On the Nature of Merge Conflicts: a Study of 2,731 Open Source Java Projects Hosted by GitHub

Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and André van der Hoek, *Member, IEEE*

Abstract— When multiple developers change a software system in parallel, these concurrent changes need to be merged to all appear 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 as much as 10% to 20% of all merge attempts result in a merge conflict, meaning that a developer has to manually complete the merge. To date, we have little insight into the nature of these merge conflicts. What do they look like, in detail? How do developers resolve them? Do any patterns exist that might suggest new merge techniques that could reduce the manual effort? This paper contributes an in-depth study of the merge conflicts found in the histories of 2,731 open source Java projects. Seeded by the manual analysis of the histories of five projects, our automated analysis of all 2,731 projects: (1) characterizes the merge conflicts in terms of number of chunks, size, and programming language constructs involved, (2) classifies the manual resolution strategies that developers use to address these merge conflicts, and (3) analyzes the relationships between various characteristics of the merge conflicts and the chosen resolution strategies. Our results give rise to three primary recommendations for future merge techniques, that – when implemented – could on one hand help in automatically resolving certain types of conflicts and on the other hand provide the developer with tool-based assistance to more easily resolve other types of conflicts that cannot be automatically resolved.

Index Terms—Software Merge, Merge Conflict, Merge Resolution.

◆

1 Introduction

CONCURRENT work is essential to large-scale software development. Once developers finish their independent work, changes must be integrated and made available to other developers. A traditional approach to do so is to apply a merge tool, which implements some underlying merge technique that aims to automate as much as possible of the task of combining parallel changes [1].

Many merge techniques have been developed over the years [1], [2], differing considerably in what they use as a basis for comparing two versions of an artifact and resolving any conflicting changes exhibited by the two versions. A significant number of merge techniques rely on lines of code as the basis; these techniques are called unstructured merge techniques [3]–[8]. Other, more complex techniques have also been developed, some relying on syntax [9]–[15] and others on semantics [16], [17]; these techniques are termed structured merge techniques. Hybrid approaches that mix aspects of both unstructured and structured techniques have been explored as well [2], [18], [19]. These are termed semi-structured techniques.

Despite the many different merge tools in use today, it is well known that, in practice, they are not perfect. Because they cannot account for every possible concurrent change that developers may and regularly do make, conflicts arise between concurrent changes that cannot be automatically resolved, leading the merge attempt to fail [1]. The developer has to step in, analyze the respective changes and the merge conflict they caused, and resolve the conflict manually. This is a difficult task, one that developers wish to avoid as much as possible [20], [21].

Still, it has been reported that 10% to 20% of all merges fail [22], [23], with some projects experiencing rates of almost 50% [22], [24]. To spur the development of new merge techniques that could potentially reduce the high failure rate, we posit that it is necessary to take a deep dive into the nature of merge conflicts and how they are resolved. What do they look like, exactly? How do developers resolve them? Do any relationships exist between the nature of certain merge conflicts and the resolution strategies chosen by developers?

A few studies have begun to answer these kinds of questions (e.g., [2], [22]–[30]). Some focus on classifying the type of conflict (i.e., merge, build, or test failure) [22], [23]. Others examine the existence of correlations between certain development and code characteristics and resulting merge failures [26]–[30]. Yet others compare unstructured and semi-structured merge techniques, finding that semi-structured techniques can reduce, but not eliminate, merge conflicts [2], [25]. To this emerging body of literature, this paper contributes a new study that differs in: (a) taking a fine-grained approach to dissecting the nature of merge

- *Gleiph Ghiotto is with the Computing Institute, Fluminense Federal University, Niterói, RJ, Brazil and Computer Science Department, Federal University of Juiz de Fora, Juiz de Fora, MG, Brazil. E-mail: gmenezes@ic.uff.br.*
- *Leonardo Murta is with the Computing Institute, Fluminense Federal University, Niterói, RJ, Brazil. E-mail: leomurta@ic.uff.br.*
- *Márcio Barros is with the Information Systems Program, UNIRIO, Rio de Janeiro, RJ, Brazil. E-mail: marcio.barros@uniriotec.br.*
- *André van der Hoek is with the Department of Informatics, University of California Irvine, Irvine, CA, USA. E-mail: andre@ics.uci.edu.*

conflicts, anchoring our analysis on individual conflicting chunks (as a merge conflict might be the result of incompatible changes in multiple, disjoint parts of an artifact) and the programming language constructs they contain, and (b) seeded by a manual analysis of a handful of projects, performing an automated, large-scale analysis of over 2700 projects, compared to the typical few projects studied in prior work.

We manually examined the history of five open source projects, finding the merges that conflicted. For each such failed merge, we collected the following: (1) the number of conflicting chunks, (2) the size of each of the versions of the conflicting chunk in lines of code, (3) the programming language constructs contained within each version of the conflicting chunk, and (4) the way developers resolved each conflicting chunk. We then looked for patterns between characteristics of the conflicting chunks and the decisions made by the developers as to how to resolve them. Using this *manual analysis* as the seed, we developed the tools necessary to engage in a large-scale *automated analysis* involving 2,731 open source projects with 25,328 failed merges. We collected the same information as in the manual analysis, so we could address the same research questions.

Our results show that the creation of fully automated merge techniques is likely to be impossible, as too many conflicts have manual resolutions that cannot be anticipated. Yet, our results also give rise to three findings that show promise for the design of future merge techniques:

1. For 87% of conflicting chunks, the conflicting chunk contained all the lines of code that appeared in the merged result, and in 94% of those cases, the conflicting chunk involved less than 50 lines of code in each of its versions. Developers used anywhere from just a few lines of one version to all lines of both versions in creating a merged result. Sometimes they concatenated both versions wholesale, while at other times they interspersed individual lines. However, for the majority of merge conflict resolutions, no new lines of code were written. This suggests that it may be beneficial to create heuristics that cover the common cases, as well as design tools that assist developers in reorganizing the lines of code of failed merges.

2. Of all failed merges, 60% involved multiple conflicting chunks. Moreover, depending on the project, in 14% to 46% of the failed merges involving multiple chunks, dependencies existed among chunks in that the resolution of one conflicting chunk might offer guidance for how the remaining conflicting chunks are to be resolved. As an example, one chunk would contain a conflict in a method declaration, and other chunks conflicts in the corresponding method invocations. This suggests that new merge tools may be able to better assist developers by presenting them with a suggested order in which chunks should be resolved according to their dependencies, as well as by resuming automated merging once key conflicts are resolved.

3. For both projects and individual developers, certain tendencies existed in how conflicting chunks were resolved. As one example, though nearly 20% of the conflicting chunks in one project were resolved with *new code* (a high percentage), some developers rarely used that strategy. As another example, across a number of the projects, some kinds of conflicts were resolved with certain resolution strategies more often, regardless of individual developer preferences. Such historical patterns, and others like it, could be leveraged by new merge tools to present developers with the distributions of past choices and allowing them to choose one to be performed.

The remainder of this paper is organized as follows. Section 2 describes the materials and methods used to conduct the study. Section 3 presents our results, characterizing merge conflicts, resolution strategies, and apparent relationships between the two. Section 4 discusses the implications of our findings. Section 5 covers threats to validity. Section 6 presents related work and, finally, Section 7 concludes with an outlook at future work.

2 Materials and Methods

This section introduces relevant terminology, presents the analyses that we perform, details our data collection procedure for the manual analysis, and describes our data collection for the automated analysis.

2.1 Terminology

Distributed version control systems such as Git [4] support concurrent development through both implicit and named branches [31]. An implicit branch is typically created when a developer clones a repository to work in parallel with other developers. This kind of branch is often short-lived, only useful until the developer merges the changes back into the remote repository. Named branches exist in both the local and remote repositories and are typically created to separate long-term parallel lines of development, for instance when multiple customers require somewhat different versions of the same software. Each named branch may accommodate commits from multiple developers. Depending on the goal of the named branches, changes made on one may need to be merged into another (e.g., a bug fix common to multiple variants of the product). Sometimes, named branches are merged back together in their entirety.

When a merge attempt fails, it means that one or more changes that were made to the artifacts being merged are in conflict. The exact nature of the conflict depends on the merge technique, but regardless of technique, a conflict can manifest itself in multiple parts of the artifacts. That is, it is possible—and frequently so—that, when a merge fails, the conflict exhibits itself in several regions across the artifacts. We term each pair of those regions that is in conflict a *conflicting chunk*.

Fig. 1 shows a conflicting chunk in Git [4], using three marks: (i) the beginning mark, represented by "<<<<<<<", which is followed by the version in which changes are to be integrated (in this case HEAD, the version in the developer local repository); (ii) the separator, "=======",

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

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

Fig. 1. Conflicting chunk of merge b14ca5 (common ancestor f7d0ca) from ANTLR4.

which divides the code that differs between the two versions in conflict; and (iii) the ending mark, “>>>>>>>”, which is followed by the version from which the changes are to be included, in this case b80ad5052d1b693be6e5c0a2b.

To ease comprehension, we adopt the side-by-side representation depicted in Fig. 2. We refer to the code in gray background on the left-hand side of Fig. 2 as *version 1* (before the separator in Fig. 1, representing the changes made in the local repository of the developer) and the code in gray background on the right-hand side as *version 2* (after the separator in Fig. 1, representing the version from which the changes be integrated into the local repository). The code in white background belongs to the common ancestor of *versions 1* and *2* (i.e., the base version).

A specific goal of our study is to dive into detail regarding the nature of conflicting chunks, asking such questions as to which language constructs (e.g., *for, while, if, variable, class*) show a tendency of appearing together in conflicting chunks. We selected the Java language specification to create the list of language constructs for our analysis, including, among others, statements (e.g., *for, if*), definitions (e.g., *class declaration, method declaration*), and invocations (e.g., *method invocation)*. We grouped language constructs that play a similar role, such as @Override, @NotNull, and others like it as *annotation*, and constants, class fields, and local variables as *variable*.

To classify different conflicting chunks, we define a *kind of conflict* as the concatenation of all the unique language constructs (in alphabetical order) that are present in a conflicting chunk. To assign a kind 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 Fig. 2, the kind of conflict is “*annotation, method declaration, variable*” (both the *return* statement and its *variable* reside within a containing method declaration already in conflict, which is why we do not include them in the kind of conflict, a decision we further discuss in Section 2.3). We remove duplicate language constructs and sort the remaining ones so we create a relatively short set of kinds of conflicts, as compared to the millions that may result if we did not sort first and then remove duplicates. This choice, then, leads to a higher frequency for each of the kinds of conflicts, which in turn allows us to more meaningfully detect possible patterns.

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 by conflicting chunk basis, and identify six different ways of resolving a conflict: (1) adopt the code of *version 1*, (2) adopt the code of *version 2*, (3) concatenate both versions wholesale, in either order, (4) incorporate in some interleaving order select lines of code from both versions, without writing any new lines or modifying selected lines, (5) mix existing code from one or both versions with newly written code, and (6) use none of the versions, that is, the developer discards both versions. We identify these choices in the remainder of the paper as: *version 1* (*V1*), *version 2* (*V2*), *concatenation* (*CC*), *combination* (*CB*), *new code* (*NC*), and *none* (NN).

Finally, our study categorizes 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* is presumably more difficult than a kind of conflict that is 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 *straightforward chunks* (conflicting chunks that are resolved with *version 1*, *version 2*, *concatenation*, or *none*) and *complex chunks* that are resolved with *combination* or *new code*. The motivation is that the latter two kinds of conflicts require a developer to engage in depth with the conflicting chunks, and generally involve more time and effort. We placed *none* in *straightforward chunks*, as a manual inspection of the few occurrences of *none* that we encountered revealed that they were small merge failures where the developer decided neither option was good; complex merge cases from which the developer backed away were usually resolved by placing a comment in the code, making it *new code*.

2.2 Analyses

The focus of our work is on understanding merge conflicts in detail, together with the resolutions that developers use to address them. The more of an understanding we develop, the more of an opportunity might exist to design new merge tools that leverage the lessons learned.

To build this understanding, we identified seven incremental analyses that we performed on the 25,328 historical conflicts of the 2,731 Java projects we studied (see below for project selection). Each analysis adds detail to the previous analyses either by performing a more fine-grained analysis of earlier results or by correlating findings from previous analyses with factors that may explain them. We describe each analysis briefly here, and detail them further

```
public RuleStopState stopState;
                 version 1                  |                 version 2
public Boolean isPrecedenceRule;            | @Override
                                            | public int getStateType() {
                                            |   return RULE_START;
                                            | }
}
```

Fig. 2. Simplified side-by-side representation for the conflicting chunk of Fig. 1.

in Section 3 when we discuss our results.

A1. *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. 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.

A2. *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 merge failure, complementary to the number of conflicting chunks is the size of those conflicting chunks: the higher the LOC, the more code must be inspected and worked with to resolve the conflict – whether by a developer or a merge tool. Hence, our second analysis focuses on assessing the distribution of LOC in conflicting chunks.

A3. *What is the distribution in language constructs involved in conflicting chunks?*

As already stated, the language constructs involved in a conflict can influence the difficulty of resolving it. Consider a case in which the only language construct present in a conflicting chunk is *import*. Concatenating the two import statements will likely resolve the conflict in most cases, and is an operation that can easily be performed by a tool (perhaps with some checks if 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 frequency of appearance of different combinations of language constructs. Our third analysis, then, focuses on the distribution of language constructs involved in conflicting chunks.

A4. *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 dependencies exist that are indicative of possible 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.

A5. *What is the distribution of developer decisions?*

From all the different decisions that developers can make, only one involves addition of *new code*. If the majority of decisions does not involve new code, a sensible first step forward is to develop a new merge tool that presents the 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 ordering desired lines of code from each of the two versions). Conversely, if most conflicting chunks involve new code, such a new merge tool would not help 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.

A6. *What is the distribution in difficulty level of kinds of conflicts?*

While the first five analyses examine properties of conflicting chunks, our next analysis studies the chunks based on the relation between the kinds of conflicts they contain and what the chosen developer decisions reveal about the apparent difficulty levels of the different kinds of conflicts. It might be, for instance, that some kinds of conflicting chunks nearly always are resolved with *new code*, while other kinds mostly involve *concatenation*. This, in turn, provides preliminary guidance toward what kind of tool support may be required when.

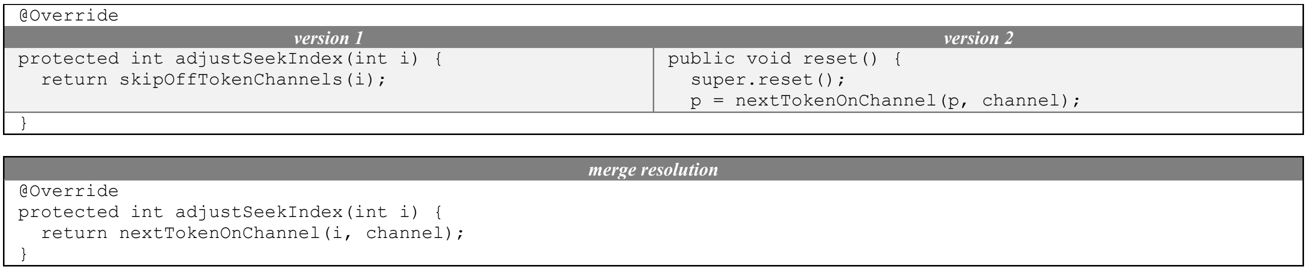
A7. *What, if any, patterns exist between the language constructs of conflicting chunks and developers' decisions?*

Our final 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 resolution. That is, rather than examining the kind of conflict in its entirety, 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 the possibility of heuristics that encapsulate these patterns in the support offered by new merge tools.

2.3 Data Collection Procedure for Manual Analysis

While we could have chosen to only perform an automated analysis of a large number of projects, we felt it was preferred to perform a manual analysis of a few projects prior. First, we felt it would help us understand the issues in much greater detail, shaping the automated analysis, and thereby not putting the proverbial cart before the horse. Second, observations from our manual inspections fueled the formulation of the analyses we performed in Section 3, as engaging with the conflicts at a very detailed level helped us to understand what sorts of phenomena were present in the data we were collecting. Finally, performing a manual analysis helped us to identify insightful examples, with several of them presented in the below.

To select projects for the manual analysis, we gathered



```
@Override
                    version 1                                         version 2
protected int adjustSeekIndex(int i) {        public void reset() {
  return skipOffTokenChannels(i);               super.reset();
                                                p = nextTokenOnChannel(p, channel);
}
```

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

Fig. 3. Conflicting chunk of merge 18f535 (common ancestor ea7037) and its *new code* resolution in ANTLR4.

all projects from the first page of the GitHub trending site[1] at the time. We removed non-Java projects and sorted the remaining projects in descending order by the number of merge conflicts. This led us to the following top four projects that we selected: MCT, Lombok, ANTLR, and Twitter4J. We added Voldemort, as it is studied by most of our related work. *MCT* is a NASA-developed real-time monitoring platform; *Lombok* is a project that helps in writing succinct boilerplate code via annotations; *ANTLR4* is a parser for programming languages; *Twitter4J* is an API for accessing Twitter; and *Voldemort* is a distributed key-value storage system.

These projects were popular when we began our study, meaning that they likely were offering useful functionality to many. Moreover, all five were hosted on GitHub, our target platform (since it strongly promotes a culture of parallel work) and most of their code is in one programming language, Java, so we can draw inferences across the projects. All had over 1,000 commits and involved at least 10 developers, increasing our changes of identifying meaningful conflicts. 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 (#Developers); the number of failed merges (#FM); and the total number of conflicting chunks (#CC).

TABLE 1

KEY STATISTICS OF THE SELECTED PROJECTS, INCLUDING THE NUMBER OF COMMITS, MERGES, DEVELOPERS, FAILED MERGES (FM), AND CONFLICTING CHUNKS (CC)

Project	#Commits	#Merges	#Developers	#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

The process of collecting our study data started with the identification of merges, for which we used a standard Git command that lists all commits with more than one parent. Afterward, we replayed each merge case to determine whether a merge was successful. To do so, we accessed the parents of the merge and ran the Git merge command again. When this returned no conflicts and produced code equivalent to the original merge result, we recorded the merge as successful and ignored it. Note that Git uses a three-way merge, because all changes are stored in its repository and thus the common ancestor is always available. We also note that both fast-forward merges and octopus merges create no conflicts by definition, so they do not influence our results.

When a merge failed, we analyzed each of its conflicting chunks to record the size of each of its respective versions, as well as the language constructs that were part of each of those versions. By doing this manually, we identified cases where a naïve line-by-line comparison would record somewhat superficial conflicts. For instance, some conflicts were the result of whitespace, something a pass of a code beautifier before merge could easily address. More importantly, however, we could identify the situation that is illustrated in Fig. 3. From the original merge result (shown at the bottom as performed manually by the developer), we deduce that the situation was one in which a pair of developers each added a method, only one of which was needed, but in somewhat adjusted form. Thus, the conflict does not concern all four of *method signature*, *return statement*, *method invocation*, and *variable*, as a naïve approach may have possibly documented, but just two *method declarations*: *adjustSeekIndex* and *reset*. Thus, this conflict was recorded as a *method declaration* kind of conflict. 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. As a rule, then, we documented the outermost language constructs involved in each of the conflicting chunks we found, with a language construct being considered outermost if it belongs to the conflicting chunk and its parent node in the AST does not belong to the same conflicting chunk. The only exception we made was *assignment*, which was obstructing other more relevant language constructs. In the case of *assignment*, we considered its children as outermost. Note that, from here on out and for reasons of brevity, we will use the term language constructs in our results to refer to outermost language constructs as defined here.

We also examined every conflicting chunk and its original merge resolution to understand how developers chose to resolve the conflict that was present. We categorized this choice as *V1*, *V2*, *CC*, *CB*, *NC*, or *NN* (as defined in Section 2.1). For example, Fig. 3 includes a resolution in which the developer chose only a few lines across both versions, and

[1] https://github.com/trending

changed a line from *"return skipOffTokenChannels(i);"* to *"return nextTokenOnChannel(i, channel);"*. Thus, the developer decision was *NC*.

We extracted developer decisions from the commit immediately after the failed merge, which is the merge commit. However, to account for situations in which developers postponed a resolution, either by choosing *NN* and manually integrating the changes without tool support later or by choosing one version (usually theirs) and ignoring the other developer's changes until a later time when they integrate them manually, 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 merge commit, we examined if it represented a postponed resolution by verifying if any code from the conflict was now included. This occurred in a few cases, six times to be precise. The raw data used in the manual analysis is available at http://gems-uff.github.io/merge-nature.

2.4 Data Collection Procedure for Automated Analysis

To perform a large-scale, automated analyses of merge failures, we first used the GitHub API to select 1,997,541 projects out of the set of projects residing in its repository. Our initial target was 2 million projects, but a handful of unexplained data collection failures led to a slightly smaller initial set. For each of the projects, we collected the last time it was updated, the size of its development team, and the size of its source code as written in the programming languages used in the project (e.g., 57% Java, 36% C#, 7% XML).

Next, we selected all active Java projects from this sample. A project was classified as active if it was updated at least once after January 2015 and before our data collection date of March 2016. A project was considered as Java if the percentage of source code written in Java was greater than the percentage of code written in any of the other languages. For instance, a software project having 34% of its code in Java, 33% in C, and 33% in HTML was included. After filtering the projects in our sample according to these criteria, 13,576 projects remained.

We then cloned the repositories of these projects and replayed the merges (960,366 of them) using the same procedure as in the manual analysis. Each merge commit was classified as failed or successful by identifying the parents of its merged version, redoing the merge of these parents, and observing if a conflict arose. When the merge did not lead to a conflict, it was discarded.

As our last step, we discarded projects that were forks of other projects in our dataset or whose conflicts did not appear in their Java files (for example appearing in the C# or XML files instead). Projects that were forks were discarded to avoid counting the same merge multiple times, since forks share parts of the history of their source project. Projects without a Java conflict were discarded for the obvious reason. This led to 2,731 projects with 25,328 failed merges and 175,805 conflicting chunks.

Having these projects in hand, we implemented a number of scripts to extract the data that we had previously extracted manually for the five projects. The scripts were designed and implemented based on our experience with the manual analysis, and incorporated the practices we established there. For instance, the scripts ignore formatting characters such as blank spaces and line breaks. As another example, when a conflict involved nested language constructs (e.g., *variable* and *method invocation* inside a *for statement*) the scripts record the outermost language construct only (e.g., the *for* statement).

At the core, the scripts replay each failed merge to: (1) analyze it for the number of conflicting chunks, size of the chunks, and outermost language constructs involved, and (2) analyze it in the context of the merge commit to figure out the resolution that was used. These analyses are largely straightforward, though nuances exist that have to be considered. For instance, the context lines delineating each chunk can generally be found in the merge commit, but if not, we assume that the developer manually performed edits that cross the chunk border (and thus lead to *NC* as the resolution we mark for this chunk). As another example, the code in conflicting chunks can generally be parsed fine, but sometimes fails when the developer provided incomplete or faulty code. In such cases, the scripts resort to using Eclipse's AST error recovery mechanism to ignore the language constructs that exhibit syntax error and correctly collect the remaining language constructs. A more intricate example pertains to a situation where some lines of code from one of the versions are used more than once in the resolution. In that case, as long as no new code has been written elsewhere, we would classify the resolution as *CB*. For additional detail, all scripts and the raw data are available at http://gems-uff.github.io/merge-nature.

3 Results

This section presents our results, organized per each of the analyses outlined in the previous section. For each analysis, results of the manual analysis are discussed first, followed by results from the automated analysis.

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

Concerning the manual analysis, Fig. 4 shows the number of failed merges involving different numbers of conflicting chunks. Most failed merges involved just four or fewer conflicting chunks (111 out of 147, 76%) and more than half involved merely one or two (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 does not necessarily mean less complicated resolutions. We return to this topic in subsequent analyses. Four failed merges involved more than 20 chunks each, with one of them involving as many as 39. Manual inspection of these revealed that they are very complex merges, all requiring code from both versions that was significant adjusted, as well as entirely new code that was written.

The automated analysis led to the distribution presented in Fig. 5. Similar to the manual analysis, most failed merges in the 2,731 projects have few conflicting chunks: 40% of the failed merges have a single conflicting chunk and 90% have 10 or fewer. The remaining 10% had 11 or more chunks, with the maximum an astounding 10,315 chunks (merge 7a9c34 of project Jnario, which was the result of a conflict between a set of feature enhancements and refactorings on one branch and a major framework upgrade on the other branch).

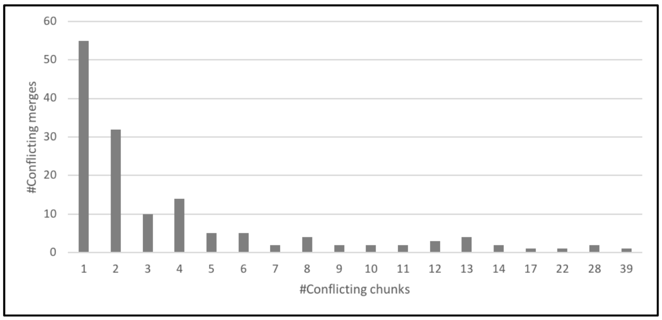


Fig. 4. Histogram of conflicting chunks (manual analysis).

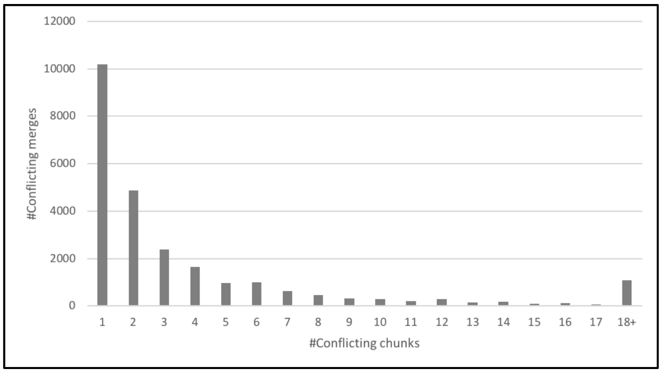


Fig. 5. Histogram of conflicting chunks (automated analysis).

Because a single failed merge may involve multiple files, we analyzed how the conflicting chunks are distributed over files. We found that 62% of the conflicting files have just one conflicting chunk and 95% of the conflicting files have five or fewer conflicting chunks. Just in rare cases (less than 0.4%), an individual file has more than 20 conflicting chunks.

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

Fig. 6 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 (with the bottom left rectangle of the left figure blown up in the right figure) for the manual analysis. First, we note the range that exists: 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. 96% 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 to resolve these chunks. When we focus on chunks with at most five LOC in each version, this still accounts for 51% of the cases, meaning that in over half of the conflicting chunks, developers have to look at merely ten LOC total. Note that in several cases one of the versions in the chunk has zero LOC; this is the result of conflicts where one version involves deleting some lines of code, with some or all of those being changed in the other version.

We also observe that it is relatively rare for the two versions of a conflicting chunk to both have high LOC. Just six out of 616 chunks have more than 50 LOC in both versions, and 18 have more than 50 LOC in one version and fewer than or equal to 50 in the other. Table 2 provides an alternative view, examining the median size (Median), average size (Mean), and standard deviation (Std) of the conflicting changes in *version 1* and *version 2*, per project. V*ersion 1* represents the larger average change in all projects, except MCT. 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 number of lines of code added per commit for MCT, which a separate calculation reveals on average to be 467.74. Still, as compared to the other projects (ANTLR4 427.74; Twitter4J 205.79; Voldemort 197.94; Lombok 104.97), this does not 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.

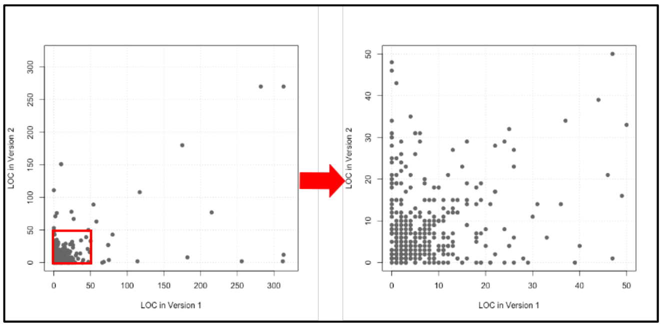


Fig. 6. LOC in version 1 versus 2 (manual analysis).

TABLE 2
AVERAGE SIZE OF CONFLICTING CHUNKS

Project	Version 1			Version 2		
	Median	Mean	Std	Median	Mean	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

Results from our automated analysis support the manual analysis, as shown in Fig. 7. 94% of the conflicting chunks have up to 50 LOC in each version (165,616 out of 175,805), 68% have up to ten LOC in each, and slightly over half (50%) five or fewer. At the other end of the spectrum, 0.05% of the chunks have more than 2,000 LOC in each version, with the extreme case involving 13,035 and 14,074 LOC, respectively (merge 310dbe of project ThingML). Across all 175,805 conflicting chunks, 4,147 (2%) involve more than 50 LOC in both versions, while 6,042 (3%) have more than 50 LOC in one version and less than 50 LOC in the other. Further examination indicates that a subset of projects is responsible for these excesses. While 700 projects (26%) have at least one chunk with both versions involving more than 50 LOC, only 95 projects (3%) have more than ten such chunks and just four projects have over a hundred such chunks. In absolute terms, the Wro4J4 project has the highest number: 179 (9%) of its conflicting chunks have more than 50 LOC in each version. In relative terms, Axis2/java5 and StatET6 are the worst: 39% and 42% of their conflicting chunks have at least 50 LOC in both versions, respectively.

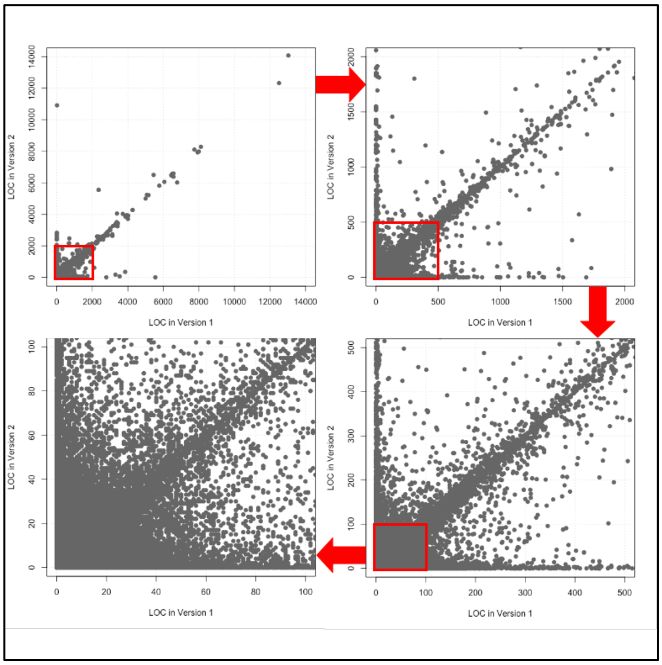


Fig. 7. LOC in version 1 versus 2 (automated analysis).

The average size across all projects of *version 1* is 19.5 LOC and *version 2* 27.6 LOC, with average standard deviations of 20.6 and 28.6 LOC, respectively, and median size of 2.0 and 2.5 LOC. Overall, the numbers show that some large chunks drag the mean size upwards and increase the standard deviation, despite the majority of chunks having less than three lines of code in one or both versions.

3.3 What is the distribution in language constructs involved in conflicting chunks?

Starting the discussion with the manual analysis, Table 3 presents the number of conflicting chunks per number of language constructs. Almost all chunks consist of up to four language constructs (594 of 616, 96%). The low number of language constructs involved is not surprising, given our decision to only record the outermost language constructs (see Section 2.3). Because the code in either version of a single chunk is contiguous, it is rare for there to be lots of outermost language constructs. Table 3, thus, should be interpreted as 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 non-nested *for* loop and *if* statement have been added).

TABLE 3

NUMBER OF CONFLICTING CHUNKS PER NUMBER OF LANGUAGE CONSTRUCTS (MANUAL ANALYSIS)

# Language constructs	# 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%)
5	1 (1%)	1(2%)	1 (2%)	9 (3%)	3 (3%)	15 (2%)
6	1 (1%)	2 (4%)	1 (2%)	0 (0%)	0 (0%)	4 (1%)
7	1 (1%)	0 (0%)	0 (0%)	0 (0%)	1 (1%)	2 (0%)
8	0 (0%)	0 (0%)	0 (0%)	1 (0%)	0 (0%)	1 (0%)

We note that 52% of conflicting chunks involve a single language construct and 80% just one or two. Because a limited set of language construct combinations 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 Fig. 8, taken from the Twitter4J project. A traditional merge technique cannot resolve this conflict, because the same area was edited in parallel. A tailored merge technique that understood that the two *if* statements address different conditions may suggest concatenating the two (which is indeed the resolution that the developer chose).

We analyzed which language constructs occurred most frequently in conflicting chunks and contrasted the results with the frequency of the language constructs across the whole code base (based on the most recent version of each project). Table 4 presents the results, when focusing on a single language construct at a time. The symbol ⬇ is used when the percentage of a language construct across the conflicting chunks is lower than across the whole code base, with the symbol ⬆ signaling the opposite. The seven language constructs listed in Table 4 represent nearly 80% of the language constructs in conflicting chunks. The three most frequently involved language constructs (*method invocation*, *variable*, and *method declaration*) together account

```
}
```

version 1

```
if (!json.isNull("lang")) {
  lang = getUnescapedString("lang", json);
```

version 2

```
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);
}
```

```
}
```

Fig. 8. Conflicting chunk from merge 3a3869 (common ancestor 6b1485) of project Twitter4J.

for over 50% of the language constructs in conflicting chunks.

TABLE 4

MOST FREQUENT LANGUAGE CONSTRUCTS (MANUAL ANALYSIS)

Language construct	# Occurrences		
	Conflicting chunks	Source code	
Method invocation	252 (22%)	250,469 (39%)	↓
Variable	208 (18%)	104,410 (16%)	↑
Method declaration	118 (10%)	40,171 (6%)	↑
Comment	112 (10%)	44,983 (7%)	↑
If statement	97 (9%)	21,646 (3%)	↑
Import	64 (6%)	33,408 (5%)	↑
Method signature	62 (5%)	40,171 (6%)	↓

Table 5 provides a refined view, listing the most frequent *kinds of* conflicts (per the definition in Section 2.1). *Method invocation*, the top language construct in Table 4, is often not a standalone change, but one that has other changes surrounding it. It appears alone a mere 63 out of 252 times, with all other occurrences in combination with other language constructs. The same is true for *variable*: 37 individual occurrences are complemented by 58 combinations with *method invocation*, 19 occurrences with *if statement* and *method invocation*, and others.

TABLE 5

MOST FREQUENT KINDS OF CONFLICTS (MANUAL ANALYSIS)

Kind of conflict	Occurrences	Percentages
Method invocation	63	10%
Import	60	9%
Method invocation, variable	58	9%
Method declaration	57	9%
Variable	37	6%
If statement	20	3%
Method signature	19	3%
If statement, method invocation, variable	19	3%

A particularly interesting example is *method invocation, variable*, which represents a combination of the two most frequent language constructs in Table 4. We found that the conflict most often concerned a *method* call to initialize or assign a value to a *variable*, with the conflict being one version changing the method being called and the other version changing the variable name. The last line of Table 5 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*.

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 kinds 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 special techniques tailored to each, our results show that the payoff of doing so diminishes relatively quickly. For instance, consider the *if statement, method invocation, variable* kind of conflict which is the eight-most frequent (Table 5; 19 out of 616 total chunks, or 3%). Amortized across the many projects under development today, 3% represents a non-trivial effort that can be eased, but as we move to 25th most or 50th most frequently occurring kind of conflict, the benefits decline rapidly, appearing only in 0.6% and 0.3% of the times, respectively. After the 50th most frequently occurring kind of conflict, all kinds of conflict occur just once.

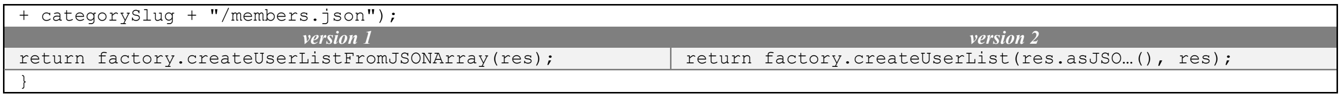
We extracted association rules involving language constructs occurring together in conflicting chunks. Table 6 shows these association rules, presented in the form of "A → B" and as measured in terms of support (s%), confidence (c%), and lift (L). For instance, the association rule *if statement, variable* → *method invocation* has 80% confidence, meaning that 80% of the chunks that include *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%. Table 6 only includes association rules that have minimum absolute support of 12 (2%) occurrences and at least 50% confidence. The rules are ordered by lift and only the top ten rules according to these criteria are shown.

TABLE 6

RELATION AMONG LANGUAGE CONSTRUCTS THAT BELONG TO CONFLICTING CHUNKS, SHOWING SUPPORT (SUP.), CONFIDENCE (CON.) AND LIFT (MANUAL ANALYSIS)

Association rule	Sup.	Con.	Lift
Annotation → method declaration	3%	64%	3.34
For statement → variable	2%	67%	1.97
If statement, variable → method invocation	6%	80%	1.96
Comment, method invocation → variable	4%	66%	1.95
If statement, method invocation → variable	6%	62%	1.84
Method invocation, method signature → variable	2%	58%	1.73
Method signature, variable → method invocation	2%	70%	1.71
Try statement → method invocation	4%	69%	1.70
Return statement → method invocation	3%	69%	1.69
Try statement, variable → method invocation	2%	68%	1.67

These association rules give further meaning to the results presented in Table 4, and particularly provide directionality to the co-occurrences shown in Table 5. As such, they may help in formulating new heuristics that explore these directionalities. Consider the conflicting chunk presented in Fig. 9, 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*. A heuristic that examines the return type of the corresponding *method declarations* and, if they differ, selects the *method invocation* that matches the type expected to be returned may be able to assist devel-



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

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

```
}
```

Fig. 9. Conflicting chunk extracted from Twitter4J project resulting of the merge 98caafc (common ancestor 5a6648).

opers in the resolution of this conflict. While such a heuristic would not solve all 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 the developer to automatically perform can significantly reduce effort.

The automated analysis confirms all the results from our manual analysis, with just minor variations (see Table 7). Half the conflicting chunks have a single language construct, 72% have one or two constructs, and 90% have up to four constructs – numbers that are remarkably similar. *Method invocation* is the most frequent language construct, appearing alone in 8% of the 175,805 conflicting chunks and in combination with other language constructs in 12% of the chunks (for a total of nearly 20%, a number just below the 23% of the manual analysis). The seven most frequent language constructs appearing individually are the same for both the manual and automated analysis, although they occupy different positions when ordered by frequency of appearance (see Table 8). The eight kinds of conflicts identified as most frequently appearing in Table 5 are also among the ten most frequently appearing according to the automated analysis.

TABLE 7

NUMBER OF CONFLICTING CHUNKS PER NUMBER OF LANGUAGE CONSTRUCTS (AUTOMATED ANALYSIS)

# Language constructs	# Conflicting chunks
1	87,899 (50%)
2	39,317 (22%)
3	19,875 (11%)
4	12,505 (7%)
5	7,064 (4%)
6	4,247 (2%)
7	2,498 (1%)
8	1,236 (1%)

TABLE 8

MOST FREQUENT LANGUAGE CONSTRUCTS (AUTOMATED ANALYSIS)

Language construct	Frequency among Language Constructs	Frequency among Kinds of Conflicts
Method invocation	75,045 (20%)	13,549 (7%)
Variable	64,613 (17%)	8,229 (4%)
Commentary	55,081 (14%)	19,447 (11%)
If statement	32,943 (8%)	7,570 (4%)
Import	24,267 (6%)	20,538 (11%)
Method signature	23,177 (6%)	3,606 (2%)
Method declaration	20,500 (5%)	3,632 (2%)
Annotation	12,458 (3%)	1,191 (0.6%)
Return statement	11,227 (3%)	207 (0.1%)
For statement	5,771 (1%)	299 (0.1%)

Some differences could be found in the association rules derived from the automated analysis as compared to the manual analysis. Shown in Table 9 as ordered by lift and having support greater or equal to 2%, we notice the lack of control flow commands, except for *return*. Common across all relations, however, are the involvement of *method invocation*, *signature*, and *declaration*, indicating that these are core language constructs involved in many different kinds of conflicts and thereby implying that it likely is challenging to devise new merge techniques or heuristics that focus solely on one of those language constructs in isolation. It instead is more likely that combinations must be addressed by specialized approaches.

TABLE 9

RELATION AMONG LANGUAGE CONSTRUCTS THAT BELONG TO CONFLICTING CHUNKS, SHOWING SUPPORT (SUP.), CONFIDENCE (CON.) AND LIFT (AUTOMATED ANALYSIS)

Association rule	Sup.	Con.	Lift
Method declaration, Method invocation → Method signature, Variable	2%	58%	8.2
Method signature, Variable → Method declaration, Method invocation	2%	33%	8.2
Method invocation, Method signature →Method declaration, Variable	2%	28%	6.5
Method declaration, Variable → Method invocation, Method signature	2%	53%	6.5
Method declaration, Method invocation →Method signature	3%	74%	5.6
Method signature → Method declaration, Method invocation	3%	22%	5.6
Method signature → Method declaration, Method invocation, Variable	2%	18%	5.5
Method declaration, Method invocation, Variable → Method signature	2%	72%	5.5
Method invocation, Method signature → Return statement	3%	33%	5.2
Return statement → Method invocation, Method signature	3%	43%	5.2

3.4 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, Fig. 10 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 less ineffective. To determine whether opportunities may exist for new merge tools to exploit this and other patterns, we first collected the data presented in Table 10 for the manual analysis. We determined the presence of dependencies ("with dependencies") by manually searching ('using 'grep') for usage of the same identifier across different chunks.

Across the five projects we analyzed, the percentage of failed merges that involve multiple conflicting chunks and exhibit dependencies among two or more of these chunks varies from 14% to 46%. To assess if any commonality exists, Table 11 presents the most frequent association rules for conflicts with multiple chunks, ordered by lift and with support and confidence thresholds of 10% and 50%, respectively. Table 11 is similar to Table 6, but instead of providing results for individual chunks, it presents association rules for entire failed merges. In line with the intra-chunk analysis performed previously, method-related language constructs again appear frequently on both sides of the rules. Comments, imports, conditional statements, and variables are also common.

TABLE 10

DEPENDENCIES IN FAILED MERGES (MANUAL ANALYSIS)

Projects	Failed merges		
	Total	With multiple chunks	With 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%)

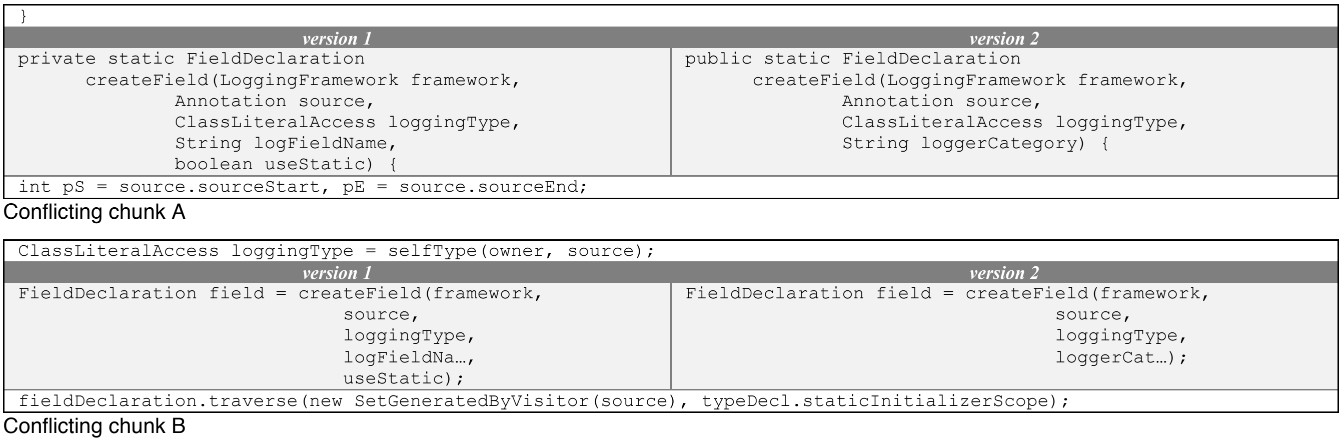
As compared to the association rules presented in Table 6, the association rules in Table 11 appear more complex as they involve more language constructs on both sides of the association rules. It may therefore not be as easy to explore any patterns that exist. Taking a closer look, however, we note the presence of *method declaration* or *method signature* in each of the rules, indicating that changes in method names, parameters, and return types appear to drive cross-chunk dependencies. The presence of *comments*, *imports*, and *if* statements in the association rules of Table 11 can be explained, then, by their semantic relation with methods (to explain a method, to point to the package containing types used in a method signature, and to denote conditional calls to a method). Therefore, it may still be possible that resolving the conflict by focusing on the method declaration first could gather useful information that a merge tool could take advantage of, in the best case automating the rest of the resolution necessary.

TABLE 11

RELATION AMONG LANGUAGE CONSTRUCTS THAT BELONG TO FAILED MERGES, SHOWING SUPPORT (SUP.), CONFIDENCE (CON.) AND LIFT (MANUAL ANALYSIS)

Association rule	Sup.	Con.	Lift
Import, method invocation, variable → if statement, method declaration	10%	68%	4.01
If statement, method declaration → import, method invocation, variable	10%	60%	4.01
Comment, if statement, variable → method invocation, method signature	12%	71%	4.00
Method invocation, method signature → comment, if statement, variable	12%	65%	4.00
If statement, import → method declaration, method invocation, variable	10%	79%	3.87
Method declaration, method invocation, variable → if statement, import	10%	50%	3.87
Comment, method declaration, variable → method invocation, method signature	12%	68%	3.84
Method invocation, method signature → comment, method declaration, variable	12%	65%	3.84
Import, method invocation → if statement, method declaration, variable	10%	63%	3.83
Method invocation, method signature → comment, method declaration, variable	12%	65%	3.84

This naturally leads to the observation that resolution of failed merges involving multiple conflicting chunks may be easier if the chunks are addressed in a particular order. In Fig. 11, for instance, in resolving the conflict in the *method signature* a developer makes decisions that lead to a single possible resolution for the chunk that declares the



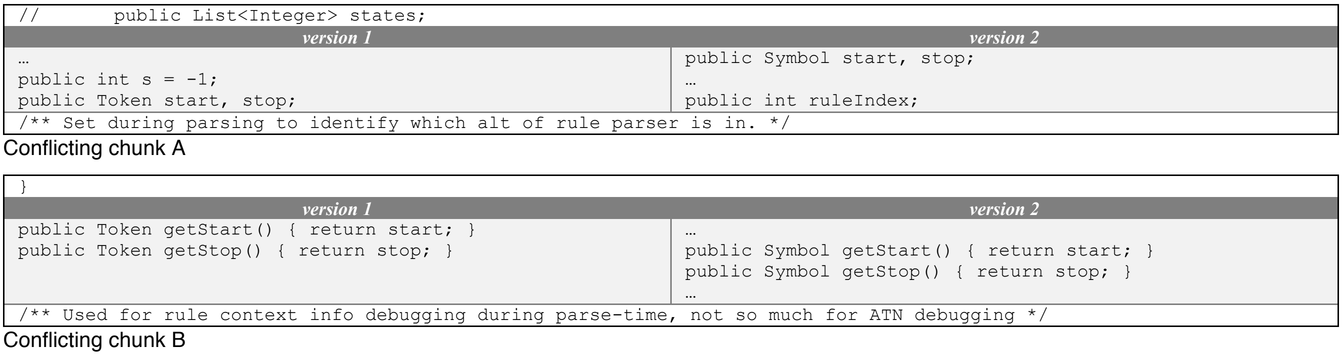
```
}
version 1
private static FieldDeclaration
      createField(LoggingFramework framework,
              Annotation source,
              ClassLiteralAccess loggingType,
              String logFieldName,
              boolean useStatic) {
version 2
public static FieldDeclaration
      createField(LoggingFramework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

Conflicting chunk A

```
ClassLiteralAccess loggingType = selfType(owner, source);
version 1
FieldDeclaration field = createField(framework,
                             source,
                             loggingType,
                             logFieldNa…,
                             useStatic);
version 2
FieldDeclaration field = createField(framework,
                                     source,
                                     loggingType,
                                     loggerCat…);
fieldDeclaration.traverse(new SetGeneratedByVisitor(source), typeDecl.staticInitializerScope);
```

Conflicting chunk B

Fig. 10. Dependent chunks of merge f956ba (common ancestor 7d5184) from project Lombok.



```
//       public List<Integer> states;
version 1
…
public int s = -1;
public Token start, stop;
version 2
public Symbol start, stop;
…
public int ruleIndex;
/** Set during parsing to identify which alt of rule parser is in. */
```

Conflicting chunk A

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

Fig. 11. Dependent chunks of merge 92ae0f (common ancestor 542e70) from project ANTLR4.

variables. Similar kinds of scenarios could arise for method signatures and invocations, return statements and return types specified in method signatures, changes in parameters (order or type), and so on. While many of the failed merges will require decisions from developers as part of their resolution, properly ordering the chunks so that decisions are taken in a sequence that allows automating or providing guidance for the following steps may prove to be a useful strategy for the next level of automated merge assistance.

Table 12 presents the association rules extracted by the automated analysis. Though these rules are not exactly the same as those found by the manual analysis, their essence is very similar: we still observe the omnipresence of *method declaration* as combined with method invocation, comments, variables, and conditional statements. Thus, the strategy of ordering the resolution of conflicting chunks to increase our ability to provide automatic resolution or guidance applies to the results found by the automated analysis.

TABLE 12

ASSOCIATION RULES OF MERGES WITH HIGHEST LIFT (AUTOMATED ANALYSIS)

Association rules	Sup.	Con.	Lift
If statement, Method declaration, Method invocation → Comment, Method signature, Variable	10%	67%	4.02
Comment, Method signature, Variable → If statement, Method declaration, Method invocation	10%	53%	4.02
If statement, Method declaration → Comment, Method invocation, Method signature, Variable	10%	64%	3.99
Comment, Method invocation, Method signature, Variable → If statement, Method declaration	10%	55%	3.99
Comment, Method invocation, Method signature → If statement, Method declaration, Variable	10%	52%	3.99
If statement, Method declaration, Variable → Comment, Method invocation, Method signature	10%	68%	3.99
If statement, Method declaration → Comment, Method signature, Variable	11%	65%	3.93
Comment, Method signature, Variable → If statement, Method declaration	11%	54%	3.93
If statement, Method declaration → Comment, Method invocation, Method signature	11%	65%	3.83
Comment, Method invocation, Method signature → If statement, Method declaration	11%	53%	3.83

3.5 What is the distribution of developer decisions?

Table 13 shows that, across all 616 conflicting chunks studied in the manual analysis, a primary choice that developers make is to select one of the versions, either *version 1* (*V1, 21%*) or *version 2* (*V2, 35%*). This was, to us, an unexpected result. Given all the commentary and folklore surrounding the merge problem and how difficult it is said to be to resolve merge conflicts [4], [21], we had expected *new code* (*NC*) to be the most common choice. It is not: only 19% of chunks are resolved by writing some new code as part of the resolution. This means that a full 81% of the conflicting chunks are resolved using only the contents from the code contained within the chunk without actually adding or editing lines of code (*V1*, *V2*, *CC*, and *CB*; note that *NN* is negligible, because it occurs so infrequently). While this does not necessarily mean that doing so is trivial, it does mean that merge resolution through one of these four strategies could be supported not by yet more automation, but perhaps by tool support that assists developers in making one of these four choices in the first place and, in case of *CC* and *CB*, by helping them select and order the necessary lines of code.

TABLE 13

HOW DEVELOPERS RESOLVE CONFLICTS (MANUAL AND AUTOMATED ANALYSIS)

Developer decision	Manual analysis	Automated analysis
Version 1	21%	50%
Version 2	35%	25%
Concatenation	12%	3%
Combination	13%	9%
New Code	19%	13%
None	0%	0%

Fig. 12 breaks down Table 13 by project, offering several interesting insights. First, the maximum amount of resolutions involving *NC* is 26%, which, while above the average of 19%, still reinforces that, for each of the projects, a majority of conflicting chunks is resolved without writing new code. Second, significant differences exist across projects. For instance, the dominant choice of *V2* in Voldemort (45%) is contrasted by a mere 4% in MCT. MCT, on the other hand, has a high percentage of resolutions involving *CB*, especially when compared to Voldemort, where this choice was rarely made. The presence of such unique trends might be useful when it comes to supporting developers, at a minimum by informing them of such tendencies, but it may also be possible to auto-select them based on certain factors.

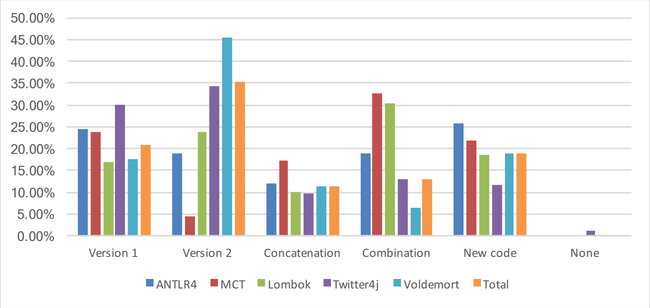


Fig. 12. How conflicting chunks are resolved in each project.

We also analyzed resolution choices per developer. One pattern immediately stood out: for four of the projects, a single developer resolved the majority of the conflicts, despite the fact that the projects had code contributions from 13 to 84 developers. This is not out of line with the historical open source practice of numerous contributors submitting patches, but few having commit privileges to integrate the patches into the main repository. 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 resolving most merge failures).

This information is summarized in Table 14, which lists, per project, all developers who performed at least one merge, the number of conflicting chunks each developer resolved (CH), the percentage of each decision (*V1*, *V2*, *CC*, *CB*, *NC*, *NN*) 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.

TABLE 14

HOW EACH DEVELOPER RESOLVES CONFLICTING CHUNKS

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

Several developers had no clear preference in resolution decisions, employing a variety of them, but others appear more likely to resolve conflicts in the same manner. For instance, developer Lei Gao resolved almost 50% of conflicting chunks by choosing *V2*, while Yusuke Yamamoto used *V1* and *CB* in approximately 30% of the cases each. Such patterns may be related to specific developers or project policies, but may also be related to conflict-specific characteristics – for instance resolving all conflicting chunks in a single failed merge in the same way by choosing one version, each time. Examples exist, however, of failed merges in which developers use a variety of strategies. For instance, facing a failed merge (970260) involving 28 conflicting chunks in Voldemort, developer Roshan Sumbaly chose *V1* four, *V2* 12, *CB* five, and *NC* seven times, respectively.

We assessed whether the number of conflicting chunks in a failed merge had any effect on developer decisions as to how to resolve the conflicts. Table 15 shows the results, with a clear shift visible from *V1* and *V2* being a more frequent choice when fewer conflicting chunks are present to *CB* and *NC* in case of higher numbers of conflicting chunks. Lombok, especially, has 70% of chunks resolved by *CB* or *NC* for merges involving 16 or more chunks; MCT, too, has 64% of its merges involving six to 15 conflicting chunks resolved by *CB* or *NC* (as Twitter4J and ANTLR4, MCT has no failed merges of 16 or more conflicting chunks). Twitter4J represents an interesting exception: only 27% of its failed merges involving six to 15 conflicting chunks used *CB* or *NC*. Again, this shows that projects may exhibit individual characteristics that may possibly be exploited.

TABLE 15

DISTRIBUTION OF DEVELOPER DECISIONS BY CONFLICTING CHUNKS, GROUPED BY THE RANGES FOR THE NUMBER OF CHUNKS (#CHUNKS), THE NUMBER OF FAILED MERGES IN THE RANGE (FM), AND THE TOTAL NUMBER OF CHUNKS IN THE RANGE (TOTAL)

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

Recall from the discussion in Section 2.3 that developers sometimes postpone resolution of a conflict. We therefore did not only analyze the immediate next commit, but also the commits up to one month after the original commit to examine whether a developer returned to the conflict later to choose a different resolution. The conflicting chunk in Fig. 13, for instance, involved a postponed resolution, since, at first, the developer selected *V2*, but later changed to *NC* (*V1* with '*<arg>*' replaced by *<arg>*).

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) was 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 ones. The majority of these commits were of this nature, but six of the commits represented postponed merges (4%, representing only 2%

```
*/
```

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),
```

Fig. 13. Conflicting chunk extracted from project ANTLR4 regarding merge d85ea0 (common ancestor 5bd415) that was resolved by a commit using *V2*, but in a latter commit the resolution was changed to *NC*.

version 1	*version 2*

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

Fig. 14. Conflicting chunk extracted from project Lombok regarding merge 4e152f (common ancestor 2bdc12) 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
…
```

Fig. 15. Sequence of two rebases (a21bf2, 234ac9) followed by merge 3fbef9 (common ancestor cd19e8), all with postponed resolutions, in project Voldemort.

of all conflicting chunks). In manually examining the six, no pattern or reason seems to dominate. In one case, the developer adjusted the *variable* name due to a rename refactoring that they postponed (see Fig. 14). In other cases, the developers added comments or simply left the source code in conflict to resolve it afterwards.

For instance, Fig. 15 represents a situation in which two developers worked in parallel on the same class on the same day, but chose to postpone the merge of a comment section multiple times; they instead performed a series of rebases, merging without removing the merge markers in the comments section. One week later one of them finally committed (05f23e) an update to the comments that finally removed merge markers, with a message "code clean up". Across all 14 conflicting chunks that were part of postponed merges, the dominant developer decision for the first resolution was *NC* (eight occurrences) followed by *V2* (five occurrences). From this, we can conclude that postponed merges seem to address situations that are complex, and generally require new code to be written.

Switching to the results of the automated analysis, Table 13 shows the distribution of how developers resolved conflicting chunks across all 2,731 projects. While we observe some sizeable differences (*V1* is chosen 50% of the time as opposed to just 21% in the five projects we manually analyzed; *V2* 25% instead of 35%; *NC* 13% instead of 19%), the primary observation from the manual analysis still stands: just 13% of the conflicting chunks involve new code, meaning that in 87% of the cases, all of the code that is necessary to resolve a conflict already exists and is present in the two versions in conflict. This once again shows that the source code used in resolving conflicting chunks is frequently present in one of the versions, either as a full resolution or its parts. Indeed, a full three quarters of the conflicting chunks were resolved simply by choosing *V1* or *V2*.

TABLE 16

CHANGES IN CONFLICTING CHUNKS AREAS

Project	Merges		Conflicting Chunks	
	Changed	Postponed	Changed	Postponed
ANTLR4	3 (14%)	1 (5%)	3 (4%)	1 (1%)
Lombok	5 (28%)	1 (6%)	15 (25%)	2 (3%)
MCT	5 (36%)	0 (0%)	17 (37%)	0 (0%)
Twitter4J	4 (11%)	0 (0%)	7 (8%)	0 (0%)
Voldemort	20 (35%)	4 (7%)	58 (17%)	11 (3%)
Total	37 (25%)	6 (4%)	100 (16%)	14 (2%)

Interestingly, the choice of *V1* is twice as frequent as the choice of *V2*, which is in contrast to our findings for the manual analysis. This can be explained if we consider that one of the primary reasons why developers resolve failed merges is to integrate the changes they made on implicit branches. In this case, a bias seems to exist in that they choose their own code over what changes exist in the repository.

Fig. 16 provides a closer look at the developer decisions, this time plotting individual projects and the percentage of their conflicting chunks that were resolved by a given decision (every project, then, is part of each box and whisker in Fig. 16). The plot reveals a number of interesting patterns. First, each developer decision has at least one project for which all the decisions were of that type (e.g., all *V1* or all *NC*). In some ways, this is not surprising, because, as we discussed in our answer to A1, 40% of failed merges involve just one conflicting chunk and it would be highly unlikely if for one reason or another one of the developer decisions is not represented in this 40%. At the same time, it confirms that programmers use every kind of decision, even in the simplest case of just one conflicting chunk.

Returning to Table 13 and the 13% of *NC* decisions that were made overall, we observe that they are scattered over 75% of the projects (the first quartile of the box plot for *NC* lies at 0). This has two implications. First, it implies that, in 25% of the projects, all the merge failures were resolved by using existing code from *V1* and *V2*. Second, in examining the box plot for *NC*, we note that some projects encounter *NC* as the resolution strategy more frequently among their respective failed merges (indeed, we see a good number of projects with roughly 50% to nearly 100% of *new code*).

Finally, confirming what we found in our manual analysis, *NN* (*none*) is the least frequently used developer decision. Its box plot is a flat line at the origin, followed by a few outliers, including two projects in which 100% of conflicting chunks were resolved with *none*: Firefly and Bookie-Android. Both of these two projects had one single failed merge involving just one conflicting chunk, accounting for the anomaly. These two projects are a clear exception: they represent just 0.07% of our project corpus.

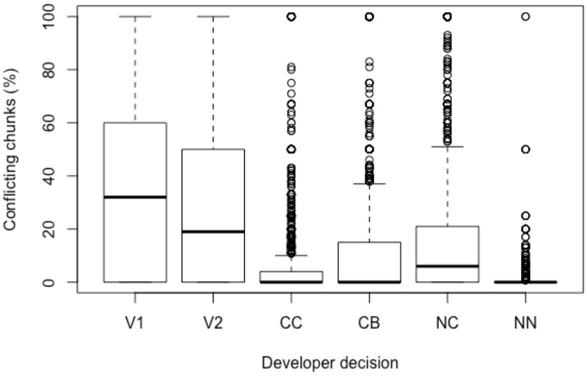


Fig. 16. Box-plots for the distribution of developer decisions.

We did not attempt to account for postponed merges in our automated analysis. On one hand, it is difficult to envision a heuristic that would work fully automatically. On the other hand, the number of failed merges to examine is too high to reasonably assess the next month's check-ins manually.

3.6 What is the distribution in difficulty level of kinds of conflicts?

Based on the concept of straightforward and complex chunks as introduced in Section 2, we analyzed the approximate difficulty of different kinds of conflicts with the help of the following difficulty ratio:

$$DR = \frac{\#complex\ chunks}{\#complex\ chunks + \#straightforward\ chunks}$$

For each kind of conflict, we took all its conflicting chunks and assigned to each chunk the label of straightforward or complex based on the developer decision made for that chunk. Then, we calculated the conflict's difficulty ratio: the higher this value, the more difficult the kind of conflict appears to resolve. Table 17 shows the kinds of conflicts that appear in more than ten conflicting chunks, in descending order of difficulty ratio (*DR*), as found by our manual analysis.

The kinds of conflicts consisting solely of *variable* or *import* seem to be the easiest to resolve, since their resolution rarely involves writing new code. *CC* is the most frequent decision in resolving these kinds of conflicts, with *V1* and *V2* also chosen frequently. Perhaps counter-intuitively, *CB* also is used at times. This represents situations in which multiple *variables* or multiple *imports* that are in conflict are combined (either all or as a subset) in a different order than straightforward concatenation. An example is shown in Fig. 17, with the developer including both versions of the conflicting chunk, but reordering some *imports* in an order they preferred.

At the other end of the spectrum, the kinds of conflicts of *comment* and *if statement, method invocation* seem to be most difficult to resolve, as their relative proportion of *CB* and *NC* as the developer decision is the highest. The presence of *comment* as the top difficult kind of conflict was surprising to us. On one hand, comments are normally written in natural language and thus may not be a good match for automation. On the other hand, it is rare to hear about comments being a major source of frustration or difficulty when it comes to merging. We return to this in the context of the automated analysis.

TABLE 17

DIFFICULTY RATIO OF RESOLVING KINDS OF CONFLICTS (MANUAL ANALYSIS)

Kind of conflict	V1	V2	CC	CB	NC	NN	DR
Comment	4 29%	4 29%	0 0%	1 7%	5 36%	0 0%	43%
If statement, method invocation	2 17%	3 25%	2 17%	2 17%	3 25%	0 0%	42%
Method invocation, variable	17 29%	20 35%	1 2%	10 17%	10 17%	0 0%	35%
Method invocation	6 10%	31 49%	5 8%	10 16%	10 16%	1 2%	32%
Method signature	5 26%	8 42%	0 0%	1 5%	5 26%	0 0%	32%
If statement	6 30%	7 35%	1 5%	1 5%	5 25%	0 0%	30%
Comment, method declaration	5 29%	5 29%	2 12%	2 12%	2 12%	1 6%	24%
Method declaration	10 18%	24 42%	11 19%	4 7%	8 14%	0 0%	21%
If statement, method invocation, variable	4 21%	10 53%	1 5%	2 11%	2 11%	0 0%	21%
Import	12 20%	14 23%	22 37%	10 17%	2 3%	0 0%	20%
Variable	9 24%	8 22%	13 35%	4 11%	3 8%	0 0%	19%

That the *if statement, method invocation* kind of conflict is among the most difficult to resolve surprised us, especially in comparison to the *if statement, method invocation, variable* kind of conflict. The latter has more language constructs involved, yet appears significantly easier to address (with a *DR* score of 21% compared to 42% for *if statement, method invocation*). 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* assigning a *variable* a value, with the *if statement, method invocation* typically involving more complex behavior. The example in Fig. 18 shows a situation in which *CC* and even *CB* would fail: the message id of Native Backup had to be updated to 30 in order to preserve uniqueness of messages in the output stream.

A final highlight concerns the *method declaration* kind of conflict. Developers decide upon one version or the other in the vast majority of cases (*V1* in 10 cases, *V2* in 24) and quite frequently 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.

Table 18 shows the results of the same analysis as Table 17, but repeated for our automated analysis. The top ten kinds of conflict appearing in more than 2,000 chunks are shown, ordered by their difficulty ratio. Although seven out of 10 kinds of conflicts appear in both tables, the difficulty ratios that result from the automated analysis are quite different from those of the manual analysis. For instance, in stark contrast to the results found in the manual analysis, *comment* turns out to be easier to resolve (difficulty ratio equal to 5%), not even appearing among the most complex kinds of conflicts for the automated analysis. Though very distinct from the results of the manual analysis in terms of difficulty ratio, results from the automated analysis are more in line with what might reasonably be expected. We are unsure what caused the bias in the five projects of the manual analysis, other than that *comment* is the second least common kind of conflict in the manual analysis (meeting our criterion of at least 10 occurrences) and it had six out of 14 being resolved through *CB* or *NC*, meaning some selection bias likely occurred.

TABLE 18

DIFFICULTY RATIO OF RESOLVING KINDS OF CONFLICTS (AUTOMATED ANALYSIS)

Kind of conflict	V1	V2	CC	CB	NC	NN	DR
Comment, Method invocation, Variable	1048 40%	789 30%	57 2%	313 12%	437 16%	6 0%	28%
Method invocation, Variable	6079 43%	3940 28%	258 2%	1341 9%	2533 18%	31 0%	27%
If statement	3773 50%	1688 22%	47 1%	128 2%	1932 26%	2 0%	27%
If statement, Method invocation, Variable	1512 45%	905 27%	40 1%	436 13%	457 14%	4 0%	27%
Import	7338 36%	5377 26%	2058 10%	3493 17%	1964 10%	308 1%	27%
If statement, Method invocation	1326 49%	638 23%	59 2%	410 15%	290 11%	2 0%	26%
Variable	4038 49%	2011 24%	455 6%	503 6%	1206 15%	16 0%	21%
Comment, Variable	1171 51%	496 21%	170 7%	245 11%	224 10%	5 0%	20%
Annotation, Method declaration	926 45%	590 29%	137 7%	132 6%	250 12%	6 0%	19%
Comment, Method declaration	1197 48%	769 31%	84 3%	167 7%	264 11%	7 0%	17%

Method declaration combined with *annotation* or *comment* figure among the easiest kinds of conflict to resolve – developers typically pick the new version (*V1*, in 45% or 48% of the chunks, respectively), the old one (*V2*, in 29% or 31% of the chunks), or concatenate both (*CC*, 7% or 3% of the chunks, respectively). These results, despite *V1* and *V2* switching in relative frequency compared to the manual analysis for *method declaration* in isolation, overall confirm its finding of *method declaration* being easy to resolve. Interestingly, *if statement* (27%), *if statement, method invocation* (26%), and *if statement, method invocation, variable* (27%) are among the most difficult kinds of conflicts to resolve. Based on a small sample of conflicting changes involving these constructs, we believe that this may be because *if statements* often govern the semantics of a program. As in the example of Fig. 18, our informal sample contained

```
import com.sun.tools.javac.tree.JCTree.JCLiteral;
                      version 1                                          version 2
import com.sun.tools.javac.tree.JCTree.JCPrimit…;   import com.sun.tools.javac.tree.JCTree.JCModifi…;
import com.sun.tools.javac.tree.JCTree.JCUnary;     import com.sun.tools.javac.tree.JCTree.JCTypePa…;
import com.sun.tools.javac.tree.TreeMaker           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.JCModifi…;
import com.sun.tools.javac.tree.JCTree.JCPrimit…;
import com.sun.tools.javac.tree.JCTree.JCTypePa…;
import com.sun.tools.javac.tree.JCTree.JCUnary;
import com.sun.tools.javac.tree.TreeMaker;
import com.sun.tools.javac.util.List;
import com.sun.tools.javac.util.Name;
/**
```

Fig. 17. Conflicting chunks among *import declarations* from project Lombok regarding merge 45697b (common ancestor 620616).

```
}
                      version 1                                          version 2
if (hasInitiateRebalanceNodeOnDonor()) {             if (hasNativeBackup()) {
  output.write…(28, getInitiateRebalanceNode…());      output.write…(28, getNativeBackup());
}
if (hasDeleteStoreRebalanceState()) {
  output.write…(29, getDeleteStoreRebalanceS…());

}
```

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

Fig. 18. Conflicting chunk from project Voldemort from merge 491863 (common ancestor 74e0d9).

many instances in which the changes in the *if* clause itself needed to be met with semantic changes in the contents of its body.

We were somewhat surprised at the high difficulty ratio for *import*. Part of this can be attributed to the frequent use of *CB* to selectively incorporate imports from both versions. The use of *NC* to address situations in which a set of imports is replaced by a single import with a wildcard or situations in which they wanted to include a complementary import (e.g., *ArrayList* in addition to *List*) explains most of the other situations. In both cases, we note that semi-automated support could exploit these patterns to provide assistance to the developer.

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

Our final analysis focuses on the possible relationships between language constructs and developer decisions, which we explored through association rules. Table 19 shows association rules with confidence threshold of 30% and absolute support threshold of six occurrences among the conflicting chunks of the five projects selected for manual analysis. The top ten rules are shown, ordered by lift. 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, a few patterns still exist.

TABLE 19

RELATION BETWEEN LANGUAGE CONSTRUCTS AND RESOLUTIONS, SHOWING SUPPORT (SUP.), CONFIDENCE (CON.) AND LIFT (MANUAL ANALYSIS)

Association rules	Sup.	Con.	Lift
Annotation, variable → NC	1%	58%	3.07
Import declaration → CC	4%	34%	2.98
For statement → CB	1%	33%	2.50
Method invocation, try statement → CB	1%	32%	2.40
Comment, method signature → V2	2%	67%	1.88
For statement, variable → V2	1%	64%	1.82
Method signature, variable → V2	2%	60%	1.70
Annotation → NC	1%	32%	1.68
Return statement → NC	1%	31%	1.62
Method invocation, meth. signature, variable → V2	1%	57%	1.61

First, we note that four out of ten cases involve *V2* as the developer decision. On one hand, having *V2* as a common choice developers make is, in a way, not surprising. By choosing *V2* over *V1*, the developer resolving a conflict is choosing to keep established code that was added by other developers to the repository, instead of overwriting these changes with those they made in their local repository. Existing changes do impose a certain degree of inertia and may cause developers to refrain from modifying them. On the other hand, these results contrast with earlier results from the automated analysis, which seem to favor *V1* over *V2*. We return to this topic shortly below.

Second, Table 19 includes the association rule *import* *declaration* → *CC* with 34% of confidence, meaning that in 34% of conflicting chunks that have import declarations, the developer decision is *CC*. Moreover, this association rule has lift 2.98, denoting a strong relation between *import* and *CC* (when import is involved, it increases the chances of choosing *CC* by 198%). A question is why developers do not choose *CC* all the time, as only 37% of developer decisions are *CC* (Table 17) when *import* is the conflict. The issue here is the fact that resolving the chunk pertaining to the *import* does not take place in isolation – other resolutions, to other conflicting chunks, dictate what import declarations are ultimately needed. That is, developers appear to take care to not over-include *imports* when they do not need them.

Table 20 provides the results of the automated analysis, as ordered by lift. Compared to the manual analysis, we dropped the support and confidence thresholds to 0.1% and 30%, respectively. With 175,585 chunks, too high a support threshold filtered developer decisions that are relatively rare but nonetheless frequently enough (i.e., at least 175 times at a support of 0.1%) to warrant a closer examination. Table 20 shows the top ten association rules relating kinds of conflicts (on the left-hand side) to developers' decisions (right-hand side). This table differs significantly from the results of the manual analysis, both in the language constructs involved in the rules and the developer decisions. While rules found through our manual analysis are spread over distinct language constructs, from annotations to exception handling, rules collected from the automated analysis are more uniform, showing a prominence of method-related constructs, as co-occurring with comments, control flow statements, and references to variables. Furthermore, we notice that all rules are resolved either through the addition of new code or by picking V1.

TABLE 20

RELATION BETWEEN LANGUAGE CONSTRUCTS AND RESOLUTIONS, SHOWING SUPPORT (SUP.), CONFIDENCE (CON.) AND LIFT (AUTOMATED ANALYSIS)

Association rules	Sup.	Con.	Lift
Method invocation, Method signature, Return statement, Try statement → NC	0.1%	31%	2.39
Method declaration, Method invocation, Method signature, Try statement, Variable → NC	0.1%	30%	2.36
Method declaration, Method signature, Try statement, Variable → NC	0.1%	30%	2.36
Method declaration, Method invocation, Try statement, Variable → NC	0.1%	30%	2.32
Method declaration, Try statement, Variable → NC	0.1%	30%	2.32
Method signature, Return statement, Try statement → NC	0.1%	30%	2.31
Comment, Do statement, If statement → V1	0.1%	95%	1.89
Comment, Do statement, If statement, Method invocation → V1	0.1%	95%	1.89
Comment, Do statement, If statement, Variable → V1	0.1%	95%	1.89
Comment, Do statement, If statement, Method invocation, Variable → V1	0.1%	95%	1.89

If we repeat the analysis of Table 20 without any confidence threshold, association rules with higher lift appear in the top 10. Particularly, the rule with the highest lift turns out to be *import* → *NN* (support 0.1%, confidence 1%, lift 3.58), while *import* → *CC* (support 1.2%, confidence 9%, lift 2.59) also appears. Interestingly, all remaining rules in the top 10 have *CC* or *CB* in the consequent, with lift ranging from 2.90 to 2.41, but confidence dropping to the range of 8% to 22%. As none of these rules, except *import* → *CC*, belong to the top 10 list of the manual analysis (Table 19), results from the automated analysis highlight that the results of our manual analysis are not representative for a larger set of projects. On the other hand, the persistence of method-related rules points to the fact that it could be beneficial to focus the design of new merge techniques on such constructs – in line with results from our previous analyses above.

4 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 the nature of merge conflicts in detail, is it 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 below.

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. New techniques that may work for the former likely will not work so well for the latter, and vice versa. Even when conflicts are quite similar, of roughly the same size and with the same language constructs involved, it is apparent that developers still choose different ways of resolving them.

Based on our results, then, it is difficult to envision a single generic merge technique that can automatically resolve all possible conflicts, since the diversity in conflicts we encountered is simply too great. At the same time, we believe it is possible to improve over the existing state-of-the-art of tools that use a single technique to attempt merging any and all conflict and that defer resolution of conflicts that do not fit its technique to the user for manual resolution.

We envision two avenues forward. First, our results seem to indicate that it is possible to improve over current merge techniques by creating a portfolio of highly specific complementary techniques that each can resolve a type of conflict that current techniques cannot handle. Plugged-in to existing tools to handle the exceptions that normally occur, this should alleviate developers from encountering some subset of merge conflicts altogether. Second, for the conflicts that still cannot be addressed with such complementary techniques, our results suggest creating advanced tool support for assisting developers in easily resolving conflicts manually. For instance, it may be possible to create intelligent interfaces that enable developers to choose from several 'most likely' precomputed candidate resolutions. Overall, such tool support would create solutions that are neither fully automated nor fully manual, but nonetheless help developers in resolving conflicts that today take significant time.

In the below, we substantiate why we believe these two avenues forward are feasible with three conclusions that we draw from the results in the previous section.

Conflicting chunks generally contain all the necessary information to resolve them. As shown in Table 13, only 13% of conflicting chunks require developers to produce extra code beyond the code already present in the chunk (as part of version 1, version 2, or both), given that the developers chose to resolve conflicting chunks via *V1*, *V2*, *CC*, *CB*, and *NN* in 87% of all cases (automated analysis). This means that the necessary resources for resolving conflicts are already present and available inside the conflicting chunks in the majority of conflicts. While a developer certainly may need to examine other parts of the code and spend time thinking on how to proceed, for the actual change that must be made, the source code lines in the conflicting chunk suffice. An example is shown in Fig. 20, where the solution is to concatenate the contents of version 1 (i.e., *variable*) and version 2 (i.e., *annotation and method declaration*).

More complex examples exist as well, where the developer interleaves lines of code from version 1 and version 2. Fig. 19 showed an example of this. While the resolution is non-trivial, with the developer needing to carefully order the lines of code from both versions, no new code was written.

To assist developers in performing these kinds of manual resolutions, it is necessary to re-envision merge tools, for example in supporting 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 creation of various heuristics, as the number of possible permutations suffers from a combinatorial explosion. A combination of search-based software engineering techniques with smart ways of pruning the search tree will be necessary to identify most likely candidates.

Important to the feasibility of this approach is that many conflicting chunks are small. In 94% of the cases, the conflicting chunk involved less than 50 lines of code in each of its versions. More strikingly, 40% of merge failures involved just a single chunk, and the median size of the versions in conflicting chunks was 2 (*version 1*) and 2.5 *(version 2*) across all the projects from the automated analysis. The search space, then, is often small and even if the search space cannot be pruned by much, the resulting number of choices to present to the developer is limited and likely can be ordered in some way representing the prospect of the result being applicable.

Resolution order of conflicting chunks can matter. About 60% of failed merges in the manual analysis consist of multiple conflicting chunks (Fig. 4), with 29% exhibiting dependencies (Table 10). In existing merge tools, once a merge fails, the developer is presented with all conflicting code at once for them to resolve. 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, the developer is left to their own devices, manually working out the desired result.

We observe that resolving conflicting chunks in a given order often can be beneficial. For instance, Fig. 10 presented a pair of conflicts, one concerning changes to the *method signature* of *createField* (conflicting chunk A) and the other concerning changes to one if its *method invocations* (conflicting chunk B). Resolving chunk A before chunk B is beneficial, as the choice of *method signature* decides the subsequent choice of *method invocation*. As another example, returning to the ANTLR4 example in Fig. 11, we know which *method* to choose should the *variable* be resolved (with, in this case, the reverse also working: once the return

```
}
```

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
```

Fig. 19. Conflicting chunk with *annotation*, *comment*, and *method declaration* from project ANTLR4 regarding merge 18f535 (common ancestor ea7037).

type is chosen, the type of the variable can be resolved).

We note that the association rules listed in Table 6 and Table 9 can be a source of support for merge tools that seek to assist developers in ordering the conflicting chunks for resolution. For instance, the association rule *return statement* → *method invocation* (Table 6) indicates that 69% of the chunks with conflicts in *return statement* also exhibit conflicts in *method invocation*. As the opposite rule (*method invocation* → *return statement*) does not even appear in Table 6 due to being lower than the confidence threshold of 50%, this indicates that changes in the *return statement* imply changes in *method invocation*, and not the other way around – giving directionality to the order in which a tool presents the developer the chunks to resolve.

Of course, not every set of conflicting chunks is covered by our set of association rules and, even when they are, the order implied will not always be the order in which a developer performs resolution. This means that any merge tool that builds on this information should probably take an advisory role: instead of automation, it should offer suggestions from which a 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 some expert system that, together with some general rules inspired by the Java grammar, can field queries as to what the best order might be for a given subset of conflicting chunks. In this context, various heuristics will need to be developed and tested for effectiveness.

Finally, we note that, as conflicting chunks are resolved by the developer, more information becomes available that can make it possible for the merge tool to resume merging automatically again. In the case of an *import* and a *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 which *import* conflicting chunk to use. This is equally true in the examples of Fig. 10 and Fig. 11. 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. Merge tools, thus, should not outright fail when they encounter a situation they cannot resolve, but instead seek input when they need it to help them direct their automated efforts.

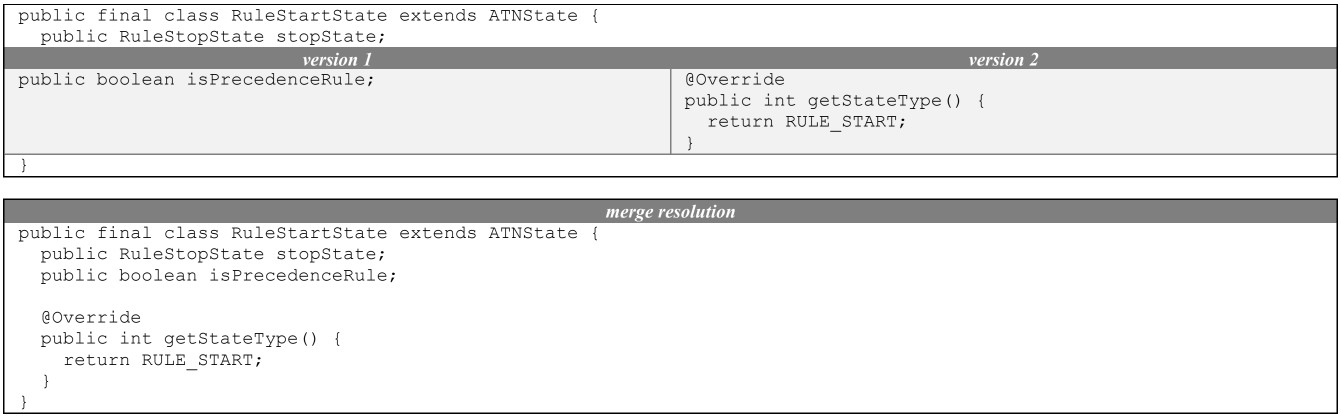
Underlying the above observations is the fact that a limited set of language constructs and combinations thereof is most common to occur in conflicting chunks (see Table 3 and Table 7). This means that it should not be necessary to have to design a plethora of different heuristics, but that a set of strategically chosen heuristics tackling the more frequent cases should suffice to reduce developer effort.

Past choices of how conflicting chunks were resolved can inform future choices. With today's merge tools, each new merge is performed afresh, without any history. This, however, leaves an important opportunity on the table. Our results indicate that some (certainly not all) developers exhibit patterns in their choices in terms of how they resolve conflicts over time. A possible extension to existing tools would be to identify such historical patterns, and present them to developers when similar situations appear (i.e., one could imagine a tool that communicates to a developer 'In the past, you resolved 16 conflicts similar to this one, eight of those by choosing version 2, six by choosing version 1, and two 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 can be helpful to resolve multiple conflicting chunks in a specific order, it may be possible for a tool to learn what different preferred orders are for different situations, as based on the kinds of conflicts, association rules, and past ordering choices of the developer.

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 where a conflict in method parameters exists and where one of the conflicting parameters is used in newly written code in version 2, that parameter must be renamed according to 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 [32] and that repetitive patterns of change exist. 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



```
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;
  }
}
```

Fig. 20. Conflicting chunk (top) and its resolution (bottom) of merge b14ca56 (common ancestor f7d0ca) from project ANTLR4.

come up with.

5 Threats to Validity

Concerning internal validity, language constructs were extracted manually by a single researcher for the manual analysis, which may have inadvertently introduced data collection errors. To mitigate this risk, two of the remaining authors verified the language constructs that were extracted, discussing with the first author any discrepancies to fix rare errors. Further, we cross-checked the results from the automated analysis with those of the manual analysis for the five projects they have in common, ensuring consistency among both analyses. Finally, we note that tallying language constructs is not a subjective or ambiguous task – a construct is there or not. As compared to coding conversations in an online help forum, for instance, less risk exists for biasing results.

Similarly, it is possible that we misclassified developer decisions. For the manual analysis, we used the same strategies as for language constructs: two other authors verified the developer decisions independently and we compared the results from the manual analysis to those of the automated analysis. For the automated analysis, however, it is still possible that our scripts and heuristics do not address certain edge cases that might be present in the repository. We performed random checking on the decisions and also studied unexpected results (e.g., chunks without language constructs, none of which turned out to be misclassified because of possible parser errors).

Also concerning internal validity, although Git tracks the version history of a project precisely as it occurred and generally provides a faithful record, it does allow history rewriting. In particular, the rebase command eliminates any trace of the branch being integrated in serializing the version history. It is known that some development teams use rebase instead of merge to reintegrate implicit branches [4], meaning that an after-the-fact analysis such as ours will not identify these actions as actual merges. The number of merges we analyzed, thus, is a lower bound as compared to the actual total number of merges in the projects. Moreover, it is possible that rebase merges exhibit different patterns from the ones we found, in which case our results would not be fully representative.

With respect to construct validity, which refers to a misalignment between a study's intent and its design, all analyzed data stems from open source software projects, capturing real failed merges that took place and were resolved by developers in practice. Thus, our results document what we intended to observe: how developers tackle merge conflicts. Of course, our study only observes after-the-fact outcomes, not the detailed strategies a developer uses leading up to the eventual resolution. We, thus, might still miss important information concerning their thought process and actions. At the same time, our detailed analysis based on language constructs and developer decisions adds an important dimension to the literature on merge conflicts.

Another threat concerning construct validity stems from the fact that we replayed history by using the default three-way merge of Git to detect conflicts. Consequently, we may have reported false positives in cases where the developers used external tools to perform their merges and the external tools produced different results. Git's three-way merge, however, is the default and is highly convenient to use since it is part of all Git front-end GUIs to perform merges. Moreover, according to Git's manual (https://git-scm.com/docs/git-merge), its three-way merge "has been found to result in fewer merge conflicts without causing mismerges by tests done on actual merge commits taken from the Linux 2.6 kernel development history." As such, we believe the risk of significant numbers of projects not using the three-way merge from Git is low.

A third threat to construct validity concerns the difficulty ratio metric, which we use to discriminate among kinds of conflicts and how difficult they appear to resolve. Clearly, more precise characterizations can be built (for instance taking into account the size of chunks or exactly which language constructs are involved) that in future can perhaps even accurately predict individual conflicts and how long it will take a developer to resolve them.

Regarding conclusion validity, first we note that several of the association rules we identified have relatively small support. Therefore, some of them may not hold in case an even larger population of projects is studied. On the other hand, our study identified a set of patterns for which their frequent occurrences are difficult to accredit to chance.

One last threat to conclusion validity is the lack of postponed merge collection for the automated analysis, which may have led to a smaller number of NC occurrences in the distribution of developer decisions for these projects. The difficulties related to automating this task, along with the small frequency with which postponed merges appear in the manual analysis, inhibited performing this analysis.

Finally, regarding external validity, we note that the ability to generalize our findings is restricted by the characteristics common to our selected projects: all of them are open source, with the majority of their code written in Java. Thus, our results do not necessarily generalize to industrial projects or open source projects written in C#, for instance. Our analysis, though, has shown that patterns exist in the sample we chose, and we strongly believe similar kinds of patterns may be present in other projects and languages. Additional study is needed to assess this.

6 Related Work

All software merge techniques exhibit tradeoffs among precision (the percentage of parallel changes they successfully can merge), generality (the types of files they support), and performance (the time it takes to perform merges). The first merge techniques (e.g., [3]–[8]) were unstructured, prioritizing performance and generality over precision. Building on lines of code as the unit of comparison and resolution [33] enabled these techniques to merge any textual file at a reasonable speed. Because most source code is stored as text files, unstructured techniques today are still the preferred choice in most version control systems. However, the use of lines as the unit of comparison can lead to false positives (e.g., two developers add two different method declarations in the same region) and false negatives (e.g., a developer removes a method declaration, while a second developer adds an invocation to the removed method declaration), compromising precision. Moreover, whether or not a conflict is flagged depends on particulars of the unstructured version control systems in question. For instance, Git flags a conflict if two different adjacent lines are changed, while Darcs and Subversion do not [27]. Thus, a precise rule to identify conflicts is absent. However, Nguyen and Ignat [27] suggest that Subversion's way is most appropriate.

In attempts to reduce false positives and false negatives, researchers started exploring structured merge techniques, which are less general and usually slower, but take into account the syntax [9]–[15] or the semantics [16], [17] of the code. Westfechtel [15], for instance, proposed a technique that uses a context-free grammar and, as another example, Binkley et al. [10] introduced a technique that considers the behavior of procedure calls. While these techniques represented important steps toward structured merge, they still failed in relatively simple situations such as a rename. Hunt and Tichy [12] therefore proposed an extensible, language-aware technique that can deal with renaming and non-local conflicts, though it too has limitations in not identifying behavioral differences caused by dynamic binding, as one example. While structured merge techniques have improved precision, the cost of being language specific (less general) and typically more computationally expensive (less performant) seems to have prohibited widespread use in practice to date.

In attempts to address precision, generality, and performance together, researchers have begun exploring semi-structured techniques, combining aspects of both unstructured and structured merge techniques [2], [18], [19]. Apel et al. [19], for instance, parameterize their merge tool using language grammars to both increase generality (in adopting unstructured merge as the base) and precision (in incorporating structured techniques to resolve potential conflicts in a language-specific manner). While the initial solution suffered in performance as a consequence, the authors tackled this in subsequent work [2], [18].

Other research focuses on reducing the number of failed merges through tools that alert developers about parallel work and potential conflicts early, at the time they are beginning to develop. CASI [34], CloudStudio [35], CoDesign [36], CollabVS [37], Crystal [22], FastDash [38], Palantír [39], SafeCommit [40], Syde [41], and WeCode [42] all monitor developer workspaces to detect when two developers begin modifying the same file or different files that exhibit some syntactic or semantic dependency. If they detect such an instance, they inform the involved developers, who then are expected to coordinate to address the emergent issue, for instance by one of them postponing their changes until the other is done. Even though these approaches play an important role in minimizing the incidence of conflicts, they still do not guarantee conflict-free merges. Different factors may lead to failed merges even when these approaches are in place, such as: (1) developers working on project forks that eventually need to be reintegrated, (2) the nature of some parallel changes (e.g., bug-fixes and new features over the same component), and (3) offline changes.

Despite all these efforts, today's merge techniques still require user intervention to resolve failed merges. Recent studies report that approximately 10% to 20% of all merges fail [22], [23], with some projects experiencing a failure rate of nearly 50% [22], [24]. These studies focused on unstructured merge tools, given their popularity in practice. While some of the merge failures could theoretically have been avoided with a structured (or semi-structured) merge tool, the fact that the use of unstructured merge tools remains so prevalent means that our findings are relevant to today's practices and may lead to tools and techniques that better integrate with those practices.

Only a few studies have begun to address the questions of how conflicts arise and how they are resolved (e.g., [2], [22], [23], [25], [26], [43]). Brun et al. [22] studied the history of nine projects and found that 16% of merges have structural failures (code), 1% syntactic (compilation) failures, and 6% semantic (test) failures. Kasi and Sarma [23] made a similar analysis of four projects, not only finding high percentages of merge failures (40%, 44%, 34%, and 54%), but also that, on average, 14% of merges have structural failures, 8% syntactic failures, and 22% semantic failures. Moreover, the authors observed that the number of days the conflict existed in the repository ranged from approximately three to about 16. Nguyen and Ignat [27] analyzed four open source projects to understand the relationship between the integration rate (i.e., number of concurrently modified files over all modified files) and conflict rate (i.e., number of files with unresolved conflicts over the ones concurrently modified). They found an unexpected result: the lower the integration rate, the higher the conflict rate. They further found that developers generally roll back to a prior version when facing a syntactic or semantic conflict. Leßenich et al. [28] proposed seven indicators for identifying merge conflicts, for example based on the number of commits in a branch, the number of commits in a timespan, and the number of files changed in parallel. However, none of the seven indicators have a strong correlation with the number of failed merges. Finally, Ahmed et al. [29] analyzed 163 projects and 6,979 failed merges, finding that smelly code is three times more likely to be involved in merge conflicts and specifically that method-level smells (e.g., Blob operation and internal duplication) are highly correlated with semantic conflicts. Compared to this work, our work delves deeper into the nature of the merge failures: what language constructs are typically part of them and what kinds of developer decisions are being made?

With a somewhat different focus, Leßenich et al. [2] and Cavalcante et al. [25] examined 50 and 60 projects, respectively, to compare semi-structured and unstructured merge techniques in terms of how many conflicts they report. Both studies found that semi-structured merge techniques can reduce the number of conflicts by approximately half, but not eliminate them. Cavalcante et al. [44] used the findings from their previous study to improve their semi-structured merge technique to address conflicts involving import declarations and initialization blocks. Although important, the studies by Leßenich et al. and

Cavalcante et al. miss information about conflicts that happen inside method bodies, as the semi-structured merge techniques they used treated method bodies as plain text. Accioly et al. [45] also studied semi-structured merge, analyzing 70,047 merges from 123 GitHub projects. They found that 87.57% of merge conflicts take place inside the same method and suggest that awareness tools should be used to avoid them. Interesting, when discussing future work, they suggest further analyses to answer questions similar to the ones we answer in this paper: What are the conflict patterns inside method bodies? What percentage of those conflicts involve method signatures or just statements inside the method bodies?

Yuzuki et al. [26] analyzed the characteristics of conflicts at the level of individual methods. Using the version history of ten Java projects, they found that 44% of conflicts were due to concurrent changes (edits in the same part of the method made by two or more developers), 48% to removing methods in their entirety, and 8% to renames. They report that 99% of the conflicts were resolved by choosing one of the versions – a number that does not align with our results, but might be an artifact of the small number of projects they studied.

McKee et al. [43] first interviewed 10 developers and then performed a follow-on survey with 162 other developers to build a detailed understanding of developer perceptions regarding merge conflicts. They found, among other things, that complexity of the conflicting lines of code and file as a whole, number of LOC involved in the conflict, and developers' familiarity with the lines of code in conflict all impact how difficult developers find a conflict to resolve. They also found that, when it comes to merge tools, developers feel they need better usability, better ways of filtering out less relevant information, better ways of exploring project history, and better graphical representation of information they need. These findings are in line with our findings, with the results from our quantitative analyses suggesting some concrete ways of actually addressing some of those needs.

All in all, no prior work is as detailed as ours, shedding light on the nature of merge conflicts in terms of what conflicts look like, what kinds of conflicts occur, how developers fix them, how conflicts involving different chunks relate, and more. Moreover, no prior work analyzed as extensive a corpus of projects as we did.

7 Conclusion

This paper contributes a two-part, in-depth study of merge conflicts, what they look like in detail, and how developers choose to resolve them. First, we analyzed, by hand, over a thousand merges from five open source projects, identified the merges that led to a conflict and, on a conflicting chunk by conflicting chunk basis, catalogued the language constructs involved and the resolution strategies that developers used to address the conflicts they encountered. We then examined the data from a number of perspectives that articulate what makes some merge conflicts more difficult than others, including the number of conflicting chunks, the size of the chunks, the language constructs that appear in the conflicting chunks, the patterns in language constructs that are present inside and across chunks, and the patterns relating language constructs and developer decisions. Second, inspired by the results from the manual analysis, we replicated the analysis on a large-scale corpus of 2,731 Java projects involving 25,328 failed merges and a total of 175,805 conflicting chunks. With a few exceptions that we highlighted in the paper, findings from the large-scale corpus align with those from the five initial projects.

From our analyses, 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 kinds of merge conflicts. We envision new merge tools that integrate several components: (1) an existing base technique, whether structured or unstructured, (2) extensions in the form of plug-ins that each can automatically handle a specific kind of conflict, (3) user interface tools that offer tailored heuristics with relevant choices that guide the developer when the merge tool cannot resolve a conflict on its own, and (4) the ability to continue merging automatically once relevant manual choices are made. Supporting this vision are the following key findings from our study: 87% of conflicting chunks (automated analysis; 83% manual analysis) had all the information needed to resolve them as resolution did not require any new code to be written; 60% of failed merges involved multiple conflicting chunks, with, depending on project, from 14% to 46% of all chunks (29% average) having dependencies on other chunks; and patterns exist in both how certain kinds of conflicts are addressed repeatedly and how developers make similar resolution choices over time.

Our future work is twofold. First, we intend to study the nature of merge conflicts in more detail yet, as well as more broadly. As one example, we plan to examine merges that result from concatenation and study whether any patterns exist in which lines of code from both versions are selected and how they are interwoven. As another example, we plan to replicate our study with projects written in different programming languages. Our second direction of future work turns to the design and exploration of new merge tools that are inspired by the observations made in this paper. Short term, we are particularly interested in designing semi-automated tool support that assists developers in the incremental resolution of conflicting chunks by suggesting possible resolutions based on the patterns we found. Long term, we plan to explore the design of learning-based approaches to merging.

Acknowledgment

The authors thank CAPES, CNPq, and FAPERJ for the financial support. Part of this work was supported by the NSF under grant number CCF-1414197.

References

[1] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Trans. Softw. Eng.*, vol. 28, no. 5, pp. 449–462, 2002.

[2] O. Leßenich, S. Apel, and C. Lengauer, “Balancing precision and performance in structured merge,” *Autom. Softw. Eng. ASE*, vol. 22, no. 3, pp. 367–397, May 2014.

[3] T. Berlage and A. Genau, “A framework for shared applications with a replicated architecture,” in *Symposium on User Interface Software and Technology (UIST)*, Atlanta, Georgia, USA, 1993, pp. 249–257.

[4] S. Chacon, *Pro Git*, 1st ed. Berkeley, CA, USA: Apress, 2009.

[5] J. W. Hunt and M. D. McIlroy, “An Algorithm for Differential File Comparison,” Bell Laboratories, Murrray Hill, NJ, Computing Science Technical Report CSTR 41, 1976.

[6] W. Miller and E. W. Myers, “A file comparison program,” *Softw. Pract. Exp.*, vol. 15, no. 11, pp. 1025–1040, 1985.

[7] E. W. Myers, “An O(ND) Difference Algorithm and its Variations,” *Algorithmica*, vol. 1, no. 2, pp. 251–266, 1986.

[8] G. Oster, P. Urso, P. Molli, and A. Imine, “Data Consistency for P2P Collaborative Editing,” in *Conference on Computer Supported Cooperative Work (CSCW)*, Banff, Alberta, Canada, 2006, pp. 259–268.

[9] T. Apiwattanapong, A. Orso, and M. J. Harrold, “JDiff: A Differencing Technique and Tool for Object-oriented Programs,” *Autom. Softw. Eng. ASE*, vol. 14, no. 1, pp. 3–36, 2007.

[10] D. Binkley, S. Horwitz, and T. Reps, “Program Integration for Languages with Procedure Calls,” *ACM Trans. Softw. Eng. Methodol. TOSEM*, vol. 4, no. 1, pp. 3–35, 1995.

[11] J. Buffenbarger, “Syntactic software merging,” in *International Conference on Software Engineering (ICSE)*, London, UK, 1995, pp. 153–172.

[12] J. J. Hunt and W. F. Tichy, “Extensible language-aware merging,” in *International Conference on Software Maintenance (ICSM)*, Montreal, Canada, 2002, pp. 511–520.

[13] H. Shen and C. Sun, “A complete textual merging algorithm for software configuration management systems,” in *Computer Software and Applications Conference (COMPSAC)*, Hong Kong, China, 2004, pp. 293–298 vol.1.

[14] H. Shen and C. Sun, “Syntax-based reconciliation for asynchronous collaborative writing,” in *International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, San Jose, CA, USA, 2005, pp. 10 pp.-.

[15] B. Westfechtel, “Structure-oriented Merging of Revisions of Software Documents,” in *Workshop on Software Configuration Management (WSCM)*, Trondheim, Norway, 1991, pp. 68–79.

[16] V. Berzins, “Software merge: semantics of combining changes to programs,” *ACM Trans. Program. Lang. Syst. TOPLAS*, vol. 16, no. 6, pp. 1875–1903, 1994.

[17] D. Jackson and D. A. Ladd, “Semantic Diff: a tool for summarizing the effects of modifications,” in *International Conference on Software Maintenance (ICSM)*, Victoria, BC, Canada, 1994, pp. 243–252.

[18] S. Apel, O. Lessenich, and C. Lengauer, “Structured merge with auto-tuning: balancing precision and performance,” in *Automated Software Engineering (ASE)*, Essen, Germany, 2012, pp. 120–129.

[19] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, “Semistructured Merge: Rethinking Merge in Revision Control Systems,” in *European Conference on Foundations of Software Engineering (ESEC/FSE)*, Szeged, Hungary, 2011, pp. 190–200.

[20] D. Berlin and G. Rooney, *Practical Subversion*, 2nd ed. Apress, 2006.

[21] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Boston, MA: Addison-Wesley, 2007.

[22] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, “Proactive detection of collaboration conflicts,” in *European Conference on Foundations of Software Engineering (ESEC/FSE)*, Szeged, Hungary, 2011, pp. 168–178.

[23] B. K. Kasi and A. Sarma, “Cassandra: Proactive Conflict Minimization Through Optimized Task Scheduling,” in *International Conference on Software Engineering (ICSE)*, Piscataway, NJ, USA, 2013, pp. 732–741.

[24] T. Zimmermann, “Mining Workspace Updates in CVS,” in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, Washington, DC, USA, 2007, p. 11.

[25] G. Cavalcanti, P. Accioly, and P. Borba, “Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment,” in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Beijing, China, 2015, pp. 1–10.

[26] R. Yuzuki, H. Hata, and K. Matsumoto, “How we resolve conflict: an empirical study of method-level conflict resolution,” in *International Workshop on Software Analytics (SWAN)*, Montreal, Canada, 2015, pp. 21–24.

[27] H. L. Nguyen and C.-L. Ignat, “Parallelism and conflicting changes in Git version control systems,” presented at the IWCES’17 - The Fifteenth International Workshop on Collaborative Editing Systems, 2017.

[28] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen, “Indicators for merge conflicts in the wild: survey and empirical study,” *Autom. Softw. Eng.*, vol. 25, no. 2, pp. 279–313, Jun. 2018.

[29] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma, “An Empirical Examination of the Relationship between Code Smells and Merge Conflicts,” in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017, pp. 58–67.

[30] R. Santos and L. G. P. Murta, “Evaluating the Branch Merging Effort in Version Control Systems,” in *2012 26th Brazilian Symposium on Software Engineering (SBES)*, Natal, RN - Brazil, 2012, pp. 151–160.

[31] B. O’Sullivan, “Making sense of revision-control systems,” *Commun. ACM*, vol. 52, no. 9, p. 56, Sep. 2009.

[32] P. Devanbu, “New Initiative: The Naturalness of Software,” in *International Conference on Software Engineering (ICSE)*, Florence, Italy, 2015.

[33] L. G. P. Murta, H. L. R. Oliveira, C. R. Dantas, L. G. B. Lopes, and C. M. L. Werner, “Odyssey-SCM: An integrated software configuration management infrastructure for UML models,” *Sci. Comput. Program.*, vol. 65, no. 3, pp. 249–274, 2007.

[34] J. Portillo-Rodriguez, A. Vizcaino, C. Ebert, and M. Piattini, “Tools to Support Global Software Development Processes: A Survey,” in *5th IEEE International Conference on Global Software Engineering (ICGSE)*, 2010, pp. 13–22.

[35] H. C. Estler, M. Nordio, C. A. Furia, and B. Meyer, “Unifying Configuration Management with Merge Conflict Detection and Awareness Systems,” in *2nd Australian Software Engineering Conference (ASWEC)*, Washington, DC, USA, 2013, pp. 201–210.

[36] J. young Bang *et al.*, “CoDesign: a highly extensible collaborative software modeling framework,” presented at the 2010 ACM/IEEE 32nd International Conference on Software Engineering, 2010, vol. 2, pp. 243–246.

[37] P. Dewan and R. Hegde, “Semi-synchronous conflict detection and resolution in asynchronous software development,” *Eur. Conf. Comput.-Support. Coop. Work ECSCW*, pp. 159–178, 2007.

[38] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson, “FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2007, pp. 1313–1322.

[39] A. Sarma, D. Redmiles, and A. van der Hoek, “Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes,” *IEEE Trans Softw Eng*, vol. 38, no. 4, pp. 889–908, Jul. 2012.

[40] J. Wloka, B. Ryder, F. Tip, and X. Ren, “Safe-commit Analysis to Facilitate Team Software Development,” in *31st International Conference on Software Engineering (ICSE)*, Washington, DC, USA, 2009, pp. 507–517.

[41] L. Hattori and M. Lanza, “Syde: a tool for collaborative software development,” in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 2010, vol. 2, pp. 235–238.

[42] M. L. Guimarães and A. R. Silva, “Improving early detection of software merge conflicts,” in *34th International Conference on Software Engineering (ICSE)*, Piscataway, NJ, USA, 2012, pp. 342–352.

[43] S. McKee, N. Nelson, A. Sarma, and D. Dig, “Software Practitioner Perspectives on Merge Conflicts and Resolutions,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 467–478.

[44] G. Cavalcanti, P. Borba, and P. Accioly, “Evaluating and Improving Semistructured Merge,” *Proc ACM Program Lang*, vol. 1, no. OOPSLA, pp. 59:1–59:27, Oct. 2017.

[45] P. Accioly, P. Borba, and G. Cavalcanti, “Understanding semistructured merge conflict characteristics in open-source Java projects,” *Empir. Softw. Eng.*, pp. 1–35, Dec. 2017.

Gleiph G. L. Menezes holds a Ph.D. (2016) and Master (2011) degrees in Computer Science from the Computing Institute of Universidade Federal Fluminense (UFF) and a B.S. (2009) degree in Computer Science from Universidade Federal de Viçosa (UFV). He is a Professor at the Computer Science Department of Universidade Federal de Juiz de Fora (UFJF). He received a distinguished Master Thesis award at *Simpósio Brasileiro de Qualidade de Software* (SBQS) in 2012. His research area is software engineering, and his current research interests include configuration management and software evolution.

Leonardo G. P. Murta is an Associate Professor at the Computing Institute of Universidade Federal Fluminense (UFF). He holds a Ph.D. (2006) and a M.S. (2002) degree in Systems Engineering and Computer Science from COPPE/UFRJ, and a B.S. (1999) degree in Informatics from IM/UFRJ. He has a productivity research grant level 2 from CNPq since 2009 and a Young Scientist research grant from FAPERJ since 2012. He has published over 150 papers on journals and conferences and received an ACM SIGSOFT Distinguished Paper Award at ASE 2006 and three best paper awards at SBES in 2009, 2014, and 2016. He has served as program committee member of ICSE 2014, program chair of SBES 2015, associate editor of JBCS since 2013 and editor in chief of JSERD since 2017. His research area is software engineering, and his current research interests include configuration management, software evolution, software architecture, and provenance.

Marcio de O. Barros holds a Ph.D. degree (2001) in Computer Science and System Engineering from COPPE/UFRJ at Rio de Janeiro, Brazil. He is an Associate Professor at Universidade Federal do Estado do Rio de Janeiro (UNIRIO). He has a productivity research grant level 2 from CNPq since 2007. His research interests involve the application of search-based optimization to software construction related activities, such as design, code improvement and version control systems, as well as earlier software development activities, including requirements prioritization and project management. He serves on several international program committees and acts as reviewer for relevant software engineering journals.

André van der Hoek serves as chair of the Department of Informatics at the University of California, Irvine and heads the Software Design and Collaboration Laboratory, which focuses on understanding and advancing the roles of design, collaboration, and education in software engineering. He is co-author of 'Software Design Decoded: 66 Ways How Experts Think' and co-editor of 'Studying Professional Software Design: a Human-Centric Look at Design Work', two books that detail the expert practices of professional software designers. He has authored and co-authored over 100 peer-reviewed journal and conference publications, and in 2006 was a recipient of an ACM SIGSOFT Distinguished Paper Award. He was recognized as an ACM Distinguished Scientist in 2013, and in 2009 he was a recipient of the Premier Award for Excellence in Engineering Education Courseware. He is the principal designer of the B.S. in Informatics at UC Irvine and was honored, in 2005, as UC Irvine Professor of the Year for his outstanding and innovative educational contributions. He holds a joint B.S. and M.S. degree in Business-Oriented Computer Science from Erasmus University Rotterdam, the Netherlands, and a Ph.D. degree in Computer Science from the University of Colorado at Boulder.