Conflicts of Interest: Merge Conflict Resolution Patterns in Open Source Projects

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 carries the risk of discord and incongruity. Within software projects that use Distributed Version Control Systems (DVCS), these issues can appear in the form 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 [8]. Since merge conflicts carry a cost to any software project, developers and researchers have pursued ways of mitigating them [12]. 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 (i.e. `Git`, `SVN`, `Mercurial`, `Bazaar`) have features that allow for the detection of merge conflicts, and can automatically resolve a subset of these conflicts using basic merging strategies. For example, `git` (the current VCS leader with 29% of developers selecting it as their primary VCS according to a 2013 study by Ohloh [6]), provides the following merge strategies: *resolve*, *recursive*, *rebase*, *octopus*, *ours*, and *subtree*. But when more complex conflicts occur, these version control systems require developer intervention to resolve the conflict. Although several papers have examined how and why merge conflicts occur [1] [13] [7], 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, and examining the resolution patterns that developers took to resolve them, we are able to highlight both the current strengths and the potential deficiencies within current version control systems. Our results show that certain resolution patterns are more prevalent within the broader open-source community, and that tool developers should focus on these particular patterns when evolving and adding to the automated systems and algorithms available for software development and project collaboration.

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

DOI: 10.475/123_4

2. BACKGROUND

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] [10]. 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 [9], recursive three-way merge, fuzzy patch application [3], weave merge [11], 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. [7], C Schneider, et al. [14], 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. Therefore, we focus our attention on the patterns that developers engage in when actively resolving such merge conflicts. Our results will hopefully provide both impetus and emphasis for further VCS development

Table 1: Executive Summary

Goal	To understand merge conflict resolution patterns in Git repositories.
Research Questions	**RQ1:** What merge conflict resolution patterns exist in GitHub? **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 used data mining. Data mining was best for RQ1 because it gave us the least biased view of what developers do to resolve conflicts in practice. It was best for RQ2 because it allowed us to get a sample across a wide variety of languages while testing for certain resolution patterns. It was best for RQ3 because it was 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.

and the creation of tools that target specific resolution patterns.

3.2 Dataset

Our data was selected from the most popular open-source projects on GitHub. We used the star ratings assigned to projects hosted on GitHub as a metric for the popularity of the project, selecting only projects that were ranked within the top 30 projects according to this metric (as of May 2016). We further filtered our selections to remove projects such as *legacy-homebrew*, *gitignore*, *html5-boilerplate*, *You-Dont-Know-JS*, *Font-Awesome*, *free-programming-books*. These projects either did not contain code in known programming languages (font templates, books, etc.) or were not collaborative projects likely to contain merge conflicts (collections of scripts, sample programs, etc.). We thus excluded them from our corpus.

Within. 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 conflicts, but maintain or alignment with analyzing the patterns associated with resolving conflicts that occur when merging within an open source project.

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 in relatively equal portions and interweaved together in the resulting merge.
   (c) *Decline:* No changes are taken from either parent commit, and no new 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:* No pattern was found that conformed to the previously outlined patterns.

5. **Primary programming language of each project**
   Primary language is determined based upon the internal analysis of GitHub, which determines programming language on a per-file basis through the use of static syntax analysis and file extensions. Each project is designated with a programming language that is most prevalent in the files that are included within it.

3.3 Data Gathering and Analysis

Our dataset is comprised of project metadata, `git` logs, and commits gathered from GitHub through a combination of GitPython, GitHub API v3, and Python libraries. Our dataset was pulled from GitHub and analyzed locally in order to avoid exceeding the access limitations placed upon developers accessing GitHub data through the use of their APIs. We gathered this dataset using the following steps:

1. Clone local copies of the `master` branch version of top projects based upon number of stars, with the limitation that some projects were not suited for this type of research and were therefore excluded.

2. Collect all merges within the `master` branch of a repository.

3. Locate merge conflicts by speculatively merging two commits, and their accompanying branch histories.

Table 2: Corpus Description

Project	Language	Stars	Commits	Merges	Conflicts
angular/angular.js	JavaScript	49,777	7,839	137	14
danedan/animate.css	CSS	32,732	1,216	92	8
robbyrussell/on-my-zsh	Shell	38,097	3,977	1,468	35
FreeCodeCamp/FreeCodeCamp	JavaScript	136,245	8,548	333	22
impress/impress.js	JavaScript	27,550	261	114	17
BVLC/caffe	C++	10,664	3,705	1,191	78
hakimel/reveal.js	JavaScript	28,558	1,796	217	25
smashingboxes/cardboard	Ruby	584	643	65	13
nodejs/node	JavaScript	23,907	14,353	495	322
tensorflow/tensorflow	C++	63,206	3,716	68	367
cbeus/testng	Java	667	3,437	214	95
sleexyz/hylogen	Haskell	209	175	12	3
snapframework/snap-core	Haskell	255	939	146	108
voldemort/voldemort	Java	1,695	4,193	371	311

* Collected from GitHub on May 31-June 07, 2016

Table 3: Resolution Patterns

		Resolution Pattern Usage				
Project	**Language**	*TakeOne*	*Interweaving*	*Decline*	*Overwrite*	*Other*
angular/angular.js	JavaScript	1	0	8	1	7
danedan/animate.css	CSS	0	0	5	3	0
robbyrussell/on-my-zsh	Shell	0	0	8	27	0
FreeCodeCamp/FreeCodeCamp	JavaScript	0	0	8	14	1
impress/impress.js	JavaScript	0	0	2	2	0
BVLC/caffee	C++	0	17	9	51	3
hakimel/reveal.js	JavaScript	2	0	5	15	3
smashingboxes/cardboard	Ruby	0	0	15	22	3
nodejs/node	JavaScript	0	12	187	130	36
tensorflow/tensorflow	C++	0	10	24	319	14
cbeus/testng	Java	0	8	1	80	6
sleexyz/hylogen	Haskell	0	0	0	2	1
snapframework/snap-core	Haskell	0	3	17	88	0
voldemort/voldemort	Java	0	26	18	267	0

4. Obtain the conflicting versions of each text section (`section A` from commit `A`, and `section B` from commit `B`) by parsing the unmerged sections resulting from the speculative merge.

5. Obtain the merge conflict resolution text section (commit `M`) by branching from parent `A`, merging parent `B` into that newly created branch, committing the resulting merge conflict as if it were resolved, taking the `diff` of the commit `M` and the merge conflict resolution.

6. Analyze `section A`, `section B`, and `section M` for each merge conflict resolution in order to categorize the resolution into one of the previously indicated classifications.

Since most of the repositories not suitable for this research were already removed prior to this analysis process, the resulting dataset required only minimal manual pruning prior to conducting population analysis.

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 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. RESULTS

4.1 Existence of Merge Conflict Resolution Patterns

RQ1: What merge conflict resolution patterns exist in GitHub?

The original set of patterns that we provided within our Project Proposal was naive and inconsistent with the data that we found during our initial exploration of popular repositories in GitHub prior to mining them. We updated our set of patterns based upon this new information and provided those patterns in our Midterm Report. We have since conducted several rounds of software repository mining, and refined our patterns down to the set of five presented in Section 3.2.

Through the mining and analysis of 14 projects on GitHub, we have found that each project averages 23.00167 conflicts per 1,000 commits and a variety of at least 4 different patterns per 1,000 commits. With an rate of 0.059 *TakeOne* patterns per 1,000 commits (3 instances overall), 1.501 *Interweaving* patterns per 1,000 commits (76 instances overall), 6.066 *Decline* patterns per 1,000 commits (307 instances overall), 20.175 *Overwrite* patterns per 1,000 commits (1,021 instances overall), and 1.343 *Other* instances per 1,000 commits (68 instances overall), and an overall population of 54,798 commits and 1,475 conflict resolution pattern instances, we are confident that merge conflict resolution patterns do arise in regularity within open source projects on GitHub.

4.2 Frequency of Merge Conflict Resolution Patterns by Programming Language

RQ2: What is the frequency distribution of merge conflict resolution patterns by programming language?

The projects selected for our Final Project Presentation were made out of convenience and prudence of time, after the majority of our efforts went toward developing the automated GitHub repository mining and evaluation system. With the majority of the development finished, we were able to select a more representative population for this Final Report. We diversed based upon language and the variety of applications of each project for our final population.

We selected 5 JavaScript projects, 2 C++ projects, 2 Java projects, 1 Shell project, 2 Haskell projects, 1 Ruby project, and 1 CSS project. These projects were targeted at a variety of markets, including: machine learning API, devOps infrastructure, website development, programmer code camps, online web animation frameworks, and content delivery networks (CDNs). The frequency of individual merge conflict resolution patterns can be observed in Table 4.

4.3 Relationship between Conflict Size and Conflict Resolution Patterns

RQ3: Is there a relationship between the size of the conflict and its conflict resolution pattern?

For this question, we had hoped to gather additional statistical information about the conflict sizes and significance of each combination of merge conflict resolution pattern and conflict size. With this type of data, we planned to use regression models and Wilcoxon signed-rank test to determine significance of any relationships between conflict size and merge conflict resolution patterns.

With the time constraints of this project, we leave this question and analysis for future work.

5. THREATS TO VALIDITY

Internal Validity The methodology used to determine the thresholds and hueristics for our merge conflict resolution patterns have not been rigorously tested for statistical validity, and could be calibated either ineffectively or incorrectly.

External Validity This project was run on a small number of projects on GitHub, with a significant skew toward the JavaScript language due to their popularity on GitHub. Although this population is representative of the most popular projects on GitHub, it is unlikely to be representative of the larger open source community of projects. Therefore, the generalizability of this project cannot be extended beyond such a population.

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.

Figure 1: Miner Architecture

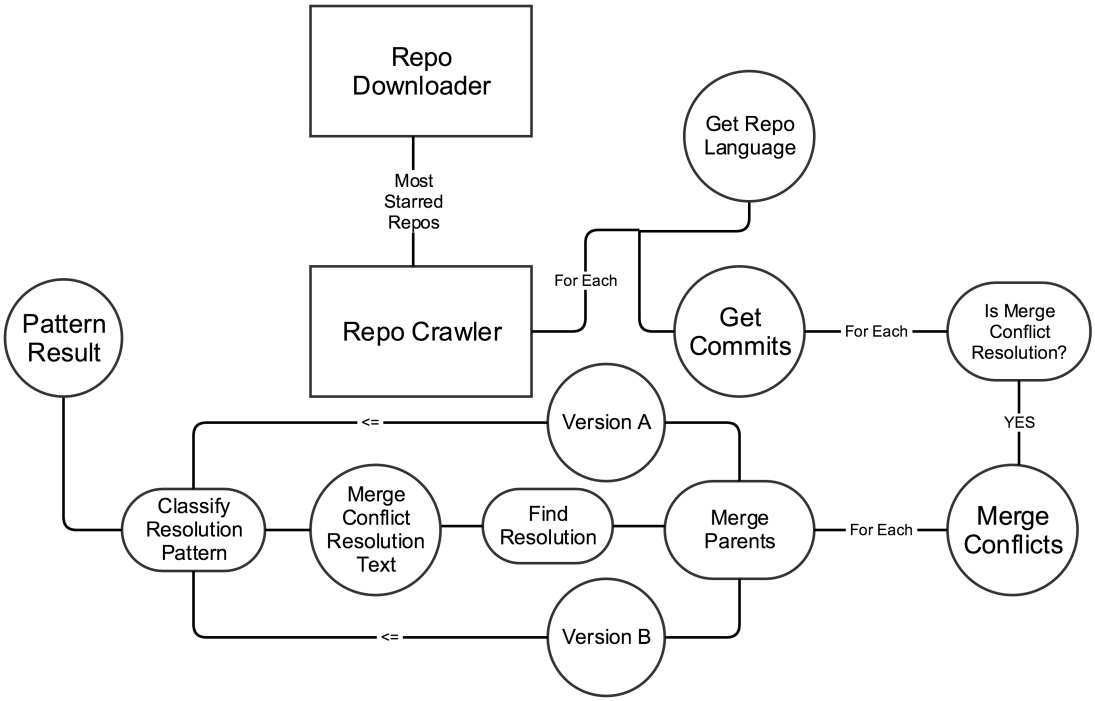

Table 4: Resolution Patterns by Programming Language

	Resolution Pattern Usage Totals				
Language	*TakeOne*	*Interweaving*	*Decline*	*Overwrite*	*Other*
JavaScript	3	12	210	162	47
CSS	0	0	5	3	0
Shell	0	0	8	27	0
C++	0	27	33	370	17
Ruby	0	0	15	22	3
Java	0	34	19	347	6
Haskell	0	3	17	90	1

[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] Daniel M. German, Bram Adams, and Ahmed E. Hassan. Continuously mining distributed version control systems: an empirical study of how linux uses git. *Empirical Software Engineering*, 21(1):260–299, 2016.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.