Program Verification in Continuous Integration Workflows

Bachelor's Thesis Project Description

Johannes Gasser

Supervisors: Linard Arquint, João C. Pereira, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zurich

17th September 2021

1 Background

Software engineers use testing to assess the reliability of their programs. Ideally, new tests are introduced to a code base whenever new features are introduced or new bugs are identified, and are executed automatically on all contributions to the project by using Continuous Integration. Unfortunately, tests are not enough to ensure the desired software reliability, since tests only explore a small fraction of possible program behaviours, which is not sufficient to show the absence of bugs.

Formal verification techniques provide stronger correctness guarantees. Programmers are expected to annotate their code with specifications, such as pre- and postconditions. These annotations describe the correct behaviour of the program. Then, program verifiers such as Gobra [1], a verifier for Go programs, automatically check whether the code meets its given specifications.

Despite the guarantees provided by program verifiers, they are still not widely used in automated development pipelines. Nonetheless, initial industrial experiments using related techniques [2] have shown promising results. Motivated by these results, we plan to extend Gobra to be easily integrated with Continuous Integration workflows. To achieve this, we will develop a reusable Continuous Integration workflow, smart caching strategies to reduce the workflow's runtime and verification reports, which summarize the verification results and provide helpful statistics to developers and project managers. We plan to evaluate our solution in the VerifiedSCION repository, containing a verified implementation of the SCION protocol [3].

1.1 Gobra

Gobra is a modular verifier for heap-manipulating, concurrent Go programs. It takes a Go program annotated with assertions, such as pre- and postconditions or loop invariants, and encodes the program into the intermediate verification language Viper [4]. Then, it applies an existing SMT-based verifier to check whether the Viper program behaves according to its specification. If not, Viper reports an error, which is then back-translated to an error in terms of the original Go code.

1.2 VerifiedSCION

VerifiedSCION is a project with the goal to verify the network protocol SCION from the high-level design all the way down to the implementation. SCION is clean-slate Internet architecture that provides secure routing and forwarding, alongside a number of other desirable properties. Currently, Gobra is used to verify parts of the flagship implementation of the SCION protocol, which is written in Go.

1.3 Continuous Integration

Continuous Integration (CI) describes the process in software engineering, in which developers simultaneously and frequently add to a shared repository. To ensure quality of a repository, automatic pipelines are run on contributions as part of the CI process to run builds or execute tests and report the results.

2 Core Goals

The goal of this project is to develop a fast CI workflow that is easy to integrate in existing projects, and also delivers a summary of verification results in the end. To achieve this, we established five core goals.

2.1 GitHub Action

The first task will be to build a GitHub Action [5], that uses Gobra to verify a project or parts of it during a CI workflow. We focus on GitHub Actions since they are natively supported by GitHub, and GitHub is one of the largest platforms where software projects, including the SCION and VerifiedSCION projects, are hosted. This Action should be project-agnostic, meaning it does not rely on a fixed project structure. Besides, it should provide configuration options on a global or package level, including which files and packages should be verified, which Gobra flags should be used, how to handle verification errors and timeouts, resource limits, and the location of signatures for third party libraries.

2.2 Caching

After implementing the GitHub Action, the next goal is to implement caching strategies to avoid re-verifying files when there are changes to the codebase that do not affect those files. Caching makes most sense at the method level, since Gobra is method-modular, which means that each method implementation is verified in isolation. All verification results can be written to a file. Fortunately, there already exists a caching approach for Viper in the ViperServer [6] project, which is already used by the Gobra Visual Studio Code extension [7]. Using ViperServer in our Action will provide us with a working caching implementation, while also allowing us to extend the use of the cached results to the Gobra IDE, as described in the extension goal 3.1. To store the cached files, created during the GitHub Action, for future workflow runs, there exist other GitHub Actions [8], which can be combined with ours in a workflow. A downside of the GitHub cache storage is, that the cache is evicted after 7 days, which could be a problem for smaller projects with few contributions and releases.

2.3 Usability in Existing Projects

One of the goals of this project is to make Gobra easily integratable in the CI of Go projects, with a special focus on preexisting, large codebases. To that end, we plan to extend Gobra to support a trusted annotation for methods. This annotation specifies that a given specification is assumed to hold for the verification and instructs Gobra to ignore the method's body. This helps developers integrating Gobra into their CI workflow, since they don't have to verify a whole package directly but can start to add verification step by step. A possible downside to this approach is that this functionality allows users to introduce new assumptions, which might be wrong. For example, the user could specify a wrong footprint for a method. To analyze which methods are depending on assumptions, we will use call graph analysis. We can then use this to inform a user about the number of methods, which are depending on assumptions, or visualize the call graph, as described in 3.4.

Additionally, Gobra should be able to handle calls to third party libraries. To achieve this, a user can extract the signatures of the methods that are called, write specifications for them and then mark them as trusted. This approach enables Gobra to use these specifications in the verification of the project without the need to verify every third party library used.

2.4 Summary

At the end of our CI workflow, we plan to provide a meaningful summary to the project developers, including the number of methods verified, the number of trusted methods and which methods depend directly or indirectly on trusted methods. Our summary will also report all verification errors produced by Gobra. As a first step, this will be in form of a log output from the GitHub Action. This log paired with a status flag that differentiates between errors, warnings and complete verification makes a reasonable first implementation for a summary that can be added to a GitHub README page.

2.5 Evaluation

To test this implementation on practical code, we will deploy our Github Action in the Verified-SCION repository. This serves as a test to see how well Gobra can handle a large codebase with these improvements. Currently, there exists a simple GitHub workflow, which can be used as a baseline to measure the improvements brought by this project. As a second form of evaluation, we will design a series of test cases to verify our solution on a changing codebase. A possible way to achieve this would be to imitate a commit history of a real repository and observe how long the runtime of our Action is and how well the cache is used.

3 Extension Goals

After we finish the core goals, we plan to tackle other problems such as verification timeouts during pipelines. Furthermore, we want to extend the summary part of the solution to a dashboard.

3.1 Reusing caches

The caching of Gobra could be extended such that the Gobra IDE plugin can make use of the cache files generated by the CI and vice-versa. For this to work the cache has to be updated to contain version information about Gobra, since different Gobra versions might give different verification results, for example after fixing a bug in Gobra. This could speed up pipelines, since edited files would already be verified by the IDE, which means only files depending on edited specification will need to be verified again by the Action. A possible solution to reuse caches, generated by our GitHub Action, would be to commit them directly into the repository after an Action run. This would also solve the issue with the short lifetime of GitHub caches. However, caches larger than a few Megabytes shouldn't be committed to a repository, since this would slow down git processes such as cloning. GitHub generally recommends that the size of repository is less than 1 GB, and strongly recommends that it is less than 5 GB.

3.2 Verification Timeouts

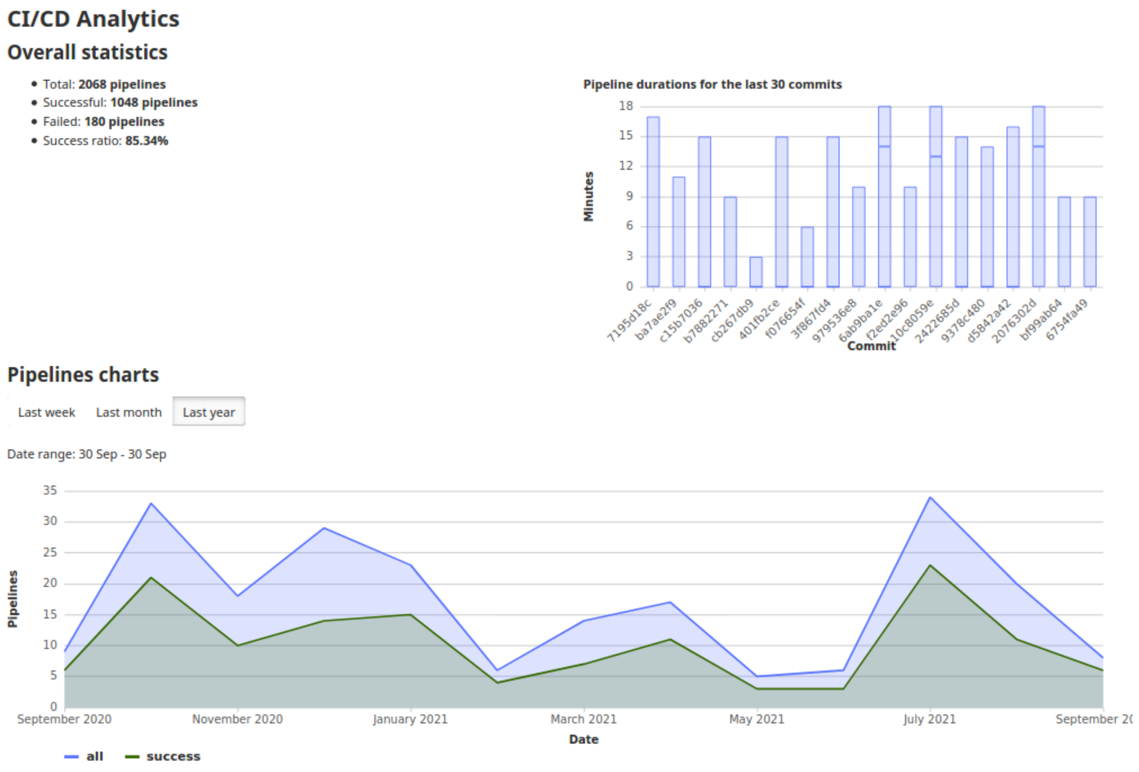
Since the underlying SMT solver can take a very long time, verification timeouts will be a problem in CI workflows. To improve the behaviour in such cases, it is preferable if we can extract results of already verified methods and information on which method's verification timed out. With this information, problematic methods could be fixed or automatically marked as trusted for future runs, such that the next runs of the workflow consume less computational resources and terminate more quickly.

3.3 Supporting Additional Frontends and Technologies

Other Viper frontends such as Prusti [9] would also benefit from an Action such as the one we build for Gobra. One possibility is to evaluate which frontends are ready to be used in CI workflows. For those which are not ready, we want to provide an interface for configurable options and a general approach, such that they can easily integrate their frontend in a GitHub Action. In addition, we plan to generalize our workflow to be easily used in other Git providers and CI frameworks such as Gitlab or Travis CI.

3.4 Extended Summary

The results previously mentioned could be displayed in an interactive dashboard, containing a visualization of the call graph and statistics, such as how many methods have been verified in the current run, for how many runs have particular methods been cached or how often verification timed out for a method. As an inspiration to this goal we could use existing dashboards such as GitLab's CI/CD Analytics.



- Total: **2068 pipelines**
- Successful: **1048 pipelines**
- Failed: **180 pipelines**
- Success ratio: **85.34%**

Figure 1: Example of a Gitlab CI/CD Analytics dashboard.

References

[1] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of go programs," in *Computer Aided Verification (CAV)*, A. Silva and K. R. M. Leino, Eds. Springer International Publishing, 2021, pp. 367–379.

[2] N. Chong, B. Cook, J. Eidelman, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow at amazon web services," *Software: Practice and Experience*, vol. 51, no. 4, pp. 772–797, 2021.

[3] X. Zhang, H. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. Andersen, "Scion:scalability, control, and isolation on next-generation network," in *IEEE Symposiumon Security and Privacy*. IEEE, 2011, pp. 212–227.

[4] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62.

[5] 2021. [Online]. Available: https://github.com/features/actions

[6] 2021. [Online]. Available: https://github.com/viperproject/viperserver

[7] 2021. [Online]. Available: https://github.com/viperproject/gobra-ide

[8] 2021. [Online]. Available: https://docs.github.com/en/actions/advanced-guides/caching-dependencies-to-speed-up-workflows

[9] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification," ETH Zurich, Tech. Rep., 2019.