Prevalence of Botched Code Integrations

Muylaert, Ward; De Roover, Coen

Published in:
Proceedings - 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, MSR 2017

DOI:
10.1109/MSR.2017.40

Publication date:
2017

Document Version:
Accepted author manuscript

Prevalence of Botched Code Integrations

Ward Muylaert
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
ward.muylaert@vub.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

***Abstract*—Integrating code from different sources can be an error-prone and effort-intensive process. While an integration may appear statically sound, unexpected errors may still surface at run time. The industry practice of continuous integration aims to detect these and other run-time errors through an extensive pipeline of successive tests. Using data from a continuous integration service, Travis CI, we look into the prevalence of integration errors. We find code integration causes failure less often than regular commits. Repairing is usually done the same day and takes less than ten lines of code, largely in the source code. These results indicate that applying proper practices mitigates many issues associated with code integration.**

I. Introduction

Version control repositories enable working on independent versions of a project in so-called branches. Merging two branches combines their changes, but this is not always successful. Three different types of merge conflicts can be discerned [5]. A *textual conflict* occurs when the same line of code has been changed in both branches. A *syntactic conflict* occurs when the result of a merge is no longer syntactically correct. A *semantic conflict* occurs when the result is syntactically correct, but no longer behaves as intended.

For this mining challenge, we analyse the prevalence of syntactic and semantic conflicts on a large scale. We combine information from GitHub, a version control repository host, with information from Travis CI, a continuous integration service. This because many projects hosted on GitHub have been configured so that Travis CI will build the program, run its test suite, and report the results back to the developers upon every commit pushed to GitHub. In the case of open source projects, Travis CI makes these results publicly available.

While the types of conflicts are well-defined, there is little information on their frequency. Brun et al. [2] analysed $3{,}562$ merge commits across nine open source projects. Their study observed that about one in six merge commits leads to a textual conflict. Three of the nine open source projects were investigated for build and test failures. 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.

Though the first of its kind, the study lacks in two aspects. First, the sample size is small. Only three projects were investigated in terms of conflicts beyond the textual. Second, the study did not consider what was done to fix these failures.

We seek to answer the following research questions:

RQ1 How often do code integrations lead to conflicts?
RQ2 How much effort is needed to fix conflicts after code integration?
RQ3 What type of files is this effort concentrated in?

II. Dataset

A. Origin of Dataset

To answer these research questions, we need to combine two datasets. GHTorrent [3] (version 2016-05-04) provides GitHub data, while TravisTorrent [1] (version 2016-12-06) provides Travis CI data. The TravisTorrent dataset contains information on whether or not the build and tests after a commit succeeded, for about 1300 Ruby and Java projects. These projects meet the following criteria defined by Kalliamvakou et al. [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. Our study still requires GHTorrent in order to identify merge commits in this dataset, based on information about their parent commits. To this end, we link commits from either dataset using their SHA-1 hash.

B. Refining the Dataset

We perform a three-step refinement on the dataset to ensure its projects have sufficient merge commits, and adhere to continuous integration practices. First, the refinement eliminates projects with less than 50 builds of merge commits. This step leaves 579 projects.

Second, the refinement filters out projects with a build success rate under 34%. We suspect these projects of not adhering to continuous integration practices. The success rate of a project is the ratio of passed builds compared to the total number of builds. The quartiles are at 0.67, 0.81, and 0.89. The interquartile range IQR defines a lower bound l for the success rate: $l = Q_1 - 1.5 * IQR = 0.34$. Of the 579 projects, 555 have sufficient success rate.

Third, our research method requires information on the build of a merge commit *and* its parents. This step eliminates projects with build information on less than 50 merge commits and their parents. This refines the dataset to 348 projects.

Table I and Fig. 1 characterize the 348 projects in the refined dataset by the number of commits, the number of merge commits, the maximum team size, and the number of branches.

TABLE I
A SUMMARY OF THE 348 PROJECTS WITH 50 OR MORE BUILDS OF MERGE COMMITS AND PARENTS AS WELL AS SUFFICIENT SUCCESSFUL BUILDS.

	Commits	Merges	Team size	Branches
Min	138	50	2	1
Q1	360	104.8	9	20.75
Median	566	155	13	53
Q3	1120	288.5	20	117.5
Max	19142	8169	288	1022

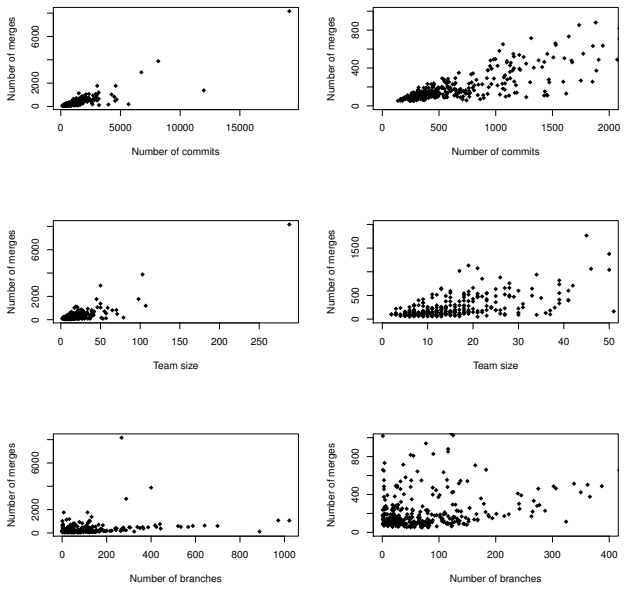

Fig. 1. Number of merge commits plotted against from top to bottom: number of commits, maximum team size, and number of branches. Each left graph contains all 348 selected projects. Each right graph zooms in on a section closer to the origin. Every dot corresponds to one project.

III. RESEARCH METHOD

Before explaining the research method for each research questions, we define three concepts: breaking commit, fixing commit, and merge commit.

A *breaking commit* is a commit of which the build has status "failed" and of which the parent commit(s) have builds with status "passed". Considering the build status irrespective of the one of the parents would skew results. This because a build can remain failing for several builds in a row. The build information is, through TravisTorrent, provided by Travis CI. Travis CI builds can have a status "passed", "errored", "failed", "started", or "cancelled". "Started" means the continuous integration pipeline is still running. "Cancelled" is a state triggered by the project's developers if they choose to cancel a run of the pipeline. "Passed" means nothing went wrong during building or testing of the project. "Errored" means something went wrong in setting up the project (e.g., a dependency could not be installed). "Failed" means something went wrong either while building the project or while running the project's tests. This build status is therefore indicative of the syntactic and semantic conflicts we are interested in.

A *fixing commit* is the first commit with a build status "passed" after a breaking commit. We define succession in terms of TravisTorrent information. Each build entry in TravisTorrent has a `tr_prev_build` field which links to the `tr_build_id` of its previous build. We repeatedly follow this link until the first build that passes.

A *merge commit* is a commit with two or more parent commits. To identify these commits in the TravisTorrent dataset, we look up the commit with the corresponding SHA-1 hash in the GHTorrent dataset. GHTorrent provides information about the parents of a commit through its `commit_parents` table.

A. Frequency of Conflicts

Our research method for RQ1 consists in analysing the frequency of breaking commits. For each project, we compute the ratio of breaking commits to all commits. This for breaking regular commits and breaking merge commits separately. Merge commits are then categorised into pull requests and others. A pull request is a GitHub concept enabling explicit review of patches to a repository. Contributors can review the patch, suggest changes, or comment on it before it is merged into the repository. We identify pull requests through the `gh_is_pr` field in TravisTorrent.

Section IV-A employs the following metrics to answer RQ1.

- $BREAK\%$: The ratio for a project of the number of breaking commits ("failed" after "passed") to the total number of commits after a passing build (anything after "passed"). A lower number is better.
- $BREAK\%_{RC}$, $BREAK\%_{MC}$: The $BREAK\%$ for regular commits and merge commits respectively.
- $BREAK\%_{MCNPR}$, $BREAK\%_{MCPR}$: The $BREAK\%$ for non pull request merge commits and pull request merge commits respectively. This is a breakdown of $BREAK\%_{MC}$.

B. Effort to Fix Conflicts

Our research method for RQ2 is measuring proxies for the effort involved in fixing a build. We use three metrics. The first metric is the number of builds needed to fix a breaking commit. This number is the number of steps as described in finding the fixing commit in Section III. We prefer to look at the number of builds over the number of commits. When several commits are pushed at once, Travis CI will only build the last one. Our assumption here is that the developer will push their changes once they think the fix is ready. The second metric is the number of changed lines between the breaking and the fixing commit. The final metric measures the time between breaking and fixing commit. The metric considers the `gh_build_started_at` field provided by TravisTorrent. `gh_build_started_at` has a precision of a day. The measured differences will thus also have a precision of a day.

- $NBTF$: The number of builds to fix: how many builds it takes before a breaking commit is fixed. A lower number is preferred.
- $LINES$: The number of lines changed between the breaking and fixing commit. A lower number indicates a possibly lower effort.
- TTF: The time between the breaking commit and its fixing commit. Lower may indicate an easier fix.

We define $\overline{METRIC}$ as the median $METRIC$ in a project.

TABLE II
A SUMMARY OF THE $BREAK\%$ FOR ALL 348 PROJECTS.

	Regular	Merge	Not PR	PR
Min	0.00	0.00	0.00	0.00
Q1	4.02	0.00	0.00	0.00
Median	6.90	2.34	2.22	0.00
Q3	11.20	5.78	5.79	0.00
Max	30.67	43.40	43.40	100.00

C. Source vs Test

To answer RQ3, we categorise a fix into one of four categories. We take the sum of the changes between the breaking merge commit and its fixing commit. Specifically, we use the `git_diff_src_churn` and `git_diff_test_churn` fields of TravisTorrent. The four categories are: "source" (a fix with only changes to source code), "test" (a fix with only changes to test code), "both" (a fix with changes to both source and test code), and "none" (a fix with changes to neither source nor test code). For each project, we count the number of fixes in each category relative to the project's total amount of fixes to define the four following metrics per project:

- SRC: The ratio of "source" fixes.
- $TEST$: The ratio of "test" fixes.
- $BOTH$: The ratio of "both" fixes.
- $NONE$: The ratio of "none" fixes.

IV. RESULTS

A. Frequency of Conflicts

For RQ1 we consider the $BREAK\%$ metric. The metric uses the previous build for regular commits as defined in TravisTorrent. The metric only considers merge commits with exactly two parents. The dataset resulting from Section II-B has but one merge commit with more than two parents.

Table II and Fig. 2 (a) depict $BREAK\%_{RC}$ and $BREAK\%_{MC}$, the $BREAK\%$ for regular commits and merge commits respectively. We notice merge commits break the builds *less* often than regular commits do. Fig. 2 (b) splits up the merge commits into two categories: pull request and not pull request. While we believed the pull requests might explain away the good behaviour of the merge commits, this does not seem to be the case. Only 35 of the selected breaking merge commits across all the 348 projects are marked as a pull request. Filtering these out does not change the result for the other merge commits.

> A breaking merge commit happens less often than a breaking regular commit in projects with a CI pipeline who maintain a 34% success rate.

This could be explained through our commit selection. We pick *breaking* commits, i.e., commits for which the build not just fails, but the build of the parent commit(s) also passes. Regular commits will usually be something completely new to the source code. Merge commits on the other hand will combine two passing branches. The only way for a merge commit to break the build is to have the source code from both branches interact in an unexpected way. Table II shows half of the projects deal with such an error once every 43 merge commits. For a quarter of the projects this occurs at least once every 17 merge commits.

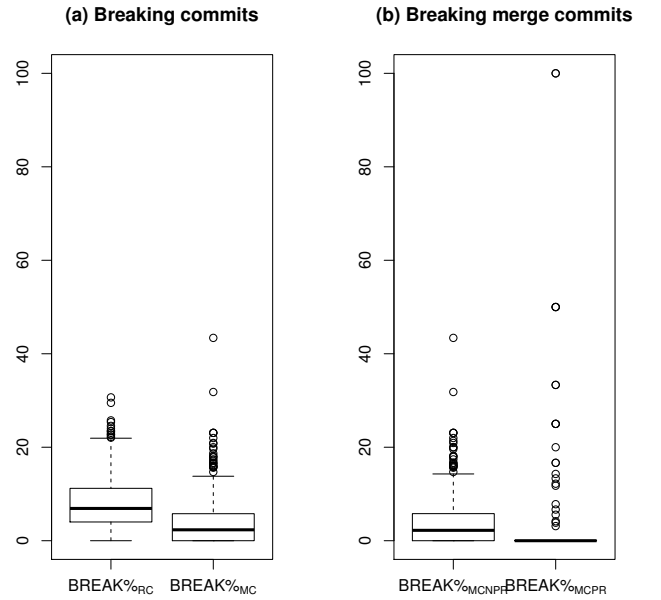

Fig. 2. A comparison over all projects of the BREAK%. (a) splits up breaking commits by regular commits and merge commits. (b) splits up the breaking merge commits by pull request.

Threats to Validity. Merge commits are but one form of code integration. The manual application of a patch or a Git "rebase" would not show up in the Git history. Rebasing rewrites the history of a project to pretend commits were made sequentially rather than in parallel over different branches. This study does not consider these forms of code integration.

TravisTorrent contains projects that adhere to the GitHub workflow. The projects need have forks and pull requests. This limits our analysis to this type of projects.

B. Effort to Fix Conflicts

We start out with 16413 breaking commits (14430 regular, 1983 merge) from the 348 projects after removal of outlier projects in Section IV-A. For 8453 (7664 regular, 789 merge) of the breaking commits a fixing commit is found. The merge commits are spread out over 203 projects.

The $\overline{NBTF}$ metric shows 87.19% of projects usually repair a breaking merge commit on the next build. Performing the same analysis on the number of commits gives similar results: 70.94% of projects usually require just one commit.

Fig. 3 depicts the $\overline{LINES}$ (quartiles at 3.25, 9, and 36). The inset zooms in on the left part of the graph. The inset still shows 87.68% of the projects. Half of the projects repair breaking merge commits usually with up to nine lines.

Table III summarises the $\overline{TTF}$ metric. $\overline{TTF}$ shows 66.5% of projects usually fix a breaking merge commit the same day. Within a week, 94.09% of projects have fixed a breaking merge commit.

> In most projects a breaking merge commit is usually fixed with one build on the same day by changing less than ten lines of code.

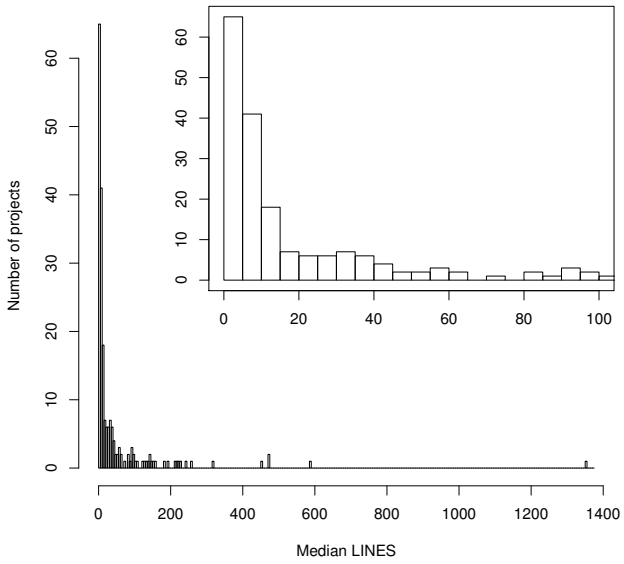

Fig. 3. $\overline{LINES}$ for every project. Despite the long tail, for 75% projects $\overline{LINES}$ is less than 36. The inset zooms in on the lower end of the graph. The inset still shows 88% of projects.

TABLE III
AN OVERVIEW OF THE $\overline{TTF}$ METRIC. IT SHOWS 66.5% OF PROJECTS USUALLY FIX A BREAKING BUILD OF A MERGE COMMIT WITHIN A DAY.

	Usually fixed
the same day	66.50%
the next day	14.29%
the same week	13.30%
the same month	4.93%
more than a month	0.99%

Threats to Validity. Our method for identifying the fixing commit relies on finding the next builds in TravisTorrent. However, TravisTorrent does not provide this information in the case of merge commits. A fixing commit will not be found if a breaking commit is fixed by a merge commit or a merge occurs between the breaking and the fixing commit.

Once left with 789 fixes of merge commits there is not a lot of data per project. This may skew the results in favour of what happens in those projects with very few data points.

Clearly all metrics are but a proxy for effort. It may take a lot of effort to track down the exact problem of an issue, while still fixing it with but one line of code in one commit. The $\overline{TTF}$ metric used does not necessarily indicate time a developer spent working on fixing the build. The dataset comprises open source projects which are, in general, developed by volunteers on an irregular basis.

C. Source vs Test

Fig. 4 depicts how the metrics defined in Section III-C are spread out across all projects. Fig. 4 shows most breaking merge commits are fixed by changes to either exclusively the source code or to both source and test code.

Breaking merge commits are fixed by changes to the source code.

Threats to Validity. This analysis is done for those breaking merge commits for which a fix was found. Only 789 such cases were found. There is not a lot of data per project. This may skew the results in favour of what happens in those projects with very few data points.

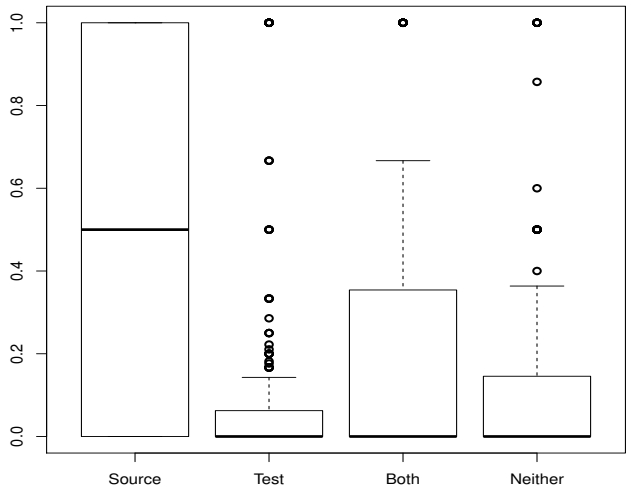

Fig. 4. SRC, $TEST$, $BOTH$, and $NONE$ metrics. Breaking merge commits in the majority of projects are usually repaired by changes to the source code.

V. CONCLUSION

Using data from GitHub and Travis CI, we analysed breaking commits: commits for which the build fails and the build of its parent commit(s) passed. We found breaking merge commits occur less often than breaking regular commits. Breaking merge commits are repaired with relatively little effort. Repairing is often done the same day and with just one build. Less than ten lines of code need to be changed to repair a breaking merge commit. Most of the changes are done in the source code, as opposed to test code or other places.

Given their observed prevalence, we recommend further research on tools that warn developers early about potential semantic merge conflicts. Semantic conflicts are more subtle than textual conflicts and may otherwise go undetected until all tests are run or a user encounters its effects.

NOTES

Ward Muylaert is an SB PhD fellow at FWO, project number 1S64317N. A replication package for this study is available via https://soft.vub.ac.be/~wmuylaer/publications.

REFERENCES

[1] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Int. Conf. on Mining Software Repositories (MSR)*, 2017.

[2] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, 2011.

[3] G. Gousios, "The GHTorrent dataset and tool suite," in *Int. Conf. on Mining Software Repositories (MSR)*, 2013.

[4] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining github," *Empirical Software Engineering*, 2015.

[5] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, May 2002.