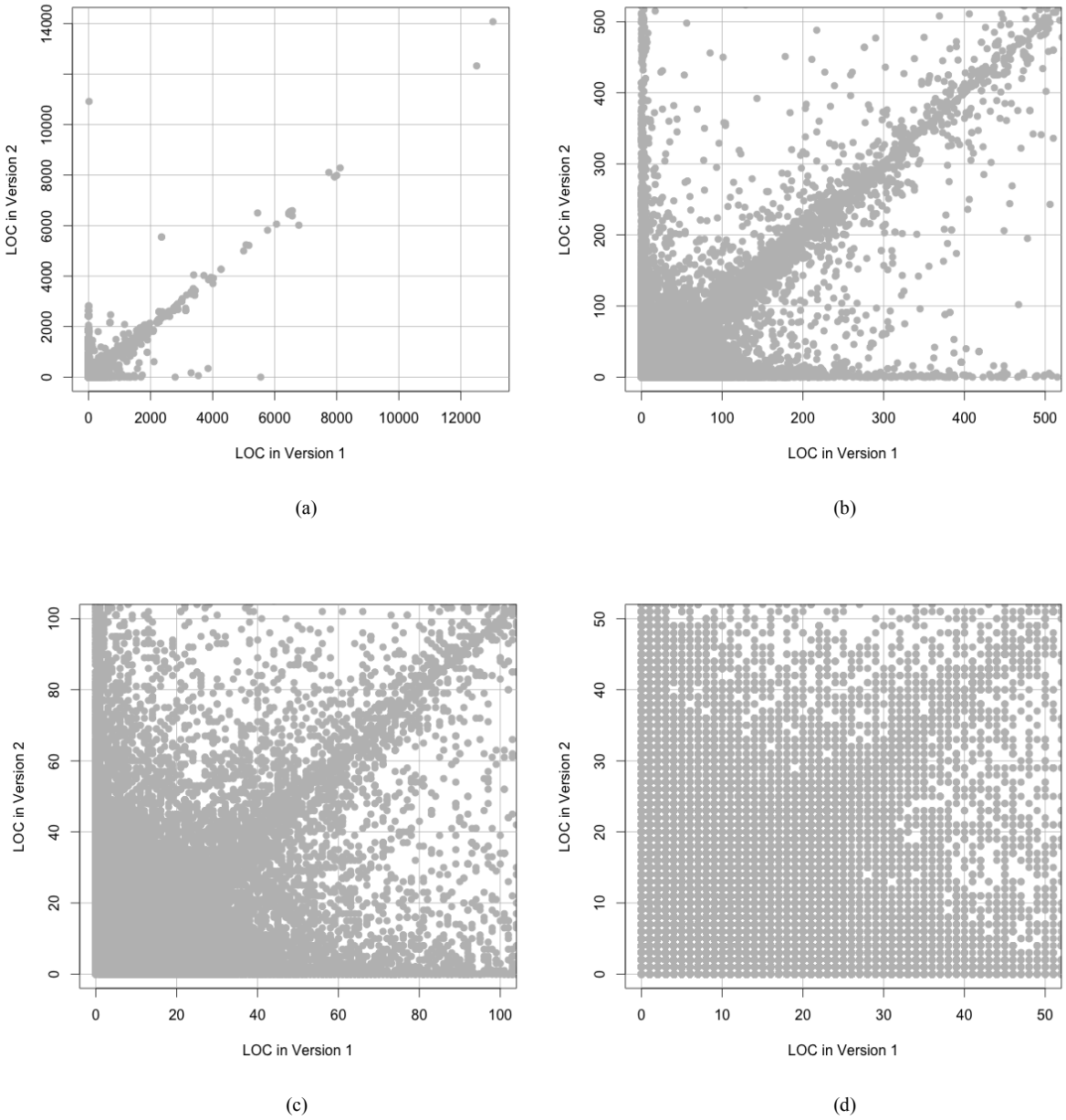

Figure 16. Distribution of LOC in the conflicting chunks before (a) and after filtering out chunks with more than 500 LOC (b), 100 LOC (c), and 50 LOC (d) in each version.

Figure 16.b, Figure 16.c, and Figure 16.d show more details about the distribution of LOC in conflicting chunks, which progressively zoom into the regions close to the origin of

the Cartesian-plane. Chart (b) shows the conflicting chunks having up to 500 LOC in either *version* 1 or 2. Chart (c) focus the region with up to 100 LOC in each version, and Chart (d) shows conflicting chunks with up to 50 LOC in each version. We observe that 94.20% of the conflicting chunks have up to 50 LOC in each version in conflict (165,616 out of 175,805). Moreover, 68% of the chunks have up to 10 LOC and slightly more than half the chunks (50.17%, or 88,200 chunks) have 5 or less LOC in conflict. Therefore, the results observed in the five projects used in the manual analysis can also be found in the larger sample used for the automatic analysis.

We also observed that it is relatively rare for the two versions of a conflicting chunk to both have high numbers of LOC. Out of the 175,805 chunks in our sample, 165,616 (94.20%) have less than 50 LOC in either version, 6,042 (3.44%) have more than 50 LOC in 1 version and less than 50 LOC in the other, and only 4,147 (2.36%) chunks have more than 50 LOC in both versions. The analysis also shows a certain concentration of projects having complex conflicting chunks: 1,546 out of 2,731 projects (56.61%) have only chunks with less than 50 LOC in both versions, while only 700 (25.63%) projects have at least 1 chunk with more than 50 LOC in both versions. Only 95 (3.47%) projects have more than 10 chunks with more than 50 LOC in each version and only 4 projects have more than a hundred of such chunks. Wro4J[4] has the highest number of the most complex cases: 179 (8.61%) chunks with more than 50 LOC in each version, though 2 other projects (Axis2/java[5] and StatET[6]) have up to 38.57% and 42.37% of their chunks as complex ones. Such complex chunks account for on average 5% of the chunks in our projects, while the simplest chunks represent about 91.33% of the chunks of any given project.

The average size of *version 1* over our sample projects (average across projects of the average within project) is 19.5 LOC, while *version 2* has an average of 27.6 LOC. The average standard deviations (average across projects of the standard deviation within project) are 20.6 and 28.6 LOC, respectively, for *version* 1 and *version* 2. The median size of *version 1* and *version 2* (median across projects of the medians within projects) are 2.0 and 2.5 LOC. These numbers, as well as the point clouds extending in the diagonal and along the axis in Figure 16.b, show that some large conflicting chunks drag the mean of the chunks upward and increase the standard deviation despite the concentration of most chunks near the origin of the chart. Nevertheless, these conflicting chunks are not frequent enough to affect the median.

[4] https://github.com/wro4j/wro4j
[5] https://github.com/apache/axis2-java
[6] https://github.com/walware/statet

The results of manual and automatic analyses present the same conclusion: (1) more than 95% of the chunks have up to 50 LOC in each version and (2) it is relatively rare to have more than 50 LOC in both versions. This is an interesting result since developers in most of the chunks should look at most 100 LOC to resolve their conflicts, which is also interesting for automatic approaches because the number of possible combinations of LOC decreases.

RQ3. What is the distribution in language constructs involved in conflicting chunks?

Considering the **manual analysis**, we first examined the number of language constructs present in conflicting chunks, leading to the results shown in Table 4. We observe that almost all conflicting chunks consist of up to 4 language constructs (594 out of the 616 chunks, 96%). A negative correlation exists between the number of language constructs (first column) and the number of conflicting chunks (remaining columns), varying from -0.85 to -0.98 across the 5 projects (Spearman rank-order correlation, which we selected because the Shapiro-Wilk test indicated non-normality in our data with a *p-value* smaller than 0.001).

Table 4. Number of Conflicting Chunks per Number of Language Constructs.

# Language constructs	Number of conflicting chunks					
	ANTLR4	**MCT**	**Lombok**	**Voldemort**	**Twitter4J**	**Total**
1	30 (41%)	28 (61%)	29 (49%)	183 (53%)	50 (54%)	320 (52%)
2	25 (34%)	6 (14%)	21 (36%)	95 (28%)	23 (25%)	170 (28%)
3	10 (14%)	7 (15%)	7 (12%)	32 (9%)	10 (11%)	66 (11%)
4	6 (8%)	2 (4%)	0 (0%)	24 (7%)	6 (6%)	38 (6.2%)
5	1 (1%)	1(2%)	1 (1.5%)	9 (2.7%)	3 (3%)	15 (2.4%)
6	1 (1%)	2 (4%)	1 (1.5%)	0 (0%)	0 (0%)	4 (0.6%)
7	1 (1%)	0 (0%)	0 (0%)	0 (0%)	1 (1%)	2 (0.3%)
8	0 (0%)	0 (0%)	0 (0%)	1 (0.3%)	0 (0%)	1 (0.2%)

The low number of language constructs involved is not surprising, given our decision to only record the outermost language constructs when nesting is present (as described in Section 3.4.2). Because the code in either version of a single conflicting chunk is contiguous, it is rare for there to be lots of 'top level' language constructs. The table, thus, should be interpreted as recording the distribution of how many language constructs are the primary reason for a conflict (e.g., a *for* loop has been added, an *if* statement has been added, a *method declaration* has been added).

Given this, and examining the table further, we see that 52% of conflicting chunks involve a single language construct and 80% just 1 or 2. Because of the limited set of language constructs combinations that occur frequently (see below), we believe this suggests that an exploration of specialized merge techniques that deal with few language constructs is an important direction forward. For instance, consider the conflict in Figure 17, taken from the Twitter4J project. A traditional, general merge technique cannot resolve this conflict, because the same area was edited in parallel. A tailored merge technique that understands that the two *if* statements each address a different condition could suggest to simply concatenate the two, which is indeed the resolution that the developer chose.

```
}
```

version 1	*version 2*

`if (!json.isNull("lang")) {`
`  lang = getUnescapedString("lang", json);`
	`if (!json.isNull("scopes")) {`
	`  JSONObject s… = json.getJSONObject("scopes");`
	`  if (!s….isNull("place ids")) {`
	`  JSONArray p… = s….getJSONArray("place ids");`
	`  int len = p….length();`
	`  String[] placeIds = new String[len];`
	`  for (int i = 0; i < len; i++) {`
	`  placeIds[i] = p….getString(i);`
	`  }`
	`  scopes = new ScopesImpl(placeIds);`
	`}`

```
}
```

Figure 17. Conflicting chunk from merge 3a3869 of project Twitter4J.

We analyzed which language constructs occurred most frequently in conflicting chunks, and juxtaposed this with the frequency of the language constructs across the whole code base (calculated based on the last version of each project). Table 5 presents the results, when focusing on a single language construct at a time. The symbol 🡻 is used when the percentage of the language construct in the conflicting chunk is lower than across the whole code base, with the symbol 🡹 signaling the opposite. Because a language construct can occur more than once in a conflicting chunk, the total number of language constructs is higher than the total number of conflicting chunks we reported in Table 1. The seven language constructs listed in Table 5 represent nearly 80% of the language constructs in conflicting chunks.

Table 5. Most frequent language constructs.

Language construct	Number of occurrences		
	Conflicting chunks	Source code	
Method invocation	252 (22.32%)	250,469 (39.26%)	↓
Variable	208 (18.42%)	104,410 (16.37%)	↑
Method declaration	118 (10.45%)	40,171 (6.30%)	↑
Commentary	112 (9.92%)	44,983 (7.05%)	↑
If statement	97 (8.59%)	21,646 (3.39%)	↑
Import	64 (5.67%)	33,408 (5.24%)	↑
Method signature	62 (5.49%)	40,171 (6.30%)	↓

The 3 most frequently involved language constructs are *method invocation*, *variable*, and *method declaration*, which together account for over 50% of the language constructs in conflicting chunks. This, in and of itself, is interesting: it implies that larger, wholesale changes are often the source of conflicts, rather than smaller, local changes. Table 6 provides a refined view on this observation, listing the most frequent kinds of conflicts (per the definition we introduced in Section 3.2). We observe that *method invocation* is often not a standalone change, but one that has other changes surrounding it in a conflicting chunk. It appears alone a mere 63 out of 252 times, with all other occurrences being in combination with other language constructs. The same is true for *variable*: 37 individual occurrences are complemented by 58 combinations of *variable* with *method invocation*, 19 occurrences of *if statement*, *method invocation*, and *variable*, and others.

Table 6. Most frequent kinds of conflicts.

Kind of conflict	Occurrences
Method invocation	63
Import	60
Method invocation, variable	58
Method declaration	57
Variable	37
If statement	20
Method signature	19
If statement, method invocation, variable	19

An interesting example of a particular kind of conflict is *method invocation*, the most frequent language construct in Table 5, with *variable*, the second most frequent. Examining this kind of conflict, we found that it often concerns a *method* that is called to initialize or

assign a value to a *variable* (one version changed the method being called; the other the variable name). The last line of Table 6 represents an 'expansion' of this combination: this kind of conflict frequently captures the initialization of a variable depending on the condition of an *if statement* (Figure 18 shows an example). Although this combination of language constructs may represent other cases but conditional initialization, most of its occurrences represent an opportunity to create heuristics to address and resolve these cases.

```
assigns.add(new SingleNameReference(field.name,fieldPos));
```

version 1	*version 2*

`args.add(new Argument(…, Modifier.FINAL));`	`Argument argument = new Argument(…);`
	`Annotation[] copiedAnnot… = copyAnnotations(…);`
	`if (copiedAnnot….length != 0)`
	`  argument.annotations = copiedAnnot…;`
	`args.add(new Argument(…, 0));`

```
}
```

Figure 18. Chunk from Lombok´s merge that generated commit 4e152f.

Excluding *import*, which by virtue of where it must appear in the source code nearly always occurs alone in a conflicting chunk, other language constructs occur as part of kind of conflicts that involve multiple language constructs over 50% of the time. While this is on one hand encouraging, in that it means different kinds of conflicts exist that can perhaps be addressed through techniques tailored for each, the payoff of such an approach diminishes relatively quickly. For instance, even the *if statement*, *method invocation*, *variable* kind of conflict appears only 19 times (out of 616 total conflicting chunks, this is just 3%). Still, amortized across the many projects under development today, 3% of conflicting chunks represents a non-trivial development effort that can be eased.

We also mined for association rules involving language constructs occurring together in conflicting chunks. Table 7 shows these association rules, presented in the form of "A → B" and measured in terms of support (s%), confidence (c%), and lift (L). This states that a pattern of having language constructs "A" along with language constructs "B" in the same conflicting chunk happened in s% of all 616 chunks. Additionally, out of all the conflicting chunks that have "A", c% of them also have "B". Finally, the fact of having "A" increased the frequency of having "B" by L. The association rules presented in Table 7 have minimum absolute support of 12 (2%) occurrences and at least 50% confidence. Thus, for all the rules, "A" and "B" occur together in at least 12 conflicting chunks and in at least 50% of the chunks that "A" occurs, "B" also occurs.

The association rule *if statement, variable* → *method invocation* has 80% confidence, meaning that 80% of the conflicting chunks that have *if statement* and *variable* also have *method invocation*. In addition, the lift for this rule is 1.96, which means that the occurrence

of *if statement* and *variable* increases the probability of *method invocation* in the same chunk by 96%. On the other hand, the association rule *variable* → *method invocation* has 63% confidence, which means that 63% of the conflicting chunks that have *variable* also have *method invocation*. This decrement of confidence compared to the more specific rule indicates that *if statement* has a strong influence on using *method invocation* to set *variable* values. This pattern is confirmed considering lift values: while the rule *if statement*, *variable* → *method invocation* has lift equal to 1.96, the rule *variable* → *method invocation* has lower lift (1.54).

Note that the direction of association rules matters. For instance, while *variable* → *method invocation* has 63% confidence, *method invocation* → *variable* only has 52% confidence. This happens because, as we already discussed, *variables* often use *method invocations* to initialize their values, as in the example shown in Figure 18.

Table 7. Relation among language constructs that belong to conflicting chunks.

Association rule	Sup.	Con.	Lift
Annotation → method declaration	3%	64%	3.34
For statement → variable	2%	67%	1.98
If statement, variable → method invocation	6%	80%	1.96
Commentary, method invocation → variable	4%	66%	1.95
Commentary, if statement → variable	2%	63%	1.87
If statement, method invocation → variable	6%	62%	1.84
Method invocation, method signature → variable	2%	58%	1.72
Method signature, variable → method invocation	2%	70%	1.71
Try statement → method invocation	4%	69%	1.69
Return statement → method invocation	3%	69%	1.69
Try statement, variable → method invocation	2%	68%	1.66
Commentary, variable → method invocation	4%	66%	1.61
Try statement → variable	6%	53%	1.57
Commentary, if statement → method invocation	2%	63%	1.54
Variable → method invocation	21%	63%	1.54
Method invocation, try statement → variable	2%	52%	1.54
If statement → method invocation	9%	60%	1.47
Method invocation → variable	21%	52%	1.27

These association rules give further meaning to the results presented in Table 5, and particularly provide directionality to the co-occurrences shown in Table 6. As such, they may help in formulating new heuristics that explore these directionalities. Consider the conflicting

chunk presented in Figure 19, which is covered by the association rule *return statement* → *method invocation*. The conflict could not be automatically resolved, because each version uses a different *method invocation*. However, a heuristic that looks at the return type of the *method declaration* and selects the *method invocation* that matches it should be able to assist developers in the resolution of this conflict. While such a heuristic would not solve all merge conflicts involving the association rule *return statement* → *method invocation*, given the confidence of 69% and lift of 1.69, offering the heuristic as an option for a developer to apply if it is so appropriate can significantly reduce effort.

```
+ categorySlug + "/members.json");
```

version 1	*version 2*
`return fac….createUserListFromJSONArray(res);`	`return fac….createUserList(res.asJSO…(), res);`

```
}
```

Figure 19. Conflicting chunk extracted from Twitter4J project resulting of the merge 98caafc.

Finally, a rule that stands out in terms of lift is *annotation* → *method declaration*, denoting that if a conflicting chunk has *annotation*, the chances of also having *method declaration* is 3.34 times higher than when not having *annotation*. This is expected, as some *annotations* are meant to add metadata to *method declarations* and, consequently, should occur together with this language construct (Figure 20 shows an example in the form of *@override*).

```
}
```

version 1	*version 2*

`@Override`
`public short getId() {`
`  return 15;`
`}`
	`public ProxyUnreachableExce…(Throwable t) {`
	`  super(t);`
	`}`

```
}
```

Figure 20. Conflicting chunk from project Voldemort of merge 66c832.

Considering the projects selected for automatic analysis, Figure 21 shows the percentage of conflicting chunks having a given number of language constructs. For instance, 88 thousand chunks (50% of our sample) have a single language construct. Moreover, more than 90% of the chunks in our sample have up to 4 language constructs. On the other hand, about one percent of the chunks have eight or more constructs. Thus, this leads to a distribution which is close to the one presented in the manual analysis.

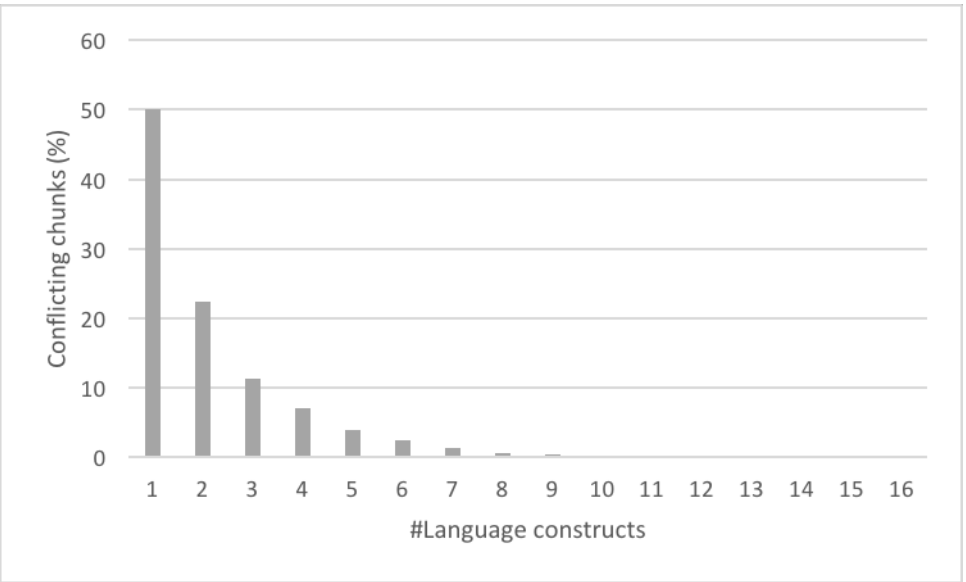

Figure 21. Distribution of the number of conflicting chunks based on the number of language constructs.

Considering the **automatic analyses,** we identified which language constructs occur most frequently in conflicting chunks, as well as their specific kind of conflict (*i.e.*, the kind of conflict that the language construct appears alone). Table 8 presents the most frequent language constructs found in conflicting chunks (first column), their counting (second column), and the counting of kinds of conflict that they appear alone (third column). The numbers in parenthesis denote the percentage of language constructs in chunks where the language constructs appear (from a total of 375,851 occurrences of language constructs in chunks) – in the second column – and the percentage of chunks where the kind of conflict with a single language construct appears (from the total of 175,805). As previously discussed, the percentage in the second column is from a total greater than the number of chunks because different language constructs may appear together in the same chunk. It is worth mentioning that the eleven constructs in Table 8 represent almost 90% of the language constructs found in conflicting chunks.

Still analyzing this table, we can observe that the seven most frequent language constructs found in the manual analysis are among the eight most frequent language constructs found in the automatic analysis. The *Attribute* construct was identified among the most frequent constructs for the automatic analysis. In the manual analysis, attributes were not distinguished from *variables* and, thus, merging their data together would place *variables* as the second most frequent construct in the automatic analysis, replicating the two topmost constructs of the manual analysis. Therefore, we confirm the findings of manual analysis concerning the frequency of certain language constructs in a bigger number of projects.

Table 8. Most frequent language constructs.

Language construct	Frequency among Language Constructs	Frequency among Kinds of Conflicts
Method invocation	75,045 (19.97%)	13,549 (7.71%)
Commentary	55,081 (14.66%)	19,447 (11.06%)
Variable	40,332 (10.73%)	1,595 (0.91%)
If statement	32,943 (8.76%)	7,570 (4.31%)
Attribute	31,176 (8.29%)	6,634 (3.77%)
Import	24,267 (6.46%)	20,538 (11.68%)
Method signature	23,177 (6.17%)	3,606 (2.05%)
Method declaration	20,500 (5.45%)	3,632 (2.07)
Annotation	12,458 (3.31%)	1,191 (0.68%)
Return statement	11,227 (2.99%)	207 (0.12%)
For statement	5,771 (1.54%)	299 (0,17%)

Table 9 shows the most frequent kinds of conflicts found by the automatic analysis. Contrasting the information of this figure with Table 6, we notice that the most frequent kinds of conflicts found while performing the manual analysis are also in this table, which confirms again the results presented in the manual analysis for a broader range of projects.

We also extracted association rules involving language constructs occurring together in conflicting chunks of the projects selected for automatic analysis. That led to a huge number of association rules, and we subsequently changed the association rules notation to represent both directions with the ⇆ symbol and save space while presenting these rules. In this case, the association rules have two confidence values: the first identifies the confidence from the left to the right (→) and the second identifies the confidence in the opposite direction (←). The different values of confidence show that the rules may be stronger in one direction than in the other. As a consequence, we can trust more in the direction of higher confidence values.

Table 9. Most frequent kinds of conflicts.

Kind of conflict	Occurrences
Import	20,538
Comment	19,447
Method invocation	13,549
Method invocation, Variable	8,865
If statement	7,570
Attribute	6,634
Attribute, Method invocation	5,317
Method declaration	3,632
Method signature	3,606
Comment, Method invocation	3,469
If statement, Method invocation, Variable	2,978
If statement, Method invocation	2,725
Comment, Method declaration	2,488
Attribute, Comment	2,110
Annotation, Method declaration	2,041

Table 10 presents the top ten association rules in terms of support. The association rules involve language construct combinations in the antecedent and consequent and, for any given rule, all kinds of conflict involving a superset of its antecedent are subject to the rule (that is, the consequent is to be found with the given probability).

Table 10. Association rules of chunks with highest support.

Association rule	Sup.	Confidence		Lift
		→	←	
If statement ⇆ Variable	9%	50%	41%	2.19
Variable ⇆ Method invocation	20%	88%	47%	2.05
Return statement ⇆ Method invocation	6%	85%	13%	2.00
If statement ⇆ Method invocation	12%	65%	29%	1.53
Method signature ⇆ Method invocation	8%	62%	19%	1.46
Attribute ⇆ Method invocation	10%	55%	23%	1.30
Variable ⇆ Comment	7%	31%	23%	0.99
Comment ⇆ If statement	6%	18%	31%	0.98
Comment ⇆ Attribute	5%	17%	3%	0.94
Method invocation ⇆ Comment	12%	27%	37%	0.88

As in the manual analysis, we can learn some lessons from this table. For instance, 88% of all chunks having *variable* and 55% of the chunks having *attributes* also have *method invocation*. This enforces the argument from Table 7 reporting that 63% of the chunks having

variable also have *method invocation*, particularly if we take into account that *variables* and *attributes* are both handled as *variable* in the manual analysis. Table 7 also shows a rule stating that 60% of the chunks having *if statement* also have *method invocation*. In the automatic analysis we observe an increment of 5% in this frequency, thus enforcing that having conflicts in an *if statement* together with a *method invocation* is a common pattern. Finally, the automatic analysis shows that having *return statements* in a chunk increases the chances of having *method invocation* by 100%. Furthermore, 85% of the chunks having a *return statement* also have *method invocation*. This rule shows an increment in support, confidence, and lift all together, showing that this is also a pattern frequently related to failed merges. We can still observe a clear direction in this rule: 85% of the conflicts in *return statement* also involve *method invocation*, but just 13% of conflicts in *method invocation* involve *return statement*.

In the end, the automatic analysis ascertains and supports major rules found while performing the manual analysis, showing at the same time that the rules are recurrent and that the set of projects selected for manual inspection can represent the larger population of open source software written in Java at least in what concerns failed merges.

The results of manual and automatic analyses have the same message relating the number of language constructs by chunks and the most frequent language constructs in conflicting chunks. Nevertheless, the results for the association rules cannot be generalized so far, as very distinct results were found in the manual and automatic analysis.

RQ4. What, if any, patterns exist in the language constructs of failed merges involving multiple conflicting chunks?

When a failed merge involves multiple chunks, it might be that certain dependencies exist that are indicative of strategies that could help resolve the conflict. For instance, Figure 22 depicts a case in which it is desirable to resolve chunk A before chunk B, as the change in the *signature* of the *createField method* affects its *invocation*. Consequently, attempting to resolve chunk B first is likely to be ineffective.

Considering the **manual analyses,** we first collected the data presented in Table 11 to determine whether opportunities may exist for new merge tools that explore dependencies among chunks. This table has the following columns: “total”, representing the number of failed merges per project; “conflicting chunks > 1”, representing the number of failed merges that involve two or more conflicting chunks; and “dependencies”, representing the number of

failed merges that involve dependencies among chunks. We determined the presence of dependencies through a manual analysis focused on syntactical dependencies, that is, using the same language constructs across different chunks. Thus, there are no false positives considering, for instance, local variables with the same identifier in different scopes.

```
}
```

version 1	*version 2*
`private static FieldDeclaration createField(LoggingFramework framework, Annotation source, ClassLiteralAccess loggingType, String logFieldName, boolean useStatic) {`	`public static FieldDeclaration createField(LoggingFramework framework, Annotation source, ClassLiteralAccess loggingType, String loggerCategory) {`
`int pS = source.sourceStart, pE = source.sourceEnd;`	

Conflicting chunk A

```
ClassLiteralAccess loggingType = selfType(owner, source);
```

version 1	*version 2*
`FieldDeclaration field = createField(framework, source, loggingType, logFieldName, useStatic);`	`FieldDeclaration field = createField(framework, source, loggingType, loggerCategory);`
`fieldDeclaration.traverse(new SetGeneratedByVisitor(source), typeDecl.staticInitializerScope);`	

Conflicting chunk B

Figure 22. Dependent chunks of merge f956ba from project Lombok.

Table 11. Dependencies in failed merges.

Projects	Failed merges		
	Total	Conflicting chunks > 1	Dependencies
ANTLR4	22	12 (55%)	3 (14%)
Lombok	14	9 (64%)	3 (21%)
MCT	18	9 (50%)	4 (22%)
Twitter4J	36	21 (58%)	6 (17%)
Voldemort	57	41 (72%)	26 (46%)
Total	147	92 (63%)	42 (29%)

Across the 5 projects we analyzed, the percentage of failed merges that involve multiple conflicting chunks and have dependencies among 2 or more of the conflicting chunks varies from 14% to 46%. When considering just the failed merges with multiple conflicting chunks, a range of 25% to 63% of them involve dependencies, indicating that a significant opportunity exists for new merge tools to leverage these dependencies in assisting developers to find the best order in which the chunks should be addressed for resolution.

Table 12 lists the most frequent association rules for conflicts involving different (combinations of) language constructs across merges, with support and confidence thresholds of 10% and 50%, respectively. The table is similar to Table 7, but instead of providing results

at the level of individual chunks, it presents association rules at the level of entire failed merges. The first rule, as an example, is to be interpreted as follows: a *method signature* and a *variable* co-occur in 20% of all failed merges (whether in a single chunk or across chunks). Additionally, the presence of a *method signature* in a chunk of a failed merge means a *variable* is also in some chunk of the same failed merge 94% of the cases, while having *method signature* present in a chunk of a failed merge increases the chances of having *variable* in some chunk of the same failed merge by 82% as compared to the frequency that *variable* usually appear in failed merges.

Table 12. Relation among language constructs that belong to failed merges.

Association rule	**Sup.**	**Con.**	**Lift**
Method signature → variable	20%	94%	1.82
Method declaration, method invocation → variable	20%	91%	1.76
If statement → method invocation, variable	24%	69%	1.75
Annotation → method declaration	11%	73%	1.72
Commentary, method invocation → variable	24%	85%	1.64
If statement, method invocation → variable	24%	85%	1.64
If statement, variable → method invocation	24%	90%	1.63
Method declaration, variable → method invocation	14%	86%	1.56
Commentary, variable → method invocation	24%	85%	1.54
Method signature → method invocation	18%	84%	1.52
If statement → variable	27%	76%	1.47
If statement → method invocation	28%	80%	1.45
Method invocation, variable → commentary	24%	60%	1.40
Method invocation → variable	39%	72%	1.39
Variable → method invocation	39%	76%	1.38
Commentary → variable	28%	65%	1.26
Commentary → method invocation	28%	65%	1.18
Import → method invocation	16%	52%	1.01

Unsurprisingly, the rule *method signature* → *method invocation*, which represents a situation where *method signature* and *method invocation* occur together in 18% of failed merges, did not appear when considering single chunks. *Method invocation*, after all, tends to appear in a different part of a source file where the corresponding *declaration* is made. Association rules such as this one may thus be the source for heuristics to prune the search space of alternative orderings for resolving conflicting chunks. More surprisingly, though, is

the presence of association rules such as *method signature* → *variable*. This association rule appears to be quite disconnected, in some ways, but a closer inspection reveals that it indicates some common cases that might be exploited. For instance, Figure 23 shows an example of *method signature* → *variable*, in which the *method signature* provides a clue on how to resolve the *variable* conflict by looking at its return type *(Token* vs. *Symbol*).

```
//      public List<Integer> states;
```

version 1	*version 2*
… public int s = -1; public Token start, stop;	public Symbol start, stop; … public int ruleIndex;

```
/** Set during parsing to identify which alt of rule parser is in. */
```

Conflicting chunk A

```
}
```

version 1	*version 2*
public Token getStart() { return start; } public Token getStop() { return stop; }	… public Symbol getStart() { return start; } public Symbol getStop() { return stop; } …

```
/** Used for rule context info debugging during parse-time, not so much for ATN debugging */
```

Conflicting chunk B

Figure 23. Dependent chunks of merge 92ae0f from project ANTLR4.

Considering the **automatic analyses**, we also analyzed the association rules extracted from failed merges occurring in the larger sample of projects. Table 13 shows the top ten association rules regarding support. As in the manual analysis, we realize that the support of the association rules increased for rules extracted from failed merge. A lesson we can learn from this table is that in most of the cases, nine to be precise, the difference between the directions of the rules is greater than 10% and in 5 rules this difference is greater than 20%. This shows us that the direction of association rules can tell which language constructs are a signal that another language construct will appear in the same failed merge. For instance, the occurrence of *variable* in a conflicting chunk indicates that a *method invocation* will also be in place in 95% of the failed merges, but the occurrence of a *method invocation* leads to having *variable* in just 66% of the failed merges.

By comparing Table 12 and Table 13, one can observe the higher values of support of the results generated in the automatic analysis, as well as a side-effect of having many rules comprising a single construct in the antecedent and in the consequent. This effect is generated because a chunk with *if statement*, *method invocation*, and *variable* supports the association rule "if statement, method invocation → variable" and "method invocation → variable", but a chunk comprising the constructs *method invocation* and *variable* just supports the second association rule. Thus, rules having a single construct in the antecedent tend to show higher support values in the large mining space provided by the automatic analysis. Future analyses

looking at rules with low support but very high lift may show less frequent but important patterns.

Table 13. Association rules of merges with highest support.

Association rule	Sup	Confidence →	Confidence ←	Lift
Variable ⇆ Method invocation	42%	95%	66%	1.49
Comment ⇆ Method invocation	34%	72%	53%	1.13
If statement ⇆ Method invocation	32%	89%	50%	1.39
Attribute ⇆ Method invocation	30%	78%	47%	1.22
If statement ⇆ Variable	27%	76%	62%	1.71
Variable ⇆ Comment	26%	60%	56%	1.27
Method signature ⇆ Method invocation	24%	89%	38%	1.39
Attribute ⇆ Comment	23%	61%	49%	1.29
If statement ⇆ Comment	22%	62%	48%	1.31
Method declaration ⇆ Method invocation	22%	72%	34%	1.12

An interesting result in Table 13 is related to the association rules with *method signature* or *method declaration* in the antecedent, and *method invocation* in the consequent. Contrasting these rules with the ones in Table 10, we observe, for instance, that "*method signature* → *method invocation*" had its support and confidence increased from 8% to 24% and from 62% to 89%, respectively. Another case of interest happens with *method declaration* because the rule "*method declaration* ⇆ *method invocation*" is not shown in Table 10 because it has support of just 4%, but appears in the rules extracted from failed merges with 22% of support. These cases enforce the arguments that generally a change/conflict in the *method signature* or *method declaration* generates a change/conflict in the *method invocation*.

The results of manual and automatic analyses show different association rules that should be analyzed by a specialist to select the more promising ones to create new heuristics to resolve frequent types of conflicts. For example, any developer would say that *method declaration* and *method invocation* are related language constructs. However, the dependency between *Method signature* and *Variable* is not so simple to be identified without help of association rules, even though they are connected when there is an encapsulation of the fields.

RQ5. What is the distribution of developer decisions?

Considering the **manual analyses,** Erro! Fonte de referência não encontrada. shows that, across the 616 conflicting chunks found in the 5 projects, the primary choice that developers make when facing the need to resolve a conflicting chunk is to select 1 of the

versions (56%), either *version 1* (*V1*, *21%*) or *version 2* (*V2*, *35%*). This was, to us, quite an unexpected result. Given all the commentary and folklore around the merge problem and how difficult it is said to be to resolve merge conflicts (Duvall et al., 2007; Chacon, 2009), we had expected *new code* to be the most common choice. It is not, however: only 19% of conflicting chunks are resolved with an approach that involves writing new code. This means that a full 81% of the conflicting chunks are resolved using only the contents from the code contained by the conflicting chunk without actually adding or editing lines of code (*V1*, *V2*, *concatenation*, and *combination*). While this does not necessarily mean that doing so is trivial, it does mean that merge resolution through one of these four strategies could well be supported not by yet more automation, but maybe through tool support that assists developers in making one of these four choices in the first place and, in the latter two cases, by helping them select and order the necessary lines of code.

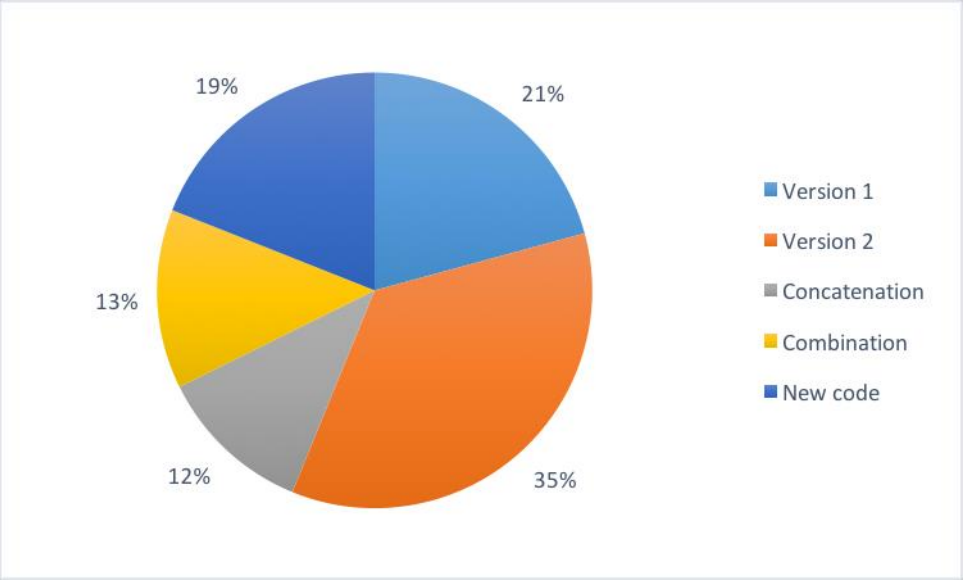

Figure 24. How developers resolve conflicts.

Considering the predominance of resolutions based on *version 1* and *version 2*, we selected a sample of merges that were resolved using these developer decisions. This sample shows that in some cases the code of both versions is equivalent (*i.e.*, they perform the same task), changing just few parts or even being almost equal, but just changing the indentation (*i.e.*, false-positive conflicts generated when using the physical merge). Another characteristic that we could notice is that in some cases one of the versions had its contents totally or partially removed, which transforms it in a resolution that does not perform the same task of its original version (that is, the source code of both versions is significantly different). Therefore, the developers should make the choice for one of the versions instead of combining them.

Figure 25 breaks down Figure 24 by project, and offers some interesting insights. First, some projects have up to 25.68% of resolutions involving *new code*, which, while more than the average of 19%, still reinforces that, for each of the projects, the majority of conflicts are resolved without writing new code. Second, some significant differences exist across the projects. For instance, the predominance of choices for *version 2* in Voldemort (45.35%) is contrasted by a mere 4.35% choice of *version 2* in MCT. MCT, on the other hand, had a high percentage of resolutions involving *combination*, especially when compared to Voldemort, where this choice was rarely made. As a final example, Twitter4j had half the resolutions (11.83%) involving *new code* compared to ANTLR4 (25.68%). It is clear that different projects can exhibit unique trends, which might be useful when it comes to supporting developers – even if simply by showing statistics on past resolution choices.

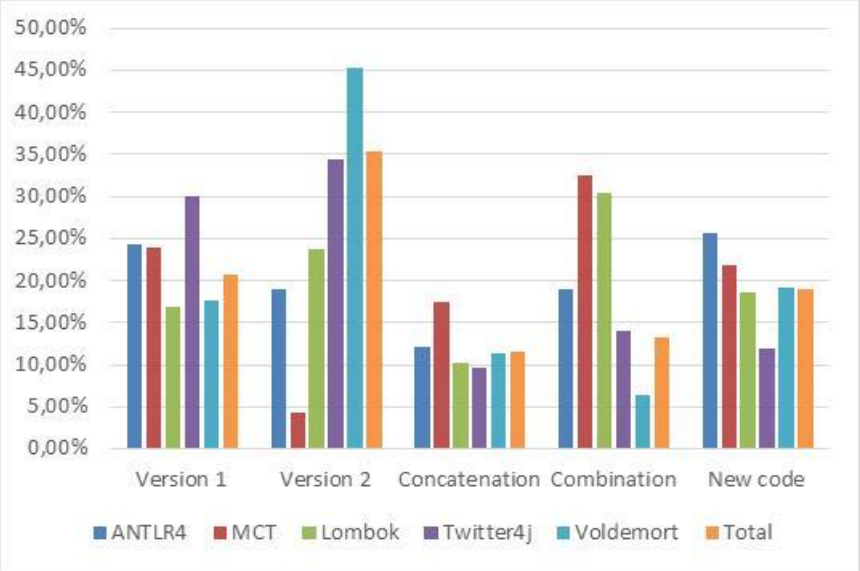

Figure 25. How conflicting chunks are resolved in each project.

We also analyzed resolution choices per developer. This information is summarized in Table 14, which lists, per project, the developers who performed at least 1 merge, the number of conflicting chunks each developer resolved (CH), the percentage of each decision (V1 – *version 1*; V2 – *version 2*; CC – *concatenation*; CB – *combination*; or NC – *new code*) that each developer took, and the percentage of conflicting chunks that they resolved in each project (Total). The last column sums up to 100% for each project.

One pattern immediately stands out: for 4 of the projects, a single developer resolved the majority of the merge conflicts, despite the fact that the projects had code contributions from many more developers (Table 1 shows code contributions by 14, 13, 16, and 84 developers to ANTLR4, MCT, Lombok, and Twitter4j, respectively). Especially for Twitter4j, this is a remarkable pattern: it indicates that one developer has the role of

gatekeeper. This is not out of line with typical open source practice, in which numerous contributors may submit patches, but few have actual commit privileges to integrate the patches into the main branch of development. ANTLR4, as well as Lombok, seem to have a pair of developers that assume that role, whereas in Voldemort the responsibility appears more divided (with four to six developers who resolve most merge failures).

Table 14. How each developer resolves conflicting chunks.

Project	Developer	CH	V1	V2	CC	CB	NC	Total
ANTLR4	Terence Parr	28	25%	21%	14%	14%	25%	38%
ANTLR4	Sam Harwell	46	24%	17%	11%	22%	26%	62%
Lombok	Roel Spilker	13	0%	46%	23%	31%	0%	22%
Lombok	Reinier Zwitserloot	46	22%	17%	7%	30%	24%	78%
MCT	Peter B. Tran	1	0%	100%	0%	0%	0%	2%
MCT	Dan Berrios	3	0%	0%	67%	33%	0%	7%
MCT	Christopher Webster	4	0%	0%	0%	25%	75%	9%
MCT	Victor Woeltjen	38	29%	3%	16%	34%	18%	83%
Twitter4J	Danaja	1	100%	0%	0%	0%	0%	1%
Twitter4J	Jsirois	2	100%	0%	0%	0%	0%	2%
Twitter4J	John Corwin	3	33%	67%	0%	0%	0%	3%
Twitter4J	Takao Nakaguchi	5	0%	80%	0%	0%	20%	5%
Twitter4J	Yusuke Yamamoto	82	29%	32%	11%	16%	12%	88%
Voldemort	Ismael Juma	2	0%	0%	50%	0%	50%	1%
Voldemort	Jay Kreps	6	0%	33%	0%	0%	67%	2%
Voldemort	Vinoth Chandar	6	50%	0%	17%	0%	33%	2%
Voldemort	Neha	7	0%	100%	0%	0%	0%	2%
Voldemort	Alex Feinberg	20	45%	50%	0%	0%	5%	6%
Voldemort	Lei Gao	26	31%	46%	8%	0%	15%	8%
Voldemort	Kirk True	44	14%	34%	18%	18%	16%	13%
Voldemort	Chinmay Soman	49	16%	51%	18%	4%	10%	14%
Voldemort	Roshan Sumbaly	87	10%	41%	8%	11%	29%	25%
Voldemort	Bhupesh Bansal	97	19%	51%	11%	2%	18%	28%

While a number of developers have no clear preference, some developers appear more likely to resolve conflicts in the same manner. For instance, Lei Gao resolved conflicting chunks in almost 50% of the cases by choosing V2 (out of 26 total conflicting chunks); Takao Nakaguchi made the same choice in four out of 5 of the cases. Some of these patterns are related to the fact that one failed merge can involve multiple conflicting chunks, meaning that each of the conflicting chunks of that failed merge was resolved in the same way (by choosing one version, each time). Examples exist, however, of failed merges in which developers used

a variety of strategies. Roshan Sumbaly, for instance, facing a failed merge (970260) involving 28 conflicting chunks, chose *V1* 4 times and *V2* 12 times, combined both versions 5 times, and wrote *new code* in 7 instances.

We also examined whether the number of conflicting chunks in a failed merge had any effect on develop decisions as to how to resolve the conflicts. Table 15 shows the results, with column "Project" identifying the project, "#Chunks" offering a classification of the number of conflicting chunks involved in a failed merge (1, 2, 3-5, 6-15, and 16+), FM counting the number of failed merges with that number of chunks, "Total" presenting the total number of conflicting chunks, and the remaining columns identifying the chosen resolution strategies.

Table 15. Distribution of failed merges by conflicting chunks.

Project	#Chunks	FM	Total	V1	V2	CC	CB	NC
ANTLR4	1	10	10	30%	30%	10%	10%	20%
ANTLR4	2	4	8	38%	38%	25%	0%	0%
ANTLR4	3-5	5	19	32%	11%	5%	16%	37%
ANTLR4	6-15	3	37	16%	16%	14%	27%	27%
ANTLR4	16+	-	-	-	-	-	-	-
Lombok	1	9	9	22%	33%	33%	11%	0%
Lombok	2	2	4	0%	0%	50%	25%	25%
Lombok	3-5	4	15	13%	7%	7%	47%	27%
Lombok	6-15	2	14	36%	43%	0%	14%	7%
Lombok	16+	1	17	6%	24%	0%	41%	29%
MCT	1	5	5	0%	20%	0%	40%	40%
MCT	2	6	12	8%	8%	42%	17%	25%
MCT	3-5	1	4	50%	0%	50%	0%	0%
MCT	6-15	2	25	32%	0%	4%	44%	20%
MCT	16+	-	-	-	-	-	-	-
Twitter4J	1	15	15	47%	13%	13%	13%	13%
Twitter4J	2	8	16	38%	19%	25%	13%	6%
Twitter4J	3-5	11	40	28%	40%	5%	18%	10%
Twitter4J	6-15	2	22	18%	50%	5%	9%	18%
Twitter4J	16+	-	-	-	-	-	-	-
Voldemort	1	16	16	13%	31%	19%	6%	31%
Voldemort	2	12	24	25%	29%	17%	8%	21%
Voldemort	3-5	8	33	6%	45%	9%	9%	30%
Voldemort	6-15	17	154	18%	56%	12%	3%	11%
Voldemort	16+	4	117	20%	37%	9%	10%	25%

```
*/
```

version 1	*version 2*
`CANNOT_CREATE_TARGET_GENERATOR(31, "ANTLR cannot generate '<arg>' code as of version " + Tool.VERSION, ErrorSeverity.ERROR_ONE_OFF),`	`CANNOT_CREATE_TARGET_GENERATOR(31, "ANTLR cannot generate <arg> code as of version " + Tool.VERSION, ErrorSeverity.ERROR),`

original merge resolution

```
*/
CANNOT_CREATE_TARGET_GENERATOR(31, "ANTLR cannot generate <arg> code as of version " +
Tool.VERSION, ErrorSeverity.ERROR),
```

eventual merge resolution

```
*/
CANNOT_CREATE_TARGET_GENERATOR(31, "ANTLR cannot generate <arg> code as of version " +
Tool.VERSION, ErrorSeverity.ERROR_ONE_OFF),
```

Figure 26. Conflicting chunk extracted from project ANTLR4 of merge d85ea0 that was resolved by a commit using the contents of *version 2*, but in a latter commit the resolution was changed to the contents of *version 1* with a small tweak.

Table 16 tallies the number of times a commit (with associated chunks) changed code from a previous commit in the month before, together with the number of times such a commit (and its associated chunks) actually represented a postponed resolution. Clearly, not every change to code from a previous commit is a postponed resolution, since it is natural for future changes to build on previous changes. The majority of these commits were indeed of this nature, but six of the commits represented postponed merges. Two of those (1 for ANTLR4 and Lombok each) are merges in which a single conflicting chunk was postponed (out of 1 chunk for ANTLR4 and 8 chunks for Lombok); the other 4 involved a total of 11 conflicting chunks that were postponed. This is an interesting result, as it indicates that it is not necessarily the failed merges with high number of chunks that are postponed (recall from the discussion in Section 3.5 that a non-trivial portion of merges involve greater than five conflicting chunks).

Table 16. Changes in conflicting chunks areas.

Project	Merges		Conflicting Chunks	
	Changed	Postponed	Changed	Postponed
ANTLR4	3 (11%)	1 (4%)	3 (3%)	1 (1%)
Lombok	5 (23%)	1 (5%)	15 (22%)	1 (1%)
MCT	5 (29%)	0 (0%)	17 (33%)	0 (0%)
Twitter4J	4 (11%)	0 (0%)	7 (7%)	0 (0%)
Voldemort	20 (31%)	4 (6%)	58 (14%)	11 (3%)
Total	37 (22%)	6 (4%)	100 (14%)	13 (2%)

Overall, postponed merges happen fairly rarely (4% of the time) and involve only 2% of all conflicting chunks, but nonetheless do take place. In manually examining all six of the

failed merges that were postponed, no apparent pattern or reason seems to dominate. In one case, the developer adjusted the *variable* name due to a rename refactoring that they postponed (see Figure 27). In other cases, the developers commented or simply left the source code in conflict to resolve it afterwards. For instance, in Figure 28 we observe 2 developers working in parallel on the same class on the same day, choosing to postpone the merge of a commentary region multiple times. In fact, they performed a series of rebases and, in the end, simply merged without removing the merge markers. One week later one of them committed (05f23e) with a message "code clean up" that finally removed the merge markers.

Version 1

```
if ( !isDirectDesc… && !callSuper && implicit ) {
      errorNode.addWarning("…  If  this  is  intentional,  add  '@EqualsAndHashCode(callSuper = false)' to your type.");
}
```

version 2

```
if ( !isDirectDesc… && !callSuper ) {
      errorNode.addWarning("…");
}
```

original merge resolution

```
if ( !isDirectDesc… && !callSuper && implicit ) {
  errorNode.addWarning("… If this is intentional, add '@EqualsAndHashCode(callSuper = false)' to your type.");
}
```

eventual merge resolution

```
if ( !isDirectDesc… && !callSuper && implicitCallSuper ) {
  errorNode.addWarning("… If this is intentional, add '@EqualsAndHashCode(callSuper = false)' to your type.");
}
```

Figure 27. Conflicting chunk extracted from project Lombok (merge 4e152f) that was resolved by a commit using the contents of *version 1*, but in a latter commit (f1124a) was changed due to a refactoring.

first rebase with postponed resolution

```
…
=======
>>>>>>> add clientId for voldemort client
…
```

second rebase with postponed resolution

```
…
<<<<<<< HEAD
=======
>>>>>>> add clientId for voldemort client
=======
>>>>>>> Adding System store functionality
…
```

merge with postponed resolution

```
…
<<<<<<< HEAD
<<<<<<< HEAD
=======
>>>>>>> add clientId for voldemort client
=======
>>>>>>> Adding System store functionality
=======
>>>>>>> leigao/client-registry
…
```

Figure 28. Sequence of two rebases (a21bf2, 234ac9) followed by a merge (3fbef9), all with postponed resolutions, in project Voldemort.

Across all 13 conflicting chunks that were part of postponed merges, the dominant developer decision for the original resolution was *new code* (8 occurrences) followed by *version 2* (5 occurrences), and for the eventual resolution was *new code* (12 occurrences) followed by 1 *version 1*. From this, we can observe that postponed merges seem to address situations that are complex to resolve (though not necessarily with a large number of chunks), and generally require new code to be written.

Considering the **automatic analyses**, Figure 29 presents the distribution of developers' decisions based on the number of conflicting chunks resolved according to each decision. In compliance with results from the manual analysis, most chunks are resolved by the choice of 1 of the versions, either Version 1 – in 50% of the chunks – or Version 2 – in 25% of the chunks. Moreover, 3% of the conflicting chunks are resolved through the concatenation of Version 1 with Version 2, or vice-versa. Additionally, 9% of the chunks are resolved by combining subsets of Version 1 and Version 2. Finally, 13% of the conflicting chunks need new code to be resolved, while only 0.4% of the chunks are resolved using none of the versions, that is, the developer resolves to exclude the content of both versions.

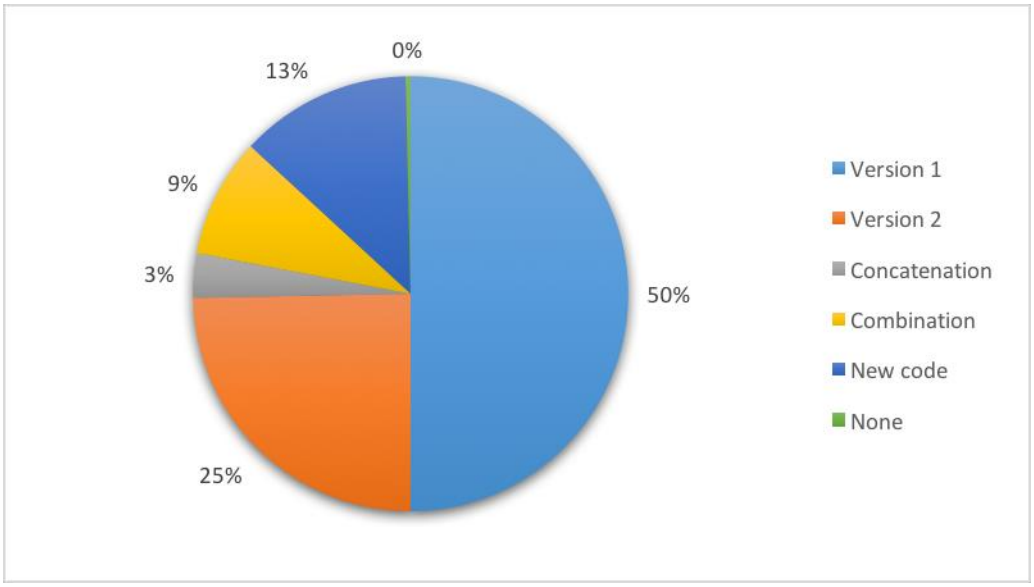

Figure 29. Distribution of developer decisions.

By comparing the charts in Figure 29 and Figure 24, we observe that the number of merge resolutions involving *new code* decreased from 19% to 13%. This shows that even considering a huge number of projects the source code used to resolve conflicting chunks is frequently present in one of the versions, either as a full version or its parts. Another decision that changed drastically in frequency is *version 1*, which increased from 21% to 50% – half of the chunks analyzed in the 2,731 projects selected for automatic analysis are resolved by

picking *version 1*. This is surprising if compared to the frequency of adding new code, but can be explained if we take into account that developers resolve merges to integrate their changes and they might frequently choose the version that has their contributions to some task.

Only 13% of our sample of conflicting chunks needs complex resolutions on which developers introduce new code. However, a closer look on the distribution of developer resolutions in our sample reveals that many projects have a significant portion of their conflicting chunks resolved manually, as can be observed in the large inter-quartile range and elongated upper-whisker for the third box-plot from left to right in Figure 30. The vertical axis of this box-plot presents the percentage of chunks in a project that were resolved by a given developer decision. We can observe that, for any given developer decision, at least one project had all of its conflicting chunks resolved according to that decision. By concentrating on the box-plot representing the introduction of *new code* (NC), we observe that despite of the low mean number of failed merges requiring new code (close to 10%), more than 20% of the chunks of 25% of the projects required the addition of new code. Furthermore, the trend for selecting resolutions for failed merges may depend on other characteristics of the projects and some projects seem more likely to use specific decisions.

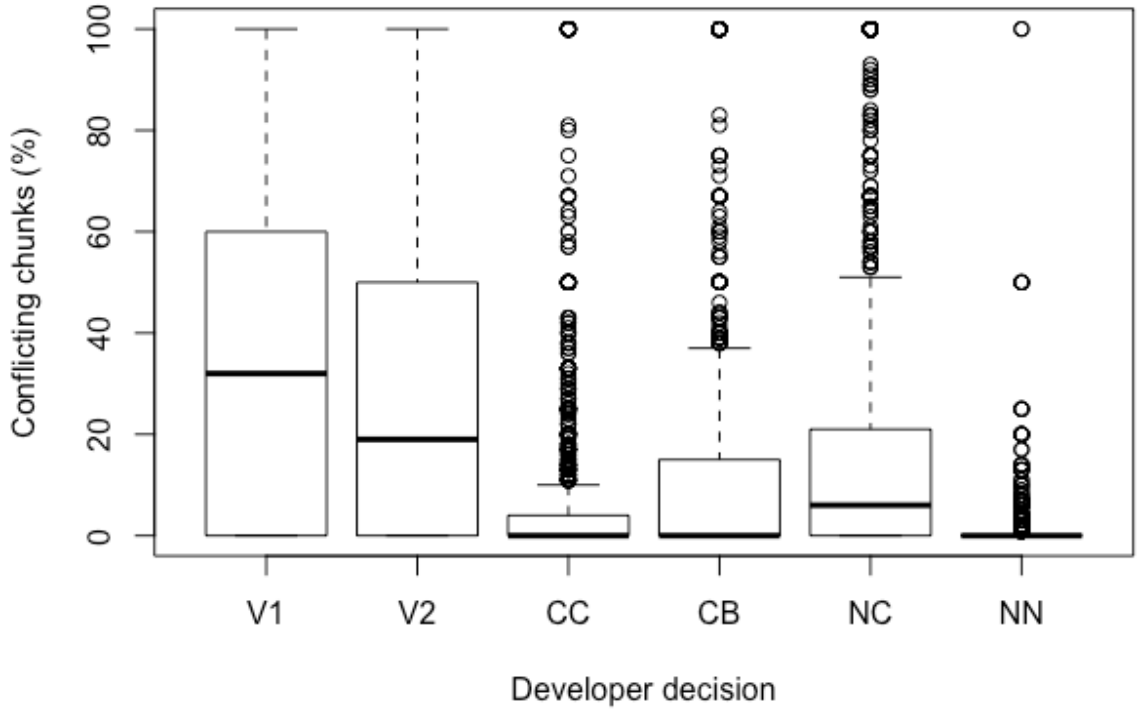

Figure 30. Box-plots for the distribution of developer decisions.

The results of manual and automatic analyses show the same message: few chunks are resolved by using *new code*. This motivate the creation of automatic approaches, since in most of the cases the source code used in the resolution of chunks is already in the *version 1* or *2*.

RQ6. What is the distribution in difficulty level of kinds of conflicts?

Based on the concept of cheap and expensive chunks presented in Section 3.2, we analyzed the approximate difficulty of different kinds of conflicts with the help of the following Cost Ratio:

$$CR = (\#Expensive\ chunks)/(\#Conflicting\ chunks)$$

For each kind of conflict found in the five projects selected for manual analysis, we catalogued the developer decisions that were made for the conflicting chunks in which that kind of conflict occurred, and then assigned the appropriate weight (cheap or expensive) based on the developer decisions made. The above cost ratio, then, is calculated as the percentage of expensive chunks over all chunks with that kind of conflict. The higher the CR value, the more difficult the kind of conflict appears to be resolved. Considering the **manual analyses,** Table 17 shows the kinds of conflicts that appear in more than ten conflicting chunks, in ascending order of their CR.

The kind of conflicts consisting only of *variable* or *import* seems to be the easiest to resolve, as these chunks rarely involve writing *new code*. *Concatenation* is the most frequent decision in resolving these kinds of conflicts, with *version 1* and *version 2* also chosen frequently. Perhaps counter intuitively, *combination* is also used at times. This represents situations in which multiple *variables* or multiple *imports* are in conflict, and developers rearrange them, or a subset, in a new order. An example is shown in Figure 31, with the developer adopting both versions of the conflicting chunk but reordering some *imports* according to their preferred order.

On the other hand of the spectrum, the kinds of conflicts of *commentary* and *if statement, method invocation* seem to be most difficult to resolve, most often involving *combination* and *new code* as the resolution strategies (relative to the total number of kinds of conflicts of that type). Commentary is unsurprising, as *comments* are normally written in natural language and thus not a good match for automation. The fact that the *if statement, method invocation* kind of conflict appeared among the most difficult surprised us, however, especially when compared to the *if statement, method invocation, variable* kind of conflict.

The latter has more language constructs involved, but appears significantly easier to address. Inspecting the two kinds of conflicts closely, we noticed that the *if statement, method invocation, variable* kind of conflict often consisted of a conditional *method invocation* that assigned a *variable* a value, whereas the *if statement, method invocation* usually involved more complex behavior. The example in Figure 32, for instance, shows a situation in which concatenation and even combination would fail: the message id of Native Backup had to be updated to 30.

Table 17. Cost ratio of resolving kinds of conflicts.

Kind of conflict	V1	V2	CC	CB	NC	CR
Variable	9 (24.3%)	8 (21.6%)	13 (35.1%)	4 (10.8%)	3 (8.1%)	19%
Import	12 (20.0%)	14 (23.3%)	22 (36.7%)	10 (16.7%)	2 (3.3%)	20%
If statement, method invocation, variable	4 (21.1%)	10 (52.6%)	1 (5.3%)	2 (10.5%)	2 (10.5%)	21%
Method declaration	10 (17.5%)	24 (42.1%)	11 (19.3%)	4 (7%)	8 (14%)	21%
If statement	6 (30.0%)	7 (35%)	1 (5%)	1 (5%)	5 (25%)	30%
Commentary, method declaration	4 (25.0%)	5 (31.3%)	2 (12.5%)	3 (18.8%)	2 (12.5%)	31%
Method signature	5 (26.3%)	8 (42.1%)	0 (0%)	1 (5.3%)	5 (26.3%)	32%
Method invocation	6 (9.5%)	31 (49.2%)	5 (7.9%)	10 (15.9%)	11 (17.5%)	33%
Method invocation, variable	17 (29.3%)	20 (34.5%)	1 (1.7%)	10 (17.2%)	10 (17.2%)	34%
If statement, method invocation	2 (16.7%)	3 (25%)	2 (16.7%)	2 (16.7%)	3 (25%)	42%
Commentary	4 (28.6%)	4 (28.6%)	0 (0%)	1 (7.1%)	5 (35.7%)	43%

```
import com.sun.tools.javac.tree.JCTree.JCLiteral;
```

version 1

```
import com.sun.tools.javac.tree.JCTree.JCPri…;
import com.sun.tools.javac.tree.JCTree.JCU…;
import com.sun.tools.javac.tree.TreeMaker
```

version 2

```
import com.sun.tools.javac.tree.JCTree.JCMod…;
import com.sun.tools.javac.tree.JCTree.JCTyp…;
import com.sun.tools.javac.tree.TreeMaker;
import com.sun.tools.javac.util.List;
import com.sun.tools.javac.util.Name;
```

```
/**
```

merge resolution

```
import com.sun.tools.javac.tree.JCTree.JCLiteral;
import com.sun.tools.javac.tree.JCTree.JCMod…;
import com.sun.tools.javac.tree.JCTree.JCPri…;
import com.sun.tools.javac.tree.JCTree.JCTyp…;
import com.sun.tools.javac.tree.JCTree.JCU;
import com.sun.tools.javac.tree.TreeMaker;
import com.sun.tools.javac.util.List;
import com.sun.tools.javac.util.Name;
/**
```

Figure 31. Conflicting chunks among *import declarations* from project Lombok (merge 45697b).

A final highlight concerns the *method declaration* kind of conflict. Developers decide upon 1 version or the other in the vast majority of cases (*version 1* in 10 cases, *version 2* in 24) and quite frequently they also *concatenate* both (11 cases). Only in four cases did they combine the two by choosing a subset of lines of each, and in eight cases did they integrate the two with *new code* to help them do so.

```
}
```

version 1

```
if (hasInitiateRebalanceNodeOnDonor()) {
  output.write…(28,
getInitiateRebalanceNode…());
}
if (hasDeleteStoreRebalanceState()) {
  output.write…(29,
getDeleteStoreRebalanceS…());
```

version 2

```
if (hasNativeBackup()) {
  output.write…(28, getNativeBackup());
```

```
}
```

merge resolution

```
}
if (hasInitiateRebalanceNodeOnDonor()) {
  output.write…(28, getInitiateRebalanceNode…());
}
if (hasDeleteStoreRebalanceState()) {
  output.write…(29, getDeleteStoreRebalanceS…());
}
if (hasNativeBackup()) {
  output.write…(30, getNativeBackup());
}
```

Figure 32. Conflicting chunk from project Voldemort from merge 491863.

Considering the **automatic analyses,** Table 18 shows the kinds of conflicts that appear in more than 2,000 conflicting chunks collected from the larger sample of projects, in ascending order of CR. Surprisingly and differently from Table 17, *commentary* appears as the most simple kind of conflict to resolve, being frequently resolved by adopting *version 1* (76.5%). On the other hand, *method declaration* is the second easiest kind of conflict to resolve, as developers typically resolve conflicting declarations by picking the new version (*version 1)*, the old one (*version 2)*, or by concatenating both *method declarations*, if they do

not have the same goal; few cases indeed demand that developers add *new code* to the declaration.

Another observation extracted by comparing Table 17 to Table 18 is that the kind of conflict comprising *attribute* and *method invocation* is one of the hardest to resolve. For the sample of projects involved in the automatic analysis, 9.5% of the chunks presenting this kind of conflict are resolved by combining both versions, while 21% are resolved by inserting *new code*. In the manual analysis, these developer decisions sum up 17.2% of the choices to resolve chunks of the *method invocation, variable* kind. Finally, 9.6% of the chunks involving *import* are resolved by adding *new code*, against 3.3% as observed in the manual analysis. We analyzed some conflicting chunks that comprise only *import declarations* and were resolved using *new code* to understand why this happened. We observed that in some cases the developers changed the use of wildcard by the direct use of the *imports*, avoiding using the whole content of the packages. Also, the developers changed the context of conflicting chunks, which does not allow us to find the correct boundaries of chunks and, as a consequence, classify their resolution as *new code*. Additionally, we also noticed that in some cases developers add new *import declarations* that are complementary to the ones that are already in place. For instance, when there is an *import declaration* to *List* and the developer add an *import* to *ArrayList*. All in all, although the chunks resolutions were classified as *new code*, the resolutions can be automated using refactoring tools that organize the *imports*.

Table 18. Cost ratio of resolving kinds of conflicts.

Kind of conflict	V1	V2	CC	CB	NC	NN	CR
Commentary	14874 (76.5%)	3505 (18%)	127 (0.7%)	249 (1.3%)	649 (3.3%)	43 (0.2%)	5%
Method declaration	1885 (51.9%)	1209 (33.3%)	153 (4.2%)	161 (4.4%)	220 (6.1%)	4 (0.1%)	12%
Comment, Method invocation	2383 (68.7%)	525 (15.1%)	62 (1.8%)	187 (5.4%)	288 (8.3%)	24 (0.7%)	16%
Method invocation	7594 (56%)	3458 (25.5%)	334 (2.5%)	745 (5.5%)	1393 (10.3%)	25 (0.2%)	19%
Method signature	2035 (56.4%)	943 (26.2%)	2 (0.1%)	53 (1.5%)	569 (15.8%)	4 (0.1%)	21%
Comment, Method declaration	1197 (48.1%)	769 (30.9%)	84 (3.4%)	167 (6.7%)	264 (10.6%)	7 (0.3%)	21%
Annotation, Method declaration	926 (45.4%)	590 (28.9%)	137 (6.7%)	132 (6.5%)	250 (12.2%)	6 (0.3%)	23%
Attribute, Comment	1053 (49.9%)	463 (21.9%)	170 (8.1%)	233 (11%)	186 (8.8%)	5 (0.2%)	25%
Attribute	3093 (46.6%)	1634 (24.6%)	445 (6.7%)	428 (6.5%)	1019 (15.4%)	15 (0.2%)	28%
Method invocation, Variable	3949 (44.5%)	2591 (29.2%)	62 (0.7%)	835 (9.4%)	1419 (16%)	9 (0.1%)	34%
If statement, Method invocation	1326 (48.7%)	638 (23.4%)	59 (2.2%)	410 (15%)	290 (10.6%)	2 (0.1%)	35%
Import	7338 (35.7%)	5377 (26.2%)	2058 (10%)	3493 (17%)	1964 (9.6%)	308 (1.5%)	36%
If statement, Method invocation, Variable	1345 (45.2%)	801 (26.9%)	25 (0.8%)	384 (12.9%)	420 (14.1%)	3 (0.1%)	37%
If statement	3773 (49.8%)	1688 (22.3%)	47 (0.6%)	128 (1.7%)	1932 (25.5%)	2 (0%)	37%
Attribute, Method invocation	2130 (40.1%)	1349 (25.4%)	196 (3.7%)	506 (9.5%)	1114 (21%)	22 (0.4%)	44%

The results of manual and automatic analyses show that the same situations in different set of projects can have different level of difficulty. For instance, *import declarations* is classified as easy in the corpus of the manual analyses and hard in the corpus of the automatic analyses. However, even in cases where the chunks resolutions are classified as *new code*, after a deeper analysis it was possible to observe that these resolutions can also be applied in a systematic way.

RQ7. What, if any, patterns exist between the language constructs of conflicting chunks and developer decisions?

Considering the **manual analyses,** our seventh analysis focuses on the possible relationships between language constructs and developer decisions, which we explored through association rules. Table 19 shows 20 association rules extracted from the 5 projects selected for manual analysis, ordered by lift, having absolute support threshold of 6 occurrences. Support values are small for most of the rules due to the large number of potential combinations among language constructs and developer decisions, resulting in a vast space that is sparsely covered by the 616 conflicting chunks. However, various patterns still exist, which we discuss in the following.

Table 19. Relationship Between Language Constructs and Resolutions.

Association rules	Sup.	Con.	Lift
Annotation, method invocation → new code	1%	75%	3.95
Annotation, variable → new code	1%	58%	3.07
Import → concatenation	4%	34%	2.98
For → combination	1%	33%	2.50
Method invocation, try statement → combination	1%	32%	2.40
For statement, try statement → V2	1%	75%	2.12
Commentary, method signature, variable → V2	1%	75%	2.12
Commentary, method signature → V2	2%	67%	1.88
Try statement → combination	1%	25%	1.88
Method declaration, method invocation → new code	1%	35%	1.86
For statement, variable → V2	1%	64%	1.82
Method signature, variable → V2	2%	60%	1.70
Commentary, for statement → V2	1%	60%	1.70
Annotation → new code	1%	32%	1.68
Return → new code	1%	31%	1.62
Method invocation, method signature, variable → V2	1%	57%	1.61
Commentary, method declaration → combination	1%	21%	1.59
If statement, method invocation → combination	2%	21%	1.55
Method declaration, variable → new code	1%	29%	1.50
Commentary, method invocation, variable → V2	2%	52%	1.47

First, we note that 8 out of 20 cases involve *version* 2 as developer decision. Having *version* 2 as the most common choice developers make (see also Figure 24) is, in a way, not surprising. By choosing *version* 2 over *version* 1, the developer resolving a conflict is choosing to keep established code that was recently added by other developers to the main

branch, instead of "overwriting" it with the changes they made in their own workspace. Existing changes usually impose a certain degree of inertia, causing developers to refrain from modifying it.

Second, Table 19 includes the association rule *import* → *concatenation* with 34% of confidence, meaning that in 34% of the conflicting chunks that have import, the developer decision is *concatenation*. Moreover, this association rule has lift 2.98, denoting a strong relation between *import* and *concatenation* (i.e., when import is involved, it increases the chances of choosing *concatenation* by 198%). A question is why developers do not choose *concatenation* all the time, as only 36.7% of developer decisions are *concatenation* (20% *version 1*, 23.3% *version* 2, 16.7% *combination*, and 3.3 *new code*). The issue here is the fact that resolving just the chunk pertaining to the *import* does not take place in isolation – other resolutions, to other conflicting chunks, dictate what *import statements* are ultimately needed. That is, developers take care to not over-include *imports* when they do not need them.

Third, the association rule *annotation, method invocation* → *new code* has the highest lift and confidence values, indicating that this is a difficult conflict that does not lend itself well to choosing one of the versions, concatenating the versions, or combining the versions' existing code in some way. The same is true for *annotation, variable* → *new code*. Based on inspecting these examples, the reason is that *annotations* tend to add extra complexity to the code that involves serious subtleties. For instance, Figure 33 shows a conflicting chunk in which both developers had the same goal – serialize the object as text – and both decided to use overriding to accomplish it. They, however, each chose to override a different method from the superclass (*toString()* versus *getText()*). Given that an @override *annotation* indicates that changes in a method may lead to side effects in the behavior of the superclass due to polymorphism, care must be taken when the changes are merged. In this example, the resolution consisted of using *version 2*, but replacing *setup* with *lazyInit* from *version 1*.

```
}
```

version 1

```
/** Grab *all* tokens from stream and return
string */
@Override
public String toString() {
  lazyInit();

  fill();
  return toString(0, tokens.size()-1);
}
```

version 2

```
/** Get the text of all tokens in this buffer.
*/
@NotNull
@Override
public String getText() {
  if ( p == -1 )
    setup();
  fill();
  return getText(Interval.of(0,size()-1));
}
```

```
@NotNull
```

merge resolution

```
}
/** Get the text of all tokens in this buffer. */
@NotNull
@Override
public String getText() {
  lazyInit();
  fill();
  return getText(Interval.of(0,size()-1));
}
@NotNull
```

Figure 33. Conflicting chunk with *annotations* from project ANTLR4 (merge 18f535).

Considering the **automatic analyses,** we also collected association rules among combinations of language constructs and developer decisions to analyze whether some decisions are frequently applied to conflicting chunks with certain language constructs in the larger sample of projects. As for the manual analysis just described, such rules were collected on a conflicting chunk basis, as different chunks pertaining to the same failed merge may be resolved differently. Table 20 shows the ten association rules having the highest lift. Support values (as percentages of the total number of chunks) are small for most rules due to the huge number of potential combinations of language constructs and developer decisions, which creates a large space of rules which is not densely covered even taking into account that our sample comprises 175,805 conflicting chunks.

As in the manual analysis, *import* is mostly resolved by *concatenation*. The automatic analysis reveals that *combination* is also frequently used to resolve conflicts involving *import*. Conversely, we notice the dominance of indications to *new code* resolutions among the ten rules presenting the highest lift in Table 20. Other interesting situation happens for rules that appear both in the manual and automatic analysis with similar lift, but leading to different developer decisions. For instance, *method signature, variable* points to resolution using *version 2* in the manual analysis and *new code* in the automatic one. This denotes that such rule is not as common as suggested by the manual analysis and further analyses should be executed to understand if patterns are maintained for the same project and organization.

Table 20. Relation between language constructs and resolutions with highest lift.

Association rules	Sup.	Conf.	Lift
Import → Concatenation	1%	9%	2.59
Import → Combination	2%	17%	1.97
Method invocation, method signature → New code	2%	21%	1.61
Method signature, variable → New code	1%	20%	1.55
Method invocation, Return statement → New code	1%	20%	1.53
Return statement → New code	1%	19%	1.45
Attribute, method invocation → New code	2%	18%	1.45
Method signature → New code	2%	18%	1.44
Comment, method invocation, variable → New code	1%	18%	1.41
Comment, variable → New code	1%	18%	1.39

We could also realize by comparing the tables that the confidence values for all rules in Table 20 are much smaller than in Table 19. However, even a rule that appears thrice as much in the manual analysis might appear so many times with a smaller confidence that the nominal number justifies the investment in developing merge tools targeted specifically to address it.

The results of manual and automatic analyses show interesting results. For instance, both point to specific resolutions that can be applied to a small or a large set of projects. For example, *import* is mostly resolved by *concatenation*. However, in some cases they diverge in terms of resolutions, which is the result of analyzing a huge number of projects with different methodologies and developers that can make different decisions when they face a conflict.

RQ8. What is the correlation between the number of developers and the number of commits, merges, and failed merges?

Most developers and researchers on software merge make an assumption that the number of developers has a direct effect in the number of commits, merges, and failed merges. It is generally assumed that the higher the number of developers, the higher the number of failed merges (Cavalcanti et al., 2015). The intuition behind the common sense is that more developers working on a project increases the chances of parallel work over the same artifact and, therefore, leads to the need to integrate these changes manually. This assumption motivated the development of awareness approaches (Brun et al., 2011b) and was used as a criterion to select projects in studies that analyze merge scenarios (Cavalcanti et al.,

2015). However, to the best of our knowledge, there is no study that provides strong evidences about the correlation between the number of developers and failed merges. Considering the **automatic analyses,** we calculated the number of developers, commits, merges, and failed merges in the automatic corpus, as well as the correlations among these measures. Such correlations are shown in the Table 21.

Table 21. Correlations among the number of developers, commits, merges, and failed merges.

First Variable	Second Variable	Correlation
Developers	Commits	0.64
Developers	Merges	0.70
Developers	Failed merges	0.49
Commits	Merges	0.78
Commits	Failed merges	0.65
Merges	Failed merges	0.81

The interpretation of the strength of correlation values is not universal. According to Cohen (1992), values in the interval [0.1, 0.25) are considered weakly correlated, while values in the interval [0.25, 0.40) represent a moderate correlation, and values in the interval [0.4,1] are strongly correlated. On the other hand, Dancey and Reidy (2007) consider weak correlation in the interval [0.10, 0.40), moderate in the interval [0.40, 0.70), and strong correlation for values in the interval [0.70, 1]. Henceforth, we consider the more conservative Dancey and Reidy classification.

According to our findings, developers and failed merges are only moderately correlated (correlation equals to 0.49). On the other hand, we observe that the number of merges is a good proxy for the number of failed merges. The number of merges has a strong correlation with failed merges (0.81) and can be easily extracted from the log of VCS, while the number of failed merges can only be determined by examining the contents of the files involved in a merge, requiring researchers to clone the project repository to collect data and replay all the merges cases. Therefore, researchers interested in projects having a high number of failed merges may select projects with a high number of merges, calculating the number of merges from the log of the VCS instead of replaying all the merge cases to identify if they failed or succeeded.

Researchers might also be interested in projects with a high number of merges. The number of developers can be used as proxy to select such projects, since it is strongly

correlated with the number of merges (0.70) and can be extracted using a single query from the GitHub API, thus avoiding the cost of cloning the project's repository and then parsing the output of the log command. An interesting observation, though, is that the transitivity property does not hold for correlation. Although the number of developers is a good proxy for the number of merges, and the number of merges is a good proxy for the number of failed merges, we cannot conclude that the number of developers is a good proxy for the number of failed merges.

3.7 DISCUSSION

Grounded in the results presented in the previous section, we now return to the original question that motivated our study in the first place: by examining in great detail the nature of merge conflicts, might it be possible to unearth information that could assist in the design of future merge tools? We believe the answer to this question is 'yes, but with caution', an answer upon which we expand in the following.

It is clear from the results that merge conflicts represent a difficult problem to tackle generically. Some conflicts are small, involving a single conflicting chunk with merely a few language constructs (sometimes even just one). Other conflicts are large, with many conflicting chunks and many different language constructs involved. What may work in supporting one, may not work so well in the other. Even when merge conflicts are similar, of roughly the same size and with the same language constructs involved, it is apparent that developers still choose different ways of resolving the conflicts. A single, general, and automated solution, then, may not result in what the developer wanted. This is especially problematic if a developer does not carefully inspect the results of their merges to discover cases in which a merge produced the wrong result.

Based on our results, we believe a space exists between searching for techniques that aim to automate resolution of all possible conflicts and techniques that offer a single base approach and defer to the user when the base approach encounters a problem. Specifically, we advocate an approach in which, in case of a conflict that results in a failed merge, several possible resolutions are identified and suggested to the developer in a convenient manner. From these, the developer chooses the resolution of choice and is further supported in executing this resolution if it requires some manual engagement.

Supporting this observation is the fact that a limited set of language construct combinations occurs frequently. The identification of 'several possible resolutions', a necessary step as required by the approach suggested in the previous paragraph, is more

feasible by focusing on these combinations and developing a dedicated technique for each of the combinations. In the following, we substantiate the idea of assisted merge conflict resolution with three incremental conclusions that we draw from the results in the previous section.

Conflicting chunks generally contain all of the necessary information to resolve them. As shown in Figure 29, only 13% of the conflicting chunks require developers to write extra code beyond what is already present in the chunk (as part of *version 1*, *version 2*, or both), given that the developers chose *version 1*, *version 2*, *concatenation*, and *combination* as the choices of resolving the conflicting chunk in 87% of the cases. This means that the necessary resources for resolving conflicts are in most cases inside the conflicting chunks. While a developer certainly may need to look elsewhere in the code, for the actual change that must be generated, the source code lines in the conflicting chunk suffice. An example is shown in Figure 34, in which the solution is to concatenate the contents of *version* 1 (i.e., *variable*) and *version* 2 (i.e., *annotation and method declaration*).

```
public final class RuleStartState extends ATNState {
  public RuleStopState stopState;
```

version 1

```
public boolean isPrecedenceRule;
```

version 2

```
@Override
public int getStateType() {
  return RULE START;
}
```

```
}
```

merge resolution

```
public final class RuleStartState extends ATNState {
  public RuleStopState stopState;
  public boolean isPrecedenceRule;

  @Override
  public int getStateType() {
    return RULE_START;
  }
}
```

Figure 34. Conflicting chunk (top) and its resolution (bottom) of merge b14ca56 from project ANTLR4.

More complex examples exist as well, in which the developer interleaves lines of code from *version* 1 and *version* 2. Figure 33 shows an example of this. While the resolution is non-trivial, with the developer needing to carefully order the lines of code from both of the versions, again no new code was written.

To assist developers in these kinds of efforts, we believe effort is necessary to re-envision merge tools, for example, to support developers in quickly reshuffling the lines of code from both versions into a single version, perhaps by drag and drop of relevant blocks of code from a column on the left (*version 1*) and a column on the right (*version 2*) to a middle column (merged version). Or, if the number of conflicting lines of code is small, a merge tool

could generate a set of reasonable permutations (excluding permutations that are syntactically incorrect or fail test cases) and present those to the developer. The latter solution likely requires the creation of various heuristics, as the number of possible permutations grows to the number of potential combinations of the lines involved. A combination of search-based software engineering techniques with smart ways of pruning the search tree will be necessary to identify most likely candidates.

The resolution order of conflicting chunks matters. Sixty-three percent of failed merges analyzed in the manual analysis consist of multiple conflicting chunks, while in the automatic analysis this number slightly falls to 59% of the failed merges having more than 1 chunk. In traditional merge tools, once a merge fails, the developer is presented with all conflicting code at once for them to resolve as they please. The merge tool provides an editor in which the entire code of both versions is presented side-by-side, with color-coded marks indicating where the conflicting chunks reside. From there on, developers are left to their own devices, manually working out the desired result.

We observe that resolving conflicting chunks in a given order often can be more effective. For instance, Figure 22 presented a conflict between the *method signatures* of *createField* (conflicting chunk A) and a conflict between their *method invocations* (conflicting chunk B). In this case, resolving chunk A before chunk B is advised, since the choice of *method signature* decides the subsequent choices of *method invocation*. Table 11 further lends support, showing that 46% of failed merges with more than 1 conflicting chunk exhibited dependencies. Returning to the ANTLR4 example in Figure 23, we see that the failed merge involves two conflicting chunks. Resolving one of these chunks first would determine how the other chunk is to be resolved. Because dependencies exist on the presence of both variables in the *method declaration*, we know which *method* to choose should the *variable* be resolved. On the other hand, because methods have return types that must be compatible with the types of the variable, resolving the conflicts the other way around also would work; that is, once the developer chooses the return type, the rest of the merge conflict should be able to be handled automatically again.

Rather than leaving it up to the developers to decide in which order they resolve multiple conflicting chunks, the association rules presented in Table 12 and Table 13 can be a source of support in the matter. For instance, the pattern *method signature* → *method invocation* says that 84% of the merges with conflicts in *method signature* found in the 5 projects manually analyzed also have conflicts in *method invocation*. This result is also

supported by the automatic analysis, in which the confidence for this rule is 89%, leading to a larger number of merges that fit. Note that the opposite rule (*method invocation* → *method signature*) does not even appear in Table 12 due to having confidence lower than 50%, but in Table 13 we can see that the confidence is 38% and, thus, much lower than the opposite direction. This indicates that changes in the *method signature* imply changes in *method invocation*, and not the other way around – giving directionality to the order in which a developer might want to resolve chunks.

Of course, not every set of conflicting chunks is covered by our set of association rules and, even when they are, the order implied for some set is not necessarily the order in which it should be resolved. This means, once again, that any merge tool that builds on this information takes an advisory rule: instead of automation, it should offer suggestions from which the developer can choose. One way that this could be done is simply through visually highlighting dependencies among conflicting chunks in the merge resolution tool. Another way might be to encode the set of association rules in an expert system that, together with some general rules inspired by the Java grammar, can field queries as to what the best order is for a given subset of conflicting chunks. Again, heuristics are likely needed here.

Finally, we note that, as conflicting chunks are resolved by the developer, more information becomes available that might make it possible for the merge tool to resume merging automatically again. In the case of *import* → *method invocation*, for instance, choosing the *method invocation* should generally suffice for the merge tool to automatically choose the *import* necessary, rather than continuing with the manual approach of asking the developer for which *import* to use. This is equally true in the examples of Figure 22 and Figure 23. In both cases, resolving one of the conflicting chunks should cause the merge tool to resolve the other conflicting chunk automatically by choosing one of the two versions.

Past choices of how conflicting chunks were resolved can inform future choices. Every merge is performed in isolation, meaning that any knowledge from how previous conflicting merges were addressed is not used. This, however, leaves an important opportunity on the table. In the answers of RQ5 and RQ7, particularly, we showed how some developers exhibit certain patterns in their choices and how certain kinds of conflicts are resolved in similar ways. In particular, Table 19 and Table 20 show some association rules relating language constructs with developer decision. For instance, the presence of *import declaration* increases the chances of *concatenation* by 200%, which makes the lift an important metric to make decisions about the future resolutions. Based on this information, a straightforward extension

to existing merge tools would be to identify such historical patterns, and present them to developers when a similar situation appears (i.e., one could imagine a tool that communicates to the developer 'In the past, you resolved 16 conflicts like this, 8 of those by choosing version 2, 6 by choosing version 1, and 2 by writing new code.').

This idea, however, can be taken further: what if it may be possible to develop a 'learning merge tool'? Such learning may play out at different levels. For instance, returning to the observation that it is often helpful to resolve conflicting chunks in a certain order, it may be possible for a tool to learn, based on the kinds of conflicts, association rules, and past ordering choices of the developer, what different preferred orders are for different situations.

As another, and more elaborate example, one can think of merge tools that analyze historical changes in detail and attempt to build patterns from these changes. It may be, for instance, that a merge tool might learn that in 64% of cases a conflict in method parameters exists with 1 of the parameters used in the newly written code, and that that parameter must be renamed accordingly in the code of version 1. Such a pattern is invisible to individual developers, but a learning approach might discover it.

The idea of a learning merge tool is not necessarily limited to a developer, project, or organization. It might even be possible to push it into the realm of the crowd, by building upon the idea that code is regular (Devanbu, 2015) and that repetitive patterns of change exist and might be resolved through the application of repeatable solution patterns. The collective wisdom of the crowd concerning how to merge may well outperform the design of any set of heuristics an individual or team could come up with.

3.8 THREATS TO VALIDITY

Although we made every attempt to reduce the threats to validity to our work as much as we could, a few uncontrolled factors may have influenced the observed results.

Regarding internal validity, we observe that language constructs and developer decisions were extracted manually for the manual analysis, which may have inadvertently introduced data collection errors. To mitigate this risk, we (after the fact) double-checked every language construct and developer decision. Furthermore, we performed an automatic analysis in which language constructs were identified by a program, eliminating the chances of a researcher wrongly cataloguing a given construct. Nevertheless, a few differences can be observed if we compare the collection of language constructs in the manual and automatic analyses. For instance, the manual analysis made no difference between *attributes* and *variables*, while these are handled as distinct constructs in the automatic analysis. On the

other hand, we note that tallying language constructs and developer decision is not a subjective task – a construct is there, or not, and a decision is one of six that could be made. As compared to coding a conversation, for instance, much less risk exists for biasing results.

With respect to construct validity, which refers to misalignments between a study's intent and its design, all analyzed data stems from real, open source software projects, capturing failed merges that actually happened and were handled by developers. Therefore, we are able to report on situations that happened in practice – documenting what we intended to observe: how developers address merge conflicts. On the other hand, our study only observes 'after the fact' outcomes of failed merge resolution, not the strategies a developer uses leading up to the eventual resolution. We, thus, may still miss important information. At the same time, we believe that our focus on capturing resolutions in great detail adds an important dimension to the literature on merge conflicts. Finally, we cannot guarantee that all the merge cases performed in the projects were analyzed. Git has a rebase command (Chacon, 2009) that is able to rewrite the history of projects and, as a consequence, the developers can hide part of the projects history that can comprise merges.

Regarding conclusion validity, which concerns the correctness of findings, our corpus, even with thousands of projects, led to a number of association rules having small support. For instance, Table 20 presents support of up to 2%. Therefore, some of these rules may not hold in different projects profiles. On the other hand, we have identified a set of patterns with frequent occurrences of which are difficult to accredit to chance.

Finally, for external validity we note that the ability to generalize our findings is restricted by the common characteristics across our selected projects: all of them are open source projects written in Java. Thus, we cannot generalize our results to industrial projects or open source projects written in C++, for instance. Our analysis, though, has shown that patterns exist in our sample, and we strongly believe similar kinds of patterns may be present in other projects and languages. Thus, we recognize the need to confirm all of our findings with a broader study involving other projects profiles, changing the programming language and, maybe, using projects from the industry instead of open source software.

3.9 FINAL REMARKS

In this chapter, we analyzed by hand more than a thousand merges from five open source projects, selected the merges that led to a conflict, and catalogued the conflicting chunks, the language constructs involved in the conflicting chunks, and the resolution strategies that developers used to address each failed merge and its chunks. We then examined

the data from a number of different perspectives, including the number of conflicting chunks per failed merge, the size of the chunks, the language constructs that are part of the conflicting chunks, the patterns in language constructs that are present inside and across chunks, the developer decisions, the difficulty level of kinds of conflicting chunks, the patterns between language constructs and developer decisions, and the correlation among different repository metrics. In addition, to mitigate the external threat to validity we performed a similar analysis automatically, which enabled us to collect information from almost 3 thousand projects, summing up 25,328 failed merges and 175,805 conflicting chunks.

From our analysis, it becomes clear that an all-purpose, general merge technique may never be reached: too much variability exists in the developer decisions being made in otherwise similar merge conflicts. Semi-automated merge techniques, combined with tailored heuristics for handling individual kinds of conflicts that are part of larger conflicts, is where we believe the direction of future work lies. This belief is rooted in the facts that: (i) 87% of conflicting chunks had all the information needed to resolve them; (ii) 63% of the failed merges that have more than 1 chunk have dependencies among chunks; and (iii) some kinds of conflicts are frequently resolved in the same way (see Table 19 and Table 20). Especially the second recommendation seems to be promising in leading to what would go beyond today's approaches. We currently have no support for ordering conflict chunks during merge, and this is a problem especially for the most complex merge cases, with dozens of conflicting chunks that depend on each other. We faced this problem and implemented a proof-of-concept tool for ordering conflicting chunks according to their dependencies, as discussed in Chapter 4.

CHAPTER 4 – AN APPROACH TO ORDER CONFLICTING CHUNKS CONSIDERING SYNTACTICAL DEPENDENCIES

4.1 INTRODUCTION

As observed in the former chapter, about 60% of the failed merges have two or more conflicting chunks. In addition, 46% of the failed merges having more than one chunk in the projects that were subject to manual analysis[7] have dependencies among their chunks. Therefore, having multiple conflicting chunks is frequent and the resolution of one conflicting chunk may help on the resolution of the subsequent ones. However, current VCS present the chunks in the order they appear in the source code files, disregarding syntactical dependencies.

Figure 35 shows two dependent chunks in a merge of the project Lombok: a chunk has a conflict in the *method declaration* (conflicting chunk A) and the other has a conflict in the *method invocation* (conflicting chunk B). In this case, resolving the *method declaration* first gives knowledge to the developer to resolve the *method invocation*, as the *method invocation* should be compatible with the previously resolved *method declaration*. As a method may have multiple invocations, resolving the *method declaration* first could support the resolution of several other chunks.

```
}
```

version 1	*version 2*
`private static FieldDeclaration createField(LoggingFramework framework, Annotation source, ClassLiteralAccess loggingType, String logFieldName, boolean useStatic) {`	`public static FieldDeclaration createField(LoggingFramework framework, Annotation source, ClassLiteralAccess loggingType, String loggerCategory) {`
`int pS = source.sourceStart, pE = source.sourceEnd;`	

Conflicting chunk A

```
ClassLiteralAccess loggingType = selfType(owner, source);
```

version 1	*version 2*
`FieldDeclaration field = createField(framework, source, loggingType, logFieldName, useStatic);`	`FieldDeclaration field = createField(framework, source, loggingType, loggerCategory);`
`fieldDeclaration.traverse(new SetGeneratedByVisitor(source), typeDecl.staticInitializerScope);`	

Conflicting chunk B

Figure 35. Dependent chunks of merge f956ba from Lombok.

Considering syntactical dependencies, we believe that the current interactive merge would be improved generating better assistance to the developers that are responsible for

[7] We did not perform this analysis automatically because it demands an extensive computational effort for extracting the abstract syntactic trees and detecting the dependencies.

merging different versions. Thus, this approach can be used inside IDEs or even in the traditional VCS that would be able to present the chunks considering syntactical dependencies. Consequently it would help developers that previously did not have any clue about the best order to resolve the chunks comprised by a failed merge.

In this chapter, we embrace the problem of ordering the resolution of chunks according to their dependencies as an example of how findings of the previous chapter can motivate the research and development of novel merge tools or improve the support provided by existing merge tools. We propose an approach that identifies dependencies among chunks and indicates a resolution order, aiming at resolving dependencies first, as they may help to resolve the dependent chunk. Our approach has the following steps: (1) extract the AST from the versions in conflict, (2) search in the AST for language constructs that belong to the conflicting chunks, (3) identify dependencies among chunks by identifying dependencies among the language constructs that belong to each chunk, and (4) order chunks according to their dependencies.

We analyzed a set of 31 randomly selected failed merges to evaluate the proposed approach. We compared the sequential order of chunks provided by the VCS with an order that considers the dependencies among chunks. As dependent variables, we observed (1) how many conflicting chunks could have been assisted considering the previous resolution decisions if a proper resolution order were applied, and (2) how many chunks were between two dependent conflicting chunks, which we call *noise*. Our results show that an approach that takes chunk dependencies into account can improve merge resolution when contrasted with an approach that uses the order provided by traditional VCS (which we call *sequential order*).

This chapter is organized into four sections including this one of introduction. Section 4.2 introduces the main concepts explored in the approach to order chunks, the extraction of dependencies, and the algorithm proposed to order the chunks comprised in a failed merge. Section 4.3 describes the evaluation of the proposed approach that is mainly composed of a study case and a broader evaluation considering a set of 31 failed merges. Finally, Section 4.5 concludes this chapter presenting our findings.

4.2 CHUNKS ORDERING

In this section, we discuss the main concepts used by the proposed approach: the kinds of dependencies extracted from language constructs, the creation of a dependency graph to represent the dependencies among the conflicting chunks, measures that allow comparing two

distinct conflicting chunk orders, and the algorithm that suggests a resolution order based on the dependency graph.

4.2.1 KINDS OF DEPENDENCY

The current version of the proposed approach considers three kinds of dependencies.

- **Method Declaration vs. Invocation (MDI)**: Dependency that occurs when a conflicting chunk has a *method invocation* and a second chunk has a *method declaration* for the same method. Figure 35 shows an example of this kind of dependency;
- **Attribute Declaration vs. Usage (ADU)**: Dependency that happens when a conflicting chunk has an *attribute usage* and a second chunk has an *attribute declaration* for any given attribute. Figure 36 exemplifies this kind of dependency. In this figure, conflicting chunk A shows the declaration of *attribute implementations* and chunk B uses this *attribute* to initialize the *variable klass*;
- **Variable Declaration vs. Usage (VDU)** – Dependency that takes place when a conflicting chunk has a *variable usage* and a second chunk has a *variable declaration* for the same variable. Figure 37 shows an example with *variable declarations nextTxn* and *next_init* in conflicting chunk A and the respective *variable usages* in chunk B.

```
public String entityResolver = null;
public String uriResolver = null;
public String errorListener = null;
```

version 1	*version 2*
`public Hashtable<QName,Class> implementations = new Hashtable<QName,Class> ();`	`public Hashtable<QName,Class<?>> implementations = new Hashtable<QName,Class<?>> ();`

```
public Hashtable<String,String> serializationOptions = new Hashtable<String,String>();
public LogOptions logOpt = LogOptions.WRAPPED;
public Vector<String> extensionFunctions = new Vector<String>();
```

Conflicting chunk A

```
public boolean isStepAvailable(QName type) {
  if (implementations.containsKey(type)) {
```

version 1	*version 2*
`Class klass = implementations.get(type);`	`Class<?> klass = implementations.get(type);`

```
    try {
      Method method = klass.getMethod("isAvailable");
      Boolean available = (Boolean) method.invoke(null);
```

Conflicting chunk B

Figure 36. Example of ADU extracted from merge 00b04c of project XML Calabash[8].

[8] https://github.com/ndw/xmlcalabash1

```
if (debug.val)
  LOG.debug(String.format("Starting %s thread", this.getClass().getSimpleName()));
```

version 1	*version 2*
`AbstractTransaction nextTxn = null;` `int added = 0;`	`AbstractTransaction next_init = null;`

```
while (this.stop == false) {
  try {
    nextTxn = this.initQueue.poll(THREAD_WAIT_TIME, THREAD_WAIT_TIMEUNIT);
```

Conflicting chunk A

```
  }
    // Requeue mispredicted local transactions
```

version 1	*version 2*

`if (`
`  (`
`  nextTxn == null`
`  ||`
`  added > CHECK INIT QUEUE LIMIT`
`  )`
`  &&`
`  this.restartQueue.isEmpty() == false) {`
	`if(`
	`  next_init == null`
	`  &&`
	`  this.restartQueue.isEmpty() == false) {`

```
      this.checkRestartQueue();
  }
} // WHILE
```

Conflicting chunk B

Figure 37. Example of VDU extracted from merge 00448a of project H-store[9].

4.2.2 DEPENDENCY GRAPH

We conceived a structure named *dependency graph* to show the complete set of dependencies among chunks. We perform the following steps to build this structure for a given set of artifacts in which conflicts were observed: (1) chunks identification, (2) language constructs identification, (3) dependency matrix creation, and (4) dependency graph creation. Figure 38 illustrates this sequence of steps. The resulting graph is used as input to generate the best chunk ordering for merge resolution.

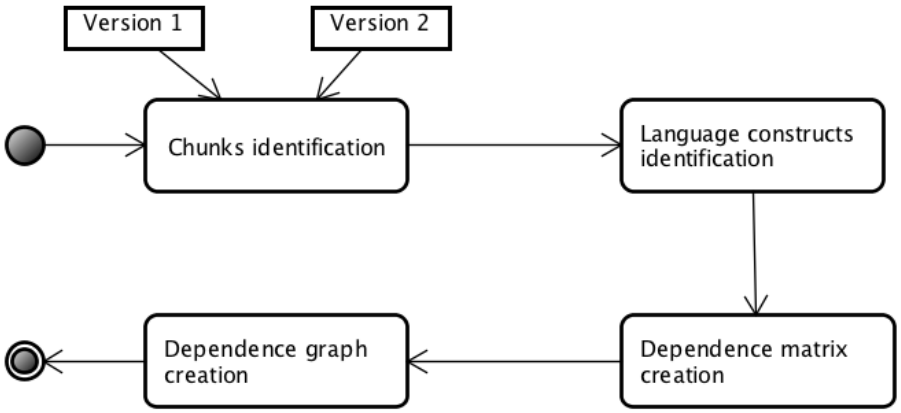

Figure 38. Activities executed to extract the dependency graph.

- **Chunks identification** – By definition, a failed merge comprises one or more conflicting chunks. Thus, given the versions (*i.e.*, *version 1* and *version 2*) to be merged, this step

[9] https://github.com/apavlo/h-store

identifies the chunks that are part of a failed merge to detect dependencies among each pair of chunks;

- **Language constructs identification** – In this step, we extract the AST for *version 1* and *version 2* of each conflicting chunk. Then, the AST are queried to identify the *method, attribute, and variable declarations* that may be part of the chunks, as well as the *method invocation*, *variable*, and *attribute usages* in these chunks;

- **Dependency matrix creation** – This step identifies dependencies among chunks and builds a dependency matrix. The dependency matrix has N columns and N rows, where N is the number of chunks. Each cell is filled with the kinds of dependencies that exist between the chunk represented by its row and the chunk represented by its column. Table 22 presents a dependency matrix for five chunks from a failed merge that was observed on project *Spring Data Neo4j*[10]. This table can be interpreted as "column C depends on row R due to the set of dependencies D presented in the cell on which row R finds column C". For instance, chunk CC3 has an attribute declaration usage (ADU) dependency with CC2;

- **Dependency graph creation** – Based on the dependency matrix, the last step generates the visualization of the dependencies by producing a directed graph. In this representation, a transition from node A to node B indicates that conflicting chunk A depends on conflicting chunk B due to a dependency whose kind is described in the edge's label. Figure 39 shows the dependency graph that represents the matrix in Table 22.

Table 22. Dependency matrix for merge 042b1d5 of project Spring Data Neo4j.

	CC0	**CC1**	**CC2**	**CC3**	**CC4**	**CC5**
CC0						
CC1						MDI
CC2				ADU		
CC3		MDI				
CC4						ADU
CC5						

[10] https://github.com/spring-projects/spring-data-neo4j

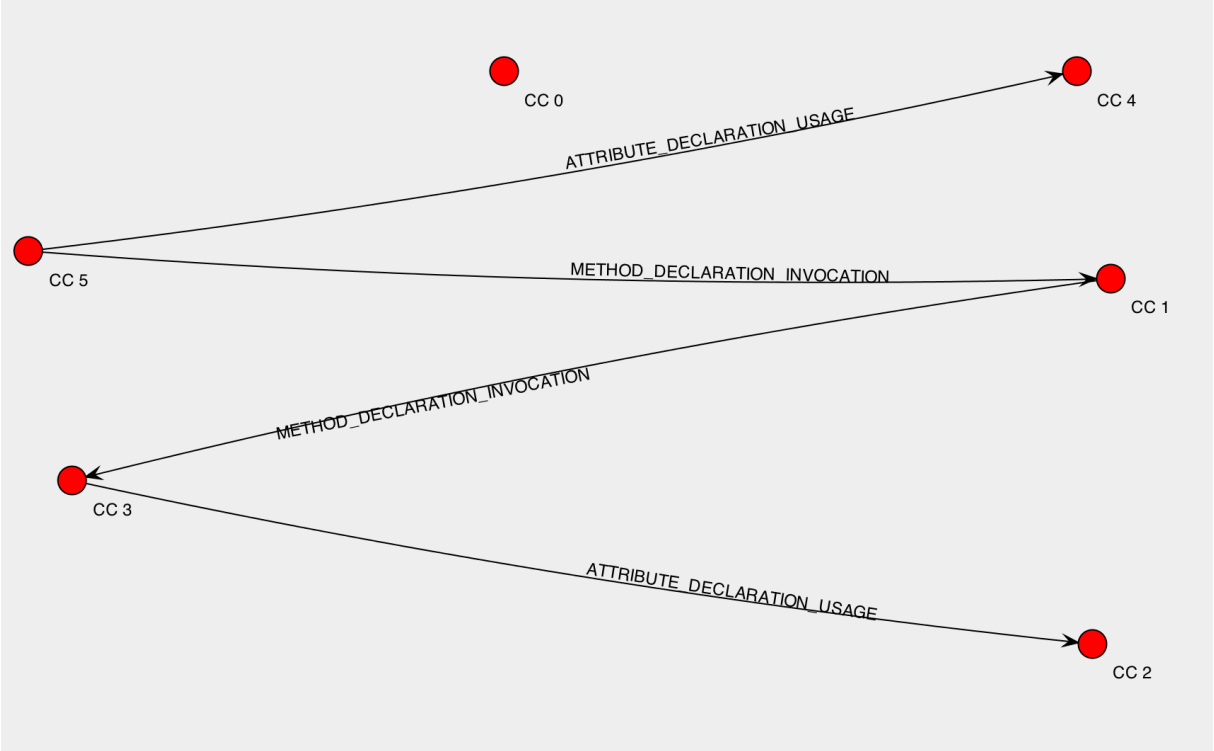

Figure 39. Dependency graph for merge 042b1d5 of project Spring Data Neo4j.

4.2.3 MEASUREMENTS OF IMPROVEMENT

To compare two distinct chunk orders, we consider two measures that can be collected from failed merges comprised of more than a single conflicting chunk: *assistance* and *noise*.

When a merge is resolved manually, developers can take advantage of the order that chunks appear to use the knowledge acquired in previous resolutions for helping the future decisions. *Assistance* measures how much a specific order of chunks helps the developers during merge resolution. For instance, assuming the case in which a conflicting chunk comprises a *method declaration* and another has a *method invocation* to the same method, the resolution of the first chunk could provide information to resolve the second chunk. *Assistance* values range from zero (no *assistance* – when none of the chunks are benefited by the resolution of previous chunks) to one (meaning total *assistance* – when all the chunks are benefited by the resolution of previous chunks). By definition, the first chunk in all the sequences has unassigned value (*i.e.*, it is not considered for computing *assistance*), given that it will never receive assistance due to the absence of a previous chunk.

More formally, let C be the set of conflicting chunks comprising a failed merge, and accept that $|C| = n$. Let M be the $n \times n$ dependency matrix for that failed merge and assume that $M[c_i, c_j] = 1$, $1 \leq i \leq n$, $1 \leq j \leq n$, if and only if c_j depends on c_i due to a MDI, ADU, or VDU; otherwise, $M[c_i, c_j] = 0$. Let S be a complete order of C and let $P: S \times c_i \rightarrow \mathbb{Z}$ be an

operator that returns the position of a given conflicting chunk c_i within a given sequence S. Assume that S[i] returns the chunk that occupies the i^{th} position in the sequence S. Finally, let D be a $n \times n$ matrix that represents whether any given chunk c_i generates knowledge that assists in the resolution of chunk c_j. $D[c_i, c_j] = 1$, $1 \leq i \leq n$, $1 \leq j \leq n$, if and only if c_j can use information from c_i to find its most adequate resolution; $D[c_i, c_j] = 0$, otherwise.

Deciding whether the resolution of a conflicting chunk can provide information to help resolving a second chunk requires analyzing the code in both versions. The existence of one or more dependencies between these chunks is an initial indication that one chunk might help resolving the other, but in some cases the former might not have enough information to be useful. For instance, a given chunk A may use a *variable* defined in a second chunk B, but besides declaring this variable, chunk B does not provide any clue to help a developer responsible for resolving the failed merge to understand what the developer who performed the change meant with it. In cases such as this D(A, B) is equal to zero, despite of the existence of a dependency between chunks A and B.

Currently, the manual intervention of a researcher is required to determine whether resolving a chunk helps in the resolution of another and build the D matrix. The *assistance* provided by S can be thus calculated.

$$assistance(S, c_j) = \frac{\sum_{i=1}^{P(c_j)-1} D[S[i], c_j]}{\sum_{i=1}^{n} M[c_i, c_j]} \quad (1)$$

$$assistance(S) = \frac{1}{n-1} \sum_{j=2}^{n} assistance(S, c_j) \quad (2)$$

Equation (1) calculates the *assistance* provided by sequence S to a given chunk c_j. It does so by determining the ratio between the number of chunks that help to resolve the target chunk c_j and the number of chunks that chunk c_j depends on. Then, equation (2) sums all individual assistances and divides by the number of chunks able to assist the following chunks ($n - 1$), thus normalizing the measure in the [0, 1] interval. Note that the first chunk in the sequence is not considered in the calculation (it is represented by the symbol "-") since there is no previous resolved chunk to provide any clue to help on its resolution.

As all human beings, software developers have short-term memory limitations (Murdock Jr., 1972) and may forget how a given chunk was resolved when addressing a different chunk that depends on the former. *Noise* measures the distance, or the number of chunks a developer resolves, in between two chunks that have dependencies. In the best scenario, this value would be zero, that is, two dependent chunks would be resolved in a contiguous sequence. On the other hand, if there are multiple chunks between two dependent

chunks, the *noise* will be up to one, indicating that many unrelated conflicting chunks may lead the developer to forget the decisions made before. *Noise* cannot be calculated when there is no *assistance* and is represented by the symbol "-" in these cases.

More formally, let PF: $S \times c_i \rightarrow \mathbb{Z}$ be an operator that returns the position of the first chunk in sequence S which c_i depends on. Also, let ND: $S \times c_i \rightarrow \mathbb{Z}$ be an operator that counts the number of chunks between the position PF(S, c_i), inclusive, and the position occupied by chunk c_i, exclusive, that does not help the resolution c_i. Thus, the noise for a given chunk c_j is calculated using equation (3) in which the "-" symbol represents *unassigned*, while equation (4) calculates the noise for a given sequence S.

$$Noise(S, c_j) = \begin{cases} \frac{ND(S,c_j)}{P(S,c_j)-PF(S,c_j)}, & if\ assistance(S, c_j) > 0 \\ -, & otherwise \end{cases} \quad (3)$$

$$Noise(S) = \begin{cases} \frac{1}{AS(S)} \sum_{i=2}^{n} Noise(S, c_i)\ , & if\ AS(S) > 0 \\ -, & otherwise \end{cases} \quad (4)$$

$$AS(S, c_j) = \begin{cases} 1, & if\ assistance(S, c_j) > 0 \\ 0, & otherwise \end{cases}$$

$$AS(S) = \sum_{c_j \in S} AS(S, c_j)$$

4.2.4 ORDERING ALGORITHM

Based on the dependency graph extracted from the conflicting chunks that are part of a given failed merge, the proposed algorithm, called *Context-aware ordering*, generates an order in which the chunks should be addressed by the developers while resolving the merge.

Figure 40 shows the *Context-aware ordering* algorithm. It receives the dependency graph as input and returns an ordered list of chunks. It identifies the order under which the chunks should be resolved based on their dependencies and groups related chunks together to exploit the short-term memory of the developer while resolving subsequent chunks. For instance, if chunk A depends on chunk B, chunk A will be resolved immediately after chunk B unless it has other dependencies yet unresolved. The algorithm controls the number of previously resolved dependencies for each chunk and this number is used to prioritize the chunks that have the fewest number of pending dependencies. Therefore, the more dependencies of a given chunk were resolved, the more priority this chunk will have in the order being built. The algorithm treats cases where circular dependencies among chunks take

place by picking the first chunk in the physical order they are presented in the files when there are several chunks with the same number of pending dependencies.

```
Input:
- Dependency graph

Output:
- List of chunks

Algorithm:
Set all the chunks as unvisited

For each unvisited chunk
      Calculate the number of dependencies
      Set the number of resolved dependencies to zero

While there are unvisited chunks
      Select the chunk c with the lowest number of dependencies and
                              the highest number of resolved dependencies
      Mark c as visited
      Add c to the list
      For each chunk that depends on c
            Update the number of dependencies
            Update the number of resolved dependencies
```

Figure 40. A pseudo-code for the Context-aware ordering algorithm.

Considering the example reported in Figure 39, the *Context-aware ordering* algorithm produces the following sequence of chunks: CC0, CC2, CC3, CC1, CC4, and CC5. As can be observed in this sequence, there is no *noise* between the resolution of two dependent chunks: after resolving CC2, CC3 is presented to the developer; after resolving CC3, CC1 is presented to the developer; and after resolving CC1 and CC4, CC5 is presented to the developer.

4.3 CASE STUDY

In this section, we present a didactical example manually executing the Context-aware algorithm to show how the ordering is done and how the metrics (*assistance* and *noise*) are calculated. This example was extracted from the project Spring Data Neo4J, a project that offers advanced features to map annotated entity classes to the Neo4J Graph Database. This merge is identified by the SHA-1 042b1d and results from merging revisions 3ba54f and 4a8f40, which ends up in a failed merge comprising six conflicting chunks that are represented from Figure 41 to Figure 46. We refer to these chunks as CC0, CC1, CC2, CC3, CC4, and CC5 from now on. They reside in the files *CypherQueryBuilder.java* (CC0 and CC1), *StartClause.java* (CC2 and CC3), and *CypherQueryBuilderUnitTests.java* (CC4 and CC5).

CC0 represents a conflict among *import declarations* that is found in the first lines of source code files. In this chunk, some *import declarations* were changed in *version 1,* while

others were removed in *version 2*. Although our current implementation does not identify this kind of dependency, *imports* can be in conflict due to conflicts among *method invocations* and other language constructs whose use is enabled through *imports*.

```
import org.springframework.data.neo4j.support.Neo4jTemplate;
import org.springframework.data.repository.query.parser.Part;
```

version 1	*version 2*

`import org.springframework.util.Assert;`
`import java.lang.annotation.Annotation;`
`import java.util.ArrayList;`
`import java.util.List;`
`import static org.springframework.util.StringUtils.*;`	

```
/**
 * Value object to create Cypher queries.
```

Figure 41. CC0 – Conflict in *import declarations*.

CC1 represents a conflict involving more language constructs (*annotation*, *attribute*, *comment*, *if statement*, *method declaration*, *method invocation*, *method signature*, *return statement*, and *variable*). Thus, it represents a good candidate to be a source of dependencies to others chunks and can depend on other chunks as well. As we currently capture dependencies among *attribute*, *variable*, and *method declarations* and their usage, we identified that chunk CC1 depends on CC3, which has a *method declaration* that is used in line *startClause.merge(partInfo);* (in bold in CC1).

```
public CypherQueryBuilder addRestriction(Part part) {
  PersistentPropertyPath<Neo4jPersistentProperty> path =
    context.getPersistentPropertyPath(part.getProperty());
```

version 1	*version 2*

`…`
`private boolean addedStartClause`
`  (PartInfo partInfo) {`
`  if (!partInfo.isIndexed()) return false;`
`  for (StartClause startClause : startClauses) {`
`  PartInfo startPartInfo =`
`  startClause.getPartInfo();`
`  if (!partInfo.sameVariable(startPartInfo))`
`  continue;`
`  if (!partInfo.sameIndex(startPartInfo))`
`  return false;`
`  **startClause.merge(partInfo);**`
`  return true;`
`  }`
`  startClauses.add(new StartClause(partInfo));`
`  return true;`
`}`
`…`
`@Override`
`public String **toString()** {`
`  …`
	`  query.addPart(part, path);`
	`  return this;`
	`}`

```
public CypherQueryDefinition buildQuery(Sort sort) {
  return query.withSort(sort);
```

Figure 42. CC1 – Conflict in annotation, attribute, comment, if statement, method declaration, method invocation, method signature, return statement, and variable.

CC2 presents the declaration of two conflicting *attributes*, *partInfo* and *partInfos*, that have different types, but similar names, which enforces the probability that they have the same goal. As this chunk does not invoke or use any of the *methods*, *variables*, or *attributes*

declared in this project, there are no dependencies on other chunks. However, other chunks may depend on its conflicting *attributes*. Furthermore, in our approach we treat *constructor invocation* as *method invocation* and *constructor declaration* as *method declaration*.

```
// TODO id-startclause, index-startclause (exact,point,fulltext)
class StartClause {
```

version 1	*version 2*
`protected final PartInfo partInfo;`	`private final SortedMap<Integer,PartInfo> partInfos =new TreeMap<Integer, PartInfo> ();`

```
/**
* Creates a new {@link StartClause} from the given {@link Neo4jPersistentProperty}, variable and the given
```

Figure 43. CC2 – Conflict in *attribute* and *method invocation*.

CC3 presents a conflicting chunk with the following language constructs: *for statement*, *method declaration*, *method signature*, *return statement*, and *throw statement*. This chunk accesses the attributes *partInfos* and *partInfos*, which were declared on CC2, and has a *method declaration* called by chunk CC1.

```
return IteratorUtil.first(partInfos.values());
}
```

version 1	*version 2*

`public void merge(PartInfo partInfo) {`
`  throw new UnsupportedOperationException`
`  ("Merge is not supported");`
	`public boolean merge(PartInfo partInfo) {`
	`  for (PartInfo info : partInfos.values()) {`
	`  if (info.sameIdentifier(partInfo)`
	`  && info.sameIndex(partInfo)) {`
	`  continue;`
	`  }`
	`  return false;`
	`  }`
	`  this.partInfos.put`
	`  (partInfo.getParameterIndex(), partInfo);`
	`  return true;`
	`}`
	`public boolean sameIdentifier(PartInfo info) {`
	`  for (PartInfo partInfo : partInfos.values()) {`
	`  if (!partInfo.sameIdentifier(info))`
	`  return false;`
	`  }`
	`  return true;`
	`}`
	`public boolean sameIndex(PartInfo info) {`
	`  for (PartInfo partInfo : partInfos.values()) {`
	`  if (!partInfo.sameIndex(info)) return false;`
	`  }`
	`  return true;`

```
  }
}
```

Figure 44. CC3 – Conflict in for statement, method declaration, method signature, return statement, and throw statement.

CC4 shows a conflict involving *attribute declarations*. This chunk does not use language constructs declared in other chunks. Consequently, it does not depend on other chunks. However, it has an *attribute declaration* that is used as a constant, which can generate dependency in other chunks.

```
CypherQueryBuilder query;
  private final String CLASS NAME = Person.class.getSimpleName();
```

version 1	*version 2*
`private final String DEFAULT_START_CLAUSE = "start `person`=node: types (className=\"" + CLASS NAME + "\")";`	`private String DEFAULT START CLAUSE = "START `person`=node: types (className=\"" + CLASS NAME + "\")";`

```
@Before
```

Figure 45. CC4 – Conflict in *attributes* and *initialization*.

Finally, CC5 presents a conflict involving *annotation*, *method declaration*, *method invocation*, and *variable*. This chunk presents a *method declaration* and has *method invocations* and *attribute* usages. It depends on chunks CC1 and CC4. The dependency with CC1 is due to the usage of a *method declaration* shown in bold in CC5 (*query.toString()*). The dependency with CC4 is due to the usage of an *attribute*, also presented in bold in CC5 (*DEFAULT_START_CLAUSE*).

```
  @Before
```

version 1	*version 2*
`assertThat (query.toString(), is(DEFAULT_START_CLAUSE +" where `person`.`age`! = {0} return `person`") ); }`	`final String className = Person.class.getName(); assertThat (query.toString(), is(DEFAULT START CLAUSE +" WHERE `person`.`age`! = {0} RETURN `person`")); } @Test public void createsSimpleTraversalClauseCorrectly() { query.addRestriction(new Part("group", Person.class)); assertThat(query.toString(), is(DEFAULT START CLAUSE + " MATCH `person`<-[:`members`] -`person group` RETURN `person`")); }`

```
@Test
public void buildsComplexQueryCorrectly() {
```

Figure 46. CC5 – Conflict in *annotation*, *method declaration*, *method invocation*, and *variable*.

According to the dependencies among the chunks above, CC1 depends on CC3, CC3 depends on CC2, and CC5 depends on CC1 and CC4. The dependency matrix of this failed merge is represented in Table 22 and Figure 39 shows the dependency graph. The Context-aware ordering algorithm then produces the following sequence of chunks: CC0, CC2, CC3, CC1, CC4, and CC5. In this resolution, all the dependencies are considered, which increases *assistance*. The algorithm also groups chunks that have dependencies, thus decreasing the *noise*. The dependency between CC3 and CC2 has no *noise*, as CC3 appears right after CC2. The next dependency, between CC1 and CC3, occurs because CC1 depends on the *method declaration* in CC3 and, consequently, resolving the method declaration in CC3 can assist in the resolution of CC1. Furthermore, the *noise* is zero because there are not chunks on which CC1 does not depend between CC3 and CC1. Finally, CC5 depends on other two chunks:

CC1 and CC4. Again, CC1 and CC4 give clues to resolve CC5, resulting in *assistance* equals to 1 and *noise* equals to 0, since CC5 depends on all chunks between CC1 and CC5. Thus, as shown in Table 23, the final numbers for the Context-aware algorithm are 50.00% of *assistance* and 0% of *noise*.

Table 23. Metrics extracted by the Context-aware ordering algorithm.

Chunk	Assistance	Noise
CC0	-	-
CC2	0	-
CC3	1	0
CC1	1	0
CC4	0	-
CC5	1	0
Failed merge	60%	0%

4.4 EVALUATION

In this section, we evaluate our approach in terms of the *assistance* it provides to developers resolving conflicts. Moreover, we evaluate if it orders the chunks in a way to exploit the short-term memory of developers, reducing the *noise* between dependencies and dependent chunks. Our evaluation runs over 31 failed merges and 405 conflicting chunks. We compare our approach with a sequential algorithm (introduced in Section 4.4.1) that emulates the chunk resolution order induced by some VCS. We aim at answering the following research questions:

1. Does the Context-aware ordering algorithm increase the *assistance* provided by the Sequential algorithm?
2. Does the Context-aware ordering algorithm reduce the *noise* generated by the Sequential algorithm?

4.4.1 BASELINE ALGORITHM

This section introduces the algorithm used as comparison baseline for our approach. The Sequential algorithm represents the way traditional VCS present a merge to be resolved. For instance, Subversion (Collins-Sussman et al., 2004) has an interactive tool to resolve merges in which the chunks are presented following the order they appear in the files.

The Sequential algorithm receives as input the chunks comprising a failed merge and the files in conflict and generates a list of chunks representing the order in which developers

should resolve the merge. Figure 47 shows the steps performed by the Sequential algorithm. It has an initial step that marks all the chunks as unvisited and then the chunks are selected in the order they appear in the conflicting files.

```
Input:
- Set of chunks
- Set of files

Output:
- List of chunks

Algorithm:
Set all the chunks as unvisited
For each file in files
      While there are unvisited chunks in the current file
            Select the first unvisited chunk c
            Mark c as visited
            Add c to the list
```

Figure 47. A pseudo-code for the Sequential algorithm.

Considering the example reported in Figure 39, the Sequential algorithm produces the following sequence of chunks: CC0, CC1, CC2, CC3, CC4, and CC5. This sequence respects only the dependencies that happen to follow the order of the files (*e.g.*, CC5 depends on CC4, CC5 depends on CC1, and CC3 depends on CC2), but disregards other dependencies (*i.e.*, from CC1 to CC3).

4.4.2 SELECTION OF OPEN SOURCE PROJECTS

We analyzed a large set of open source projects capturing information about the nature of failed merges. One piece of information is the number of conflicting chunks comprised in each failed merges, which enabled us to select merges that fit on the profile that can benefit from this approach: failed merges having at least two conflicting chunks. From the subset of failed merges comprising two or more chunks, we randomly selected a set of 31 failed merges and run our dependency analysis. Table 24 characterizes these 31 failed merges, which are ordered by the number of chunks they comprise, presenting their repository URL, SHA-1, and number of chunks. From now on, each merge will be identified by the number presented in column "#Merge". The failed merges selected for this analysis were collected from 29 different projects having from 2 to 32 conflicting chunks ($\mu = 13.06$, $\sigma = 9.55$). Based on the information of this table, any of the failed merges we studied can be retrieved and analyzed in future research.

Table 24. Failed merges evaluated in this study.

#Merge	Repository URL	Merge SHA	# Chunks
1	https://github.com/RedditAndroidDev/Tamagotchi	21f5e0	2
2	https://github.com/nuxeo/nuxeo	61cde8	2
3	https://github.com/usc-isi-i2/Web-Karma	29d83b	2
4	https://github.com/FasterXML/jackson-databind	059d39	3
5	https://github.com/alexo/wro4j	05d025	3
6	https://github.com/atlasapi/atlas	003639	4
7	https://github.com/AKSW/SAIM	044a3c	4
8	https://github.com/maxcom/lorsource	159d31	4
9	https://github.com/enonic/xp	04176d	5
10	https://github.com/rhomobile/rhostudio	043aa3	5
11	https://github.com/pentaho/modeler	0587bc	5
12	https://github.com/apavlo/h-store	00448a	6
13	https://github.com/spring-projects/spring-data-neo4j	042b1d	6
14	https://github.com/jponge/izpack	5b45fc	6
15	https://github.com/MinecraftForge/MinecraftForge	039d92	8
16	https://github.com/android/platform_packages_apps_camera	04c12f	10
17	https://github.com/fakemongo/fongo	0033c8	13
18	https://github.com/mozilla-b2g/android-sdk	058ab2	17
19	https://github.com/Ineedajob/RSBot	09d43f	17
20	https://github.com/mkarneim/pojobuilder	09b977	18
21	https://github.com/hector-client/hector	010bf7	18
22	https://github.com/jenkinsci/jira-plugin	063259	20
23	https://github.com/openMF/mifosx	01368e	21
24	https://github.com/eclipse/objectteams	08e2a6	21
25	https://github.com/ndw/xmlcalabash1	00b04c	22
26	https://github.com/structr/structr	01c205	22
27	https://github.com/alexo/wro4j	00961b	23
28	https://github.com/xetorthio/jedis	055032	24
29	https://github.com/phenoscape/Phenex	0985bf	30
30	https://github.com/alexo/wro4j	01851e	32
31	https://github.com/OpenMEAP/OpenMEAP	0af9d5	32

4.4.3 DATA COLLECTION

The data extracted from failed merges in this experiment was collected in a semi-automated way. This means that part of the data was collected by an automated approach and another part was collected manually. Having steps that are performed automatically guarantees that all the failed merges are treated equally, which makes the results more reliable

since there are no human interferences. On the other hand, the manual steps are mandatory to compute *assistance*, as the researcher determines whether the information provided by the resolution of previous chunks can support further resolutions.

Regarding the automatic steps, we implemented a tool to extract the dependencies among chunks. It receives as input the revisions to be merged and performs the steps discussed in Section 4.2.2, identifying the conflicting chunks and using the AST to find dependencies between pairs of chunks. Finally, it generates the chunks order for the two algorithms: Sequential and Context-aware ordering. The researcher, who manually extracts the *assistance* metric used to evaluate the algorithms, uses these sequences. The noise metric does not require human intervention and is calculated automatically.

4.4.4 DOES THE CONTEXT-AWARE ORDERING ALGORITHM INCREASE THE ASSISTANCE PROVIDED BY THE SEQUENTIAL ALGORITHM?

Table 25 shows the *assistance* for each failed merge and algorithm. The Context-aware algorithm presents the highest average for *assistance* (36.35% against 22.30%). Actually, in 15 out of 31 failed merges (48.39%) the *assistance* provided by the Context-aware ordering algorithm is higher than that provided by the Sequential algorithm, while in 16 out of 31 merges (51.61%) the *assistance* is precisely the same. This means that the Context-aware ordering algorithm would better assist developers than by the Sequential algorithm in almost half the failed merges analyzed.

Figure 48 shows a boxplot that summarizes the results presented in Table 25. There are two outliers in the analyzed failed merges when considering the Sequential algorithm. The first is the failed merge #16, comprising 10 conflicting chunks and having eight dependencies among these chunks, shows 88.89% *assistance* for both the Sequential and Context-aware ordering algorithms. In this specific case, all dependencies point to the same chunk, which happens to come first in the source code files. The second is the failed merge #1 that comprises 2 chunks, has 1 dependency, and has assistance equals to 100%. This case is interesting because when a failed merge has two chunks, the assistance is 0 or 100% as well as in failed merges #2 and #3, which the assistance using Context-aware algorithm is better.

Table 25. ***Assistance*** **provided by the ordering algorithms (higher values in bold).**

Merge Number	Assistance	
	Sequential	**Context-aware**
1	100.00%	100.00%
2	0.00%	**100.00%**
3	0.00%	**100.00%**
4	0.00%	**50.00%**
5	0.00%	0.00%
6	33.33%	33.33%
7	0.00%	**50.00%**
8	33.33%	33.33%
9	50.00%	50.00%
10	25.00%	25.00%
11	25.00%	**33.33%**
12	40.00%	40.00%
13	40.00%	**60.00%**
14	0.00%	**20.00%**
15	14.29%	14.29%
16	88.89%	88.89%
17	8.33%	**25.00%**
18	25.00%	**43.75%**
19	6.25%	6.25%
20	23.53%	**35.29%**
21	5.88%	**11.76%**
22	21.05%	21.05%
23	10.00%	10.00%
24	15.00%	**20.00%**
25	38.10%	38.10%
26	9.52%	9.52%
27	15.91%	**22.73%**
28	39.13%	39.13%
29	12.07%	**31.03%**
30	8.60%	8.60%
31	3.23%	**6.45%**
Average	22.30%	**36.35%**

By observing Figure 48 we can see a clear difference among the results of *assistance* provided by the Sequential if compared to the Context-aware algorithm. We applied the Wilcox test to the values presented in Table 25 and observed statistically significant difference (p-value = 0.0181) at 95% confidence level. Thus, we conclude that our algorithm has potential to improve the way developers resolve merge nowadays since the current

approach found in VCS does not explore syntactic information. As observed above, in 48.39% of the cases we can take advantage of syntactic dependencies to extract extra information that helps developers while resolving the chunks of failed merges.

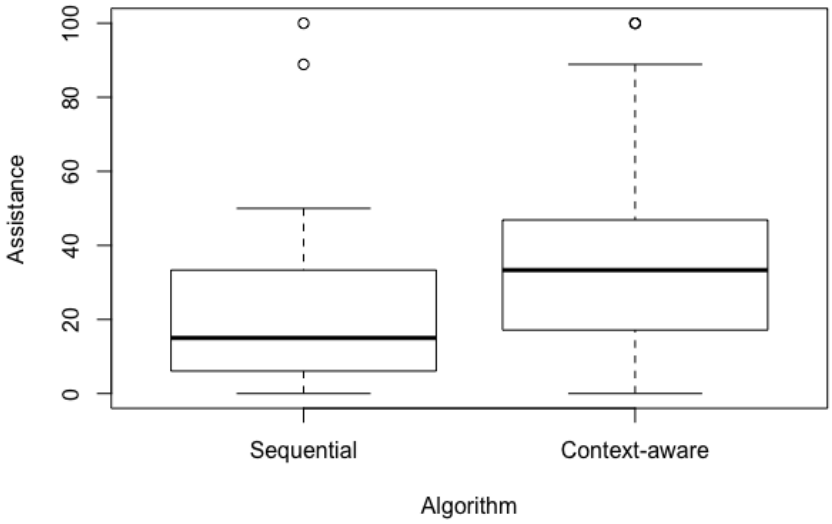

Figure 48. Boxplot for the *assistance* metric over the 31 merges in our sample.

4.4.5 DOES THE CONTEXT-AWARE ORDERING ALGORITHM REDUCE THE NOISE GENERATED BY THE SEQUENTIAL ALGORITHM?

Table 26 shows the *noise* values for the Sequential and Context-aware algorithms. The Context-aware ordering algorithm presents a lower average noise if compared to the Sequential algorithm (10.47% against 25.51%). In 11 out of 31 failed merges (35.48%) the *noise* provided by the Context-aware ordering algorithm is lower than that provided by the Sequential algorithm, in 1 out of 31 merges (3.23%) the *noise* of the Sequential algorithm is lower, in 13 out of 31 (41.94%) the noise is the same for both algorithms, and in 6 cases (19.35%) the Sequential algorithm did not provide any *assistance* at all (unassigned noise).

This information is confirmed by the boxplot presented in Figure 49, which shows a median close to zero for the Context-aware ordering algorithm. Actually, this represents a negative asymmetric distribution, as the median is equal to the minimum value. The only situation in which the Context-aware algorithm has higher noise than the Sequential algorithm is in the failed merge #18 and this situation only happens because the Sequential algorithm provides less assistance than our proposed algorithm leading to lower noise. As *noise* is treated as a secondary metric, the Context-aware algorithm is still better than the Sequential algorithm, since the former just has higher noise because it provides more assistance, which is the main goal of the proposed algorithm. By using the Wilcox test, we observe statistically

significant differences (p-value = 0.0227) at 95% confidence level between the noise generated by the Context-aware ordering algorithm and the produced by the Sequential algorithm. Thus, we conclude that the Context-aware ordering algorithm generates on average more *assistance* and lower *noise* than the Sequential one.

Table 26. *Noise* provided by the ordering algorithms (lower values in bold).

Merge Number	Noise	
	Sequential	Context-aware
1	0.00%	0.00%
2	-	0.00%
3	-	0.00%
4	-	0.00%
5	-	-
6	0.00%	0.00%
7	-	0.00%
8	0.00%	0.00%
9	25.00%	**0.00%**
10	0.00%	0.00%
11	75.00%	**0.00%**
12	25.00%	25.00%
13	25.00%	**0.00%**
14	-	0.00%
15	25.00%	**0.00%**
16	77.14%	**66.03%**
17	0.00%	0.00%
18	**37.50%**	46.26%
19	0.00%	0.00%
20	45.83%	**27.78%**
21	0.00%	0.00%
22	49.11%	49.11%
23	0.00%	0.00%
24	29.17%	**8.33%**
25	12.50%	12.50%
26	0.00%	0.00%
27	34.38%	**20.00%**
28	82.22%	**18.52%**
29	54.33%	**16.67%**
30	40.48%	**23.81%**
31	0.00%	0.00%
Average	25.51%	**10.47%**

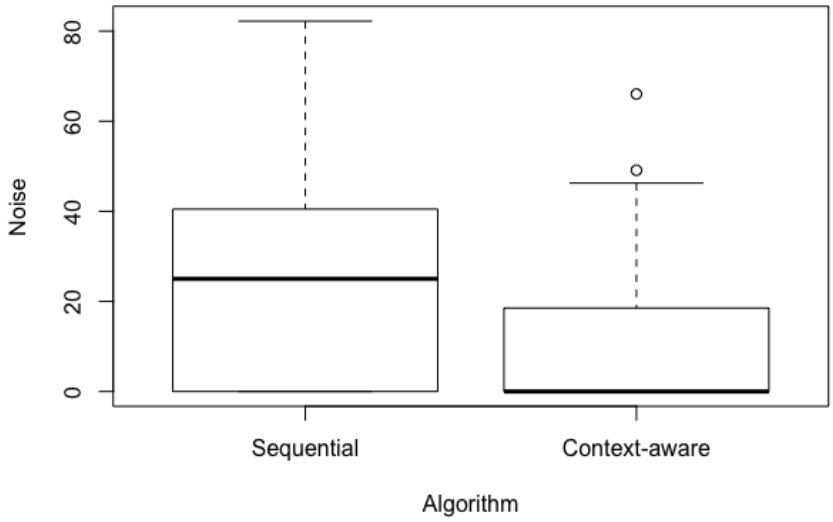

Figure 49. Boxplot for the *noise* metric over the 31 merges in our sample.

4.4.6 THREATS TO VALIDITY

Although we have taken care to reduce the threats to validity to our work, a few uncontrolled factors may have influenced the observed results. Regarding internal validity, it is worth to observe that the metrics *assistance* and *noise* were extracted manually. To mitigate this, we tried to follow the same steps and rules to evaluate all failed merges. Furthermore, we inspected the failed merge more than once to avoid data collection errors.

With respect to construct validity, which refers to misalignments between a study's intent and its design, all analyzed data came from real software projects, capturing failed merges that actually happened and were handled by developers. Therefore, we are able to report on observed situations that happened in practice. However, it was not possible to run an experiment with a real development team, which could give us further feedback to improve the approach and the quality of our analysis.

Finally, regarding external validity, we note that the ability to generalize our findings is restricted by the common characteristics of our selected failed merges: open-source projects coded in Java that comprise from 2 to 32 chunks. Thus, we cannot generalize our results to industrial projects or open-source projects written in C++, for instance. Our analysis, though, shows that Context-aware algorithm is able to assist developers during merge resolution, which justifies the pursuit of additional studies to assess it.

4.5 FINAL REMARKS

This chapter presented an approach to order chunks comprised in failed merges. To do so, we used dependencies extracted from language constructs related to declarations, usages

and invocations, generating a dependency graph that is used to build the order of chunks by using the Context-aware ordering algorithm. This algorithm presents improvements if compared to the one currently used in VCS, which we called Sequential algorithm. Our results show that the Context-aware ordering algorithm presents the highest average *assistance* in 48.39% of the failed merges and only have greater *noise* in 1 failed merge (3.23%). This confirms our expectations that the order in which chunks are resolved can provide relevant information for software developers.

CHAPTER 5 – CONCLUSION

5.1 CONTRIBUTIONS

This work presented an extensive analysis on the nature of software merge conflicts. From the results of this analysis, we devised an approach to assist developers during the resolution of failed merges comprising more than a single conflicting chunk. As part of this work, we studied the related work, designed and executed experiments to analyze the failed merges in a set of open source projects written in Java, and evaluated the potential benefits of the proposed approach for merge resolution. The main contributions of the Thesis are:

1) Analysis of the nature of software merge – a detailed manual study over 5 projects, followed by an automated analysis over 2,731 projects, answering 8 key research questions about where failed merge usually appear, how they are frequently resolved, and their patterns of occurrence. Our main findings are:
   a) Conflicting chunks generally contain all the information required to resolve them – 87% of the chunks do not need extra code beyond those presented in the versions under conflict;
   b) The resolution order of conflicting chunks matters – 46% of the manually analyzed failed merges have dependences between pairs of chunks that can provide knowledge to the resolutions of subsequent chunks; and
   c) Past choices on how conflicting chunks are resolved can inform future choices – information of how a specific kind of conflict was previously resolved can suggest resolutions for subsequent chunks having similar kinds of conflict.

2) Conflicting chunk ordering approach – the design and implementation of a proof-of-concept tool to exploit our finding 1.b discussed before, which could produce an ordering for the conflicting chunks of a failed merge based on their syntactical dependencies. Our main findings after adopting the proposed algorithm are:
   a) The *assistance* when considering chunks dependences increases in 48.39% of the failed merges and
   b) The *noise* in between two dependent chunks reduced in 35.48% of the failed merges.

5.2 LIMITATIONS

Despite having the aforementioned contributions, there are some limitations in our approach that can be treated in the future to improve the robustness of the analyses and provide better results than those reported here.

5.2.1 WE JUST ANALYZED SOURCE CODE

The first limitation in our approach is that the results from both the manual and automatic analyses of failed merges are based only on the source code and the metadata available into the repository of the open source projects. Thus, there are situation in which an interpretation of the failed merge cannot precisely represent the real intention of developers or even be confirmed by them. This limitation can be reduced through the analysis of the messages written by developers in commits. These messages may provide extra information on the real intention of developers. Other possibility would be to show the conflicting chunks or the whole failed merge to the actual developers that worked on the projects, asking for a validation if a given heuristic or checking whether other findings make sense for his/her case. Moreover, we could evaluate current approaches to validate if the resolution is effective or not considering the developer opinion. We have not interviewed developers or presented our findings to those developers involved in the projects we have analyzed. Thus, in the future we would like to ask questions to developers about specific chunks or failed merges to see the rationale that led to their resolution. We may find that some of resolutions are recurrent and follow best practices adopted by the development team. If they are found and happen to be shared by a large group of developers, such practices could be implemented in a specialist system which would decide for the most suitable resolution based on the characteristics of the conflicting chunk and/or failed merge.

5.2.2 WE JUST CONSIDERED THREE KINDS OF DEPENDENCIES IN THE PROOF-OF-CONCEPT TOOL

Our proof-of-concept tool to order chunks, which in the current version handles only three kinds of dependencies, also has limitations. Many other kinds of dependencies could be explored, such as: *import declaration* and *method invocation; annotation declaration* and *annotation usage*; *method interface and method declaration*; among others. These dependencies could enable us to improve *assistance* and, consequently, increase the expressiveness of our results. Another limitation of our approach is that we can only extract the AST from Java source code, limiting its application in multi-language projects or in non-

Java projects. Although this challenge is well-known in the syntactical merge approaches, it can be handled by inserting new kinds of dependencies that are not related to syntactical information, for instance, extracting patterns from files that are frequently changed together (i.e., logical coupling) and using these rules to order the conflicting chunks. Other alternative is to extend the approach to deal with other programming languages by adding extractors for these languages and finding language-specific dependencies.

5.2.3 THE CHUNK ORDERING APPROACH DOES NOT HAVE A GRAPHIC INTERFACE

The chunks ordering algorithm is not currently integrated to an IDEs or does not have a standalone user interface through which developers could evaluate the usability of our approach. As discussed in the Chapter 4, our approach receives as input the revisions that will be merged and, when a failed merge takes place, it returns the sequence of chunks in the order that they should be resolved. This interface may not attract people to use the approach. Thus, we do not have feedbacks about its usefulness, even though we could show in our experiments that it assists developers during failed merge resolutions.

5.3 FUTURE WORK

The analysis reported in this document and the proof-of-concept tool opened some opportunities for future work. These ideas are reported in the following.

5.3.1 CROSS-LANGUAGE ANALYSES

Our analyses were focused on projects written in Java, and our goal was to analyze just the occurrences of language constructs in this language. However, a natural evolution of our work is to analyze if the patterns hold for other single-language projects or even if multi-language projects also face similar conflicting and resolution patterns. The former analysis could rely on statistics about the usage of programming languages to select the number of projects from each language. Then, the results of different languages could be compared to determine if there are patterns for certain programming language groups or language-independent patterns. The latter analysis can be conducted over a randomly selected set of projects, showing if the number of cross-language failed merges (that is, failed merges comprising conflicting chunks written in different programming languages) is relevant.

5.3.2 SEARCH-BASED SOFTWARE ENGINEERING FOR MERGE RESOLUTION

Chapter 3 reveled that 87% of the chunks have all content necessary to resolve them in the lines of code that represent *versions 1* and *2*. Therefore, Search-based Software Engineering approaches that create permutations of these lines of code and test whether they compile and pass the test suite would help developers by providing a set of candidate resolutions from which he/she could pick one to resolve the failed merge. This might be promising since 87% of the chunks would be resolved in a semi-automatic way and the remaining 13% would have an initial resolution on which developers could use as baseline. Of course, there are challenges to this approach. For instance, the number of candidate resolutions can increase exponentially according to the number of lines in *version 1* and *2*. Equation (5) shows the number of candidate resolutions given LOC1 and LOC2, respectively representing the LOC from *version 1* and *2*. Therefore, such approach should consider strategies to prone candidate resolutions that are not likely to have good results and, as a consequence, reduce the number of candidates to generate the correct resolution in an appropriate time. Moreover, this work could explore exact algorithms to find the best resolution when the number of candidate resolutions is small, and metaheuristics when there is a high number of candidate resolutions.

$$\sum_{i=0}^{LOC1} \sum_{j=0}^{LOC2} \binom{LOC1}{i} \binom{LOC2}{j} \binom{i+j}{i} \quad (5)$$

5.3.3 RESOLUTION OF CHUNKS BASED ON PREVIOUS RESOLUTION

Our analyses reveled that there are relations between language constructs and developer decisions. Moreover, the evaluation of our proof-of-concept tool showed that the order in which chunks are resolved could provide *assistance* during the resolution of further chunks. Based on these facts, previous resolution could be used to predict future resolutions automatically or at least to exploit the order of chunks to automatize future resolutions based on the previous one. The former approach would rely on the association rules that relate language constructs and developer decisions. Thus, a set of rules would be extracted from projects developed by the same developers, organizations, or well-known projects that probably adopt best practices during software evolution. Then, the association rules could be implemented in the form of an expert system which would provide a set of possible resolutions from which the developer would pick the one that resolves the failed merge in the most appropriate way. The latter approach would explore the syntactical dependencies to order the chunks in a way that automatic approaches would take advantage. For instance,

assume two chunks A and B comprised in a failed merge. Chunk A presents a conflict involving *method invocations* and chunk B has a set of *method declarations* in conflict. As additional information, chunks A and B have dependencies since the *method declarations* are implementing the behavior of the *method invocations*. Thus, resolving chunk A gives enough resource to resolve chunk B, which will maintain just the *method invocations* that their *declarations* are still in place.

5.3.4 SUPPORT AWARENESS APPROACHES

Table 17 and Table 18 show that some kinds of conflicts are harder to resolve than others. Additionally, awareness approaches like Palantir (Sarma et al., 2003), Crystal (Brun et al., 2011b), and WeCode (Guimarães; Silva, 2012) try to help developers to detect conflicts before than take place and establish a strategy to avoid failed merges. Thus, an additional information that could improve warnings given by awareness approaches is the difficulty degree that a conflict can have based on the history of the project. Therefore, a developer would have the choice of continuing coding if the difficulty degree is low or try to resolve the conflict otherwise. Thus, developers would be able to better manage their risks.

5.3.5 IS THE GRANULARITY IMPORTANT IN THE CORRELATION WITH NUMBER OF DEVELOPERS?

Table 21 shows that the number of developers in a project has lower correlation with the number of commits, merges, and failed merges. However, this analysis was performed just over the project as a whole. Thus, a future work could consist on performing the same analysis considering packages, files, or maybe method declarations and other syntactical elements as grains. With this study, we could uncover other correlations for specific parts of the project.

5.3.6 ANALYZE THE DIFFICULTY TO RESOLVE CONFLICTING CHUNKS BASED ON LOC

We analyzed the difficulty to resolve conflicting chunks based on the kind of conflict that the conflicting chunk has. However, there are other characteristics that can determine this difficulty, for instance LOC. Thus, a possible future work is to analyze the relation between the number of LOC and the difficulty to resolve a conflicting chunk. Of course, for this analysis, we expect that chunks with higher number of LOC would result in most difficult resolutions than the chunks with lower number of LOC.

5.3.7 IS IT POSSIBLE TO AUTOMATE NEW CODE RESOLUTIONS?

During our analyses, we considered that the developer decisions *version 1*, *version 2*, *combination*, *concatenation*, and *none* are possible to be automated. However, some cases of *new code* are also susceptible to automation. For instance, for a missing *import declaration*, an approach could identify the problem and suggest *import declarations* that can be used in this context. Therefore, a possible future work would be to identify opportunities in the chunks with *new code* and propose heuristics to resolve them.

5.3.8 USE DEEP LEARNING TO IMPROVE MERGE

As we could observe, Table 19 and Table 20 show association rules that connect a set of language constructs to a specific resolution. However, there is no silver bullet to resolve all the merge cases and in some cases the developers can diverge in opinion. Therefore, a future work could be to provide automatic resolutions to developers and allow them to classify if the resolution make sense or not. By doing so, the developers would provide feedback that the tool could learn from and use this knowledge in the following suggestion. This can be used to provide accurate feedback for specific developers or to the rest of the development team.

An open question is related to the data used by the approach to base its decisions. A possible solution is to use only the data generated by the developer who is performing the merge. However, this generates a problem called *cold start*: there is no data available when the developer starts using the approach. On the other hand, the approach could use data from the organization, but this sometimes may not be the best choice since developers have different profiles and ways to resolve conflicts. To deal with this tradeoff, a possible solution could be start using the data from the organization and, as soon as the developers have enough data available, use this data to make decisions.

5.3.9 THE CHARACTERISTICS OF THE ORGANIZATION, PROJECT, PROCESS, AND DEVELOPERS CAN INFLUENCE IN THE FAILED MERGE?

As deep as we go in the analyses, new questions arise. For instance, is there any difference among the failed merges in different organization? Or even within the same organization, is there any difference among the failed merges of different developers? These and other questions were not treated in our current analyses, but they are important and can also generate knowledge to avoid future conflicts or to support the resolution of some conflicts. For example, Costa et al. (2016b) observed that some projects have an integrator that is responsible to merge most of the versions, and consequently there is a specialist to

resolve failed merges. On the other hand, there are projects on which the developer who performs the merge is chosen based on the knowledge the he/she has in the changes that took place.

5.3.10 PERFORM QUALITATIVE ANALYSIS OVER THE PROJECTS

Another future work is to analyze a project that the developers are available to answer questions. This would be a great opportunity to see if the obtained results make sense or not for the development team. Moreover, this kind of analysis could lead to new studies focused on the developers' doubts.

5.3.11 USE LOGICAL DEPENDENCY IN THE CHUNKS ORDERING APPROACH

In the proof-of-concept tool described in Chapter 4, we use syntactical dependencies that are closely related to the programming language of the project. This way, we only support projects that are written in Java. We have two main options to provide support for non-Java projects: (1) implement the mechanism to collect these dependencies from other languages or (2) use an alternative way to collect dependencies that does not rely on syntactical aspects. This alternative way could extract logical dependencies from files that are frequently changed together. This information can be extracted from the logs of VCSs. Consequently, logical dependencies could be used to drive the identification of dependencies and provide support to any project under VCS control.

REFERENCES

APEL, S.; LESSENICH, O.; LENGAUER, C. Structured merge with auto-tuning: balancing precision and performance. In: AUTOMATED SOFTWARE ENGINEERING (ASE), 2012, Essen, Germany. Essen, Germany, 2012. p. 120–129.

APEL, S.; LIEBIG, Jörg; BRANDL, Benjamin; LENGAUER, Christian; KÄSTNER, Christian. Semistructured Merge: Rethinking Merge in Revision Control Systems. In: EUROPEAN CONFERENCE ON FOUNDATIONS OF SOFTWARE ENGINEERING, ESEC/FSE '11, 2011, Szeged, Hungary. Szeged, Hungary: ACM, 2011. p. 190–200. Available: <http://doi.acm.org/10.1145/2025113.2025141>. Accessed: 28 jan. 2015.

APIWATTANAPONG, Taweesup; ORSO, Alessandro; HARROLD, Mary Jean. JDiff: A Differencing Technique and Tool for Object-oriented Programs. *Automated Software Engineering (ASE)*, v. 14, n. 1, p. 3–36, 2007.

BERCZUK, Stephen P.; APPLETON, Brad; BROWN, Kyle. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Boston: Addison-Wesley Professional, 2003.

BERLAGE, Thomas; GENAU, Andreas. A framework for shared applications with a replicated architecture. Symposium on User Interface Software and Technology (UIST), 1993, Atlanta, Georgia, USA. Atlanta, Georgia, USA: ACM, 1993. p. 249–257. Available: <http://doi.acm.org/10.1145/168642.168668>. Accessed: 25 jun. 2013.

BERLIN, Daniel; ROONEY, Garrett. *Practical Subversion*. Apress, 2006.

BERSOFF, Edward H.; HENDERSON, Vilas D.; SIEGEL, Stanley G. *Software Configuration Management: An Investment in Product Integrity*. Prentice Hall, 1980.

BERZINS, V. Software merge: semantics of combining changes to programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, v. 16, n. 6, p. 1875–1903, 1994.

BINKLEY, David; HORWITZ, Susan; REPS, Thomas. Program Integration for Languages with Procedure Calls. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 4, n. 1, p. 3–35, 1995.

BRUN, Y.; HOLMES, R.; ERNST M. D.; NOTKIN D. *Speculative Identification of Merge Conflicts and Non-Conflicts*. Technical Report UW-CSE-10-03-01. Washington,USA: University of Washington, 2010.

BRUN, Y.; HOLMES, Reid; ERNST, Michael D.; NOTKIN, David. Crystal: precise and unobtrusive conflict warnings. In: (ESEC/FSE), ESEC/FSE '11, 2011a, New York, NY, USA. New York, NY, USA: ACM, 2011. p. 444–447. Available: <http://doi.acm.org/10.1145/2025113.2025187>. Accessed: 31 aug. 2012.

BRUN, Y.; HOLMES, Reid; ERNST, Michael D.; NOTKIN, David. Proactive detection of collaboration conflicts. ESEC/FSE '11, 2011b, Szeged, Hungary. Szeged, Hungary: ACM, 2011. p. 168–178. Available: <http://doi.acm.org/10.1145/2025113.2025139>. Accessed: 31 aug. 2012.

BUFFENBARGER, Jim. Syntactic software merging. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), Lecture Notes in Computer Science, 1995, London, UK. London, UK: Springer Berlin Heidelberg, 1995. p. 153–172. Available: <http://link.springer.com/chapter/10.1007/3-540-60578-9_14>. Accessed: 2 apr. 2015.

CART, M.; FERRIE, J. Asynchronous reconciliation based on operational transformation for P2P collaborative environments. In: INTERNATIONAL CONFERENCE ON COLLABORATIVE COMPUTING: NETWORKING, APPLICATIONS AND WORKSHARING, 2007. COLLABORATECOM 2007, 2007, 2007. p. 127–138.

CASS, Stephen. The 2015 Top Ten Programming Languages. *IEEE Spectrum: Technology, Engineering, and Science News*, 20 Jul. 2015. Available: <http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages>. Accessed: 20 oct. 2016.

CAVALCANTI, Guilherme; ACCIOLY, Paola; BORBA, Paulo. Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), 2015, Beijing, China. Beijing, China, 2015. p. (To appear). Available: <http://twiki.cin.ufpe.br/twiki/pub/SPG/SemistructuredMergeReplication/gjcc_full_esem15.pdf>. Accessed: 11 aug. 2015.

CEDERQVIST, P. *Version Management with CVS*. . Free Software Foundation, 2003.

CHACON, Scott. *Pro Git*. 1. ed. Berkeley, CA, USA: Apress, 2009.

COHEN, Jacob. A power primer. *Psychological Bulletin*, v. 112, n. 1, p. 155–159, 1992.

COLLINS-SUSSMAN, B.; FITZPATRICK, B. W; PILATO, C. M. *Version control with subversion*. O'Reilly Media, Inc., 2004.

COLLINS-SUSSMAN, B.; FITZPATRICK, B. W.; PILATO, C. M. *Version Control with Subversion*. 2. ed. Sebastpol, CA, USA: O'Reilly Media, 2008.

COSTA, C.; FIGUEIREDO, J. J. C.; GHIOTTO, G.; MURTA, L. Characterizing the Problem of Developers' Assignment for Merging Branches. *International Journal of Software Engineering and Knowledge Engineering*, v. 24, n. 10, p. 1489–1508, 1 Dec. 2014.

COSTA, C.; FIGUEIREDO, J.; MURTA, L. Collaborative Merge in Distributed Software Development: Who Should Participate? 2014, Vancouver, Canada. Vancouver, Canada, 2014. p. 268–273.

COSTA, Catarina; FIGUEIREDO, Jair; MURTA, Leonardo; SARMA, A. TIPMerge: Recommending Developers for Merging Branches. In: ACM SIGSOFT INT'L SYMP. FOUNDATIONS OF SOFTWARE ENG. (FSE), 2016a, Seattle, WA, USA. Seattle, WA, USA, 2016.

COSTA, Catarina; FIGUEIREDO, Jair; MURTA, Leonardo; SARMA, A. TIPMerge: Recommending Experts for Integrating Changes across Branches. In: ACM SIGSOFT INT'L SYMP. FOUNDATIONS OF SOFTWARE ENG. (FSE), 2016b, Seattle, WA, USA. Seattle, WA, USA, 2016.

DANCEY, Christine P.; REIDY, John. *Statistics without maths for psychology*. Pearson Education, 2007.

DART, Susan. Concepts in configuration management systems. SCM '91, 1991, Trondheim, Norway. Trondheim, Norway: ACM, 1991. p. 1–18.

DEVANBU, Premkumar. New Initiative: The Naturalness of Software. In: ICSE - NIER TRACK, 2015, Florence, Italy. Florence, Italy, 2015. Available: <http://macbeth.cs.ucdavis.edu/neir2015.pdf>. Accessed: 27 aug. 2015.

DUVALL, Paul M.; MATYAS, Steve; GLOVER, Andrew. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

ELLIS, C. A.; GIBBS, S. J. Concurrency Control in Groupware Systems. SIGMOD '89, 1989, New York, NY, USA. New York, NY, USA: ACM, 1989. p. 399–407. Available: <http://doi.acm.org/10.1145/67544.66963>. Accessed: 16 jan. 2014.

FOGEL, K.; BAR, M. *Open Source Development with CVS*. Scottsdale, Arizona, USA: The Coriolis Group, 2001.

FRASER, Neil. Differential Synchronization. DocEng '09, 2009, New York, NY, USA. New York, NY, USA: ACM, 2009. p. 13–20. Available: <http://doi.acm.org/10.1145/1600193.1600198>. Accessed: 17 jan. 2014.

GUIMARÃES, Mário Luís; SILVA, António Rito. Improving early detection of software merge conflicts. ICSE 2012, 2012, Piscataway, NJ, USA. Piscataway, NJ, USA: IEEE Press, 2012. p. 342–352. Available: <http://dl.acm.org/citation.cfm?id=2337223.2337264>. Accessed: 31 aug. 2012.

HORWITZ, Susan; REPS, Thomas. The Use of Program Dependence Graphs in Software Engineering. 1992, 1992. p. 392–411.

HUNT, James W.; MCILROY, M. Douglas. *An Algorithm for Differential File Comparison*. Computing Science Technical Report, n^{o} CSTR 41. Murrray Hill, NJ: Bell Laboratories, 1976.

HUNT, J.J.; TICHY, W.F. Extensible language-aware merging. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), 2002. PROCEEDINGS, 2002, Montreal, Canada. Montreal, Canada, 2002. p. 511–520.

JACKSON, D.; LADD, D.A. Semantic Diff: a tool for summarizing the effects of modifications. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 1994. PROCEEDINGS, 1994, Victoria, BC, Canada,. Victoria, BC, Canada, 1994. p. 243–252.

KASI, Bakhtiar Khan; SARMA, A. Cassandra: Proactive Conflict Minimization Through Optimized Task Scheduling. ICSE '13, 2013, Piscataway, NJ, USA. Piscataway, NJ, USA: IEEE Press, 2013. p. 732–741. Available: <http://dl.acm.org/citation.cfm?id=2486788.2486884>. Accessed: 26 feb. 2015.

KERMARREC, Anne-Marie; ROWSTRON, Antony; SHAPIRO, Marc; DRUSCHEL, Peter. The IceCube approach to the reconciliation of divergent replicas. Symposium on Principles

of Distributed Computing (PODC), 2001, New York, NY, USA. New York, NY, USA: ACM, 2001. p. 210–218. Available: <http://doi.acm.org/10.1145/383962.384020>. Accessed: 25 jun. 2013.

KOEGEL, Maximilian; HERRMANNSDOERFER, Markus; VON WESENDONK, Otto; HELMING, Jonas. Operation-based conflict detection. International Workshop on Model Comparison in Practice (IWMCP), 2010, New York, NY, USA. New York, NY, USA: ACM, 2010. p. 21–30. Available: <http://doi.acm.org/10.1145/1826147.1826154>. Accessed: 23 jun. 2013.

LEHMAN, M.M.; RAMIL, J.F.; WERNICK, P.D.; PERRY, D.E.; TURSKI, W.M. Metrics and laws of software evolution-the nineties view. In: SOFTWARE METRICS SYMPOSIUM, 1997. PROCEEDINGS., FOURTH INTERNATIONAL, Nov. 1997, Nov. 1997. p. 20–32.

LEßENICH, Olaf; APEL, Sven; LENGAUER, Christian. Balancing precision and performance in structured merge. *Automated Software Engineering (ASE)*, v. 22, n. 3, p. 367–397, 28 May 2014.

MENS, T. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, TSE, v. 28, n. 5, p. 449–462, 2002.

MILLER, Webb; MYERS, Eugene W. A file comparison program. *Software: Practice and Experience*, 1097-024X, v. 15, n. 11, p. 1025–1040, 1985.

MOLLI, Pascal; OSTER, Gérald; SKAF-MOLLI, Hala; IMINE, Abdessamad. Using the Transformational Approach to Build a Safe and Generic Data Synchronizer. GROUP '03, 2003, New York, NY, USA. New York, NY, USA: ACM, 2003. p. 212–220. Available: <http://doi.acm.org/10.1145/958160.958194>. Accessed: 16 jan. 2014.

MURDOCK JR., Bennet B. Short-Term Memory1. In: BOWER, GORDON H. (Ed.). . *Psychology of Learning and Motivation*. Academic Press, 1972. v. 5. p. 67–127. Available: <http://www.sciencedirect.com/science/article/pii/S0079742108604405>. Accessed: 28 oct. 2016.

MURTA, L. G. P.; CORRÊA, Chessman Kennedy Faria; PRUDÊNCIO, João Gustavo; WERNER, Cláudia Maria Lima. Towards odyssey-VCS 2: improvements over a UML-based version control system. In: INTERNATIONAL WORKSHOP ON COMPARISON AND VERSIONING OF SOFTWARE MODELS (CVSM), 2008, Leipzig. Leipzig: ACM, 2008.

MYERS, E. W. An O(ND) Difference Algorithm and its Variations. *Algorithmica*, v. 1, n. 2, p. 251–266, 1986.

OSTER, Gérald; URSO, Pascal; MOLLI, Pascal; IMINE, Abdessamad. Data Consistency for P2P Collaborative Editing. CSCW '06, 2006, Banff, Alberta, Canada. Banff, Alberta, Canada: ACM, 2006. p. 259–268. Available: <http://doi.acm.org/10.1145/1180875.1180916>. Accessed: 17 jan. 2014.

O'SULLIVAN, B. *Mercurial : the definitive guide*. 1. ed. Sebastopol CA: O'Reilly Media, 2009.

PERRY, Dewayne E.; SIY, Harvey P.; VOTTA, Lawrence G. Parallel changes in large scale software development: an observational case study. ICSE '98, 1998, Washington, DC, USA. Washington, DC, USA: IEEE Computer Society, 1998. p. 251–260.

PREGUICA, Nuno; MARQUES, Joan Manuel; SHAPIRO, Marc; LETIA, Mihai. A Commutative Replicated Data Type for Cooperative Editing. ICDCS '09, 2009, Washington, DC, USA. Washington, DC, USA: IEEE Computer Society, 2009. p. 395–403. Available: <http://dx.doi.org/10.1109/ICDCS.2009.20>. Accessed: 17 jan. 2014.

PRUDÊNCIO, João Gustavo; MURTA, Leonardo; WERNER, Cláudia; CEPÊDA, Rafael. To lock, or not to lock: That is the question. *Journal of Systems and Software*, v. 85, n. 2, p. 277–289, Feb. 2012.

RESSEL, Matthias; NITSCHE-RUHLAND, Doris; GUNZENHÄUSER, Rul. An Integrating, Transformation-oriented Approach to Concurrency Control and Undo in Group Editors. CSCW '96, 1996, New York, NY, USA. New York, NY, USA: ACM, 1996. p. 288–297. Available: <http://doi.acm.org/10.1145/240080.240305>. Accessed: 16 jan. 2014.

RHYNE, James R.; WOLF, Catherine G. Tools for Supporting the Collaborative Process. UIST '92, 1992, New York, NY, USA. New York, NY, USA: ACM, 1992. p. 161–170. Available: <http://doi.acm.org/10.1145/142621.142645>. Accessed: 1 mar. 2014.

ROH, Hyun-Gul; JEON, Myeongjae; KIM, Jin-Soo; LEE, Joonwon. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *J. Parallel Distrib. Comput.*, v. 71, n. 3, p. 354–368, Mar. 2011.

SANTOS, Jadson; KULESZA, Uirá. Quantifying and Assessing the Merge of Cloned Web-Based System: An Exploratory Study. 2016. Available: <http://ksiresearchorg.ipage.com/seke/seke16paper/seke16paper_232.pdf>. Accessed: 13 sep. 2016.

SARMA, A.; NOROOZI, Zahra; VAN DER HOEK, A. Palantír: Raising Awareness among Configuration Management Workspaces. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 2003, Portland, Oregon. Portland, Oregon: IEEE, 2003. p. 444–454.

SARMA, A.; REDMILES, D.F.; VAN DER HOEK, A. Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes. *IEEE Transactions on Software Engineering*, v. 38, n. 4, p. 889–908, Jul. 2012.

SARMA, A.; VAN DER HOEK, A. Palantir: coordinating distributed workspaces. In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 2002. COMPSAC 2002. PROCEEDINGS. 26TH ANNUAL INTERNATIONAL, 2002, 2002. p. 1093–1097.

SERBAN, A. *Visual Sourcesafe 2005 Software Configuration Management in Practice*. Birmingham, UK: Packt Publishing, 2007.

SHEN, Haifeng; SUN, Chengzheng. A complete textual merging algorithm for software configuration management systems. In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 2004. COMPSAC 2004. PROCEEDINGS OF THE 28TH ANNUAL INTERNATIONAL, 2004, Hong Kong, China. Hong Kong, China, 2004. p. 293–298 vol.1.

SHEN, Haifeng; SUN, Chengzheng. Syntax-based reconciliation for asynchronous collaborative writing. In: 2005 INTERNATIONAL CONFERENCE ON COLLABORATIVE COMPUTING: NETWORKING, APPLICATIONS AND WORKSHARING, 2005, San Jose, CA, USA. San Jose, CA, USA, 2005. p. 10 pp.-.

SILVA JUNIOR, J.R.; PACHECO, T.; CLUA, E.; MURTA, L. A GPU-based Architecture for Parallel Image-aware Version Control. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR), Mar. 2012, Genova, Italy. Genova, Italy: IEEE, Mar. 2012. p. 191–200.

SULEIMAN, Maher; CART, Michèle; FERRIÉ, Jean. Serialization of Concurrent Operations in a Distributed Collaborative Environment. GROUP '97, 1997, New York, NY, USA. New York, NY, USA: ACM, 1997. p. 435–445. Available: <http://doi.acm.org/10.1145/266838.267369>. Accessed: 16 jan. 2014.

TICHY, Walter F. RCS: A System for Version Control. *Software - practice and experience*, v. 15, n. 7, p. 637–654, 1985.

VIDOT, Nicolas; CART, Michelle; FERRIÉ, Jean; SULEIMAN, Maher. Copies Convergence in a Distributed Real-time Collaborative Environment. CSCW '00, 2000, New York, NY, USA. New York, NY, USA: ACM, 2000. p. 171–180. Available: <http://doi.acm.org/10.1145/358916.358988>. Accessed: 16 jan. 2014.

WALRAD, Chuck; STROM, Darrel. The Importance of Branching Models in SCM. *IEEE Computer*, v. 35, n. 9, p. 31–38, 2002.

WEISER, Mark. Program Slicing. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 1981, Piscataway, USA. Piscataway, USA: IEEE Press, 1981. p. 439–449. Available: <http://dl.acm.org/citation.cfm?id=800078.802557>. Accessed: 1 jul. 2014.

WEISS, Stéphane; URSO, Pascal; MOLLI, Pascal. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. 2009, Los Alamitos, CA, USA. Los Alamitos, CA, USA: IEEE Computer Society, 2009. p. 404–412.

WESTFECHTEL, Bernhard. Structure-oriented Merging of Revisions of Software Documents. SCM '91, 1991, Trondheim, Norway. Trondheim, Norway: ACM, 1991. p. 68–79. Available: <http://doi.acm.org/10.1145/111062.111071>. Accessed: 2 apr. 2015.

WINGERD, Laura. *Practical Perforce*. 1. ed. O'Reilly Media, 2005.

YUZUKI, R.; HATA, H.; MATSUMOTO, K. How we resolve conflict: an empirical study of method-level conflict resolution. In: 2015 IEEE 1ST INTERNATIONAL WORKSHOP ON SOFTWARE ANALYTICS (SWAN), 2015, Montreal, Canada. Montreal, Canada, 2015. p. 21–24.

ZHANG, Kaizhong; JIANG, Tao. Some MAX SNP-hard Results Concerning Unordered Labeled Trees. *Inf. Process. Lett.*, v. 49, n. 5, p. 249–254, Mar. 1994.