An Exploratory Study Into the Prevalence of Botched Code Integrations

Ward Muylaert
wmuylaer@vub.ac.be
Software Languages Lab
Vrije Universiteit Brussel

Coen De Roover
cderoove@vub.ac.be
Software Languages Lab
Vrije Universiteit Brussel

Abstract

When integrating code, semantic conflicts may occur. We look into how often these conflicts occur and what it takes to fix them. We find that conflicts occur often and take a non-trivial amount of work to fix.

1 Introduction

Software projects are often managed in version control systems (VCS) like Git or SVN. Version control systems aid in managing different versions of a project. VCS provide a history of the versions building upon one another. VCS also enable the use of so called branches. In a branch, work can be performed independently of other changes. For example, to work on a particular feature or bug fix. Different branches may be merged together again at some point in time to combine the different changes that were made.

Merging does not always have successful results. Three different types of conflicts are often considered [5]. A textual conflict occurs when two different branches contain changes to the same line of code. The merging tool does not know which version to prefer. A syntactic conflict occurs when the result of a merge is no longer syntactically correct. For example, two branches each surround a statement with an if statement. The resulting program has two if statements, but only one closing curly bracket. A semantic conflict may occur when the merged program is syntactically correct, but no longer behaves as intended. This could be static, for example due to the disappearing of a variable in one branch that is still used in another branch. Alternatively, the program may run without errors, but the conflict may be unexpected or unintended behaviour. This conflict would not be found until the code was run, during a test or, worse, in production.

While these conflicts are well defined, there is little information as to how often they occur. Previous work [2] looked at nine open source projects using Git. The 3,562 merge commits spread across these projects were investigated. This study showed about one in six merge commits leads to a textual conflict. Of those nine open source projects, three were investigated further for build and test failures. For these three, build failures were found in 0.1%, 4%, and 10% of merge commits. Test failures were found in 4%, 28%, and 3% of merge commits. This study is a start, but could do with being looked at on a larger scale.

In this presentation, we look at projects on a larger scale. Using information from GitHub, a source code repository host, enables the analysis of a large amount of software projects. This information is combined with Travis CI, a continuous integration service. Travis CI can be coupled for free with open source projects on GitHub. For every commit pushed to a project on GitHub, Travis CI will build the program, run the test suite, and report the results back to the developers of the project. Travis CI makes these results publicly available as well.

Using this combined information, we look into the following research questions:

1. How often does code integration lead to semantic conflicts?

2. How much effort is needed to fix semantic conflicts after code integration?

3. How long does it take to fix semantic conflicts after code integration?

2 Dataset

As a basis for our dataset, we consider projects on GitHub. GitHub is a source code repository host. Open source projects can be hosted on GitHub for free. Many projects do: in 2013, GitHub was the host to ten million source code repositories.[1] A large amount of projects are publicly available to everyone. GHTorrent [3] attempts to bring this information to researchers. GHTorrent uses the GitHub API to create an offline mirror of data on GitHub. GHTorrent uses this data to create a MySQL database, making it easily queriable. The GHTorrent MySQL dump currently contains over 400 million commits.

Travis CI is a continuous delivery service. Travis CI provides free continuous delivery for open source projects on GitHub. The results of the builds and tests Travis CI performs are publicly available for these projects. TravisTorrent [1] is an attempt for Travis CI similar to what GHTorrent is for GitHub. We were intent on using TravisTorrent, but encountered some trouble working with TravisTorrent as provided. The data from Travis CI we are interested in does not yet, at the time of writing, seem to be a part of TravisTorrent. Instead, we created our own local dataset by calling the API provided by Travis CI. The information gathered is stored in a MySQL database to combine the Travis CI data with the data of the GHTorrent database. We mirror three of the entities available through the Travis CI API: repositories, commits, and builds. Combined these provide information about the success or failure of the builds and tests run on a commit. It also connects the data to the data in GHTorrent.

To select projects, we started out with the projects as defined by TravisTorrent [1]. Project selection for TravisTorrent was done by considering projects in Ruby or Java with the following criteria [4]: projects must have forks, received a commit in the last six months, received at least one pull request, and have more than 50 stars on GitHub. Going by these criteria, TravisTorrent had collected 1300 projects. We collected information on these 1300 projects through the Travis CI API. Some projects were no longer available (e.g., due to removal by the authors). This left us with 1248 projects and 1.1 million builds spanning those projects.

We then used the information in GHTorrent to find merge commits. A merge commit is a commit with more than one parent commit. We further filter out projects that have less than 50 builds of merge commits. This leaves us with 584 projects. Of these, the quartiles of the amount of builds of merge commits per project are 75, 114, and 217.5.

3 Results

3.1 Frequency of Conflicts

To consider the failure of a commit, we look at the state of the build as defined by Travis CI. Travis CI associates a different state to a build depending on how that build went. As such we can consider the states representing failed builds and tests to measure the proportion of conflicts.

With this information, we consider for each project the proportion of merge commits which lead to a state of failure. We find that for this the quartiles are at 6.7%, 15.3%, and 29.0%. In other words, for half of the projects a conflict occurred in about one in six merges. Worse, for one in four projects a conflict occurred almost once every three merges.

In looking at these results, we also notice some outliers. We filter out those projects causing the outliers. The projects in question had conflicts in over 62.6% of merge commits. These projects do not seem to adhere to continuous delivery. After filtering, 559 projects are left in the dataset.

3.2 Effort to Fix Conflicts

As a proxy for effort, we consider the amount of builds it takes before tests pass again. We call this the NBTF or Number of Builds To Fix. If the next build after the failing merge commit passes all tests successfully, NBTF is one. For every extra build needed, NBTF also increases by one.

We find the quartiles at 2, 4, and 16. Half of the failing merge commits take more than four builds before all tests pass again.

[1] https://github.com/blog/1724-10-million-repositories.

We note that NBTF is but a proxy for effort. A bug may be fixed in one small commit, but that does not mean it could not have taken a lot of effort to track down the cause of the bug. Conversely, a bug may not be fixed until several commits later due to a developer not strictly adhering to principles of continuous delivery.

3.3 Time to Fix Conflicts

To consider how long it takes to fix a conflict, we look at the time between the failing of the merge commit and the first build, after that merge commit, that passes all tests. We call this the TTF or Time To Fix. It is the difference between the timestamp of the failing build and the timestamp of the first successful build following.

For half of failing merge commits, it takes more than 12 hours before there is a successful build. Almost one in five failing merge commits take between 24 hours and 7 days to fix. Another 11% even take between 7 and 30 days.

In this case it should be noted that the dataset consists of open source projects. Many of these are run by volunteers in their spare time. As such, it is possible the volunteers in question simply did not have the time any sooner. A fix taking seven days may mean the developer simply did not have time for 6 days and then briefly looked at things to fix it.

4 Conclusion

We used data gathered from GitHub, a source code repository host, and Travis CI, a continuous delivery service. By looking at the intersection between them, we had information about projects and when these projects passed their tests. We used those projects with enough merge commits to look into conflicts and their resolution. We found that

1. Conflicts after code integration happen often. For half of the projects, the build did not succeed for one in six merge commits.
2. Multiple builds are needed to fix the majority of botched code integrations.
3. The majority of botched code integrations are fixed within a day.

References

[1] Moritz Beller, Georgios Gousios and Andy Zaidman. *Oops, My Tests Broke the Build: An Analysis of Travis CI Builds with GitHub.* Tech. rep. PeerJ Preprints, 2016.

[2] Yuriy Brun et al. "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)*. 2011.

[3] Georgios Gousios. "The GHTorrent Dataset and Tool Suite". In: *International Conference on Mining Software Repositories (MSR)*. 2013.

[4] Eirini Kalliamvakou et al. "An in-depth study of the promises and perils of mining GitHub". In: *Empirical Software Engineering* (2015).

[5] Tom Mens. "A State-of-the-Art Survey on Software Merging". In: *IEEE Transactions on Software Engineering* (May 2002).