Git Corpus: Analyzing git-merge scenarios extracted from Open Source Software

Richard Li[1], Sebastien Mosser PhD[2]

[1] Department of Computing and Software, McMaster University, Hamilton, Canada.

Introduction

- Each developer has felt at least once the so-called "merge-conflict panic": your code works well, and then you push your code to the shared repository, and suddenly the world crashes into pieces. Your code is now full of conflict markers!
- The study of a smarter merge algorithm is currently flawed (Shen et al, ICSE'20), as each algorithm is validated on a tailer benchmark.
- Unfortunately, except from a paper that does not contain reproducible data (Ghiotto et al, ICSE'18), a reference benchmark does not exist.

Git

- Distributed **version control system** commonly used to track changes during the Software Development process [1].
- Captures **snapshots** in time, rather than only gathering the changes made.

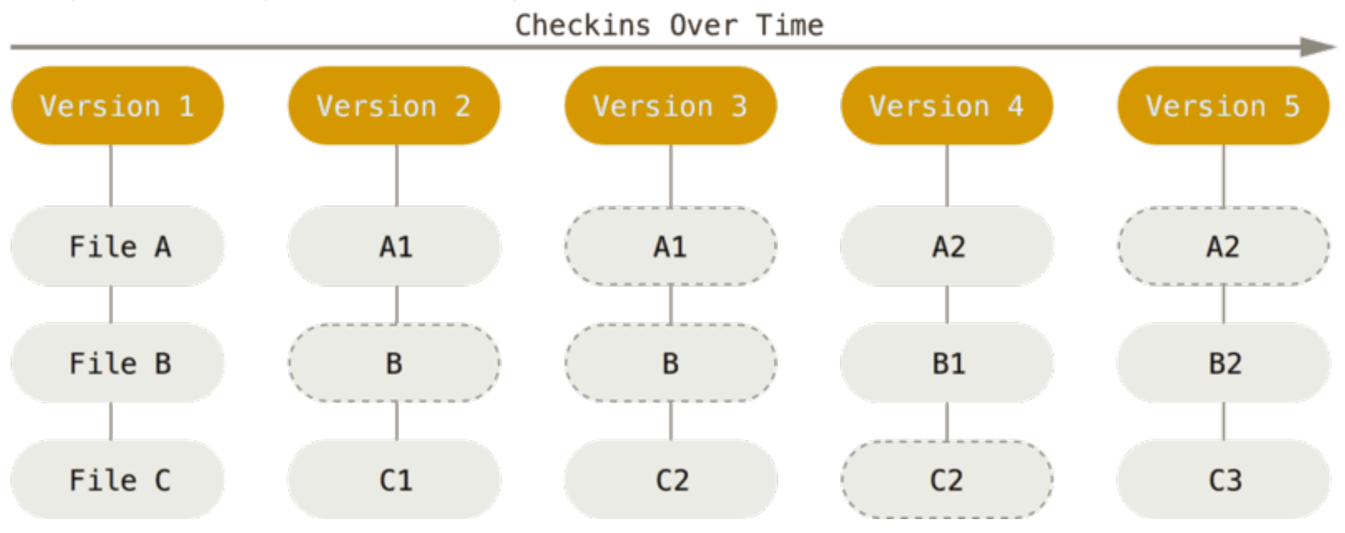


Figure 1: Snapshot view of storing data for a project

- Every snapshot saved to the Git database is a **Commit** and stored as a **Commit Hash** - a 40 character hexadecimal string.
- Over the course of a project, there may be an accumulation of thousands of these snapshots in time.

Merge Commits

- Normally happens in projects with multiple developers.
- Occurs when two different **Commits** are combined together successfully.

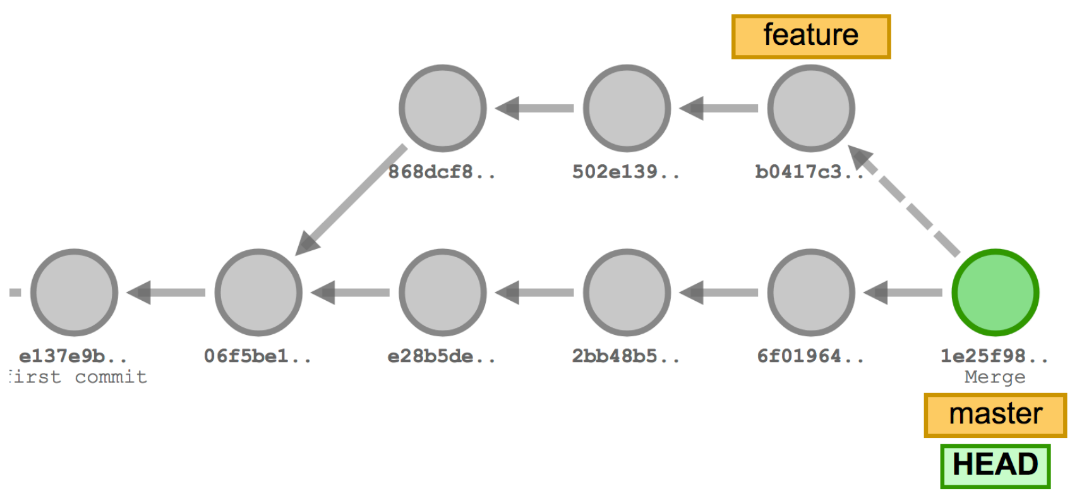
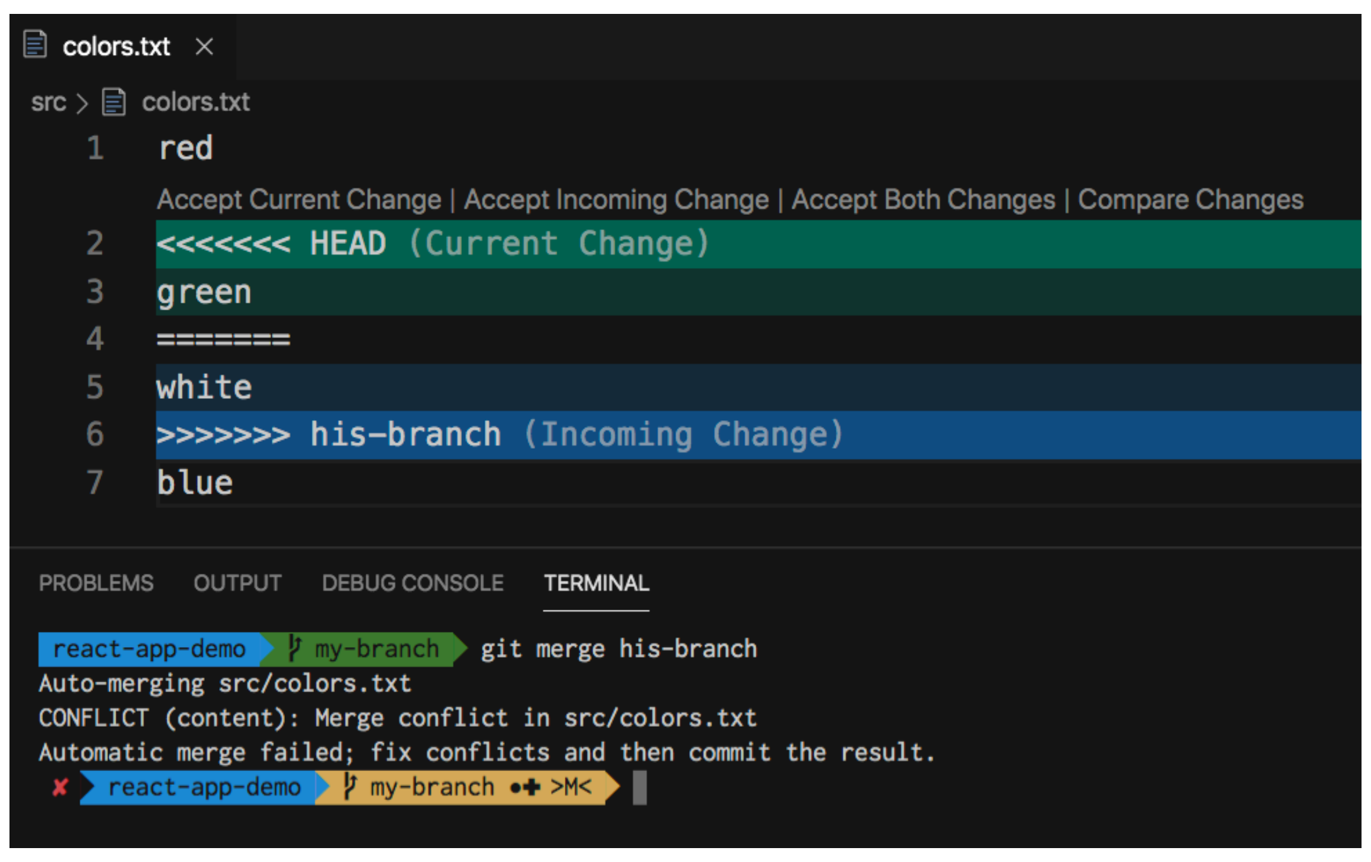


Figure 2: Branching Merge Commit (Timeline moves left to right) [2]

- In a merge, there are 4 commit hashes involved:
  1. Merged hash (green circle)
  2. Left parent (feature branch)
  3. Right parent (master branch)
  4. Common Ancestor
- There are 2 cases of Merge Commits:
  1. Diamond shape (Figure 2)
  2. Fast Forwarded (Common Ancestor is the Left parent)

Merge Conflict

- Normally happens in projects with multiple developers.
- Occurs when two separate **Commits** have different changes to the same lines of a specific file.
- The Git merge algorithm is unable to decipher which change is the correct one.
- The developer conducting the merge must manually correct these changes, the code will now have "conflict-markers".


colors.txt
src > colors.txt
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL

```
red
<<<<<<< HEAD (Current Change)
green
=======
white
>>>>>>> his-branch (Incoming Change)
blue
```

```
react-app-demo  my-branch  git merge his-branch
Auto-merging src/colors.txt
CONFLICT (content): Merge conflict in src/colors.txt
Automatic merge failed; fix conflicts and then commit the result.
✘  react-app-demo  my-branch ●✚ >M<
```

Figure 3: Merge Conflict-Marker [2]

Objectives

- To **identify** merge scenarios in open source Java projects and organize them into a reference corpus.
- **Develop** numerous frameworks to run arbitrary analyses on collected scenarios.
- **Implement** a JavaParser to parse Java code into nodes in an Abstract Syntax Tree to identify programming elements.
- **Determine** which programming elements are most susceptible in merge conflict scenarios.

Note: The Java projects that are selected are from the "Awesome Java" repository, a Category contains many similar types of projects.

Methodology

1. Capturing the **Commit Hashes** in the Git Commit history by annotating for the type of Commit that is present
2. From the Hashes with **Two Parents**, gather the 4 commit hashes (refer to Merge Commits section) to gather the type of merge occurred (**Diamond** or **Fast-Forwarded**). Gather the contributions of each parent in terms of line additions and deletions.
3. Screen and identify the **Files** that were involved in the conflict only for the **Diamond** merge scenarios. **Simulate** a merging the left and right parents to check if a merge conflict occurred and collecting the specific files that were involved.
4. Identify and keep track of **Java elements** that were involved in a merge conflict through simulating a merge scenario and gathering the line numbers. Parse the Java code looking for the elements present at each line.

Results

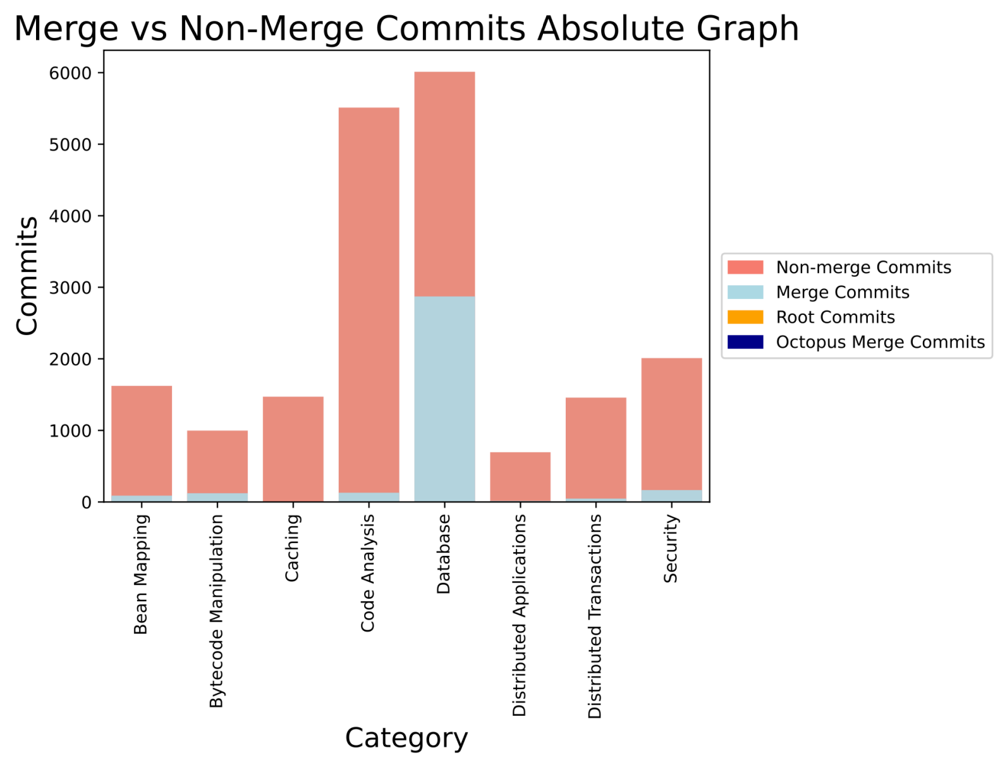


Figure 4: Types of commits present in a corresponding category

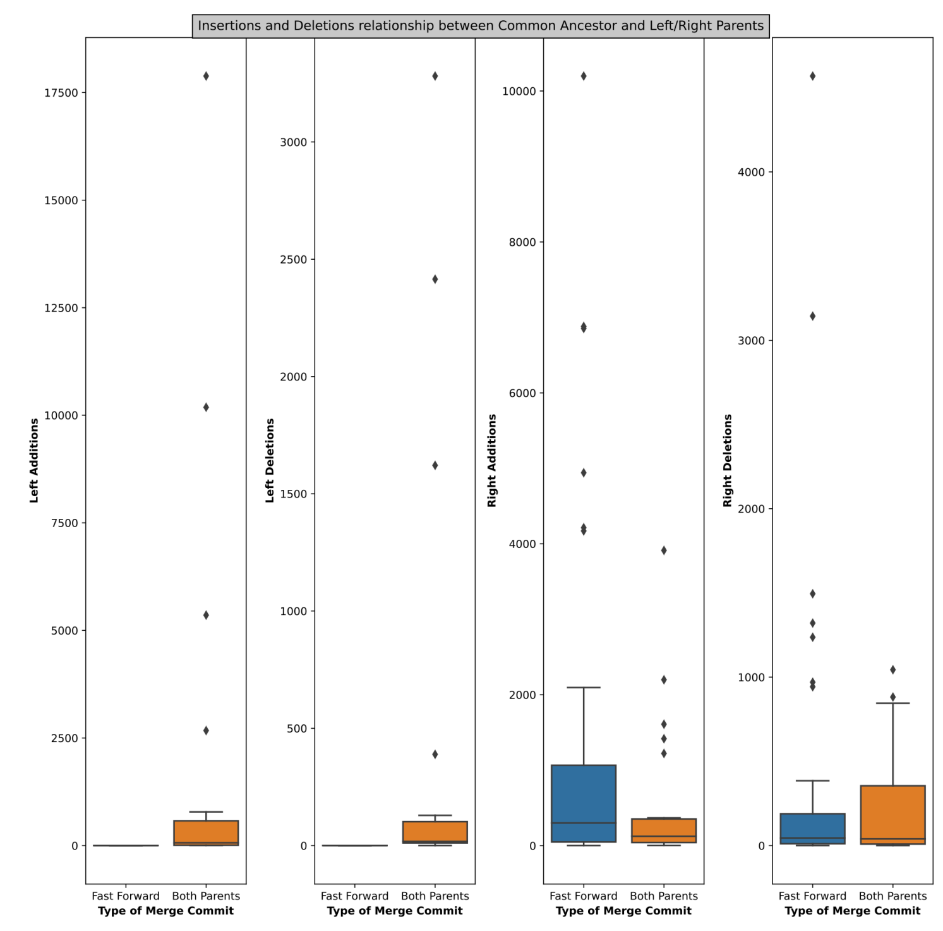


Figure 5: Addition and Deletion distribution between the two types of merge commits.

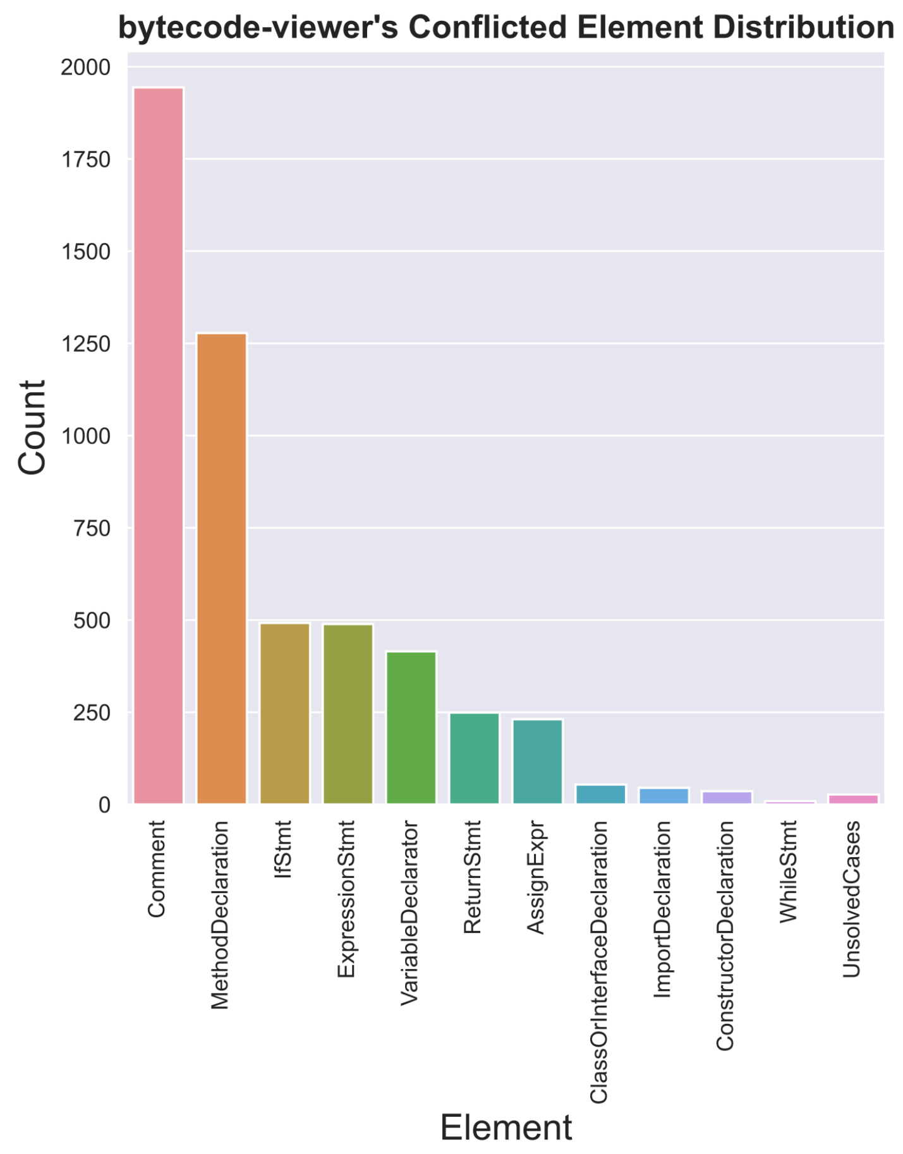


Figure 6: Most conflict prone elements in the Bytecode Viewer project

Conclusions

- From figure 4, the "one-parent" commit is the most common one. Well engineered projects limit the amount of merging needed to be done.
- The reason why the **Left Deletions** and **Additions** are zero in the fast-forwarded merge commit is that the **Left Parent** is the **common ancestor** - they are identical.
- In a **diamond** scenario, the results are quite erratic, the changes can be attributed to both the left and right parents.
- After analyzing numerous projects (Bytecode Viewer was just an example figure 6), the **Comment** element seems to be one of the most conflict prone elements.

Future Work

- Continue to run the frameworks on a larger sample size to confirm results (on different categories and more projects)
- With a reference data collected, begin to analyze the individual merges.
- Develop a new merge algorithm based on the large dataset to handle such cases.
- Work with NLP frameworks to be able to parse conflicted comments to determine correctness.
- Extend Support for Python, C++, JavaScript as well.

References

(1) S. Chacon and B. Straub, Pro Git, Berkeley, CA, USA:Apress, 2014. [Accessed August 19, 2022]

(2) S. Mosser, "Git Corpus" https://github.com/ace-design/git-corpus [Accessed August 19, 2022]

(3) "Git Merge" atlassian.com. https://www.atlassian.com/git/tutorials/using-branches/git-merge [Accessed August 19 2022]

Richard Li
B.Eng Software Engineering

McMaster University, Department of Computing and Software

Email: li1502@mcmaster.ca

BRIGHTER WORLD