Automated Categorisation of Breaking Merge Commits
Work in Progress

Ward Muylaert
ward.muylaert@vub.be

Coen De Roover
coen.de.roover@vub.be

Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium

Abstract

Integrating code from different sources can be an error-prone and effort-intensive process. While an integration may appear sound, unexpected errors may still surface at run time. What exactly constitutes these errors is not always clear, nor is it known which are most common. We want to create a hierarchic categorisation of errors that break merge commits. We will do this in two phases. First, a manual categorisation to enable the definition of categories based on real world examples. Second, a declarative specification of every category to automate categorisation and enable performing an empirical study.

1 Introduction

Version control systems enable developers to work on different versions of source code independently and in parallel. When different versions of the source code are merged together again, conflicts may occur. These conflicts can be split in different categories (Mens 2002). In its most basic form, these conflicts may be of a textual nature. Textual conflicts are caught by the version control software and demand developer intervention in order for the merging to complete. More subtle are syntactic and semantic conflicts. These may not be discovered until the code is built or run, respectively, if at all.

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017 (sattose.org). 07-09 June 2017, Madrid, Spain.

We are interested in the cause of these more subtle conflicts. To investigate it, we look at *breaking* merge commits specifically. We previously defined a commit to be breaking if its compilation or tests failed, but no such issues were present in the parent commit(s) (Muylaert and De Roover 2017).

Previous work looked for causes and likelihood of failure by considering the context: the number of modified subsystems or results of previous build (Hassan and Zhang 2006) or the number of simultaneous developers contributing to the same part of the code, the type of development performed, and stakeholder roles (Kerzazi, Khomh and Adams 2014). Rausch, Hummer, Leitner and Schulte (2017) categorises failures by their cause, such as test failures or compile errors. However, none of these studies focuses on integration errors.

In this work in progress, we categorise breaking merge commits by looking at what was done to fix the conflict. We do this in two phases. At first, we manually analyse breaking merge commits and their fixes in order to define a category hierarchy. The changes between a breaking merge commit and its fixing commit are analysed and assigned to a category. Second, we construct declarative specifications for the different categories. The declarative specifications are written in Qwalkeko (Stevens and De Roover 2014). Using these declarative specification will enable an automatic analysis into which categories occur how often.

2 Dataset

Selecting our dataset happens in two steps. First, projects are selected from the TravisTorrent dataset. Second, breaking merge commits and their fixing commits are selected from these projects.

We work with a subset of projects of the TravisTorrent dataset (Beller, Gousios and Zaidman 2017). This subset is based on earlier work (Muylaert and

De Roover 2017) and on a Java dependency we have in this work. TravisTorrent provides build information gathered from Travis CI, a continuous integration service. TravisTorrent contains information on 1300 Java and Ruby projects. These projects meet the following criteria defined by Kalliamvakou et al. (2015): 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. We previously selected a subset containing 348 projects (Muylaert and De Roover 2017). Each project in our subset fulfils the following criteria: (1) a "sufficient" success rate present across all the builds and (2) build information present on at least 50 merge commits and all their parents. The original dataset contains both Ruby and Java projects. Our automated analysis has a hard dependency on Java. We removed Ruby projects out of the 348 projects. This was done automatically by using the language information present in TravisTorrent. A project is kept in the dataset if the majority of its commits are classified as Java language.

In the selected projects, we identified breaking merge commits: merge commits whose build in Travis CI failed, but for which the build of both[1] parents succeeded. For breaking merge commits, we also identified the fixing commit. The fixing commit is the first commit following the breaking merge commit for which the build passes again. "Following" here is defined using the next build information present in TravisTorrent. Using this method, we find 245 breaking merge commits and their fixing commits. These 245 are analysed manually as described in section 3.

3 Manual Categorisation

Prior work in categorising code integrations focused on what went wrong in porting errors (Ray, Kim, Person and Rungta 2013). Ray et al. manually identified failures by searching for mentions of porting errors in commit messages. This analysis was performed on the FreeBSD and Linux code bases, which are written in C. Ray et al. identified five categories of porting errors: inconsistent control flow, inconsistent renaming of identifiers, inconsistent renaming of related identifiers, inconsistent data flow, and other.

We want to categorise breaking merge commits. To do so, we look at what was done to fix the failing build. In other words, we look at the difference between the breaking merge commit and its fixing commit. We start by manually analysing these differences and looking for patterns. Our aim is to create a category hierarchy. A hierarchy enables us to be as precise as possible in identifying and classifying different commits. It also gives the option to look at things on a higher level by combining data. The category hierarchy is a work in progress, an example of what we have in mind can be found in Figure 1. A commit may be categorised in more than one category.

Consider, for example, the `inconsistent_type` category. An actual occurrence of this category is given here:

```
216
217 - List<State> states = new ArrayList<>();
217 + List<UUID> newObjectIds = new ArrayList<>();
218   if (!ObjectUtils.isBlank(newStorageItem)) {
```

As implied by the name, this category considers errors in which the type is inconsistent with changes made elsewhere. In this particular example, the `State` class was no longer the correct class for the code that followed. Code in another branch had already changed in light of this, but this part of the code still needed to be updated.

4 Automated Categorisation

4.1 Declarative Specification

Once the category hierarchy has been properly defined, we want to perform an analysis that tells us how often each of the categories appear. To do so, we write a declarative specification of the different categories. This is done using the Qwalkeko tool (Stevens and De Roover 2014). Qwalkeko enables automatically looking at differences in source code and presents the changes as Insert, Update, Move, or Delete actions on the AST. It is over these actions and their contents that the declarative specification matches.

Consider again the `inconsistent_type` mentioned in the previous section. Declarative code in Qwalkeko to find such occurrences looks as follows. Given a `change`, this piece of code will validate that the change is indeed to the type of a field.

```
(defn change|field-type [change]
  (logic/fresh [?before ?type ?declaration]
    ; Look for a Qwalkeko update
    (changes/change|update change)
    ; Check before the change
    (changes/change-leftparent change ?before)
    ; Is it a change to the type?
    (jdt/ast-parent ?before ?type)
    (jdt/ast :SimpleType ?type)
    ; And is it of a field
    (jdt/ast-parent ?type ?declaration)
    (jdt/ast :FieldDeclaration ?declaration)))
```

The comments in the code describe the details, but three parts can be distinguished:

[1] A commit may be a merge of more than two parent commits. Only one such case was present in our 348 selected projects. We do not consider this case.

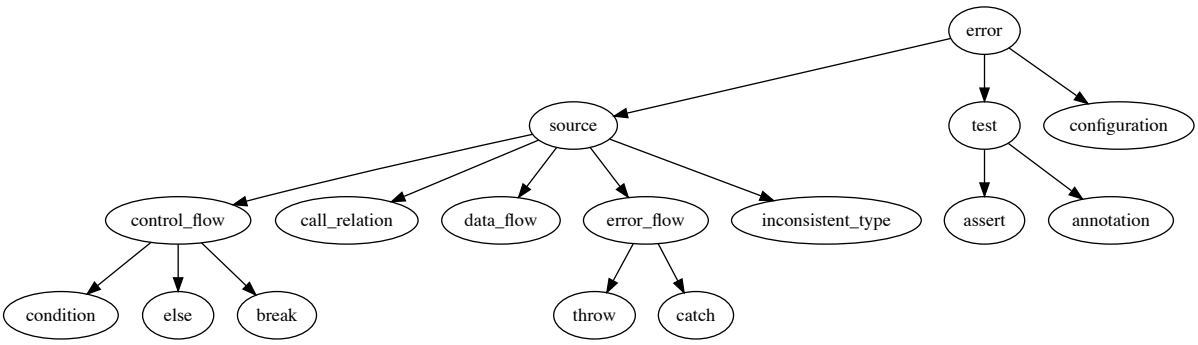

Figure 1: Example of a category hierarchy of code integration errors.

1. Match the action (i.e., Insert, Update, Move, or Delete).
2. Match the before or after of the action.
3. Match the AST in the before or after of the action.

4.2 Automated Analysis

Once the declarative specification is complete for all the different categories we define, we will use it to automatically categorise breaking merge commits in all Java projects of the TravisTorrent dataset, not just those of the 245 commits we looked at manually.

Acknowledgements

Ward Muylaert is an SB PhD fellow at FWO, project number 1S64317N.

References

Beller, Moritz, Georgios Gousios and Andy Zaidman (2017). 'TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration'. In: *International Conference on Mining Software Repositories (MSR)*.

Hassan, Ahmed E. and Ken Zhang (2006). 'Using Decision Trees to Predict the Certification Result of a Build'. In: *International Conference on Automated Software Engineering (ASE)*.

Kalliamvakou, Eirini, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German and Daniela Damian (2015). 'An in-depth study of the promises and perils of mining GitHub'. In: *Empirical Software Engineering*.

Kerzazi, Noureddine, Foutse Khomh and Bram Adams (2014). 'Why do Automated Builds Break? An Empirical Study'. In: *International Conference on Software Maintenance and Evolution (ICSME)*.

Mens, Tom (2002). 'A State-of-the-Art Survey on Software Merging'. In: *IEEE Transactions on Software Engineering*.

Muylaert, Ward and Coen De Roover (2017). 'Prevalence of Botched Code Integrations'. In: *International Conference on Mining Software Repositories (MSR)*.

Rausch, Thomas, Waldemar Hummer, Philipp Leitner and Stefan Schulte (2017). 'An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software'. In: *International Conference on Mining Software Repositories (MSR)*.

Ray, Baishakhi, Miryung Kim, Suzette Person and Neha Rungta (2013). 'Detecting and Characterizing Semantic Inconsistencies in Ported Code'. In: *International Conference on Automated Software Engineering (ASE)*.

Stevens, Reinout and Coen De Roover (2014). 'Querying the History of Software Projects using QwalKeko'. In: *International Conference on Software Maintenance and Evolution (ICSME)*.