An Exploratory Study Into the Prevalence of Botched Code Integrations

Ward Muylaert
wmuylaer@vub.ac.be

Wolfgang De Meuter
wdmeuter@vub.ac.be

Coen De Roover
cderoove@vub.ac.be

Software Languages Lab
Vrije Universiteit Brussel
Belgium

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 delivery aims to detect these and other run-time errors through an extensive pipeline of successive tests. Travis CI is a continuous delivery system that is free to use for open-source projects on GitHub. Of interest to researchers is the fact that Travis CI makes the outcome of each stage in the continuous delivery pipeline available through its API. At the seminar, we will present the initial results of an exploratory study on the prevalence of integration errors. We have linked GitHub's information about integration efforts with Travis CI's information about the integration outcome to this end.

1 Introduction

Teams developing code will often have to integrate their code with one another. In most cases, this is managed by means of version control software ("VCS"). VCS is a type of software that keeps track of the changes made to the source code throughout the history of a project. Git in particular, a distributed version control system, has seen growth in recent years (Skerrett 2014). Git encourages a workflow in which each developer works on their own version of the code, their own branch, allowing the different developers in a team to focus on their own work without interrupting others. After some work has been done, different versions of the code may be merged together. The textual changes made in one branch (the source branch) are applied to the version of the code in the other branch (the target branch). However, the changes made to either branch since they diverged may be incompatible with one another.

1.1 Types of Integration Conflicts

We consider two different types of incompatibilities, following the classification used in Brun et al. (2011).

The first incompatibility is the possibility of textual conflicts. This is also the more straightforward one of the two. It occurs whenever different changes are made to the same line of code in each branch. When merging, Git has no way of knowing which of the two versions should be kept and a textual conflict occurs. Git refers

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

to this as a "merge conflict" and does not perform the actual merging commit yet. Instead, it requires manual developer intervention in order to get the textual conflict resolved.

The second incompatibility is a semantic conflict. This conflict is more subtle and may actually get missed completely until a much later point in time. The issue occurs when the combination of the changes in both branches makes the program not build or, if it does build successfully, makes it behave in a way that was not intended. This can, for example, occur if code relying on a function `f` is added in one branch, while the other branch has changed the functionality of the function `f`. The code relying on the function `f` may now no longer work as previously intended. This may not be noticed until that particular part of the program is tested.

To illustrate, consider the situation depicted in Figure 1. A priori, there is a single branch b_0. Alice, a developer, decides to create a new branch, b_1, branching off from b_0. She then performs some changes on her branch. Meanwhile, Bob, another developer, performs some changes in the original branch b_0. After both developers have performed their changes, Alice decides to merge her branch back into b_0.

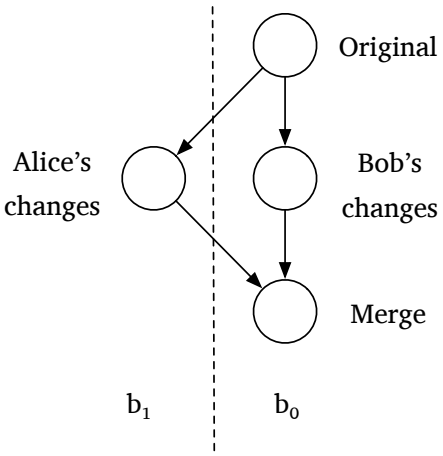

Figure 1: Two developers can make changes in different branches coming from the same ancestor. In most cases, these branches are merged together again at a later point in time.

Listing 1 depicts the code in branch b_0 before any changes were made.

```
1 function foo(x) {
2     if (x < 0) {
3         return -x;
4     }
5     return x;
6 }
```

Listing 1: Version of the code in branch b_0 before any changes are made.

As an example of a textual conflict, suppose Alice changes line 5 in her branch b_1 so that it now returns 0. Bob, meanwhile, also changes that same line, making it return 1. When Alice merges her code back into branch b_0, the version control software does not know which is the "correct" return value: should the merged version of `foo` return 0 or 1? Rather than trying to guess, the version control software forces Alice to make a decision herself.

For a semantic conflict, consider once more Listing 1, the code before either Alice or Bob have made any changes. In this situation, suppose Alice writes some code in her branch b_1 which calls the function `foo`. Bob on the other hand realises `foo` is poorly named and renames the function to `abs`, for absolute value. When Alice now merges her code back into branch b_0, the version control software will not notice anything wrong and merge without errors. However, the resulting code no longer works. Alice's calls to `foo` would fail as there is no longer a function by that name in this version of the code. This semantic conflict may not surface until the code is built, run, or tested.

1.2 Previous Studies on Integration Conflicts

We believe current code integration techniques and tools are insufficient, leading too often to the occurrence of the aforementioned conflicts. This is not limited to just the merging of branches as described above, but also other code integrations such as porting patches or cherry picking commits (i.e., integrating only some commits from a different branch).

Brun et al. (2011) conducted a study into nine open source projects that use Git. It found that of the 3,562 merges that occurred over all nine projects combined, 564 (15.8%) of them caused a textual conflict. In other words, almost one merge in six required immediate developer intervention due to the version control software not knowing how to handle overlapping changes.

Furthermore, three of those nine projects could also be analysed for semantic conflicts, which Brun et al. (2011) refer to as higher-order conflicts. For these three projects, the researchers encountered build errors after 0.1%, 4%, and 10% of the projects' respective merges. They also found 4%, 28%, and 3% test failures. Note that a failure implies the success of the steps before it: the occurrence of a test failure means there was no textual conflict and the build succeeded as well. As we consider both build and test failures a semantic conflict, this totals to 4%, 31%, and 14% of the respective merges in those three projects giving rise to a semantic conflict.

A more recent study (Guarnera 2015) looked at 23 projects to find how often textual conflicts occur when merging. The quartiles of their results are 1.8%, 4.8%, and 11.2%. In other words, half of the analysed projects had a textual conflict in less than one in 20 merges. However, four of the projects had a textual conflict at least once per six merges, indicating the problem is indeed present, if less pronounced than as found by Brun et al. (2011).

In a similar vein, Perry, Siy and Votta (2001) found a correlation between the number of people working on a file per day and the number of faults encountered. They made sure to control for other likely culprits. Caveat with this study is that it concerns an old dataset, from the 1990s, the available tools and practices have changed since then.

We decided to take these studies a step further by looking at a larger and more recent dataset. Specifically, our focus is on semantic conflicts.

2 An Exploratory Study

2.1 Research Method: Mining GitHub and Travis CI

To look for issues due to code integration on a larger scale, we propose a study using GitHub[1] and Travis CI[2].

GitHub is a popular source code repository host with millions of repositories available to the public. GitHub provides an API and an effort has been made to provide GitHub's data in an accessible format for research. This comes in the form of GHTorrent (Gousios and Spinellis 2012) which aims to provide an offline means of working with the data GitHub provides. The GHTorrent project provides a downloadable data dump as well an online queryable interface on their website[3].

Travis CI is a continuous delivery service and provides free continuous delivery for open source projects on GitHub. Continuous delivery (Humble and Farley 2010) is an increasingly popular practice in software engineering aimed at alleviating the bugs that may occur due to teams developing software together independently. The practice of continuous delivery advocates an automatic build of the code upon each commit to the version control system. If the build succeeds, it also requires the execution of a test suite. Should either the building or the test suite fail, the developer who committed the changes is expected to immediately fix the bugs that caused the failure. In case a build passes through the entire so called pipeline of tests, it is deemed ready for release. Depending on the way the project is set up, this release may happen automatically and immediately or it may require a human "push of the button". Continuous delivery expects everything relating to the project to be placed under version control: code and configuration files, but also, for example, the build artefacts. If continuous delivery is followed properly, it enables a development team to rely on having a working version of their code at any given time.

A recent effort was made for Travis CI to create a project similar in function to GHTorrent. This project is called TravisTorrent[4] (Beller, Gousios and Zaidman 2016) and it too provides a downloadable data dump and

[1] `https://github.com/`
[2] `https://travis-ci.org/`
[3] `http://ghtorrent.org/`
[4] `http://travistorrent.testroots.org/`

an online interface. We were intent on using this, but encountered some trouble working with TravisTorrent as provided. The data we are interested in does not yet, at the time of writing, seem to be a part of it. Instead, we created our own dataset by calling the API provided by Travis CI[5]. For this dataset, we recollected information for the projects included by Beller, Gousios and Zaidman (2016) in TravisTorrent. Ideally this effort would be skipped in the future in favour of using TravisTorrent as is.

For our collection of continuous delivery information, we mirror the build, commit, and repository entities from the Travis CI API in three corresponding MySQL tables. The relation between these three entities is as follows. One repository has many commits, one commit can have multiple builds. Builds are also directly connected to a repository, this information is also part of their schema. The API is called for the 1,300 projects from TravisTorrent and the information of the mentioned entities is saved to the MySQL database. This information can then be combined with the information in GHTorrent. The connection from our data to GHTorrent is made using the SHA-1 hash of the commit.

The information for some of the projects in TravisTorrent has since been removed from Travis CI. We decided to simply skip these projects and are left with 1,243 projects and a total of 1,065,364 distinct commits in our dataset of Travis CI. Of these commits, 597,835 are also present in the version of GHTorrent we downloaded (i.e., the version released on 4 May 2016). Each of the projects has at least 100 builds in Travis CI.

Using GitHub and Travis CI provides a means to revisit the study by Brun et al. (2011) on a significantly larger scale as well as conducting further studies relating to code integration.

In our presentation, we will aim to answer the following research questions.

RQ1 How often does code integration lead to semantic conflicts?

RQ2 How much effort is needed to fix semantic conflicts after a code integration?

RQ3 How long does it take to fix semantic conflicts after a code integration?

2.2 Results

In this extended abstract, we try to provide an answer to Research Question one. To do so, we use the parent information as present in GHTorrent. GHTorrent provides a table linking a commit to its parents which we can use to count the amount of parents a commit has. If there is more than one parent for a commit, then that commit is a merge commit. Using this definition, we find 156,177 builds of merge commits across 1,227 projects. Most of these merges are the regular merge between two branches. Five of them are a merge between three branches and one is between five branches, in Git these are referred to as an octopus merge. The merge commits are, however, not equally distributed amongst the 1,227 projects. We filter out projects with less than 50 builds of merge commits and are left with 586 projects.

For these merge commits, we looked at the proportion of semantic conflicts that occurred per project. Listing the percentage of semantic conflicts for every project is clearly not feasible here, instead we provide the boxplot in Figure 2 and the following summary. The median lies at 15.2%, indicating half of the projects have a semantic conflict in almost one in six merges, possibly more often. The first quartile is at 6.7%, the third at 28.9%. In other words, one in four projects deal with a semantic conflict almost every third merge.

3 Conclusion

It must be stressed this is but a preliminary result, we intend to present the complete results at the seminar.

We will revisit the above answer to Research Question one by looking at commits with a description that may indicate a code integration action while not being an explicit merge. After all, code integration is not limited to just merge commits. For example, a patch may be applied manually or the developer can elect to rebase changes, a method that effectively hides the fact that different branches existed in the Git history. We want to analyse the commit message and look for words that may indicate such actions.

To answer the second Research Question, we intend to look at metrics like the amount of commits, people, changed lines of code, or changed files. Research Question three can be analysed by looking at the time between the failure and fix.

[5] `https://docs.travis-ci.com/api`

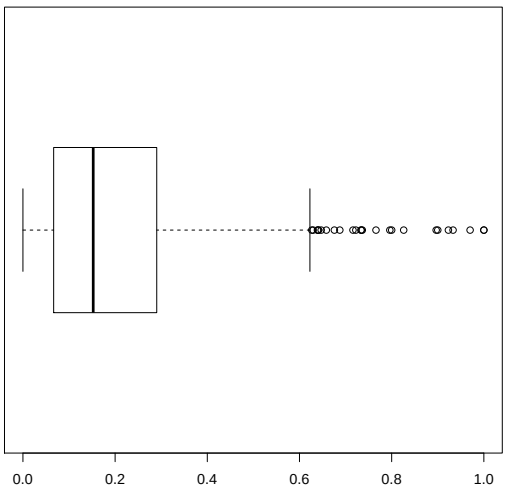

Figure 2: Boxplot of the proportion of semantic conflicts after a merge commit in 586 projects using Travis CI and GitHub. Each project has at least 50 builds of a merge commit. Quartiles are at 6.7%, 15.2%, and 28.9%.

References

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

Brun, Yuriy, Reid Holmes, Michael D. Ernst and David Notkin (2011). "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)*.

Gousios, Georgios and Diomidis Spinellis (2012). "GHTorrent: Github's Data from a Firehose". In: *Mining Software Repositories (MSR)*.

Guarnera, Drew T. (2015). "Merge Commit Contributions in Git Repositories". MA thesis. University of Akron.

Humble, Jez and David Farley (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional.

Perry, Dewayne E., Harvey P. Siy and Lawrence G. Votta (2001). "Parallel Changes in Large-Scale Software Development: An Observational Case Study". In: *ACM Transactions on Software Engineering and Methodology*.

Skerrett, Ian (2014). *Eclipse Community Survey 2014 Results*. URL: `https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/`.