Artificial Intelligence Based Software Engineering

Project Report

Continuous Test Generation on Pull Request

Group 5
Mathias Sixten
Yixuan Huang
David Budischek
Gaspard Murat
Gustav Janér

December 14, 2018

Abstract

-

1 Introduction

Over the past decades software release cycles have shortened repeatedly. Two decades ago it was common place to release software and then maybe follow up with a patch (in the form of a physical hard disk) a few years down the line. From that we went to pushing out monthly, or even weekly, security updates. In recent years though, it became common place to not bundle releases anymore. Instead every single feature/bug/typo fix is published instantaneously and pushed to every user. With Continuous Delivery come a plethora of challenges. One of which is testing. In a traditional release cycle there is enough time to hand over the release candidate to QA and give them time to verify functionality. But this won't do nowadays. Instead we rely on automated tests to quickly verify functionality. In practice, tests can never guarantee a bugfree product, instead they can only try and cover as much of the code as possible.

In addition to manually writing tests there are advanced unit test generation tools available (such as EvoSuite or Pex). While these tools require no manual input, they do require a lot of computation time. In their 2014 paper [1] the authors of EvoSuite propose a solution to this issue, Continuous Test Generation. Instead of a greenfield test generation for each new iteration, they leverage previous results to speed up the test generation process. In their research this is done by smartly allocating resources to the classes that changed or are not yet fully covered.

When it comes to integrating CTG in the CI/CD workflow commonly used in modern Software projects there is little literature available. In a case study done as part of the SSBSE'15 challenge [2] we can clearly see that CTG performs well enough to warrant implementation. Due to this we implemented an out of the box, platform agnostic tool to easily add CTG to an existing project without interfering with the infrastructure that has previously been set up.

2 GitHub repositories

`https://github.com/Budischek/test-generation-on-pr` - Project Repository

`https://github.com/Budischek/CS454` - Repository under test

3 Related work

3.1 Continuous Test Generation on Guava

For the [2] research paper, the authors used EvoSuite for continuous test generation on the Guava library for the SSBSE'15 challange. The Guava library contains more than 300 classes and so generating tests for the library requires a relatively long time. The purpose of the paper was to improve test case generation for the Guava library through implementing continuous test generation to build test cases incrementally over time. Test suite augmentation and time-budget allocation was used to focus the test generation on what had been modified. In the paper, continuous test generation was successful implemented to improve the test generation time.

4 Seeding

Since most pull requests won't modify and affect an entire program, many of the test cases of the previous test suite could still be useful to test the new modified version of the program. Instead of creating a new tests from scratch to test every pull request, seeding can be leveraged to increase test generation efficiency [1].

4.1 Identify test cases for seeding

To create efficient seeding for EvoSuite, useful test cases from the previous test suite have to be identified. Useful test cases for seeding are the test cases that reaches the modified parts of a pull request.

To identify useful test cases to be used for seeding in our project, each test case was executed separately on the CUTs. JaCoCo was used to collect code coverage information of each of the executed test cases. JaCoCo instruments the byte code of a CUT and inserts probes. For every probe that a test case reaches, the value of the probe is changed from false to true to indicate that the probe was activated and covered by the test case.

The probe activation of every test case is stored and compared to the probe activation of the previous test case execution before the pull request. If the probe activation of a specific test case is differing from the previous execution, that test case reaches code that was modified by the pull request. A test case that is identified as reaching modified code, is stored and can later be used for seeding for the test case generation.

For a program P, test cases TC, and a modified version P' of P: to identify useful test cases of TC for seeding, all test cases TC are executed on P and then P'. Any test case TC that has differing probe activation of P and P', reaches modified parts of P'.

5 Modularity

Modularization plays a big role in developing a platform agnostic tool. With our out of the box CTG solution we want to be able to easily integrate it with different version control systems, persistent data storages or even test generation tools. For that reason we designed a set of interfaces that keep coupling between components minimal and allows us to quickly develop new modules to extend functionality. The system consists of four modules:

- Source Code Repository
- Persistence
- CTG Algorithm
- Test Generation

Out of these modules the CTG algorithm can be considered the heart of our program. It connects to the three other modules via the following simple interfaces.

- AbstractGitService
  - getRepositoryPath(String repoUrl)
    Clones the repository to local storage and returns the file path
  - reportResults(int id, String msg)
    This is used to report the final results of the CTG process (e.g. new Branch coverage). In the example of a Github Pull request the id is provided by the webhook and the results are stored as a comment on the Pull request page
- AbstractPersistenceController
  Provides two seperate persistence functionalities. Firstly to store a complete testsuite and secondly a Key Value store for the CTG algorithm to persist meta data (e.g. branch coverage for each individual test or a reference to the most up to date test suite). For now the KV store simply handles Strings, but this functionality should be extended to AbstractKey/AbstractValue types for more flexibility.
  - persistTestSuite(String identifier, TestSuite testSuite)
  - getTestSuite(String identifier)
  - storeMetadata(String key, String value)
  - getMetadata(String key)
- AbstractTestGenerator
  - generateTestSuite(TestGenerationRequest request)
- AbstractCTGStrategy

- newPRTrigger(String identifier)
  This is triggered by the webhook whenever new source code is available. The identifier depends on the VCS + Platform. In the case of Github we can simply use the pull request id.

6 Development Environment

The main consideration while planning our development environment and CI/CD pipeline was ease of use. As the majority of our team has little to no experience with larger scale group projects it is important that the time spent on merging conflicts, build errors, etc. is minimized. This was the main reason we even opted to spend the time setting up automatic builds and deployment, as they would guarantee a consistent and workable state in the repository and prevent team members from merging code that has obvious errors (e.g. doesn't build).

6.1 Build Tool

The obvious choices for build tools were Maven and Gradle. After some consideration, and keeping in mind that only one team member has actual experience working with these tools we decided to go with Gradle, as its simple, groovy definition language can easily be modified without any further knowledge about how gradle works.

6.2 CI/CD

We set up a Google Cloud (GC) Build server to monitor our repository and automatically test and build every new commit. To prevent the argument "but it works on my machine" our project is completely dockerized, allowing us to quickly set up a new container with a single gradle command. The build server also automatically stores the finished container in a GC Container Registry and deploys to a virtual machine running 24/7 on GC Computation Engine. While this runs into the issue that multiple commits to different branches will lead to confusion over the currently deployed version, we did not run into any issues as each developer worked on a different schedule.

While there was a lot of initial time investment into getting the build server up and running it paid off, as can be seen by the multitude of commits with messages like "Fixed broken build" that happened right after pushing new changes.

7 Conclusion

One of the main issues with CTG is the up front time investment to get it up and running in an existing CD pipeline. With our project we present a simple way of adding CTG to an existing project without making any changes directly to the repository under test. While we only provide a few default modules the clean and simple interfaces between each module make it easy to develop your own and simply plug them into our solution.

References

[1] J. Campos, A. Arcuri, G. Fraser, and R. Abreu, "Continuous test generation: Enhancing continuous integration with automated test generation," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, New York, NY, USA: ACM, 2014, pp. 55–66, ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2643002. [Online]. Available: http://doi.acm.org/10.1145/2642937.2643002 (visited on 10/29/2018).

[2] J. Campos, G. Fraser, A. Arcuri, and R. Abreu, "Continuous test generation on guava," in *Search-Based Software Engineering*, M. Barros and Y. Labiche, Eds., vol. 9275, Cham: Springer International Publishing, 2015, pp. 228–234, ISBN: 978-3-319-22182-3 978-3-319-22183-0. DOI: 10.1007/978-3-319-22183-0_16. [Online]. Available: http://link.springer.com/10.1007/978-3-319-22183-0_16 (visited on 12/03/2018).