CHALMERS
UNIVERSITY OF TECHNOLOGY

# Merge-conflict resolution patterns

A Subtitle that can be Very Much Longer if Necessary

Master's thesis in Software Engineering

Isak Eriksson and Patrik Wållgren

Department of Some Subject or Technology
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2015

Master's thesis 2015:NN

# An Informative Headline describing the Content of the Report

A Subtitle that can be Very Much Longer if Necessary

NAME FAMILYNAME

Department of Some Subject or Technology
*Division of Division name*
Name of research group (if applicable)
Chalmers University of Technology
Gothenburg, Sweden 2015

An Informative Headline describing the Content of the Report
A Subtitle that can be Very Much Longer if Necessary
NAME FAMILYNAME

Supervisor: Name, Company or Department
Examiner: Name, Department

Master's Thesis 2015:NN
Department of Some Subject or Technology
Division of Division name
Name of research group (if applicable)
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Wind visualization constructed in Matlab showing a surface of constant wind speed along with streamlines of the flow.

Typeset in LaTeX
Printed by [Name of printing company]
Gothenburg, Sweden 2015

An Informative Headline describing the Content of the Report
A Subtitle that can be Very Much Longer if Necessary
NAME FAMILYNAME
Department of Some Subject or Technology
Chalmers University of Technology

## Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Keywords: lorem, ipsum, dolor, sit, amet, consectetur, adipisicing, elit, sed, do.

# Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Name Familyname, Gothenburg, Month Year

# Contents

# List of Figures

# List of Tables

# 1
# Introduction

## 1.1 *

When developing software products using a version-control system, branching is often used, both to support variability [1] and when developing new features, which we, in this document, call feature branching. Feature branching will be the main focus in this study. It is of vital importance that the merging of these branches works smoothly, even if a conflict occurs. As of today, there do not exist many tools for automatically solving conflicts during merging. It is up to the programmers themselves to manually resolving the conflicts in the code. Understanding which patterns are used for resolving merge conflicts allows for future development of an autonomous merge-conflict tool.

The goal of this work is to conduct a feasibility study that aims at investigating to what extent and how it is feasible to analyze merge-conflict resolution patterns in large codebases comprising many open-source projects, such as those being hosted on GitHub or BitBucket.

Our long-term goal is to create an automated merge-resolution tool. Towards this end, this study aims at answering the following research questions:

- RQ1. Is it feasible to statically analyze conflict-resolution patterns in real-world, large version histories of open-source projects, and how?
- RQ2. What kind of mining and analysis infrastructure is needed for such a study?
- RQ3. Which patterns exists for resolving merge-conflicts? ...

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

## 1.2 Section

### 1.2.1 Subsection

#### 1.2.1.1 Subsubsection

##### 1.2.1.1.1 Paragraph

###### 1.2.1.1.1.1 Subparagraph

# 2 Theory

**How Conflicts Arise in a GIT-based Branching and Merging Scenario.** Many software projects follow a branching model when using Git, such as the one explained by Giessen [2]. 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 [3]. There are various ways to use branches. A branch can be created for each new feature, for each new release, and for each product [1]. 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 [4]. At GitHub, this 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 [5]. Merging can also be performed without using pull requests. When branches are to be merged, conflicts might arise. Conflicts are the problems that prevent git from automatically merging two branches together.

**Problems of Resolving Merge Conflicts.** 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 an autonomous 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 the merge-conflict resolution is performed. Hence, in the scenario above, we're interested in studying merge-conflict resolutions.

**Problems of an Empirical Study of Conflict Resolutions.** However, even that is difficult. It is unclear whether and how we could study such resolutions as they are performed by developers in real-world software. For having representative results, one would also need to study the resolutions in large codebases, such as from GitHub/BitBucket. This requires some automated analysis. It is not trivial how this automated analysis could be designed. It is also unclear how many projects can be studied; it's even not clear which projects are well-suited for such a study.

**Problem Reports and Empirical Studies.** 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.

With the use of a version control system, cloning can be managed in a more smooth way by using branching and merging capabilities [6]. GitHub uses the version control system Git, which maintains a development history for each project. In this history lies the information about when merges have occurred. When cloning features, multiple versions of the same feature exists and their consistency needs to be managed [6].

**Textual Merging.** The most commonly used merging technique is textual merging [7]. 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 [8].

**Semistructured Merging.**Semistructured Merging. Furthermore, there exist merge tools that uses another approach than textual merging, such as syntactic- and semantic merging, which have language specific knowledge [9] and does not only compare lines of text. A combination of both textual merging, syntactic and semantic merging is called semistructured merging [7]. Other studies have proved that the use of semistructured merge decreases the number of conflicts significantly. The study "Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment", proves that semistructured merge can reduce the number of conflicts by 55% [10]. These works are important to us. But instead of proposing a new conflict-resolution technique, we are interested in how developers resolve conflicts arising from different variants of features or projects.

**Conflict Patterns.** During merging, several types of conflict patterns might occur. In her study, Accioly [11] identifies numerous conflict patterns. This list will be used throughout our study both to learn about which conflict patterns exist, and later using them when developing the analyzing tool. While Accioly focuses conflict patterns, we strive to identify conflict-resolution patterns.

**Mining.** To automatically analyze projects on GitHub, a high number of requests to GitHub has to be performed. When querying requests to GitHub, GitHub has a limit of 5000 requests per hour [12].

## 2.1 Figure

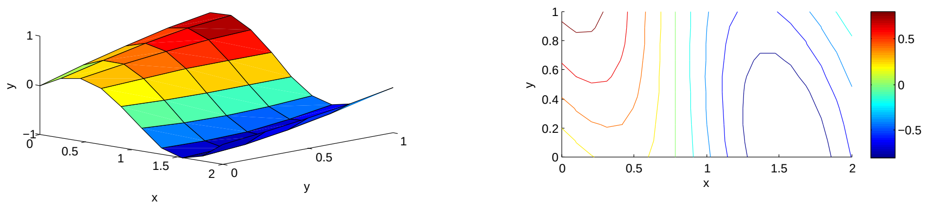


**Figure 2.1:** Surface and contour plots showing $z(x, y) = \sin(x + y)\cos(2x)$.

## 2.2 Equation

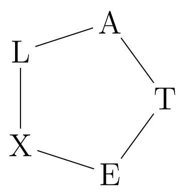
$$f(t) = \begin{cases} 1, & t < 1 \\ t^2 & t \geq 1 \end{cases} \tag{2.1}$$

## 2.3 Table

**Table 2.1:** Values of $f(t)$ for $t = 0, 1, \ldots 5$.

| $t$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $f(t)$ | 1 | 1 | 4 | 9 | 16 | 25 |

## 2.4 Chemical structure



## 2.5 List

1. The first item
   (a) Nested item 1
   (b) Nested item 2
2. The second item
3. The third item
4. . . .

## 2.6 Source code listing

```
% Generate x- and y-nodes
x=linspace(0,1); y=linspace(0,1);

% Calculate z=f(x,y)
for i=1:length(x)
 for j=1:length(y)
  z(i,j)=x(i)+2*y(j);
 end
end
```

## 2.7 To-do note

The `todo` package enables to-do notes to be added in the page margin. This can be a very convenient way of making notes in the document during the process of writing. All notes can be hidden by using the option *disable* when loading the package in the settings.

Example of a to-do note.

# 3

# Prestudy

## 3.1 Repositories to analyze

Since we are going for a quantitative analysis, we want to analyse fairly big projects that contain many commits and many forks along with many branches. This is because we want to cover as much of variant possibilities as possible to gain the most accurate result. To satisfy these requirements, the 20 top starred Java repositories on GitHub were chosen.

The projects listed in Table (no) were cloned so that Git commands could be used to analyze the repositories. Elasticsearch was chosen as the project for our initial analysis since it has a vast number of commits (more than 20000) and forks.

Elasticsearch is a distributed search engine used for analysing data in realtime.

(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 |

## 3.2 Gathering parameter data

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 indicates whether to use a certain block of code or not. In Elasticsearch, the function use 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, a tool was developed in Java which would gather the data automatically. All data that is stored in Git is hashed using SHA-1. If not stated otherwise, in this document, the hash will be referred to by SHA-1. The data to be gathered includes:

- The parameter name that was introduced
- The commit SHA
- 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 ...

The data was gathered by developing a Java program which uses Linux bash scripts that executes Git commands to get the desired data.

Git diff. As Git saves the data as snapshots and not as changes, one needs to compare two commits in order to see which changes that were introduced in a commit. To do this, we use the built in diff command in the following way: git –no-pager diff <SHA-1>ˆ<SHA-1> where îs a git shortcut to get the parent commit of a commit SHA-1.

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 SHA-1. For every commit that is checked out, we search for parameters and if there exist at least one, the commit SHA-1 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. In the beginning of developing the tool, we extracted the newly introduced boolean variables that was later used in if-statements. This proved to be not useful since the boolean variable names was not always the same as the parameter names used in the configuration file.

getAsBoolean line. While extracting the name of the parameter in the getAsBoolean 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. This data is printed to the excel document.

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 analyse manually. To get the commit message for a giver SHA-1, this command was used: git log –format=%B -n 1 <SHA-1>

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 # .

We print the parsed information in the excel document using the following format: table..

## 3.3 Extract the conflicting files

Originally, the idea was to find merge-conflict resolution patterns in Git merges. To begin searching for such patterns, we first had to figure out what patterns exist. When looking at the git merge log, using the command git log –merges, it lists the merge commits with the commit messages which could contain a list of conflicting files. However, this approach is not reliable as the commit message could be altered by the commitér who can remove the list of conflicting files. Because of this, we extended the tool to recreate all of the historical merges in the repository to see which conflicts arises. To do this, we first needed to find the two ancestral commits of a merge commit, that is, the to commits that were merged. The following command prints the two commits: git –no-pager log –merges –format=where <SHA-1> is the merge commit SHA-1. We then parse the two commits and performs the merge using the following sequence of commands. The two commits are hereby referred to as C1 and C2:

```
git reset --hard <SHA-1 of C1>
git clean -f
git branch <temp branch name>
git checkout <SHA-1 of C2>
git merge <temp branch name>
```

In 1, we set HEAD to commit C1 and changes the working copy to the state of that commit. In 2, the working copy is cleaned to be ready for the merge. In 3, a new branch is created which points at commit C1. In 4, we checkout commit C2. In 5, we merge the two commits by merging the newly created branch into commit C2. Git will now print out the conflicting files, which we then can parse. Afterwards, we abort the merge and delete the branch.

Fast forward. When merging two branches, Git first attempts to perform a so-called fast forward merge. It is a way of simplifying merges in cases where at least one of the branches points to the common ancestor of the commits pointed to by the two branches. In such a case, all that has to be done, is to change both branches to point at the latest commit. For example:
image paragraph* Three-way merge. If fast forward fails, that is, when commits has been made to both branches that are to be merged, Git has to merge all files that the commits contains. This consists of merging the two versions of every file separately, and 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.

Git conflicts. If the two commits, that are to be merged, have made changes to the same place in a file, Git will not be able to merge the two versions automatically. This is called a Git conflict. When a Git conflict occurs, Git will output the conflicting file paths in the commit message, which the tool parses.

When resolving conflicts, it is usually done by manually merging the conflicting lines of the local file (the file in the current checked out branch) with the file of the remote file (the file of the branch which is being merged into the current checked out branch) and the common ancestor file (the original file before it was changed in the local- and remote branches). These files, along with the resulting resolution file in the merge commit, are copied and saved in a Conflict file tree. The Conflict file tree consist of folders and the versions of the conflicting files, structured according to the figure below.
Having all the conflicting file versions in a structured manner will make it easier to analyse them further.

## 3.4 Details about an introduced parameter

Branches consist of one or more commits and often when new functionality is added or changed to software, good practice is to branch out, therefore developers tend to do just that. 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.

To analyse when in the branch a parameter was introduced, we extended our tool to calculate when a given parameter was introduced in a branch, given the merge commit where the branch was merged back into the original branch and the parameter name. By recursively step backwards from the merge commit, through the commits of the branch, searching for the given parameter, finding the commit where it was introduced would be a nifty, we thought. Due to limitations in the Git history regarding branches, we soon found out that analysing commits with regards to branches is very limited, if not impossible in some cases.

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.

## 3.5 Other ways of recognizing variants in code

Another way to find variant related events in the history of a Git-hosted project is to search for code clones. Something that would indicate variance is if a clone of some code block are changed in a branch and after the branch has been merged, both clones still exist. This would indicate that there were two variants in different branches and that the developers chose to keep both when merging. Cases like this would be interesting to study qualitatively to try to find patterns that could be used in an automatic analysis tool.

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 was 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.

## 3.6 Outcome of the prestudy

After the prestudy, we determined that the direction of this study will be to identify and classify merge-conflict resolution patterns.

**Possibilities and limitations of Git.** As git is a fully distributed version control system, one has almost instant access to the complete history of a cloned project. Once a project is cloned, no request limits on project hosting sites are a problem anymore. This makes analyzing a Git-hosted project nifty.

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 new commit which, unlike other commits, has two ancestral commits, 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. That commit consists of a snapshot, in which the changes introduced in the branch that is to be merged in, are applied on top of the commit that the branch which was rebased into points at. Then, both branches that were to be rebased are then changed to point at the new commit, which has only one parent, being the commit that the branch which was to be rebased into was pointing at. The other commit is left as it was. Thus, there is no nifty way of find those rebased commits.

To be able to understand what happened historically in a git-hosted project, one must have a basic knowledge about how Git works. 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” 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. 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 nifty way to tell which branch was merged into which. However, clues may be found in commit messages but, as they are often edited, they are not reliable.

Moreover, using the command “git branch –contains <SHA-1>” 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. Common practice amongst developers is to delete a branch after it has been merged, and once that happens, all information about that branch is lost from the Git history.

**Tool.** The tool created during the prestudy will be used when analyzing merge-conflict resolution patterns. It will be used to parse information about commits in GitHub repositories and also be extended with new functionality to automatically be able to identify the patterns.

# 4 Methods

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

# 5 Results

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

# 6
# Conclusion

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

# Bibliography

[1] Frisk, D. (2015) A Chalmers University of Technology Master's thesis template for LaTeX. Unpublished.

# A
# Appendix 1

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.