(Merge) Conflicts of Interest: Resolution Patterns in Practice

Midterm Report for Empiricial Methods of Software Engineering (EMSE), Spring 2016

Shane McKee
Oregon State University
2500 NW Monroe Avenue
Corvallis, Oregon
mckeesh@oregonstate.edu

Nicholas Nelson
Oregon State University
2500 NW Monroe Avenue
Corvallis, Oregon
nelsonni@oregonstate.edu

1. INTRODUCTION

Collaboration between software developers working on the same project also carries the risk of merge conflicts. With 28.4% of all projects on Github being non-personal repositories (as of Jan. 2014), the prevalence of merge conflicts is significant [7]. Since merge conflicts carry a cost to any software project, developers and researchers have pursued ways of mitigating them [11]. But mitigation strategies must be based upon a foundation of understanding both the problem space — conflicting versions of code — and the factoring that contribute to their occurrence.

Popular version control systems (git, svn, etc.) have features that allow for the detection of merge conflicts, and can automatically resolve the conflict when it adheres to certain basic patterns of textual conflicts. But when more complex conflicts occur, either in the form of syntactic or semantic conflicts, these version control systems require developer intervention to resolve the conflict. Although several papers have examined how and why merge conflicts occur [1] [12] [6], little attention has been paid to understanding how developers resolve merge conflicts in practice.

By mining software repositories on Github for instances of merge conflicts, we hope to characterize the patterns that developers take to resolve merge conflicts. When examined across the backdrop of tools in use for managing code, our data will show whether the features of these tools are adequately addressing the needs of software developers collaborating within open-source communities.

2. BACKGROUND

The underlying structure of version control systems affects the ways in which people interact with them. In this project, we focus on one aspect of the interactions with version control systems, but we provide a brief introduction to the structure as a background to our work.

Modern version control systems base their representations of code, and the underlying changes upon it, on graph theory. These models provide an entire family of models and methods for evaluating and attempting to resolve merge conflicts, but are limited either by the bounds of a particular model or the accuracy of its heuristics [5] [9]. We base our assumption that certain merge conflicts cannot be resolved by version control systems, and thus require human intervention, on these fundamental limitations in graph modeling.

Merge algorithms are an area of active research, and consequently there are many different approaches to automatic merging, with subtle differences. The more notable merge algorithms include three-way merge [8], recursive three-way merge, fuzzy patch application [3], weave merge [10], and patch commutation. These concepts form both a model of understanding and a lens for us to examine the differences between the theoretical models and real-world application.

Our research is guided by prior work into conflict detection and automated conflict resolution. Brun, et al. [2], ML Guimãraes, et al. [6], C Schneider, et al. [13], and Dewan et al. [4] have all attempted to locate current and upcoming merge conflicts as early as possible in order to prevent them from occurring. We take the approach that some conflicts cannot be detected either by collaborator awareness or by proactively engaging automated merging tools, and that understanding how developers currently adapt to such situations is fundamental to developing tools that support such situations.

3. STUDY DESIGN

3.1 Aspects of software development considered

3.1.1 Motivations

Merge conflicts have become a popular area of study, perhaps due to the importance of version control in the developer workflow or developers' dislike for resolving messy merge conflicts. Resolving a merge conflict can require extra time to understand how the two sets of new code should be integrated together.

3.2 Dataset

EMSE '16 March 28–June 10, 2016, Corvallis, OR, USA

DOI: 10.475/123_4

Table 1: Executive Summary

Goal	To understand merge conflict resolution patterns in Git repositories.
Research Questions	**RQ1:** What merge conflict resolution patterns exist in Git? **RQ2:** What is the frequency distribution of merge conflict resolution patterns by programming language? **RQ3:** Is there a relationship between the size of the conflict and its conflict resolution pattern?
Empirical Method	This project will use data mining. Data mining is best for RQ1 because it will give us the least biased view of what developers do to resolve conflicts in practice. It is best for RQ2 because It will allow us to get a sample across a wide variety of languages while testing for certain resolution patterns. It is best for RQ3 because it is easy to extract the size of a conflicted area and compare it to the type of merge conflict resolution that was used.
Data Collected	Number of merges, number of merge conflicts, size of each conflict, pattern used in each resolution, primary programming language of each project, author of each original commit, parent commit, and merged commit

Our data was selected from open-source projects on GitHub. This publicly accessible data is convenient, but it does run the risk of results not being sufficiently generalizable to private code repositories. The following data was gathered:

1. **Number of merges**
   This is the total number of merges, both automatically and manually merged. The intent is to identify what percentage the merges in our corpus must be manually merged.

2. **Number of merge conflicts**
   This is the number of merge conflicts that can be found using Git and GitPython. This will limit us to textual merge conflicits.

3. **Size of each conflict**
   We will determine the size of a conflict between commit A and commit B by the following equation:

$$\text{SLOC}(\text{git diff}(Original, A))$$

$$+$$

$$\text{SLOC}(\text{git diff}(Original, B))$$

4. **Pattern used to resolve each conflict**
   Each merge conflict resolution was classified as one of the following patterns:

   (a) *Take One:* Changes from one parent commit are taken while changes from the other parent commit are discarded.

   (b) *Interweaving:* Changes are taken from both commits and interweaving the conflicting code.

   (c) *Decline:* No changes are taken from either parent commit, and no lines are added while resolving the merge conflict.

   (d) *Overwrite:* No changes are taken from either parent commit, and code is added while resolving the merge conflict.

   (e) *Other:* The pattern found did not fall into any of the four defined patterns.

5. **Primary programming language of each project**
   Primary language is determined based on GitHub's internal analysis, which is determined by the prevalence of the language in the project.

6. **Author of each original commit, conflicting commits, and merged commit**
   We use Git's metadata for each commit to determine the author.

3.3 Data Gathering and Analysis

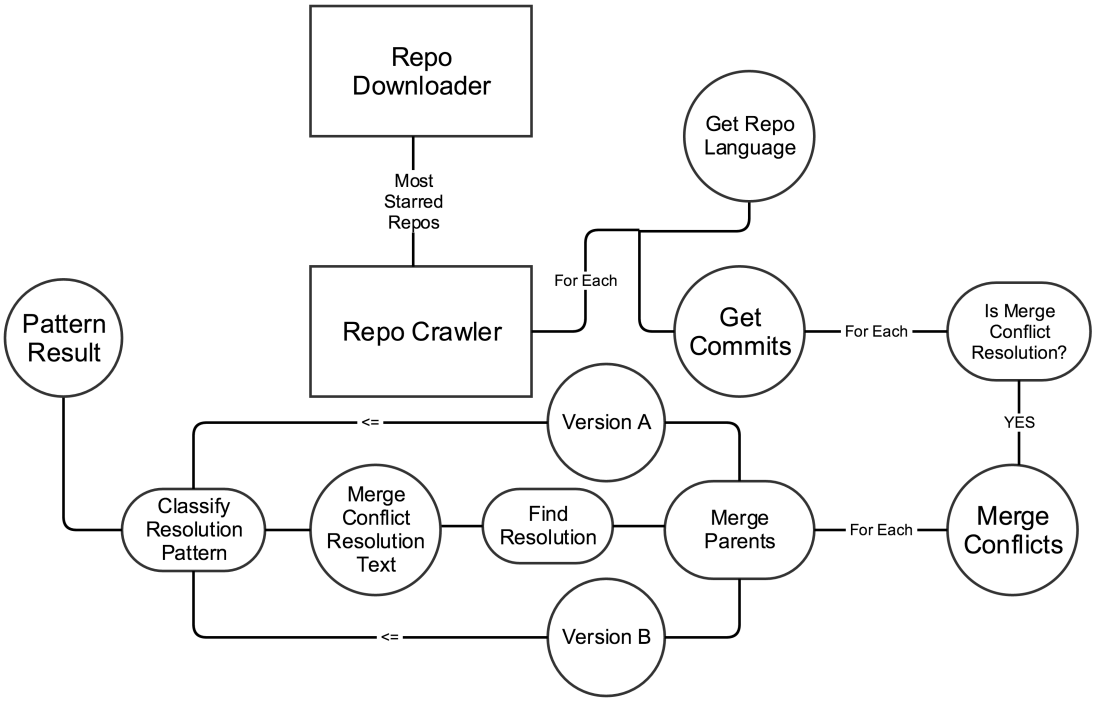
Our dataset is comprised of project metadata and commits particular to instances of merge conflicts. This dataset was gathered from Github using the GitPython tool in the following steps:

1. Collect all of the merges in a repository
2. Locate merge conflicts by merging two commits
3. Get the conflicting parts (**Part A** and **Part B**) from each commit by parsing out unmerged sections
4. Find the resulting merge conflict resolution (**M**) by branching from one parent, merging the other into the branch, committing the resulting merge conflict, then taking the diff between the commit and the merge conflict resolution.
5. Classify Part A, Part B, and M for each merge conflict resolution and its parents.

The resulting dataset required manual pruning to remove some repositories. For instance, we found that many of the most starred repositories are tutorials, code camps, or just are not projects for software development.

Our study will be limited by both space and time. Since we must store the project metadata and each target commit while mining a Github repository, and retain any set of commits that are determined to be merges, we will be bound by the storage capacity of the system we use. The constraints

Figure 1: Miner Architecture



of the term also introduces a time limitation to our project, but we should be able to mine a large enough dataset for determining initial results.

4. FORMATIVE RESULTS

Though we have not yet collected any mining data, we do have an understanding of what characteristics we should expect to see. Outside of this project, one researcher has interviewed 9 software engineers about the types of merge conflicts that they encounter and how they resolve them. This has provided a better understanding about the type of data that we are looking for during the repository mining.

5. THREATS TO VALIDITY

Generalizability The project was run on a small set of projects, which may not be representative of all projects on GitHub.

6. REFERENCES

[1] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. Speculative identification of merge conflicts and non-conflicts. *University of Washington, Seattle*, 2010.

[2] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 168–178, New York, NY, USA, 2011. ACM.

[3] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. A manifesto for model merging. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, GaMMa '06, pages 5–12. ACM, 2006.

[4] Prasun Dewan and Rajesh Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *ECSCW 2007*, pages 159–178. Springer, 2007.

[5] Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Frank Hermann. *Graph and Model Transformation: General Framework and Applications*. Springer, 2015.

[6] Mário Luís Guimarães and António Rito Silva. Improving early detection of software merge conflicts. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 342–352, Piscataway, NJ, USA, 2012. IEEE Press.

[7] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 92–101. ACM, 2014.

[8] Artem Y Livshits. Method and system for providing a representation of merge conflicts in a three-way merge operation, October 30 2007. US Patent 7,290,251.

[9] Tom Mens. Conditional graph rewriting as a domain-independent formalism for software evolution. In *Applications of Graph Transformations with Industrial Relevance*, pages 127–143. Springer, 1999.

[10] Tien N Nguyen. Conflict detection and resolution in global software design (position paper). *Workshop on*

Accountability and Traceability in Global Software Engineering, page 23, 2007.

[11] Nan Niu, Fangbo Yang, Jing-Ru C Cheng, and Sandeep Reddivari. A cost-benefit approach to recommending conflict resolution for parallel software sevelopment. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 21–25. IEEE, 2012.

[12] Anita Sarma, David Redmiles, and André van der Hoek. Empirical evidence of the benefits of workspace awareness in software configuration management. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 113–123, New York, NY, USA, 2008. ACM.

[13] Christian Schneider, Albert Zündorf, and Jörg Niere. Coobra - a small step for development tools to collaborative environments. In *Workshop on Directions in Software Engineering Environments*, 2004.