CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Study of Merge-Conflict Resolutions in Open-Source Software

Master’s thesis in Software Engineering

ISAK ERIKSSON
PATRIK WÅLLGREN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Master's thesis 2016

A Study of Merge-Conflict Resolutions in Open-Source Software

ISAK ERIKSSON
PATRIK WÅLLGREN

Department of Computer Science and Engineering
Software Engineering Division
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2016

A Study of Merge-Conflict Resolutions in Open-Source Software

ISAK ERIKSSON
PATRIK WÅLLGREN

Supervisor: Thorsten Berger, Department of Computer Science and Engineering
Co-Supervisor: Julia Rubin, Massachusetts Institute of Technology
Co-Supervisor: Sarah Nadi, Technische Universität Darmstadt
Examiner: Regina Hebig, Department of Computer Science and Engineering

Master's Thesis 2016
Department of Computer Science and Engineering
Software Engineering Division
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Gothenburg, Sweden 2016

A Study of Merge-Conflict Resolutions in Open-Source Software

ISAK ERIKSSON
PATRIK WÅLLGREN
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

In collaborative software development, conflicts often arise when merging different versions of the code. These are often solved manually, which slows down productivity. To develop a tool that assists in the resolution process, we aim to study how developers resolve conflicts and try to categorize the resolutions. We do this by studying large open-source projects on GitHub.

We found that for conflicts regarding code inside methods or constructors, the currently checked out version was chosen in 77% of the cases. We found that developers tend to choose their own version of the code when resolving merge-conflicts.

Keywords: version control system, git, merge-conflict, conflict resolution

Acknowledgements

We want to extend our thanks to our supervisor; Thorsten Berger at Chalmers University of Technology. We would also like to thank our co-supervisors; Julia Rubin at Massachusetts Institute of Technology and Sarah Nadi at Technische Universität Darmstadt. Last but not least we want to thank Regina Hebig at Chalmers University of Technology for being our examiner.

The Authors, Göteborg 30/5/16

Contents

List of Figures

List of Tables

1 Introduction

When developing software products using a version-control system, branching is often used, both to support variability [1], and to develop new features, that is "A distinguishing characteristic of a software item (e.g., performance, portability, or functionality"[2]. It is of vital importance that the merging of these branches works smoothly, even when a conflict occurs.

Resolving merge conflicts, such as those arising from changes to different variants of features, is difficult. Merging might require refactoring the class hierarchy, introducing design patterns, or adding parameters to the feature. If it would be possible to develop a tool that can provide automated conflict resolution in this case, it would be of great value, since resolving such conflicts is a recurring problem that is solved manually today. The problem is that the development of such a tool requires more understanding of how merge-conflict resolutions are currently performed by developers. For having representative results, one would need to study the resolutions in large codebases, such as from GitHub. To acquire accurate results on large codebases, some automated analysis is required.

The goal of this work is to conduct a feasibility study that aims at investigating whether analysis of merge-conflict resolutions in large codebases can be automated. We focus on open-source projects, such as those being hosted on GitHub.

Understanding how merge conflicts are resolved allows for future development of an automatic merge-conflict resolution tool, which is our long-term goal. Towards this end, this study aims at answering the following research questions:

- RQ1. How can we statically analyze merge-conflict resolutions in real-world, large version histories of open-source projects?
- RQ2. Can we make a meaningful categorization of how developers resolve merge-conflicts?

Our main working hypothesis is that it is in fact feasible to automatically analyze conflict resolutions and categorize them from all the metadata available about projects and from statically analyzing code. By studying examples of popular projects with many branches or forks, developing a mining infrastructure, and investigating to what extent static code-analysis tools can be utilized for categorizing the conflict resolutions, we will work towards testing this hypothesis.

The automated analysis in this study was conducted on a total of 1964 conflicts in 20 projects on GitHub. The Preliminaries section gives the reader the knowledge needed to grasp the content of this study. We also conducted a Pre study on the feasability of identifying variant- and feature-branching related conflicts. The Methods section describe how an initial manual analysis of merge-conflict resolutions were made, and also the steps in performing the automatic analysis. Last but not least, we list the results of our findings and discuss our conclusions.

2

Preliminaries

This section covers the knowledge needed in code-clone management, branching management and Git conflicts. It also brings up related work and discusses how this study differs from them.

2.1 Code-Clone Management

Cloning happens during all stages of a software-development process, and it is the responsibility of the developers themselves to make sure that changes between copies of the clones are propagated correctly [1]. Because of this, there are risks that conflicts arise during all stages of the development process. When cloning features, multiple versions of the same feature exists and their consistency needs to be managed [3].

With the use of a version control system, cloning can be managed in a more smooth way by using branching and merging capabilities [3]. *Git* is a fully distributed version control system. It stores the whole project along with a complete development history in a so called *repository*, and since it is fully distributed, one has instant access to the repository. Often when using Git, a remote hosting server is used. *GitHub* is a popular remote hosting server. It has a request limit of 5000 requests per hour, but once a project is cloned, no request limits on project hosting sites are a problem anymore. This makes analyzing a Git-hosted project attractive.

2.2 Branching Management

Many software projects follow a branching model when using Git, such as the one explained by Giessen [4]. In these models, users create feature branches that provide an environment where new features can be implemented and tested without affecting the end-user version of the software [5]. There are various ways to use branches. A branch can be created for each new feature, in this document called *feature-branching*. A branch may also be created for each new product that is to be developed, or each variant of an existing product. Such a branch is in this document called *variance-related branching* [1], for example, for supporting new hardware in an existing product.

When a new feature has been implemented in a new branch, the branch needs to be merged into another branch, such as the main branch. *Merging* is the process of joining two branches together, both in case of two local branches or a local and a remote branch [6]. Local branches are branches in the client's copy of the repository, whereas the remote branches are branches in the remote repository. In GitHub, merging is usually done using a pull request. A *pull request* is made to let the collaborators in the repository know that the commits in a branch are ready to be merged. The collaborators can review the new code and input their feedback until it is finally approved for merging into the end-user branch [7].

The two commits that are to be merged are called the *parents* of the *merge commit* that is created when the merge is made. In this document, the parents will often be referred to as the left- and the right version. The *left version* is the commit that was checked out at the time of the merge, and the *right version* is the commit that is being merged into the currently checked out branch. For instance, when pulling the remote branch to merge with the local branch, the left version will be the local version and the right version will be the remote version that is being pulled. The commit that two branches originates from is called the *common ancestor*. Figure 2.1 illustrates a merge and the terms common ancestor, left- and right version, and merge commit.

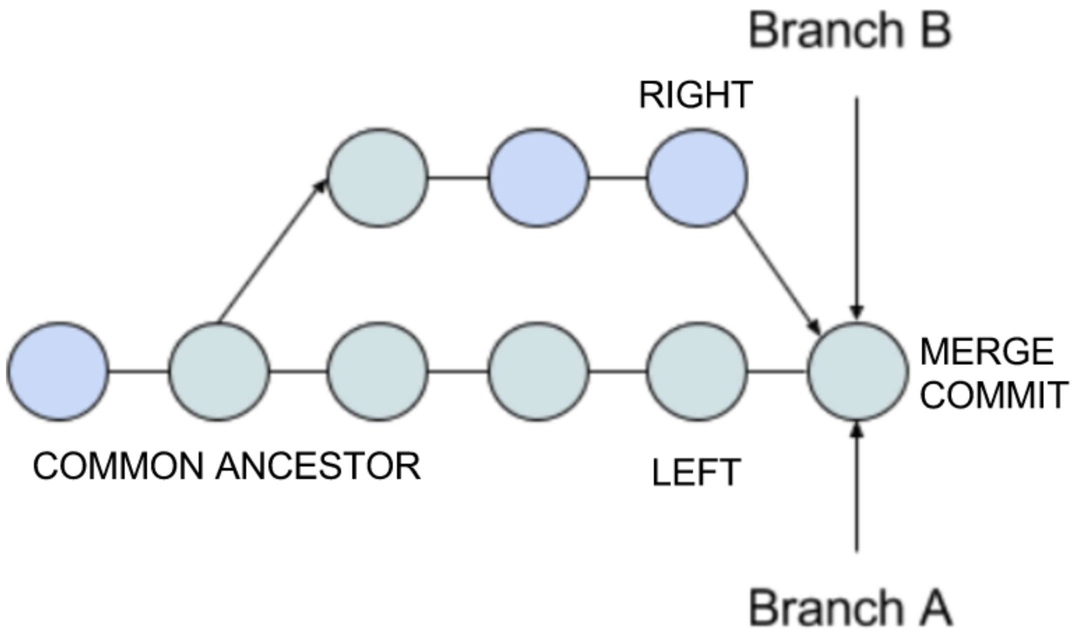
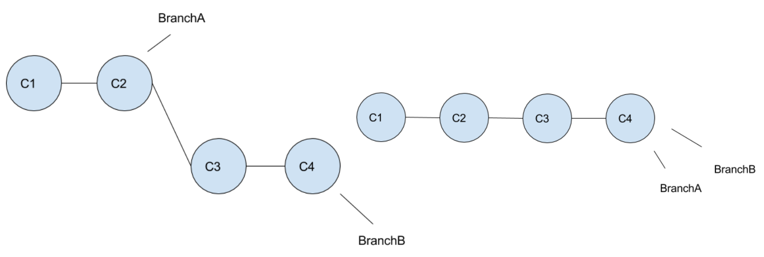


Figure 2.1: Merging of two branches

Textual Merging. The most commonly used merging technique is textual merging [8]. *Textual merging* is based on the history and on textual differences. It does not make use of any knowledge of the syntax or semantic. One must also distinguish between two-way merging and three-way merging. In *two-way merging*, only the two conflicting clones are analyzed to resolve the conflict. In *three-way merging*, also the common ancestor is used, which is more powerful [9].

Fast-Forward. When merging two branches, Git first attempts to perform a so-called fast-forward merge. *Fast-forward* is a way of simplifying merges in cases where at least one of the branches still points to the common ancestor. Since only one version has changed, there can not be any conflicts when merging. In such a case, all that has to be done, is to change both branches to point at the latest commit, as shown in Figure 2.2.



(a) Before merge **(b)** After merge

Figure 2.2: Fast-forward merge

Three-Way Merge. If fast forward fails, that is, when commits have been made to both branches that are to be merged, Git has to merge all files that the commits contain. This consists of merging the two versions of every file separately. To be able to know what has changed in the two branches, Git considers both the two versions and their common ancestor. If the files have not been changed at the same places in both branches, Git is able to do this automatically.

2.3 Resolving Git Conflicts

When branches are to be merged in Git, conflicts might arise. *Conflicts* are the problems that prevent Git from automatically merging two branches together. This happens when the two parent commits have made changes to the same place in a file. Resolving conflicts is usually done by manually merging the conflicting lines of the two versions. To do this, one also takes the common ancestor's version of the file into consideration. By looking at the common ancestor and the two versions, it becomes clear which lines each version has added, and the developer can then choose parts from the two versions. He can choose one version completely, choose parts from both versions and he can add or remove code.

2.4 Related Work

2.4.1 Semistructured Merge

There exist merge tools that use approaches other than textual merging, such as syntactic- and semantic merging, which have language specific knowledge [10] and do not only compare lines of text. A combination of both textual merging, syntactic and semantic merging is called semistructured merging [8]. Studies have shown that the use of semistructured merge decreases the number of conflicts significantly. Cavalcanti et al. prove, by performing semistructured merging on 3266 merge scenarios on 60 projects, that semistructured merging can reduce the number of conflicts by 55% [11]. Our work is different in that instead of proposing a new conflict resolution technique, we are interested in how developers resolve conflicts arising from different variants of features or projects.

2.4.2 Conflict Patterns

During merging, several types of conflict patterns might occur. Previous work by Accioly [12] identified numerous conflict patterns, using her developed tool Conflicts Analyzer.

In her study, Accioly lists conflict patterns that describe types of conflicts that might arise during a merge. The study uses a semi-structured merge tool, called SSMerge, which performs merges by first constructing Feature-Structured Trees (FSTs) for the two versions of the file that are to be merged. The two trees are then merged using superimposition. This is possible since the order of methods and class variables inside a class does not matter. Code segments where the ordering matters in Java, ie. method- and constructor bodies, cannot be merged this way and are therefore placed in the leaves of the tree [8]. It is the conflicts that concern these leaves that are of interest to our study. We will use Conflicts Analyzer and the conflict patterns to analyze and categorize merge-conflict resolutions.

The conflict patterns are derived from the conflicts that SSMerge can detect [12]. Table 2.1 lists the patterns from Accioly's study that is listed in the online appendix: From Table 2.1, it is the EditSameMC- and SameSignatureCM patterns that concern the leaves of the tree and they make up 84% of the total number of conflicts [12] and this is the main reason why our study focuses only on these patterns.

Table 2.1: Conflict patterns

Pattern	Description
EditSameMC	Different edits to the same area of the same method or constructor
SameSignatureCM	Methods or constructors added with the same signature and different bodies
EditSameFd	Different edits to the same field declaration
AddSameFd	Field declarations added with the same identifiers and different types of modifiers
ModifierList	Different edits to the modifier list of the same type declaration (class, interface, annotation or enum types)
ImplementsList	Different edits to the same implements declaration
ExtendsList	Different edits to the same extends declaration
DefaultValueA	Different edits to the same annotation method default value

2.4.3 Avoiding Software Merge-Conflicts

There exist numerous practices to reduce merge-conflicts, such as continuous integration[13] in an Agile development process. The continuous integration practice makes sure that developers merge their code in short intervals and therefore the conflicts does not become as many at a given time as it would if merges were done less frequently.

Furthermore, Guimaraes and Silva[14] argues that developers do not merge as frequently as desirable. They state that "*Unfortunately, merging is cumbersome and disrupts programming flow, so some developers do not merge as frequently as desirable — teams avoid parallel work because of difficult merges, and developers rush their tasks to avoid being the ones responsible for the merge.*" To aid developers when merging, they present a solution that continuously merges committed and uncommitted code in the background, and then presents detected conflicts to the developer inside the IDE, while the developer continues developing.

While Guimaraes and Silva strive to avoid conflicts, our long term goal is to ease the solving of merge-conflicts by automatically solving them using a tool. This study will contribute to the long term goal by analyzing and categorizing merge-conflict resolutions done by real developers.

2.4.4 Variance in Code-Clone Management

Dubinsky et. al. [1] have conducted an exploratory study of cloning in industrial software product lines. In their study, they state that it is difficult to propagate changes between clones. They have also interviewed developers who say that "*If we* *find a bug then many times it can be here and also in other places. The new product contains code that exists also in the old product. So, if we fix the old one then we also fix the new or vice versa*". Moreover, they say that sometimes they find the same bug in different variants which nobody thought about before.

We believe that our study can build on Dubinsky et. al's. study. Identifying variance-related branches will ease the process of propagating changes between clones in an industrial software product line since the developers would know which branches they should propagate.

3

Pre Study

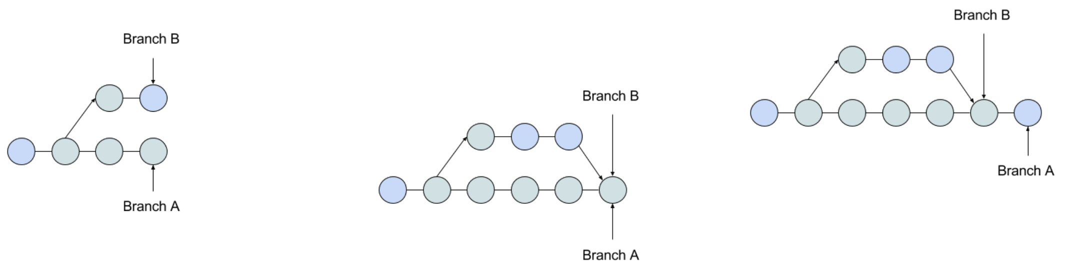
We started our work with an exploratory phase, where we investigated the possibility to limit the scope of this study. In this phase, we studied the possibilities of limiting the scope to feature-branching-related merge-conflicts or to variance-related merge-conflicts. The methodology used throughout the pre study was to try to identify branches and conflicts related to either variance or features.

3.1 Method - Identifying Feature Branches

To detect feature branches, we proposed some indicators which may indicate whether a branch is feature-related; the commit message and the branch name. A commit message could contain the branch name. However, since commit messages might be edited by the committee, it is unreliable to find feature branches this way.

In the Git history, branch names might indicate if it is a feature branch. However, due to how Git handles branches, this proved to be more difficult than anticipated. When two branches are to be merged, there are two different options: merge or rebase. When merging, Git takes the two commits that are to be merged and creates a *merge commit*. Unlike other commits, the merge commit has two parents, being the two commits that were merged. This makes it possible to distinguish merge commits from other forks and thus makes it possible to analyze them separately. Sometimes, rebase is used instead of merging. This means that instead of creating a merge commit, an ordinary commit is created. The changes that was introduced in the branch that is to be merged are applied to the other branch. Then, both branches are then changed to point at the new commit, which has only one parent. That parent is the commit that was checked out at the time of the rebase. The other commit is left as it was. Thus, there is no straight-forward way of find those rebased commits.

To analyze historical branches and merges is more difficult than one might expect. A question that one might ask is “In what branch was this commit made?”. A Git branch is only a reference that points to the latest commit and does not “contain”



(a) The branches Branch A and Branch B both points at their respective tip commit.

(b) When merged, both branches points at the merge commit.

(c) If Branch A is checked out and then committed to, Branch B will still point at the merge commit.

Figure 3.1: Merging of branches

commits. There is also a reference called HEAD, which points to the currently checked out branch. When a branch is merged, the branch will point at the same commit as the branch it was merged into, as shown in Figure 3.1. Therefore, a more correct way to ask this is “At what branch did HEAD point during the creation of this commit?”. That information is not stored in Git and therefore, that question is not possible to answer. There is no information that tells which branch was merged into which.

Moreover, using the command

```
git branch --contains <hash>
```

will show the “branches whose tip commits are descendants of the named commit”[1]. Therefore it is impossible to know, using this information alone, which branch was merged into which and also which branch a commit was created on. Another problem is that it is common practice amongst developers to delete a branch after it has been merged, and once that happens, all information about that branch is lost from the Git history.

3.2 Method - Identifying Variance Branches

To detect variance-related branches, we proposed the following indicators: Pull-requests, introduced Boolean parameters, name of introduced Boolean parameters, time in a branch’s lifetime that the parameters were introduced and the existence of clones. In variance-related merge-conflicts, we wanted to study how variants in code emerge in different branches.

We chose to analyze the GitHub project Elasticsearch. We chose this project since it has a vast number of commits (more than 20000) and more than 5000 forks. Elasticsearch is a distributed search engine used for analysing data in realtime.

3.2.1 Data Gathering Tool

When studying the code of Elasticsearch, we noticed that parameters were introduced and loaded from an external configuration file. These parameters were then used to set Boolean variables that usually indicate whether to use a certain block of code or not. In Elasticsearch, the function used to set these Boolean variables was called “getAsBoolean” and takes a string parameter name, and a Boolean default value.

```
boolean example = getAsBoolean("example_parameter", true)
    ;
```

The "example_parameter" could be set by the user in the external configuration file and if it has not been set, a default value, in this example true, will be used. The Boolean variable would in some cases be used to indicate which block of code to use, as in this example taken from a snippet of Elasticsearch code:

```
this.autoThrottle = indexSettings.getAsBoolean(
    AUTO_THROTTLE, true);

if (autoThrottle) {
concurrentMergeScheduler.enableAutoIOThrottle();
} else {
concurrentMergeScheduler.disableAutoIOThrottle();
}
```

To be able to identify the parameters and collect data about them, we developed a tool that gathers data automatically. All data that is stored in Git is hashed using *SHA-1*. The data to be gathered includes:

- The parameter name that was introduced
- The commit hash
- The if-statement that the Boolean is used in
- The code where the Boolean variable is set by the function that takes the

parameter name as one of its parameters.

- The commit message
- Whether or not the commit was a pull request

We gathered data by developing a Java program that uses Linux bash scripts which execute Git commands to get the above information. As Git saves the data as snapshots and not as changes, one needs to compare two commits in order to see which changes were introduced in a commit. To do this, we use the built in diff command in the following way:

```
git --no-pager diff <hash>^ <hash>
```

where ˆ is a git shortcut to get the parent commit of a commit hash. We now discuss how we extract each of the above pieces of information.

Parameter name. The parameter name was extracted from the line where the Boolean is set by the getAsBoolean function. It is useful to include it in the data so that it can be used when manually looking through the code to understand what the parameter was used for.

Commit Hash. For every commit that is checked out, we search for parameters and if there exist at least one, the commit hash is saved so that we know which commits to check out when we want to look manually at the code.

If-statement the Boolean is used in. We extracted the newly introduced Boolean variables that were later used in if-statements. This proved to be not useful since the Boolean variable names were not always the same as the parameter names used in the configuration file.

getAsBoolean line. While extracting the name of the parameter in the getAs-Boolean function, we also save the line itself to be able to quickly see the name of the Boolean variable as well as the default value the Boolean will be assigned to if the parameter is not set.

Commit message. The commit message is also extracted and printed in the excel document. In case the commit message contains important information which could indicate that the commit contains variant related code, it is vital to look at it to find which commits are good to analyze manually. To get the commit message for a given hash, this command was used:

```
git log --format=%B -n 1 <hash>
```

Pull request. When changes on a branch in a fork of a project is to be merged into the original project, pull-requests are used. It is interesting to know whether or not the commit was a pull request. Finding out if variant related code is more or less likely in pull requests would be interesting for the study. To know whether a given merge commit was a pull request, the commit message was parsed to see if it contains "Merge pull request #".

3.2.2 Find Details about Introduced Parameter

When functionality is added or changed, it is good practice to create a new branch. When new parameters are added that decides which variant of code to use, we believe this most often happens in a new branch, and it will be interesting for the study to know at which point in life of the branch this happens.

Using the merge commit and the parameter name, we calculate in which commit the parameter was introduced. We thought that it would be straight-forward to find where in the branch the parameter was introduced, by recursively stepping backwards from the merge commit through the commits of the branch, searching for the given parameter. We soon found out that the possibility of analyzing commits with regards to branches is very limited as stated in 3.1.

Another question one might ask is “How many branches does this project have?”. That can be answered using the git command:

```
git branch -a
```

However, as it is common practice to delete branches after they have been merged, the command is of little use as it only lists the currently existing branches. Information about old branches may be found in commit messages but, again, as they are often edited, they are not reliable.

In yet another attempt to find variant-related merges, we sought to find merge-commits where a new parameter was introduced to solve conflicts. In Elasticsearch, the parameters we looked for were fetched using “getAsBoolean”. It turned out that there was not a single example of such a case where “getAsBoolean” was introduced in a merge-commit in the history of Elasticsearch.

3.3 Result - Outcome of Pre Study

The pre study has shown that it is difficult to detect whether merge-commits are related to variance or feature-branches. Thus, we conclude that we can not identify variant- and feature-branching related conflicts. Instead, we decided that the direction of this thesis will be to identify and classify merge-conflict resolutions in general.

The tool created during the pre study will be used when analyzing merge-conflict resolutions. It will be used to parse information about commits in GitHub repositories and also be extended with new functionality to automatically be able to analyze the resolutions.

4

Method - Analyzing Merge-Conflict Resolutions

In this section we describe how we analyzed merge-conflict resolutions. A manual analysis was conducted and the result of this, which is presented in the Result chapter, was used in an automatic analysis. In this chapter we aim to answer RQ1 and RQ2.

The tools used throughout the method, Conflicts Analyzer [15] and Resolutions Analyzer [16], as well as the results from the analyses [16] are available online.

4.1 Conflict File Tree

To get a better overview of how the different versions of a file look, we developed a tool to create a file tree of all conflicts of a given project. We call this file tree the Conflict File Tree. The leaves of the tree consist of the left-, right-, common ancestor- and merge commit version of a conflicting file when re-creating merge commits, as shown in Figure 4.1.

To do this, we first needed to find the two parents of a merge commit, that is, the two commits that were merged. The following command prints the two commits:

```
git --no-pager log --merges --format=%p <hash> | head -n1
```

where <hash> is the merge commit hash. We then re-create the merge using the following:

```
git reset --hard <hash of RIGHT>
git clean -f
git branch <temp branch name>
git checkout <hash of LEFT>
git merge <temp branch name>
```

In line 1, we set HEAD to RIGHT, which changes the working copy to the state of that commit. In line 2, the working copy is cleaned to be ready for the merge. In line 3, a new branch is created which points at RIGHT. In line 4, we checkout LEFT. In line 5, we merge the two commits by merging the newly created branch into commit LEFT. Git will now print out the conflicting files, which we parse. Afterwards, we abort the merge and delete the branch.

The common ancestor-, left- and right file, along with the resulting resolution file in the merge commit, are copied and saved in the Conflict file tree. The Conflict File Tree consist of folders and the versions of the conflicting files, structured according to Figure 4.1.

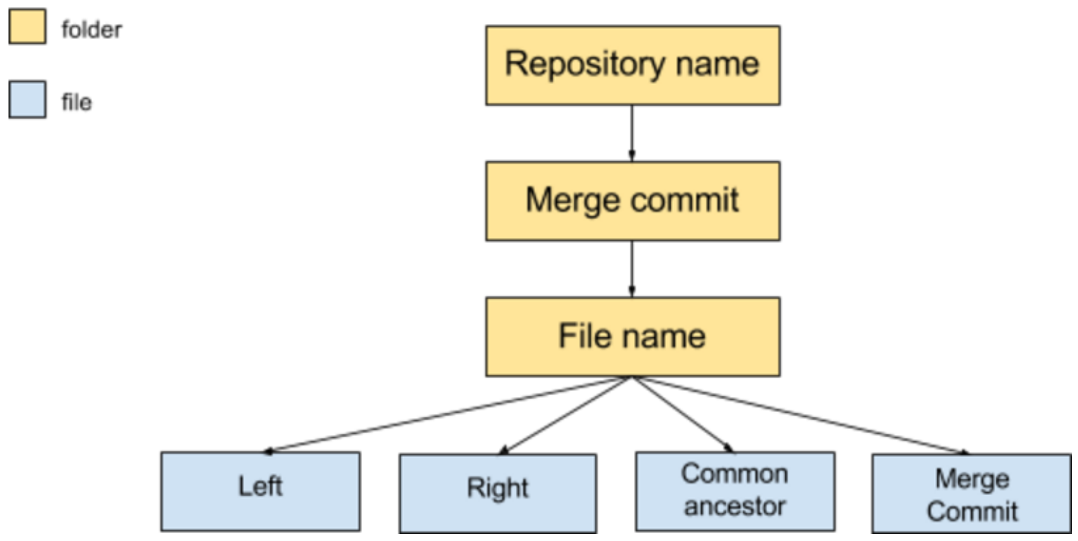


Figure 4.1: Structure of a Conflict File Tree

Having all the conflicting file versions in a structured manner made it easier to manually analyze how Git conflicts look like in files. To get the information about the conflict, we use following command:

```
git merge-file -p --diff3 <left> <ancestor> <right>
```

Initially, we used the CFT to study the merge-conflicts and their resolutions. However, we found the tool Conflicts Analyzer which classifies merge-conflicts, and therefore we decided to use the already existing tool to analyze the merge-conflicts. We incorporated our re-creation of merge-commits method in the Conflicts Analyzer tool so that it analyzes only Git conflicts (see Section 4.2).

4.2 Classify Merge-Conflicts

To classify conflicts, we used the tool Conflicts Analyzer developed by Accioly. The tool produces a conflict report with information of each conflict of a specified project. We extended the tool to add additional information as follows; the merge commit hash, the left commit hash, and the right commit hash. Since the tool uses a merge technique different from that used in Git, it finds conflicts that are not a conflict in Git. As we are interested in how the developers themselves solve conflicts, we are not interested in these additional conflicts. Therefore we also modified the tool to only analyze conflicts from merges that also yield a conflict when merged by Git. We did this by reusing our code that re-creates merges (see Section 4.1).

The output in the conflict report contains the following information:

- Conflict type
- Merge commit hash
- Left commit hash
- Right commit hash
- Conflict body
- File path

4.3 Manual Analysis

We began by reading through the output of Conflicts Analyzer from a randomly selected project called Blueprints. We found that for the conflict pattern SameSignatureCM (see Table 2.1), the resolution was often equal to one of the versions, i.e. the developer chose one of the versions completely, and that version was often a superset of the other. That version was also often the more recent version. For some observations, there were also cases where the chosen version was an intersection of the two versions. We decided to test if these types of resolutions are common by doing a qualitative analysis. As a resolution can for example be both the most recent version and a superset, we call them properties that a resolution can have. Based on the observations when reading through the output of Conflicts Analyzer, we proposed properties to check how common they are in a qualitative analysis (see Table 4.1).

Table 4.1: Proposed properties

Property	**Description**
Recent	The resolution is equal to the most recent version
Superset	The resolution is a superset of the code in both versions
Intersection	The resolution is an intersection of the code in both versions

We examined twenty-six randomly sampled examples of SameSignatureCM conflicts from 9 different projects. These projects were Atmosphere, Activiti, Blueprints, BroadLeadCommerce, Buildcraft, EventBus, android-async-http, RxJava and Elasticsearch. For each conflict, we looked at the two versions of the method or constructor, and tried to understand why they chose the one they did. What do the versions they chose have in common?

To be able to analyze conflicts from the output of Conflicts Analyzer, we wrote a Bash script to automatically extract the Java-files that conflicted in the merge, as well as the common ancestor file and the resolution file after the merge was made.

To extract the files, the script first resets the git repository so that HEAD points to the same commit as the latest commit on the remote master branch. Since the output from Conflicts Analyzer strip down the commit hash, our script parses the full commit hash of the parents from the git log. From the git log, the hash of the merge commit is also parsed. Using the git command:

```
git --no-pager log --merges --format=%p <hash> | head -n1
```

where *hash* is the merge commit hash. We then parse the two commits and perform the merge using the following sequence of commands:

```
git merge-base <LEFT> <RIGHT>
```

where *LEFT* and *RIGHT* refer to the hashes of the parents. Then, the two parent commits, the merge commit, and the common ancestor commit are checked out respectively and the desired file is copied to a specified output folder. Finally, the script parses the date of the parent commits and prints it to a file in the specified output folder.

We analyzed the extracted files manually to see whether the properties described in Table 4.1 would emerge repeatedly in many resolutions. The data gathered in the manual analysis consisted of Project name, Function name, Merge hash, Merge commit message, Left commit hash, Right commit hash, Left commit date, Right commit date, Conflict pattern and Resolution properties. We also found out that we can apply these same proposed properties for the conflict pattern EditSameMC (see Table 2.1). For EditSameMC, the method existed in the common ancestor but was modified in both versions. Since both EditSameMC and SameSignatureCM are patterns that concern conflicts inside methods and constructors, the only difference being that the method or constructor did not exist in the common ancestor in the SameSignatureCM pattern. Therefore we chose to not treat them differently when analyzing their resulutions.

4.4 Automatic Analysis

To see whether the results from the manual analysis still apply in a large-scale analysis, we developed a tool, which we named Resolution Analyzer, that reads the output of Conflicts Analyzer. The tool then filters out those conflicts we are interested in. Those conflicts are then analyzed and the result is printed in a spreadsheet.

As aforementioned in Section 4.3, we saw that the developer in many cases choose one of the versions completely. We also saw that sometimes the version that is chosen has more if-statements than the other version. We decided that it would be interesting to see whether choosing such a version recurs in many cases. It would also be interesting to see other cases where the version chosen had more of error handling and log printouts. From these observations, we extended and formalized our list of properties, by adding the properties if-statements, print-instances, log-instances and try-instances. The properties are defined in Table 4.2, where X and Y refer to the two versions. For each property the conflict resolutions are categorized as explained in Table 4.3.

Table 4.2: Definition of the properties

Property	Description
Recent	X is more recent than Y
Superset	X is a superset of Y and Y is not a superset of X
Intersection	X is an intersection of Y and Y is not an intersection of X
if-statements	X has more if-statements than Y
print-instances	X has more instances of the keyword ‘print’ than Y
log-instances	X has more instances of the keyword ‘log’ than Y
try-instances	X has more instances of the keyword ‘try’ than Y

Table 4.3: Definition of the categories

Category	Description
X chosen	The resolution is equal to X
Y chosen	The resolution is equal to Y
None chosen	Property satisfied but the resolution is not equal to any of the versions
Not applicable	Property not satisfied

4.4.1 Repositories to analyze

We want to analyze fairly big projects that contain many commits and many forks along with many branches. To satisfy these requirements, we chose the 20 top starred Java repositories on GitHub.

The projects listed in Table 4.4 were cloned so that they could be analyzed for conflict patterns by the Conflicts Analyzer tool.

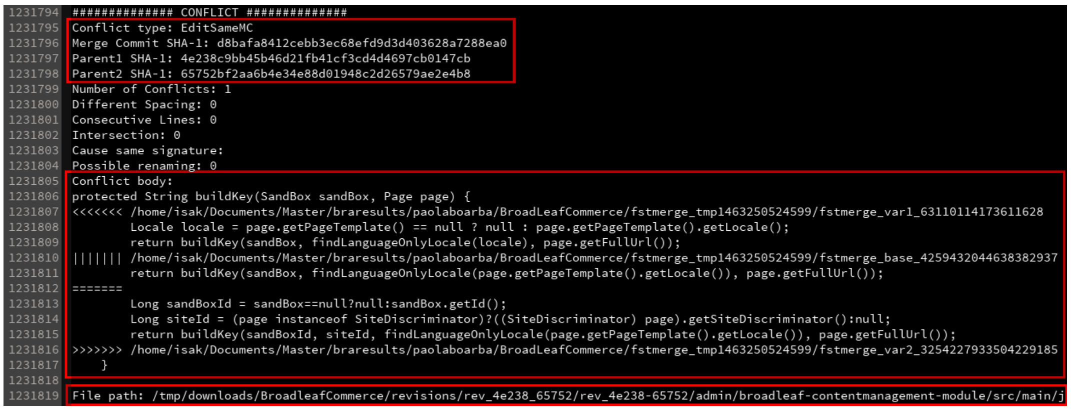
Table 4.4: GitHub repositories (As of 23/3-16))

Name	Commits	Branches	Forks
Elasticsearch	20712	46	5229
Android-async-http	856	3	4024
Android-best-practices	201	1	1696
Android-universal-image-loader	1025	3	5640
Curator	1050	9	304
Eventbus	404	5	2493
Fresco	494	3	2453
Guava	3372	4	1862
Iosched	129	2	4071
Java-design-patterns	1196	6	3495
Leakcanary	238	15	1291
Libgdx	12247	4	4479
Okhttp	2449	37	2518
React-native	5707	23	5609
Retrofit	1285	21	2081
Rxjava	4630	24	1919
Slidingmenu	336	8	5306
Spring-framework	11825	10	6860
Storm	1764	44	1760
Zxing	3203	3	4730

4.4.2 Input

We use the data from Conflicts Analyzer as input for our tool. As stated in Section 2.4.2, we are only interested in conflicts that are of the types SameSignatureCM or EditSameMC. Therefore, our tool filters out conflicts that are not of these patterns. It also removes conflicts in which any version of the function is empty, ie. the function was removed in one version. Conflicts that contain obscure output data from Conflicts Analyzer, such as if the conflict is not a Git conflict, are also skipped.

Figure 4.2 shows a screenshot of an example output from Conflicts Analyzer. The data surrounded by a red border are the data we use as input for our tool.



```
############### CONFLICT ###############
Conflict type: EditSameMC
Merge Commit SHA-1: d8bafa8412cebb3ec68efd9d3d403628a7288ea0
Parent1 SHA-1: 4e238c9bb45b46d21fb41cf3cd4d4697cb0147cb
Parent2 SHA-1: 65752bf2aa6b4e34e88d01948c2d26579ae2e4b8
Number of Conflicts: 1
Different Spacing: 0
Consecutive Lines: 0
Intersection: 0
Cause same signature:
Possible renaming: 0
Conflict body:
protected String buildKey(SandBox sandBox, Page page) {
<<<<<<< /home/isak/Documents/Master/braresults/paolaboarba/BroadLeafCommerce/fstmerge_tmp1463250524599/fstmerge_var1_6311011417361628
        Locale locale = page.getPageTemplate() == null ? null : page.getPageTemplate().getLocale();
        return buildKey(sandBox, findLanguageOnlyLocale(locale), page.getFullUrl());
||||||| /home/isak/Documents/Master/braresults/paolaboarba/BroadLeafCommerce/fstmerge_tmp1463250524599/fstmerge_base_4259432044638382937
        return buildKey(sandBox, findLanguageOnlyLocale(page.getPageTemplate().getLocale()), page.getFullUrl());
=======
        Long sandBoxId = sandBox==null?null:sandBox.getId();
        Long siteId = (page instanceof SiteDiscriminator)?((SiteDiscriminator) page).getSiteDiscriminator():null;
        return buildKey(sandBoxId, siteId, findLanguageOnlyLocale(page.getPageTemplate().getLocale()), page.getFullUrl());
>>>>>>> /home/isak/Documents/Master/braresults/paolaboarba/BroadLeafCommerce/fstmerge_tmp1463250524599/fstmerge_var2_3254227933504229185
    }

File path: /tmp/downloads/BroadleafCommerce/revisions/rev_4e238_65752/rev_4e238-65752/admin/broadleaf-contentmanagement-module/src/main/j
```

Figure 4.2: Output data from Conflicts Analyzer

From the "Conflict body" in the output, the name and signature of the conflicting function is parsed, as well as the parameter types that the function takes. The function body of the two versions in "Conflict body" is also parsed. This information is stored, and using the function name and the parameter types the function takes, the tool is able to parse the resolution function in the merge-commit by checking out the commit and reading the specified Java file from "File path".

This information is then used to find the result from the resolution function in the merge-commit. The different versions of the function are extracted and saved. To filter out the conflicts that arose only due to different spacing or new lines on different places, each line in the extracted functions are trimmed and empty lines are removed. The conflicts in which the conflicting versions of the function thereafter are equal, are removed.

4.4.3 Categorizing Conflict Resolutions

The automatic tool now categorizes the merge-conflict resolutions according to the properties listed in Table 4.3. To do that, our tool first compares the left and right version of the function, to the function in the result. First they are checked for equality, ie. is the result equal to the left, right, both or none of the versions. This is used to see if they chose one version completely. By doing that, we can find out which category that version belongs to for each property.

Recent. Then, the tool extracts the commit date of the parents to see if the chosen version was the most recent one. The date of the commits are extracted using the command:

```
1 git log -1 <hash> --format=%ci
```

Superset. To see if the code in one version is a superset of the code in the other versions, first consider the following example of a superset: Left version:

```
private int getValue(int index) {
  return (index >= values.size()) ? -1 : values[index];
}
```

Right version:

```
private int getValue(int index) {
  return values[index];
}
```

The left version contains all code from the right version plus a check for the index size. To detect that this is a superset it is not enough to compare them line by line. We solved this by instead considering the set of words in the code. All code in the left- and the right version are therefore split into words and added to a hashset. A version is a superset if and only if the that version's set of words is equal to the set of all words in the left and right code.

Intersection. Similarly to how we detect a superset, we also consider the sets of words to detect whether a version is an intersection of the left and the right versions. A version is an intersection if and only if that version's set of words is equal to the intersection of the sets of words in the left and right code.

if-statements, print-instances, log-instances and try-instances. Lastly, for each version of the method or constructor, we calculate the number of occurrences of each keyword and the number of if-statements (see Table 4.2).

5

Results

For RQ1, our study has shown a way of analyzing conflict resolutions in large-scale codebases, and this can be achieved by parsing output from Conflicts Analyzer. Likewise, for RQ2, we have created a categorization for merge-conflicts resolved by developers which can be used in future studies to, for instance, create an automatic merge tool. This chapter shows the results of the manual- and automatic analysis.

5.1 Manual Analysis

For conflicts in the conflict pattern SameSignatureCM, we found that if one of the two versions had the same code as the other version but with some additional code, that version was often chosen as resolution. This can be seen in Table 5.1 where the resolution was a superset in 69% of the cases. Table 5.1 lists how many resolutions, in the 26 cases that were analyzed, were *Superset*, *Intersection* and/or *Recent*. As a resolution can be both recent and a superset or an intersection at the same time, the percentage add up to more than 100%.

Table 5.1: Results of the manual analysis for the properties in Table 4.1

Property	Number of cases (total 26 cases)
Superset	18 (69%)
Recent	15 (58%)
Intersection	4 (15%)

For some of the cases that were chosen as resolution, one or more if-statements had been introduced in the version that was chosen which the other version did not have. We also saw an example of a case where the chosen version had more error handling than the other version.

An example can be seen from the repository Android-async-http, where two versions that resulted in a conflict when being merged looked like this:
Left version:

```
protected void sendSuccessMessage(int statusCode, Header[] headers, String responseBody) {
  try {
    Object jsonResponse = parseResponse(responseBody);
    sendMessage(obtainMessage(SUCCESS_JSON_MESSAGE, new Object[]{statusCode, headers, jsonResponse}));
  } catch(JSONException e) {
    sendFailureMessage(e, responseBody);
  }
}
```

Right version:

```
protected void sendSuccessMessage(int statusCode, Header[] headers, String responseBody) {
  if (statusCode != HttpStatus.SC_NO_CONTENT){
    try {
      Object jsonResponse = parseResponse(responseBody);
      sendMessage(obtainMessage(SUCCESS_JSON_MESSAGE, new Object[]{statusCode, headers, jsonResponse}));
    } catch(JSONException e) {
      sendFailureMessage(e, responseBody);
    }
  } else {
    sendMessage(obtainMessage(SUCCESS_JSON_MESSAGE, new Object[]{statusCode, new JSONObject()}));
  }
}
```

Here, all the words in the left version are also in the right version, hence the right is a superset of the left version. In this example, the right version was chosen as the resolution. As can also be seen, in the right version there is a check by an if-statement that is not present in the left version.

5.2 Automatic Analysis

The automatic analysis that was conducted on the 20 top starred Java projects on GitHub yielded, after our filtering, 1964 conflicts. For each property (see Table 4.2), the resolutions for the 1964 conflicts was categorized according to Table 4.3 and the results are presented in this section.

5.2.1 Developers often choose one version completely

In 1509 cases (77% of the studied cases), the left version was chosen as resolution, that is, the version that was checked out at the time of merging. Figure 5.1 shows how many cases where the left- and right version was chosen completely as the resolution and in how many cases none of them were chosen completely.

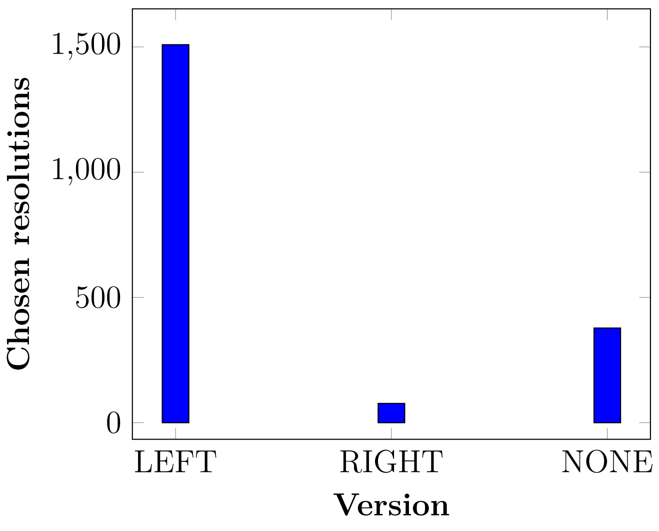


Figure 5.1: Number of chosen resolutions

5.2.2 Properties

Figure 5.2 shows the resolution categorization, as defined in Table 4.3 for each property, as defined in Table 4.2. The figure shows that the chosen version often is the more recent. It also shows that if one of the versions is a superset of the other, the version that is the superset is often chosen. On the contrary, if one version is an intersection of the two versions, the other version is more often chosen. For the if-statements, print-instances, log-instances and try-instances there were too many resolutions in the Not applicable category to draw any conclusions.

Figure 5.3 shows the resolution categorization where X is the left version and Y is the right version, and Figure 5.4 shows the resolution categorizations where X is the right version and Y is the left version. From these figures, one can see that the left version more often is a superset of the right version than the other way around. On the contrary, the right version

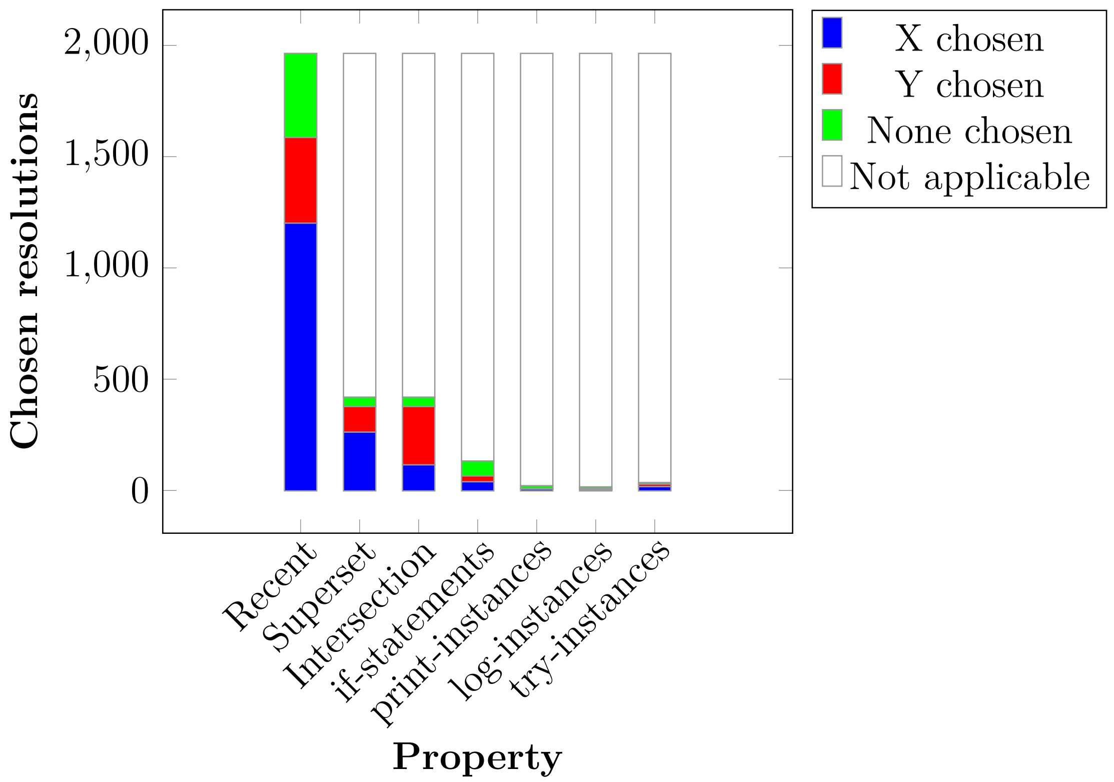


Figure 5.2: Resolution categorization for each property.

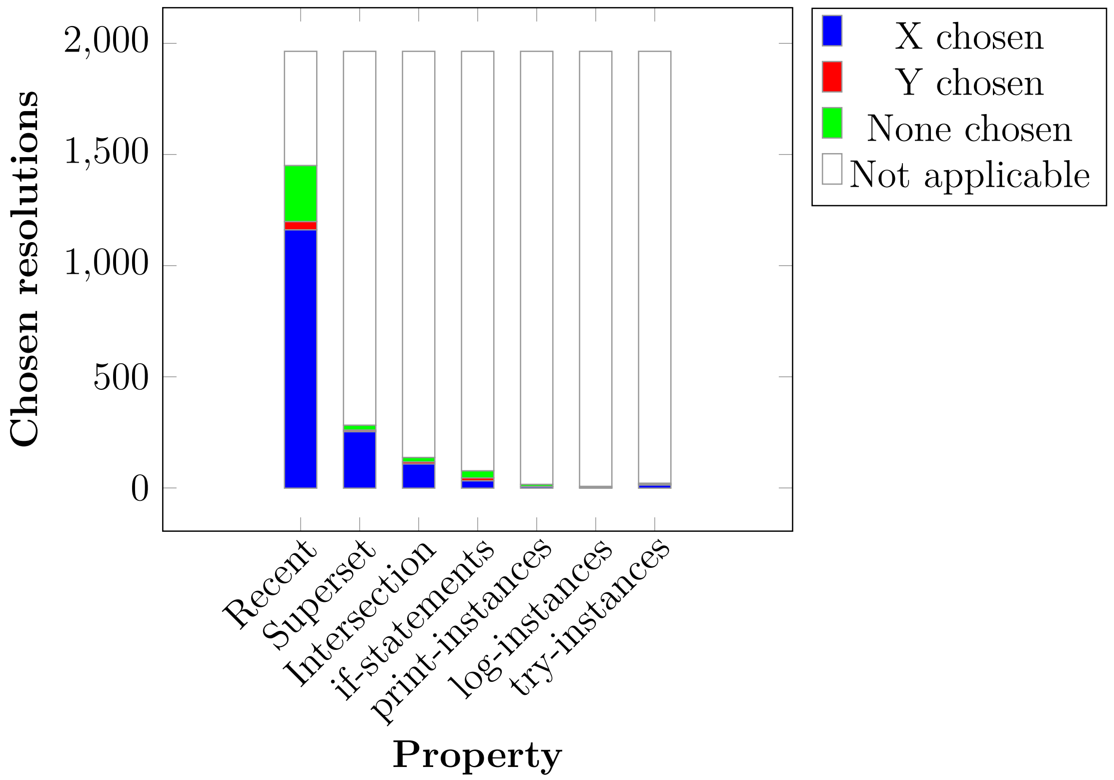


Figure 5.3: Resolution categorization for each property, where X is the left version and Y is the right version.

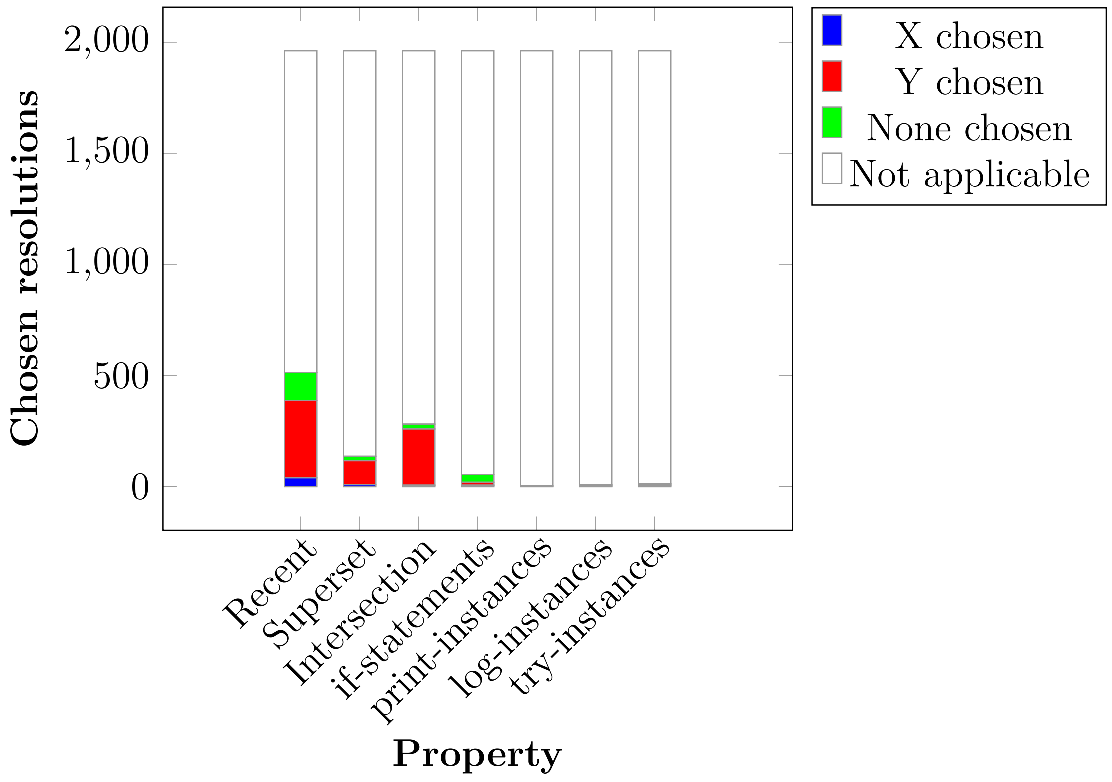


Figure 5.4: Resolution categorization for each property, where X is the right version and Y is the left version.

5.3 Discussion

The most significant result in this study is that in cases of conflicting code in methods or constructors, in 3 out of 4 cases that we studied, developers chose the left version when merging. The reasons for this is unclear, but there are two main scenarios; when pulling a remote branch and when merging a local branch into another one.

The left version is the commit that the developer who performs the merge has checked out when merging. If the conflicting merge is a result of pulling a remote branch, the left version is likely his own code, and the right version is someone else's code. Figure 5.2 shows that the chosen version also often is the most recent version. This in combination with that the left version most often is chosen indicates that developers tend to choose their own code as resolution. This study has not gone into why developers resolve conflicts this way, but it might be interesting for future studies to investigate this further. If the conflicting merge is a result of merging a local branch into another one, the left version is the branch which the other branch was merged into.

5.3.1 Threats to Validity

Construct Validity

Although the tools used throughout the study, Conflicts Analyzer and Resolutions Analyzer, have been developed with great care, there is still the risk of the tools containing bugs which might lead to an inaccurate result.

External Validity

Our results are based on a sample of 20 projects of different sizes, and a total of 1964 conflicts were analyzed. The results might differ if other projects were to be analyzed. The 20 top starred projects were all Java projects. Thus we can not say if our results hold for other languages. Also, these were all open-source projects and may not hold for closed-source projects. The fact that the projects are the 20 most top starred projects on Github might also affect the results.

Conclusion Validity

To show that the results are not random, we have done a statistical analysis on the results using Chi squared tests. We tested the results to show that they are significant. The Chi squared tests was conducted on the results, see Figure 5.2, for each property, see Table 4.2. The expected values are here calculated by dividing the total amount of cases in the categories X chosen and Y chosen by 2. Thus, cases in the categories None chosen and Not applicable are not included in the Chi squared tests and therefore, the total number of cases in the X chosen and Y chosen categories will differ depending on the property. We also conducted a Chi square test on the result for Left and Right versions chosen, see Figure 5.1. The expected values are here calculated by dividing the total amount of cases where one version was chosen completely by 2. The tested hypotheses are defined in Table 5.2. We reject H_0 if the p value is lower than 0.05. With a degree of freedom of 1, the required Chi square is 3.84. The Chi square results are shown in Table 5.3 and Table 5.4. Note that for the properties print-instances and log-instances, the expected values are lower than 5, which is a limitation for the Chi square test and we do not draw any conclusions about those properties.

Table 5.2: Hypotheses

	H_0	**H_a**
Properties	When a property is satisfied and either X or Y was chosen as resolution, the property does not affect what version developers choose when resolving conflicts.	When a property is satisfied and either X or Y was chosen as resolution, the property affects what version developers choose when resolving conflicts.
Chosen versions	Whether a version is Left or Right does not affect which of the versions developers choose when resolving conflicts.	Whether a version is Left or Right affects which of the versions developers choose when resolving conflicts.

Table 5.3: Chi square tests for the properties

Property	**X chosen**		**Y chosen**		**Total**	**Chi square**	**Reject H_0**
	Observed	**Expected**	**Observed**	**Expected**			
Recent	1201	793	385	793	1586	419.8	Yes
Superset	262	188.5	115	188.5	377	57.32	Yes
Intersection	115	188.5	262	188.5	377	57.32	Yes
if-statements	39	32.5	26	32.5	65	2.600	No
print-instances	6	4	2	4	8	2.000	No
log-instances	4	4.5	5	4.5	9	0.111	No
try-instances	17	14	11	14	28	1.286	No

Table 5.4: Chi square test for the chosen versions

Left		**Right**		**Total**	**Chi square**	**Reject H_0**
Observed	**Expected**	**Observed**	**Expected**			
1509	793	77	793	1586	1293	Yes

6

Conclusion

The goal of our thesis was to answer the following three research questions:

- RQ1. How can we statically analyze merge-conflict resolutions in real-world, large version histories of open-source projects?

- RQ2. Can we make a meaningful categorization of how developers resolve merge-conflicts?

To answer RQ1 and RQ2, we conducted a quantitative analysis based on the results of our manual analysis, described in Section 4.3. We showed how to statically analyze conflict resolutions in real-world, large version histories of open-source projects and how they could be categorized by analyzing output from Conflicts Analyzer.

Our study contributes to the long term goal of creating an automatic merge tool by increasing the understanding of how developers resolve merge-conflicts. As shown in Figure 5.1, developers tend to choose their own code when resolving conflicts.

For conflicts regarding code inside methods or constructors, we have shown that the currently checked out version is chosen in more than 3 out of 4 cases and that the chosen version often is the most recent one. Therefore, we conclude that developers tend to choose their own version of the code when resolving merge-conflicts. If developers choose their own versions instead of choosing the best code, our long term goal of an automatic merge-conflict resolution tool is even more important. For future studies, it would be interesting to see if our results hold for the other conflict patterns other than EditSameMC and SameSignatureCM, as described in Table 2.1.

Bibliography

[1] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, K. Krzysztof, “An Exploratory Study of Cloning in Industrial Software Product Lines”, 2013

[2] IEEE 829 Standard for Test Documentation

[3] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stanciulescu, A. Wasowski, I. Schaefer, “Flexible Product Line Engineering with a Virtual Platform”, 2014

[4] V. Driessen. “A successful Git branching model” Internet: http://nvie.com/posts/a-successful-git-branching-model/, Jan. 5, 2010 [Jan. 15, 2016].

[5] GitHub Inc. “Understanding the GitHub Flow”, Internet: https://guides.github.com/introduction/flow/, Dec. 12, 2013 [Jan. 15, 2016]

[6] Atlassian, “Using Branches”, Internet: https://www.atlassian.com/git/tutorials/using-branches/git-checkout, [Jan. 15, 2016]

[7] GitHub Inc. “Using pull-requests” Internet: https://help.github.com/articles/using-pull-requests/, [Jan. 15, 2016]

[8] S. Apel, J. Liebig, C. Lengauer, C. Kästner, W. Cook, “Semistructured Merge in Revision Control Systems”, 2010

[9] T. Mens. “A State-of-the-Art Survey on Software Merging” IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,VOL. 28,NO. 5, May 2002, pp, 449-462. http://uff-labgc-2010-2-grupo5.googlecode.com/svn/trunk/seminarios/artigos/mens2002.pdf

[10] S. Apel, O. Leßenich, C. Lengauer, “Structured Merge with Auto-Tuning: Balancing Precision and Performance”, 2012

[11] G. Cavalcanti, P. Accioly, P. Borba, “Assessing Semistructured Merge in Version

Control Systems: A Replicated Experiment", 2015

[12] P. Accioly, "Understanding Conflicts Arising from Collaborative Development", 2015

[13] M. Fowler. Continuous Integration. [Online]. Available: http://martinfowler.com/articles/continuousIntegration.html

[14] M. Guimaraes, A. Silva, "Improving Early Detection of Software Merge Conflicts"

[15] "Conflicts Analyzer" https://github.com/patwal/conflictsAnalyzer, accessed: 2016-06-17

[16] "Resolutions Analyzer" https://github.com/Isak-Eriksson/ResolutionsAnalyzer, accessed: 2016-06-17

A

Appendix 1

In this appendix, we describe how to use the Resolutions Analyzer. A detailed description of how to run Accioly's Conflicts Analyzer is available at the forked GitHub repository [15].

A.1 Resolutions Analyzer

Resolutions Analyzer analyzes the output of Conflicts Analyzer. Thus, it is required that Conflicts Analyzer is run on projects of the user's choice first (see the forked GitHub repository for instructions [15]).

A.1.1 Running the .jar

Resolutions Analyzer takes as argument the path to the conflict reports root folder of the projects analyzed, produced by Conflicts Analyzer. The conflicts report root folder is called *ResultData*. Resolutions Analyzer also takes as argument the path to the repository download folder (the same as in the configuration for Conflicts Analyzer, *downloads.path*). It can be run by executing the runnable jar file. Download and extract ResolutionsAnalyzer.zip, available at the Resolutions Analyzer repository [16], then run:

```
java -jar ResolutionsAnalyzer.jar </path/to/ResultData/>
    </download.path/>
```

It is important that ResolutionsAnalyzer.jar is executed in the same folder as scripts/ (packaged next to ResolutionsAnalyzer.jar in the zip file).

A.1.2 Running it in Eclipse

Running Resolutions Analyzer from Eclipse can be done by cloning the repository and creating a new Java project in Eclipse.

```
git clone https://github.com/Isak-Eriksson/
    ResolutionsAnalyzer
```

Then you can either import the source files from the src/ folder and the scripts from the scripts/ folder into your newly created project, or you can create symbolic links of the src/ and scripts/ folders into your project folder.

Resolutions Analyzer uses the JXL library. Add it to the Java build path by importing jxl.jar from the repository.

Click Run -> Run configurations... then browse to your Resolutions Analyzer project. Click the tab Arguments and input </path/to/ResultData/> </download.path/> as Program arguments.

Running the project will create Results.xls in the project path.