IDE Support for a Golang Verifier

Bachelor's Thesis Project Description

Silas Walker
Supervised by Felix Wolf & Linard Arquint
under Prof. Dr. Peter Müller
Department of Computer Science, ETH Zurich
Zurich, Switzerland

March 26, 2020

1 Introduction

Go [2] is a compiled and statically typed programming language targeting high-performance networking and multiprocessing. It is used for web applications including Netflix, Uber, and Cloudflare [5, 3, 1].
Bugs in such systems lead to major consequences such as system crashes or corruption of data. The deductive verification tool Gobra was developed by the Programming Methodology Group at ETH Zurich to prove the absence of bugs. In deductive verification, a program is annotated with contracts specifying the functional behavior, and proof guiding annotations. Figure 1 shows an example of a Viper [6] contract formulated as precondition (*requires* $0 < n$) and postcondition (*ensures res* ≤ 0). These annotated programs are verified regarding functional correctness, meaning that if the precondition of a function holds before its invocation, then the postcondition holds afterwards.
In further detail, Gobra translates Go programs with contracts and annotations into the Viper intermediate verification language, which in turn is verified by the Viper backend. In case of an unsuccessful verification attempt, the part of the specification which could not be verified is reported.

Gobra is a new verification tool, which is actively under development. Currently, Gobra is supported as a command-line tool without IDE integration. As explained in the motivation following afterwards, the lack of an

IDE integration results in limited productivity whilst working with the tool. The main focus of this project is to develop a plugin for Microsoft Visual Studio Code [10] allowing the user to verify code with Gobra within the IDE directly.

```
1  method func(n: Int) returns (res: Int)
2    requires 0 < n
3    ensures  res <= 0
4  {
5    res := 42 * n;
6  }
```

Figure 1: Function with contract in Viper

2 Motivation

Currently, the Gobra verifier can be invoked using the command-line only. This leads to the following workflow for verifying a Go program: Firstly, the annotations and contracts are added to the Go program. Afterwards, the command-line tool is separately run every time the programmer wants to verify the program. If the verification does not succeed, the command-line displays the verification error messages containing the identified reason and location. In particular, those errors have to be manually located in the program text. As an additional consequence of this workflow, Gobra lacks the ability to cache any previous verification results.

As opposed to the aforementioned verification workflow, verifying a Go program using IDE integration will be as follows: Gobra is run as a server reacting to verification requests. As before, the specification has to be added to the Go program. An IDE plugin creates verification requests at suitable times and sends them to the server, which in turn verifies the annotated Go programs using Gobra. Afterwards, the results are returned to the IDE plugin, which either displays a success message or highlights the errors in the Gobra program. The error highlighting will be done similarly to Viper IDE [9], as shown in line 3 of Figure 1 for the postcondition (*ensures res* $\leq$ *0*). In addition to the programmer not having to manually locate the verification errors, the Gobra Server can be extended to support load balancing and caching of verification results. We estimate that caching can speed up the verification in various scenarios, especially when only minor code changes are done in between verification runs.

3 Approach

3.1 Core Goals

1. **Basic Gobra IDE Integration:**
   The first goal is to implement a basic IDE integration for Gobra allowing the user to invoke a command in VS Code in order to verify the currently opened Gobra file. The verification errors are displayed in a similar fashion to the Viper plugin. The integration consists of the following two parts:

   1.1. Gobra running as part of a server written in Scala [7], accepting verification requests, invoking the current Gobra implementation, and returning the verification results.

   1.2. VS Code plugin, implemented in TypeScript [8], supporting the core verification features mentioned above and communicating with the server over the Language Server Protocol (LSP) [4]. The LSP is a protocol standardizing the communication between an IDE and a language server providing features for a language.

2. **Basic caching and load balancing at the Viper level:**
   The basic Gobra implementation communicates with a Viper backend verifier via a standardized Scala interface, the *verifier* interface. This interface defines as its main functionality a method taking an AST as input and returning the verification results as output. Neither Gobra nor the currently used backend verifier support any caching or load balancing, thus the basic implementation of the Gobra Server described in the first core goal lacks the support for these features as well. Viper Server, which is part of the IDE integration for Viper, already implements caching and load balancing. Gobra Server can benefit from adopting those features by using Viper Server as a backend verifier.
   This requires modifying Viper Server because currently, Viper Server accepts verification requests via HTTP, specifying paths to files only. We will isolate the caching and load balancing features of Viper Server from the HTTP server components. In addition to accepting file paths, the server should be extended to accept a Viper AST by implementing the *verifier* interface. Then, Gobra can use Viper Server as a backend verifier, directly reusing Viper Server's caching and load balancing.

3. **Advanced IDE Features:**
   In order to facilitate the development and verification of Go programs using the Gobra plugin, features improving the user experience are added to the basic IDE integration. Many features can potentially improve the user experience. We will evaluate potential features, prioritize them by their productivity gain, and add the most important features. The current list of possible extension features is:

   (1) **Automatic Gobra Plugin Installation**
   In order to make the plugin more user-friendly, the installation of the plugin and all its dependencies should be possible through the VS Code Extension manager.

   (2) **Syntax Highlighting**
   Support syntax highlighting for Gobra.

   (3) **Goifying and Gobrafying**
   Since Gobra programs include annotations, tools for normal Go programs such as a VS Code plugin for Go, and the Go compiler cannot be used. To tackle this issue, Gobra programs could be converted to Go programs by commenting out annotations and vice versa putting annotations back again. Recovering annotations should be robust to changes in the Go code. Additionally, the commented out annotations should be reusable by a potential future Gobra frontend implemented in Go. The transformation between Gobra and Go programs enables verifying Go files with commented out annotations. This functionality could be added to an explicit Go plugin.

4. **Evaluation:**
   The VS Code plugin for Gobra will be evaluated using different evaluation measures. Two possible measures could be performance and scalability regarding verification project size.

3.2 Extension Goals

1. **Cross-Language Feature**
   Gobra internally translates annotated Go programs to different translation stages, in particular the final Viper code. During the development of Gobra it would be of great use to have a feature in the IDE which displays the output of the translation stages for a specific Gobra file or function.

2. **Verification Process Display**
   Use Visual Studio Code's status bar to display information about the current verification process. For example, a progress bar as seen in the Viper IDE or a summary of the verification results.

3. **Performance Improvement**
   To handle large projects with Gobra the performance of the plugin is crucial. As described previously, load balancing and caching will be adopted from the Viper level by using parts of Viper Server. For further improvements in performance, one can attempt to implement those concepts at the Gobra level as well.

4. **Code Completion**
   Support code completion for annotations introduced by Gobra.

5. **Code Navigation**
   Improve productivity by supporting code navigation in Visual Studio Code. Code navigation includes features such as `Go-to-Implementation`, and `Go-to-Definition`.

References

[1] *Go at CloudFlare*. URL: https://blog.cloudflare.com/go-at-cloudflare/ (visited on Mar. 11, 2020).

[2] *Go Programming Language*. URL: https://golang.org (visited on Mar. 4, 2020).

[3] *How We Built Uber Engineering's Highest Query per Second Service Using Go*. URL: https://eng.uber.com/go-geofence-highest-query-per-second-service/ (visited on Mar. 11, 2020).

[4] *Language Server Protocol*. URL: https://microsoft.github.io/language-server-protocol/ (visited on Mar. 9, 2020).

[5] *Netflix Chaosmonkey*. URL: https://github.com/netflix/chaosmonkey (visited on Mar. 11, 2020).

[6] Malte Schwerhoff Peter Müller and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. 2016, pp. 41–62.

[7] *Scala Programming Language*. URL: https://scala-lang.org/ (visited on Mar. 4, 2020).

[8] *TypeScript Programming Language*. URL: https://www.typescriptlang.org/ (visited on Mar. 4, 2020).

[9] *Viper Project*. URL: https://bitbucket.org/viperproject/viper-ide/src/default/ (visited on Mar. 4, 2020).

[10] *Visual Studio Code*. URL: https://code.visualstudio.com (visited on Mar. 4, 2020).