Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

IDE Support for a Golang Verifier

Bachelor's Thesis

Silas Walker

Tuesday 1st September, 2020

Advisors: Prof. Dr. Peter Müller, Felix Wolf, Linard Arquint
Department of Computer Science, ETH Zürich

Abstract

Gobra is a deductive verification tool used to verify Go programs. Currently, Gobra is provided as a command line tool only. An IDE integration could improve the productivity whilst working with the verification tool.

In this thesis, we developed an IDE integration of Gobra for Microsoft Visual Studio Code. Gobra IDE integrates the verification with Gobra directly into a programmer's workflow. Additional usability improvements, including the caching of intermediate verification results and the ability to translate a Gobra program to a valid Go program, are provided by the IDE integration.

Generally, Gobra IDE improves the usability compared to the command line tool. Additionally, in most of the cases Gobra IDE outperformes the command line tool in terms of the verification time.

Contents

Contents **iii**

1 Introduction **1**

2 Background **3**
2.1 Verification in Gobra 3
2.2 Verification IDEs 4

3 Architecture **7**
3.1 Gobra Client 7
3.2 Gobra Server 9
3.3 Client-Server Communication 10
3.4 Automatic Control Flow 11
3.5 Manual Commands 12
3.6 Settings 14

4 Verification in Gobra Server **17**
4.1 Verification without Caching 17
4.2 Caching 21
4.3 Soundness 22
4.3.1 System Specification 22
4.3.2 Cache Consistency 26
4.3.3 Verification Equivalence 27

5 Advanced IDE Features **31**
5.1 Automatic Gobra Plugin Installation 31
5.2 Syntax Highlighting 32
5.3 Verification Process Display 32
5.4 Goifying and Gobrafying 33
5.4.1 Verification of Go Files 34
5.5 Cross-Language Feature 34

5.6 Performance Improvement 35

6 Evaluation **37**
6.1 Comparison of Visual Features to Viper IDE 37
6.2 Comparison to Command Line Interface 38
6.2.1 Usability 38
6.2.2 Performance Benchmark 38

7 Conclusions and Future Work **43**
7.1 Conclusions 43
7.2 Future Work 43

Bibliography **45**

Chapter 1

Introduction

Go [3] is a statically typed, compiled programming language targeting high-performance networking and multiprocessing. In order to prove the absence of bugs in Go programs, the deductive verification tool Gobra was developed. The verification tool is actively under development and only provides a command line interface (CLI). This limits productivity as there is no integration into an integrated development environment (IDE) yet. Specifically, Gobra has to manually be invoked after some code modifications to reattempt a verification. Furthermore, the user has to manually locate the verification errors using the information displayed in the command line. In addition to these limitations, the command line tool lacks the ability to cache any verification results between two verification runs. Hence, Gobra has to verify the entire program even if there are no modifications compared to the previously verified program.

The main focus of this project is to develop a Visual Studio Code [5] plugin for Gobra. This plugin improves the productivity while working with Gobra by integrating verifications with Gobra into a programmer's workflow. The plugin not only overcomes the above mentioned shortcomings of the command line Gobra version but also provides utility features to for example execute a Go program with specifications.

This report is structured as follows: The plugins main architecture is presented in Chapter 3. This chapter describes the main components of the plugin and the interaction of the user with the plugin. The verification in the IDE with reasoning about the soundness of caching verification results is presented in Chapter 4. Advanced features improving the usability of Gobra IDE are explained in Chapter 5. An evaluation of the plugin is given in Chapter 6. In the last chapter, the work is concluded and a brief outlook on future work is given.

Chapter 2

Background

2.1 Verification in Gobra

Gobra is a deductive verification tool used to verify Go programs. In deductive verification, a program is annotated with contracts, specifying the functional behavior, and proof guiding annotations. These annotations are named ghost code [13]. Ghost code is used to facilitate the specification of the program while keeping the program's semantics unchanged. Thus, ghost code can simply be removed without changing the behavior of the program. In the remainder of this report, *Gobra annotations* is used to denote contracts and ghost code in Gobra programs. Figure 2.1 shows an example of a swap method with Gobra annotations. The annotations define the precondition *(requires acc(x.val ∧ acc(y.val)))* and the postconditions *(ensures acc(x.val) ∧ acc(y.val), ensures x.val == old(y.val) ∧ y.val == old(x.val))*. These specified pre- and postconditions express that there is write access to the *val* field of x and y. Additionally, the values of the two fields are swapped after the invocation. With these annotations, the program is verified regarding its functional correctness. Gobra successfully verifies this function whenever executing the function's body in a state satisfying the precondition results in a state satisfying the postcondition.

```
requires acc(x.val) && acc(y.val)
ensures acc(x.val) && acc(y.val)
ensures x.val == old(y.val) && y.val == old(x.val)
func swap(x *cell, y *cell) () {
  x.val, y.val = y.val, x.val

}
```

Figure 2.1: Swap Function with Contracts

Gobra splits the verification of a program into five steps. Those steps are parsing, type checking, desugaring, encoding into a Viper [1] program, and verification of the Viper program. The first four steps are called *preprocessing* in the remainder of the report. Step five will be called Viper verification to avoid confusion with the overall verification using Gobra. These steps are executed sequentially and Gobra skips all subsequent steps if an error occurs.
In the first step, Gobra parses the input program and converts it to an abstract syntax tree (AST) representation if the program is syntactically correct. Next, the AST representation is type checked to derive typing information for all AST nodes. Desugaring converts the AST representation to a simpler intermediate representation by removing syntactic sugar. The intermediate representation is then encoded in Viper, a programming language targeted at verification. In the last step, the Viper encoding is verified using one of the Viper backends: Silicon [4] performs symbolic execution using the Z3 SMT solver [12]. Carbon [2] is based on verification condition generation (VCD) and internally uses Boogie [8] and Z3.
Verification errors detected in the parsing, type checking, and backend verification steps are returned in two different ways. In particular, Gobra returns a single return value at the end of the verification and streams intermediate results for example of a method as soon as the result is found.

2.2 Verification IDEs

A verification IDE is used to aid the development of a correct program. The development of a correct program refers to writing a program from scratch and simultaneously verifying it. Additionally, annotating an existing implementation to find and fix potential bugs is included in the development of a correct program as well. Such a verification IDE enables a seamless verification of the currently opened program by combining an IDE and a verification tool. In the optimal case, the program is verified automatically during the development. This is achieved by detecting user actions such as file changes, save events, or file open events in the IDE. Hence, the user can focus on the development without having to trigger a verification after each change. Furthermore, potential verification errors are directly shown in the code to speedup the process of locating and fixing issues.
Viper IDE [14] is an example of an already existing verification IDE. It is a plugin for Visual Studio Code and enables the verification of Viper programs directly in Visual Studio Code. Figure 2.2 shows an example of a Viper program opened in Visual Studio Code with the plugin installed. The program is verified automatically upon opening it in the IDE. The resulting error is displayed directly in the code with a squiggly line. Additionally, the toolbar informs the user that the verification has failed.

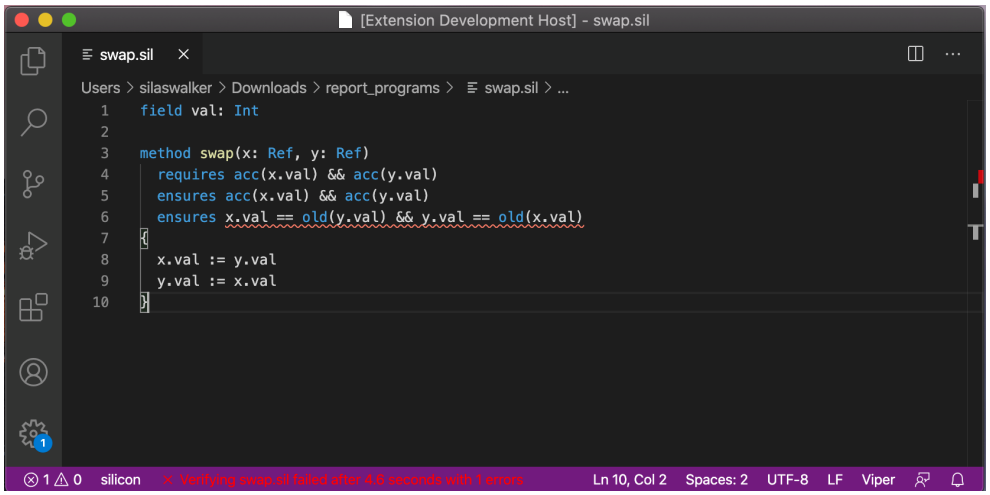

Figure 2.2: Viper IDE in Visual Studio Code

Chapter 3

Architecture

Figure 3.1: Coarse Architecture of Gobra IDE

Figure 3.1 shows the coarse architecture of Gobra IDE. This architecture includes Visual Studio Code with the Gobra extension, serving as the client, and Gobra Server. The client side of the extension is directly added to Visual Studio Code using the Visual Studio Code extension API [7]. Conversely, Client and server communicate over an asynchronous channel with each other. This asynchronous communication enables one instance of Gobra Server handling requests from multiple instances of Gobra IDE clients.

3.1 Gobra Client

Users interact with Gobra IDE directly over Visual Studio Code with the Gobra extension installed. As interactions, the user can give inputs either implicitly through actions or explicitly through commands in Visual Studio Code. The implicit user actions are opening a file, applying changes to a file, and saving a file. Gobra Client reacts to those implicit user actions according to the control flow defined in Section 3.4.
After the verification of a file has finished, the verification results are displayed to the user in the editor of Visual Studio Code directly. Gobra IDE has several mechanisms to display the verification results depending on whether the verification was successful or returns errors. Let's first consider the case where the verification resulted in errors. Those errors are displayed to the

user in four different ways, as shown in Figure 3.2: (1) Whenever the user hovers over the code range resulting in a verification error, some additional information explaining the reason of the error is displayed in a small window next to the mouse cursor. (2) The code range corresponding to a verification error is underlined directly in the editor by a squiggly line. (3) A list of all verification errors together with their corresponding information and position in the source code is displayed in the problems view of Visual Studio Code. (4) To allow the user to identify quickly whether there were any errors in the verification, the overall result of the verification together with the verification time is displayed in the toolbar of Visual Studio Code. In the case of a successful verification there are no verification errors to display and thus the error displays (1), (2), and (3) in Figure 3.2 are not present. The remaining error display (4) shows a success message notifying the user of the successful verification. Additionally, this message again includes the overall verification time needed.

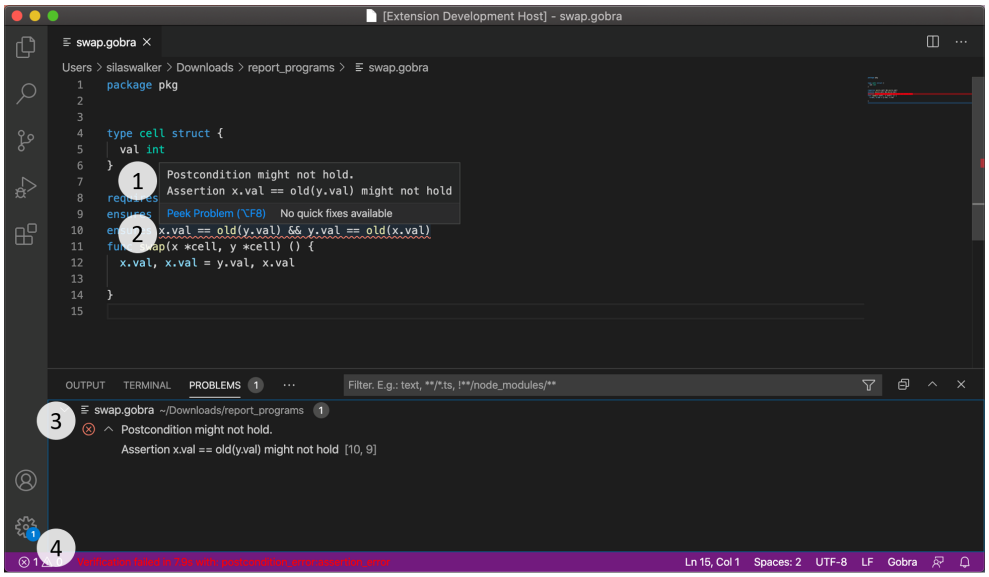

Figure 3.2: Verification Errors displayed in Visual Studio Code

3.2 Gobra Server

Gobra Server can be described in terms of its three main components which are shown in Figure 3.3. These components are the Language Server Protocol component, Gobra, and Viper. The focus of this section is Gobra and Viper, which are used for the verification and which return the results. Details on the Language Server Protocol component are provided in Section 3.3.

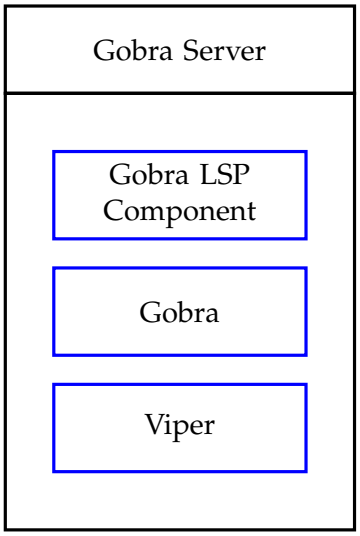

Figure 3.3: Architecture of Gobra Server

As described in Section 2.1, Gobra encodes a Gobra file into the Viper language. Then, the encoded Viper file gets verified using a backend verifier such as Silicon or Carbon. Gobra communicates with its backend verifier via a standardized *verifier* interface. The verifier interface defines as its main functionality a method taking a Viper AST as an input and returning the verification results as output. Neither Gobra nor the verification backends Silicon and Carbon provide any caching features. Thus, Gobra reverifies a method even though it has not changed with respect to the previous verification.
To mitigate this issue, Gobra Server utilizes Viper Server as additional backend verifier. Viper Server is part of the Viper project at ETH Zurich used by Viper IDE. This server manages verification requests via HTTP. The implementation of Viper Server already includes the caching of verification results. In order to reuse this caching feature in Gobra Server, Viper Server is modified to implement the verifier interface. This modification includes the separation of the core functionality of Viper Server from the HTTP communication. The core functionality implementing the verifier interface is

referred to as Viper Core Server.
Internally, Viper Core Server initiates Silicon or Carbon to verify the Viper AST. Thus, Viper Core Server acts as an intermediate component computing the necessary information for caching verification results and storing the cached results.

3.3 Client-Server Communication

The Language Server Protocol (LSP) [6] was created to standardize the communication between an IDE and a server. This allows the IDE to implement complex features and services with lightweight implementations. Additionally, the standardization enables communication between client and server implementations in different programming languages. In Gobra IDE, LSP4J is used for the communication between Visual Studio Code and the LSP component of Gobra Server. LSP4J is a Java binding for the Language Server Protocol.

Gobra Server supports requests for verifications, cache flushing, and utility functions on Gobra files. In the following, the details of each request are described:

1. To issue the verification of a particular file, Gobra Client sends a verification request message to Gobra Server. The message consists of a file location and the verification settings used by Gobra. A list of settings is given in Section 3.6. For such a request, Gobra Server responds with messages containing the verification results. Results consist of updates to the Visual Studio Code toolbar and diagnostics. These diagnostics are displayed as shown in Figure 3.2.

2. For Goification, Gobrafication, and the cross-language feature, explained in Chapter 5, Gobra Client sends a corresponding request containing the file location. On completion of the request, Gobra Server sends a notification to the client. This notification contains whether the completion was successful or resulted in a failure.

3. Flushing the Viper Server cache can be requested by Gobra Client by sending a corresponding request. The request does not need any additional information which gets passed alongside with it. Whenever Gobra Server has flushed the Viper Server cache, it displays a notification message in Visual Studio Code informing the user that the cache has been flushed successfully.

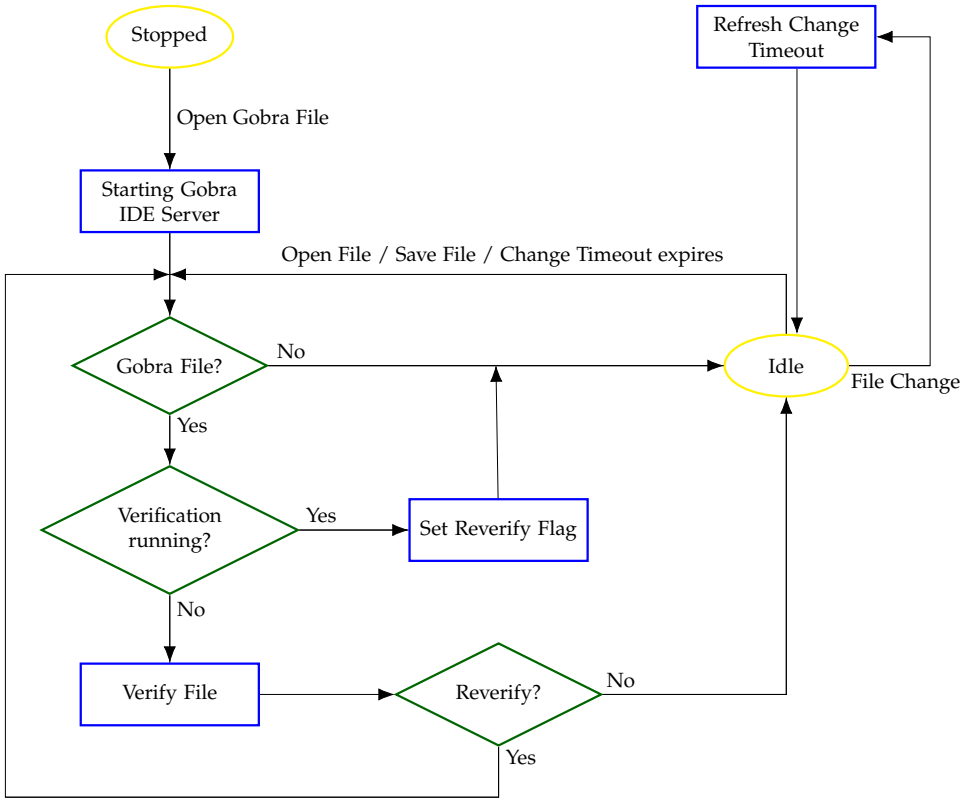

Figure 3.4: Automatic Control Flow of Gobra IDE

3.4 Automatic Control Flow

Gobra IDE automatically invokes verifications of currently opened Gobra files. Figure 3.4 shows the automatic control flow graph used to determine when verifications are invoked. The yellow nodes denote the states in which the IDE can be, the green nodes denote the decisions which the IDE can make internally, and the blue nodes denote the internal computations of the IDE. Labeled edges in the control flow graph indicate transitions between states that are caused by an event. Contrary, unlabeled edges symbolize transitions to another node after the computation of the preceding node has finished.

Initially, Gobra IDE is in the stopped state at the top of Figure 3.4. After the IDE has finished starting up, the IDE checks if the opened file should be verified automatically. A file is verified automatically if (1) it is a Gobra file and (2) it is not being verified at the moment. If one of these checks fails, the verification is aborted and the IDE enters an idle state. In case of the file being already verified, a reverification flag is set. This flag indicates that the

file needs to be reverified once the current verification is completed. When both checks pass, the verification request is sent to Gobra Server. After the verification in Gobra Server is completed, the results are displayed and the file is reverified if the reverification flag is set. Conversely, Gobra IDE enters an idle state if the reverification flag is not set. In the idle state, Gobra IDE waits for further user action.
When Gobra IDE is in idle state, the user can trigger additional verifications by either opening a new file or saving a currently opened file. Whenever one of those two actions occurs, the corresponding file is verified as described for the start of the IDE.
Additionally, a verification can be triggered by modifying a file opened in the editor. Gobra IDE should avoid initiating a verification on each file change. Therefore, a timeout is kept to track the time since the last modification of the file. This timeout is reset whenever the user modifies the currently opened Gobra file. On expiration of the timeout, Gobra IDE automatically verifies the file.

3.5 Manual Commands

In addition to the previously described automatic control flow, the user can manually execute commands inside the IDE. Visual Studio Code has different ways for the invocation of such manual commands. In Gobra IDE, keyboard shortcuts, the command palette, and the editor context are used for the invocation of the manual commands. In Figure 3.5, an example of the command palette and editor context are shown on the left and right, respectively. The editor context of Visual Studio Code can be displayed by right clicking the editor.

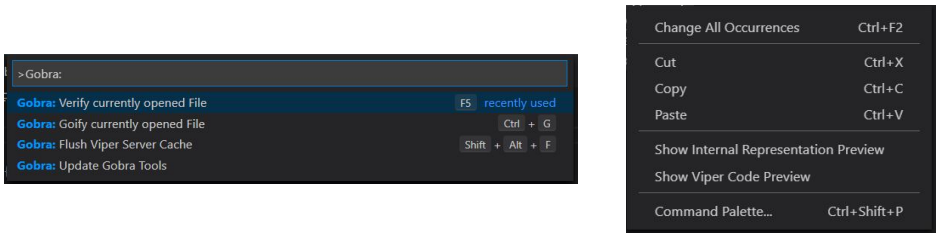

Figure 3.5: Command Palette and Editor Context of Visual Studio Code

The set of executable commands differs depending on whether the file type is Go or Gobra. To prevent the user from executing commands which were not intended for the currently opened file type, Visual Studio Code only displays the currently executable commands. Table 3.1 contains all manually

invokable commands of Gobra IDE with the invocation options and a short description.

Command	Invocation	Description
`flushCache`	Command Palette Shortcut: *alt+shift+f*	Flushes the Viper Server cache in Gobra Server and displays a notification message in Visual Studio Code when the cache was successfully flushed.
`goifyFile`	Command Palette Shortcut: *ctrl+g*	Makes the annotated Gobra program executable Go code by commenting out the ghost code of the currently opened file as described in Section 5.4. The command is only available for Gobra files.
`gobrafyFile`	Command Palette Shortcut: *ctrl+g*	Translates the currently opened Go file into Gobra by inserting the commented out ghost code of the currently opened file back into the source code. Further detail on this is provided in Section 5.4. This command can only be invoked on Go files.
`showViperCode`	Editor Context	Shows a read-only preview of the Viper encoding of the currently opened Gobra file. Additionally, the translated Viper encoding of selected code is highlighted in the preview. More information is provided in Section 5.5.
`showInternalCode`	Editor Context	Before the Viper encoding, Gobra has an intermediate representation of the Go code. On invocation of this command, the intermediate representation is displayed similarly to the read-only preview of the Viper encoding.
`updateGobraTools`	Command Palette	Update the Gobra tools according to the description in Section 5.1.
`verifyFile`	Command Palette Shortcut: *F5*	Invoking this command triggers a verification of the currently opened file. Additionally to verifying Gobra files, the manual verification invocation allows verifying Go files. For further detail on this, see Section 5.4.1.

Table 3.1: Manual Commands in Gobra IDE

3.6 Settings

To add flexibility to the Gobra IDE plugin, several settings can be adjusted directly in the preferences of Visual Studio Code. Each of the settings has a default value to work out of the box on Windows, Mac and Linux. The interface for adjusting settings in Visual Studio Code depends on the type of value which can be chosen for a particular setting. Figures 3.6, 3.7, and 3.8 show the interfaces for a boolean setting, an object setting, and a string setting, respectively.

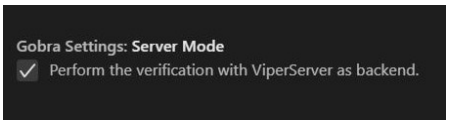

Figure 3.6: Boolean Setting in Visual Studio Code

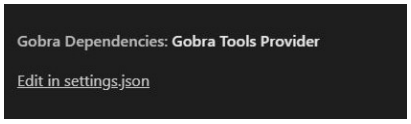

Figure 3.7: Object Setting in Visual Studio Code

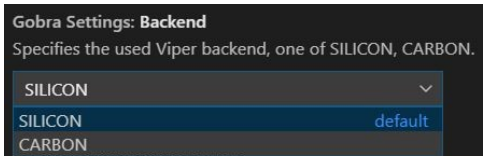

Figure 3.8: String Setting in Visual Studio Code

Even though all settings can be adjusted in the same place, the settings are grouped into three categories. The settings used for the verification directly, the ones used for the dependency installation of Gobra IDE, and the ones used in the workflow of the IDE. The settings which are used for the verification directly, are included in the verification requests send to Gobra Server. Gobra Server uses them to construct the verification configuration for Gobra.
Table 3.2 includes all available verification settings with their options and description. Default options of the settings are marked in boldface. The dependency installation settings and IDE workflow settings are listed in Table 3.3 and Table 3.4, respectively.

Setting	Options	Description
Verification Backend	**Silicon** Carbon	Specifies the backend verifier used by Gobra.
Debug	True **False**	Verification additionally creates a new file where additional debug information is printed to. The new file has the filename extension `debugtype`.
Erase Ghost	True **False**	Verification additionally creates a new file where the input program without ghost code is printed to. The new file has the filename extension `ghostless`.
Loglevel	**Off** Error Warn Info Debug Trace All	Determines the level of detail of the logs created by the backend verifier. Only required for debugging purposes.
Parse Only	True **False**	Sets that files are only parsed and not verified fully. This omits the typechecking, desugaring, viper encoding and verification step in Gobra.
Print Internal	True **False**	Verification additionally creates a new file where the internal representation of the input program is printed to. The new file has the filename extension `internal`.
Print Viper	True **False**	Verification additionally creates a new file where the viper language encoding of the input program is printed to. The new file has the filename extension `vpr`.
Server Mode	**True** False	Specifies whether Viper Server is used as backend verifier of Gobra. The verification backend used by Viper Server to actually verify files is determined with the *Backend* option.
Unparse	True **False**	Verification additionally creates a new file where the parsed input program is printed to. The new file has the filename extension `unparsed`.

Table 3.2: Verification Settings of Gobra IDE

Setting	Description
Tool Paths	Specifies the platform dependent paths for the Gobra Tools. This includes a path to the Gobra Tools directory, the z3 executable, the boogie executable, and the Gobra IDE server jar. In order to include the Gobra Tools directory in a path, one can add the prefix $*gobraTools*$ which later gets resolved to the actual path of the Gobra Tools.
Tool Provider	Specifies the platform dependent URL to the stable and nightly Gobra Tools versions.
Build Version	Specifies whether to download and install the stable or nightly version of Gobra Tools. The default option is *stable*.

Table 3.3: Dependency Settings of Gobra IDE

Setting	Description
Automatic Verification	Specifies whether Gobra IDE uses the automatic control flow defined in Section 3.4. Per default the automatic verification is enabled.
Timeout	Specifies the value of the file change timeout introduced in Section 3.4. The default value of the timeout is one second.

Table 3.4: Workflow Settings of Gobra IDE

Chapter 4

Verification in Gobra Server

In this chapter, the verification performed by Gobra Server is explained in more detail. First, the main mechanisms are explained in section 4.1 based on the verification using Gobra Server without any caching. Then, caching in Gobra Server together with the necessary changes to the verification workflow is introduced in section 4.2.

4.1 Verification without Caching

Gobra Server receives verification requests containing verification settings and the location of the file to verify. The tuple of file location and verification settings is referred to as *verification job*. Since the backend verifier Silicon can not verify programs in parallel, Gobra server enqueues arriving verification jobs into a job queue. A separate thread dequeues the oldest pending verification job whenever no verification is currently running and there is at least one pending job in the job queue. Using this dequeued job, Gobra Server constructs a verification configuration for the Gobra. After the construction of the configuration, the verification with Gobra is invoked directly.
As explained in Section 2.1, Gobra first translates the program to obtain the corresponding Viper AST. This AST is then passed to the Viper component of Gobra Server. The choice of the Viper component depends on the settings which have arrived in the verification request for this file. Since this section does not treat cached verification, the set of possible Viper components is restricted to Silicon and Carbon.
During the verification, Gobra Server uses the reporter object to enable result streaming. In particular, partial verification results are reported as soon as they are available. Examples of partial results include the successful verification of a function or a particular error for a certain function. The reported results are used to construct Visual Studio Code diagnostics. This construc-

tion is a data type conversion of the verification errors. In this conversion, the positional information of the verification error is copied to the diagnostics positional information. The reason of the error is used as the message of the diagnostic. This message is later displayed to the user whenever the cursor is hovered over the diagnostic.

Since the user can modify a file during or after verification, the position of already reported diagnostics may change. As stated in the specification of the LSP, the server is responsible for diagnostics and thus it has to compute the positional information of diagnostics after a file change. In order to react to all the file changes in the Visual Studio Code editor, the client monitors those changes using a file system watcher. These monitored file changes are then sent to Gobra Server using the Language Server Protocol. Gobra Server then updates the currently published diagnostics according to the arrived file changes. Afterwards, the updated diagnostics are published again in Visual Studio Code. Whenever a file change happens for a file which is currently being verified, Gobra Server stores the file change. These stored file changes are then used to update the diagnostics resulting from the current verification. In the following, the update function of one fixed diagnostic is described by a case distinction on the file change. The updating however, scales easily to the real implementation since Gobra Server just applies this update to all its diagnostics instead of the single one used in this description. For the sake of brevity, focus is set on changes adding characters. Other changes are dealt with analogously. Figure 4.1 shows a swap method that is supposed to swap the value field of the two given pointers. The postcondition, stating that the values have been swapped, is underlined by a squiggly line. This means that the implementation of the swap method is incorrect. Furthermore, the five positions where a file change may happen are annotated in the figure. These options are (1) at a strictly lower line number than the diagnostic, (2) at the first character of the same line as the diagnostic, (3) at the character before the diagnostic, (4) inside the diagnostic, and (5) at a higher line number than the diagnostic. For each of these cases, the change being applied to the diagnostic is described:

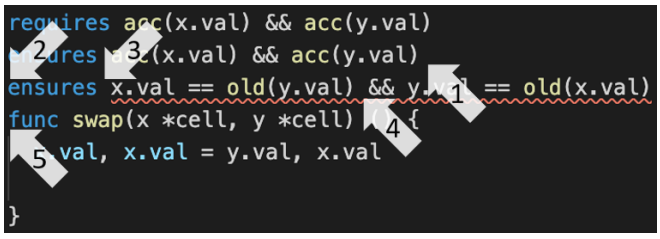

Figure 4.1: Positions of File Changes with respect to Diagnostic

Case (1) displays a file change which happens on some line number strictly smaller than the line number of the diagnostic. In this case, the line number of the diagnostic has to be incremented by the number of new lines that were added. In Figure 4.2, an example of adding one line at position (1) with the corresponding diagnostic movement is given.

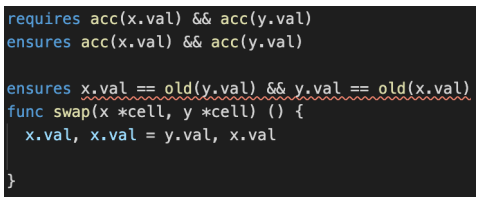

Figure 4.2: Applying a file change at position (1)

Position (2) considers a file change at the first character of a line which contains a diagnostic. Whenever a non new line character is added, the character number of the diagnostic is incremented by the number of characters which were added. If the file change contains a new line character, the diagnostic is moved the same way as in case (1). An example for this file change can be seen in Figure 4.3, where on the left hand side, one space character and on the right hand side, one new line character was added.

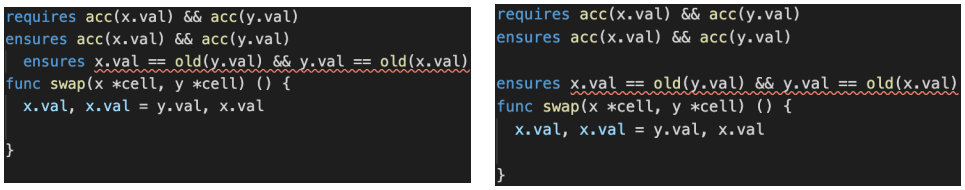

Figure 4.3: Applying a file change at position (2)

Figure 4.4 shows an example of applying a file change at position (3), where the left hand side depicts the addition of a non new line character and the right hand side depicts the addition of one new line character. This file change is similar to the one from case (2), except that it is not applied at the first character of the new line. In case of a non new line character being added, the diagnostic is moved in the same way as in case (2). However, if a new line is added, the diagnostic has to be moved differently. In particular, in addition to increasing the line number by the number of added new line characters, the character number of the position has to be decreased by the character number of the line change.

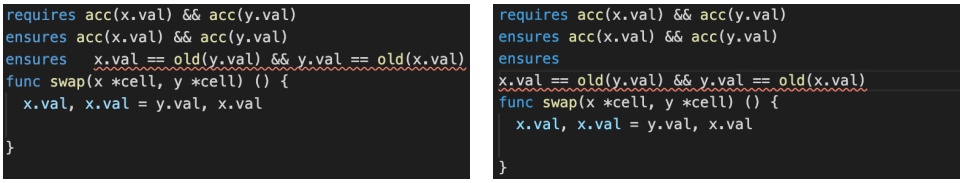

Figure 4.4: Applying a file change at position (3)

Whenever a file change is applied in between a diagnostic, as in position (4), only the end position of the diagnostic has to be changed. Thus, the diagnostic is changed as explained in case (3) with the only difference being that the starting position of the diagnostic remains fixed. A concrete example of the diagnostic movement can be seen in Figure 4.5, where the left figure shows the addition of some space characters and the right figure shows the addition of a new line character.

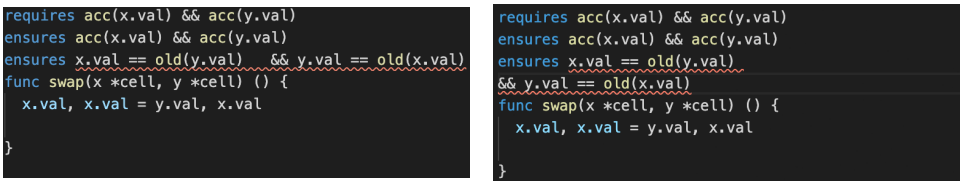

Figure 4.5: Applying a file change at position (4)

The file change at position (5) represents file changes after a diagnostic. This means that either the line number of the diagnostic is strictly smaller than the line number of the change or if the line numbers are equal, the character number of the diagnostic is strictly smaller than the character number of the change. Since the change does not directly affect the diagnostic, no movement is required.

4.2 Caching

Since the verification flow described in section 4.1 does not include any caching of intermediate results, the whole file has to be reverified each time. This can lead to large overheads, especially, when the file being verified is large and the changes between two verifications are small. To reduce this overhead, Gobra Server adopts caching and load balancing from Viper Server. Viper Server caches the results of verified methods. These cache entries are identified by a program identifier and the hash of the function AST. With this internal cache, the verification using Viper Server is as follows: First, for each function of the program, the hash is looked up in the Viper Server cache. When a lookup results in a cache hit, the corresponding verification results are reused. On a cache miss, Viper Server reverifies the function and caches the results. For the returned verification results, a flag indicates whether the result was retrieved from Viper Server cache. The subsequent paragraphs describe the necessary modifications to the verification flow introduced in Section 4.1.

As explained in the verification flow without caching, Gobra Server handles a verification request by first encoding the program resulting in the Viper AST corresponding to the program. This Viper AST is passed to Viper Server. Viper Server then verifies the Viper AST using the chosen backend. In order to construct the diagnostics directly from the verification errors returned by Viper Server, the reporter object is adapted. This reporter first splits the verification errors into the ones originating from Viper Server cache and the ones originating from the direct verification. Errors not originating from Viper Server cache are used to construct Visual Studio Code diagnostics in the same way as for the verification without caching.
Conversely, for cached results, Gobra Server cannot just translate the cached verification errors to the corresponding diagnostics. The reason for this is that Viper Server does not consider file changes by updating the positional information in the cache. For example, consider a function which was verified by Viper Server before. As a result, the verification results are in Viper Server cache. Then, a user adds a new line character before the function and invokes a verification. Since the modification to the program does not change the function itself, the results will be retrieved from the Viper Server cache. The position of these results is still the same as in the first verification. However, since the line number of the function has increased by one, the results are outdated. In order to keep those diagnostics up to date with the file changes, Gobra Server internally stores the diagnostics corresponding to cached verification errors in the diagnostics cache. This diagnostics cache is then used to retrieve the diagnostics for cached verification errors directly. We make sure that the diagnostics cache is always consistent with the Viper

Server cache. Thus, if a verification error is in the Viper Server cache, the corresponding diagnostic can always be retrieved from diagnostics cache.
After the verification, Gobra Server updates the diagnostics cache by adding all the verification results with the corresponding diagnostics. Additionally, Gobra Server keeps the positional information of the diagnostics in the diagnostics cache up to date by applying file changes to all the diagnostics currently stored in the cache. This update is done similarly to the updating of diagnostics described in Section 4.1.

4.3 Soundness

The introduction of caching in Gobra Server leads to verification results depending on previous verification runs. This can be the cause for unsoundess in verifications since Gobra Server has to decide which results are taken from the cache and which results are obtained from a reverification. Wrong decisions cause the output of an incorrect result. For example, an error in a method is not shown because the method verified successfully previously.
In this section, we reason about the soundness of our caching approach in Gobra IDE. As a soundness property we want that a verification with caching returns the same results as a verification without caching. We call this soundness property *verification equivalence*. The reasoning about this soundness property proceeds as follows: First, we specify the main components of the system. Then, we define an invariant specifying that the diagnostics cache and the Viper Server cache are consistent with each other. Lastly, using this invariant, we reason about the verification equivalence in Gobra IDE.

4.3.1 System Specification

We specify the state of the system as a quadruple *(jobQueue, diagnosticsCache, diagnostics, viperServerState)* where the elements are defined as follows:

- *jobQueue*: The internal first-in first-out queue containing all pending verification jobs.
- *diagnosticsCache*: The mapping of Gobra verification errors to Visual Studio Code diagnostics.
- *diagnostics*: The currently published diagnostics of Gobra Server. These diagnostics are displayed in the Visual Studio Code editor.
- *viperServerState*: The current state of Viper Server, including Viper Server cache.

In order to simplify the specification, we introduce some notation and make some assumptions on the system. As already seen in Section 2.1, the verification in Gobra can be split into different verification steps. These steps include parsing, type checking, desugaring, translating, and verifying the corresponding program. Since the first four steps of the verification are not involved with caching, we use the function *preprocess* to express the combination of parsing, type checking, desugaring, and translating. This function takes a Gobra program as input and returns either a Viper AST or a list of errors if preprocessing failed. Gobra's preprocessing results in a Viper program, where all methods have a unique name. This has the benefit that name ambiguities are avoided for caching.
Further, we assume that each Gobra program has a unique program identifier. This program identifier is used by the Viper Server cache and the diagnostics cache to store entries for each program separately. In particular, the cached entries of a Gobra program p do not depend on the cached entries of a Gobra program $p' \neq p$.
After the preprocessing, Gobra verifies the resulting Viper AST using a backend verifier. We abstract this verification of the Viper AST by a function called *backendVerification(id, ast, serverState, backend)*, taking the program identifier *id*, the Viper AST *ast*, the Viper Server state *serverState*, and the backend verifier *backend* as input. The function returns the already translated Gobra verification errors together with the updated Viper Server state. For the further reasoning, we abstract the backend verifier input of the function. We use the term *SingleRun* to denote the backend verifier Silicon and the term *ViperServer* to denote Viper Server with Silicon. The soundness argument for the backend verifier Carbon is analogous.

We represent internal computations of Gobra Server as state transitions of the system state. These state transitions are triggered by external and internal events. External events are starting the system, shutting the system down, changing a file, and enqueuing verification jobs. These events are initiated by the user directly. Conversely, Gobra Server triggers an internal event to start a new verification. To further specify those events, we give a short description for each of them together with the rule showing the modification of the state in the system. We use *jq*, *dc*, *d*, *vss* to range over the job queue, diagnostics cache, diagnostics, and Viper Server state, respectively. Further, we denote the cache of a given Viper Server state *vss* explicitly as *vss(cache)*.

- **start**: This event is initiated once at the beginning to start Gobra Server and initialize the state.

$$[] \xrightarrow{\text{start}} [\mathsf{Nil}, \emptyset, \emptyset, \mathsf{vss}(\emptyset)]$$

- **shutdown**: When Visual Studio Code is closed, this event is triggered to shut down Gobra Server and clean up the state.

$$[\mathsf{jq}, \mathsf{dc}, \mathsf{d}, \mathsf{vss}] \xrightarrow{\mathsf{shutdown}} []$$

- **enqueueJob(p, b)**: Verification requests containing verification jobs arriving at Gobra Server trigger the **enqueueJob** event. This event enqueues the verification job consisting of a Gobra program p and a verification backend b to the *jobQueue* of the system state.

$$[\mathsf{jq}, \mathsf{dc}, \mathsf{d}, \mathsf{vss}] \xrightarrow{\mathsf{enqueueJob}(p,b)} [\mathsf{jq} : +\mathsf{Job}(p, b), \mathsf{dc}, \mathsf{d}, \mathsf{vss}]$$

- **verifyProgram**: An internal event initiated by Gobra Server itself whenever a verification job is enqueued and no verification is currently running, or when a verification has terminated and the *jobQueue* is not empty.

$$[\mathsf{Job}(p, b) :: \mathsf{jq}, \mathsf{dc}, \mathsf{d}, \mathsf{vss}] \xrightarrow{\mathsf{verifyProgram}} [\mathsf{jq}, \mathsf{verify}(p, b, \mathsf{dc}, \mathsf{vss})]$$

  The *verify* function is described in Algorithm 1. This function takes a Gobra program, a verification backend, a diagnostics cache, and a Viper Server state as input. The function returns an updated diagnostics cache, the diagnostics resulting from the verification of program p, and an updated Viper Server state.

- **fileChange(pos, text)**: This event is triggered whenever a user modifies a Gobra file. The position of the file change *pos* and the text which was added to the Gobra file *text* are included in the file change event.

$$[\mathsf{jq}, \mathsf{dc}, \mathsf{d}, \mathsf{vss}] \xrightarrow{\mathsf{fileChange}(\mathsf{pos},\mathsf{text})} [\mathsf{jq}, \mathsf{translate}(\mathsf{pos}, \mathsf{text}, \mathsf{dc}, \mathsf{d}), \mathsf{vss}]$$

  The *translate* function modifies the *diagnostics* and all diagnostics of *diagnosticsCache* according to the file change and returns the modified *diagnosticsCache* and *diagnostics*. We define the function in Section 4.1.

Algorithm 1 shows the *verify* function used in the **verifyProgram** event. This function takes as input a Gobra program, a verification backend, the diagnostics cache, and Viper Server state. The function returns the updated diagnostics cache, the diagnostics corresponding to this verification, and the updated Viper Server state. We denote the updated values with a prime. The *verify* function internally constructs diagnostics using Algorithm 2. This algorithm takes a program identifier, a list of Gobra errors, a verification backend, and the diagnostics cache as input. When the diagnostics have been constructed, the function returns the updated diagnostics cache and

the constructed diagnostics. We use *Diagnostic(err)* to denote the conversion from a verification error *err* to a Visual Studio Code diagnostic. We define the conversion in Section 4.1.
To simplify the notation, the function *preprocess* returning either a Viper AST or Gobra verification errors returns both values in a tuple. On a successful preprocessing, the returned verification errors are the empty set and a valid Viper AST is returned. Contrary, the resulting verification errors are returned whenever the preprocessing is not valid.

Algorithm 1: Verify function in Gobra Server

```
Input: gobraProgram, backend, dc, vss
Output: dc', d, vss'

id ← gobraProgram.getId
viperProgram, errors ← preprocess(gobraProgram)
if errors ≠ ∅ then
    dc', vss' ← dc, vss
    _, d ← constructDiagnostics(errors, backend, dc)
else
    errors, vss' ← backendVerification(id, viperProgram, vss, backend)
    dc', d ← constructDiagnostics(id, errors, backend, dc)
```

Algorithm 2: Diagnostics construction function in Gobra Server

```
Input: id, errors, backend, dc
Output: dc', d

dc', d ← dc, ∅
foreach error ∈ errors do
    if error.cached = false then
        diagnostic ← Diagnostic(error)
        d ← diagnostic :: d
        if backend = ViperServer then
            dc' ← ((id, error) → diagnostic) :: dc'
    else
        diagnostic ← dc.get(id, error)
        d ← diagnostic :: d
```

4.3.2 Cache Consistency

We want to define the cache consistency of the diagnostics cache in Gobra Server and the Viper Server cache. Intuitively, cache consistency holds if and only if all errors in the Viper Server cache are valid keys of the diagnostics cache.
Formally, we define cache consistency on traces of Gobra Server. We represent such a trace as a sequence of events $e_0, e_1, \ldots, e_n$, where e_0 is the start event, e_n is the shutdown event, and $e_1, \ldots, e_{n-1}$ are either enqueueJob, verifyProgram, or fileChange events. Since we consider Gobra Server with caching, ViperServer is used as backend verifier. We index the verifyProgram events as v_i, for example e_{v_1} and e_{v_2} denote the first and second verifyProgram event in our trace, respectively. For each index v_i, we use $\mathsf{errors}(v_i)$ to denote all errors returned by the function *backendVerification* up to the index v_i. We use $\mathsf{err.cached} \equiv \mathsf{true}$ to denote that err was retrieved from Viper Server cache. Further, the diagnostics cache at index v_i is referred to as $\mathsf{dc}(v_i)$ with the corresponding key set $\mathsf{keys}(\mathsf{dc}(v_i))$. The initial values for the returned errors and the diagnostics cache are $\mathsf{errors}(v_0), \mathsf{dc}(v_0) = \emptyset, \emptyset$.
With these definitions, cache consistency can be written as follows:

$$\forall i \in \mathbb{N}, \forall \mathsf{err} \in \mathsf{errors}(v_i) \colon \mathsf{err.cached} \equiv \mathsf{true} \implies \mathsf{err} \in \mathsf{keys}(\mathsf{dc}(v_i))$$

We reason about the cache consistency property using an induction on the index i. For the base case, we have $i = 0$ for which the property can be written as

$$\forall \mathsf{err} \in \emptyset \colon \mathsf{err.cached} \equiv \mathsf{true} \implies \mathsf{err} \in \emptyset$$

which trivially holds.

In the step case, $i \to i + 1$ the induction hypothesis is defined as

$$P(i) \equiv \forall \mathsf{err} \in \mathsf{errors}(v_i) \colon \mathsf{err.cached} \equiv \mathsf{true} \implies \mathsf{err} \in \mathsf{keys}(\mathsf{dc}(v_i))$$

and used to show

$$P(i+1) \equiv \forall \mathsf{err} \in \mathsf{errors}(v_{i+1}) \colon \mathsf{err.cached} \equiv \mathsf{true} \implies \mathsf{err} \in \mathsf{keys}(\mathsf{dc}(v_{i+1}))$$

In order to show that the previous equation holds, we make a case distinction on whether *preprocess* on the file verified at index v_i succeeds or not. If *preprocess* fails, the diagnostics cache is not updated and *backendVerification* is not invoked. Therefore, $\mathsf{errors}(v_{i+1}) = \mathsf{errors}(v_i)$ and $\mathsf{keys}(\mathsf{dc}(v_{i+1})) = \mathsf{keys}(\mathsf{dc}(v_i))$ which result in our goal $P(i+1)$ following directly from the induction hypothesis $P(i)$.
When *preprocess* succeeds, a valid Viper AST is generated and passed to *backendVerification*. The set of errors at index $i + 1$ is the union of the previous errors and the new errors, i.e. $\mathsf{errors}(v_{i+1}) = \mathsf{errors}(v_i) \cup \mathsf{new\_errors}$.

All returned errors that had their cached flag set true must be contained in $\mathsf{errors}(v_i)$. This follows from the observation, that if the cached flag of an error is set true, Viper Server has retrieved the error from its cache. Since the error was retrieved from the cache, the error had to be stored in the cache in a previous verification. Thus, the error was found in a previous verification. Therefore, the error is included in $\mathsf{errors}(v_i)$. Conversely, all returned errors with the cached flags set false are contained in new_errors. Note that the cached flags of all errors contained in $\mathsf{errors}(v_{i+1})$ are set true after the verification.
Updating the diagnostics cache is done according to Algorithm 2. In particular, for each error in new_errors a mapping to its corresponding diagnostic is added. Using the set notation this means $\mathsf{dc}(v_{i+1}) = \mathsf{dc}(v_i) \cup (\mathsf{new\_errors} \rightarrow \mathsf{diagnostics})$ where diagnostics denote the diagnostics corresponding to new_errors.
With this, we can show our goal $P(i+1)$. First, let $\mathsf{err} \in \mathsf{errors}(v_{i+1})$ be arbitrary. We assume $\mathsf{err.cached} \equiv \mathsf{true}$ and show $\mathsf{err} \in \mathsf{keys}(\mathsf{dc}(v_{i+1}))$. Since $\mathsf{errors}(v_{i+1}) = \mathsf{errors}(v_i) \cup \mathsf{new\_errors}$, err is either in $\mathsf{errors}(v_i)$ or new_errors. If $\mathsf{err} \in \mathsf{errors}(v_i)$, we can use the induction hypothesis to get that err is in $\mathsf{keys}(\mathsf{dc}(v_i))$ and thus in $\mathsf{keys}(\mathsf{dc}(v_{i+1})) = \mathsf{keys}(\mathsf{dc}(v_i)) \cup \mathsf{keys}((\mathsf{new\_errors} \rightarrow \mathsf{diagnostics}))$. Conversely, if $\mathsf{err} \in$ new_errors, err is in $\mathsf{keys}(\mathsf{dc}(v_{i+1}))$ by the construction of $\mathsf{dc}(v_{i+1})$. This concludes the proof.

4.3.3 Verification Equivalence

In this section, we reason about the verification equivalence in Gobra IDE. Verification equivalence holds if and only if a verification with caching returns the same results as a verification without caching.
Formally, we define verification equivalence on traces of Gobra Server. We use s_n to denote a trace of length n. This trace is represented as a sequence $e_0, e_1, \ldots, e_{n-1}$, where e_i is either an enqueueJob, verifyProgram, or fileChange event. For an arbitrary sequence s_n and an arbitrary Gobra program p, we assume that after executing s_n in the initial state, the first entry of the job queue is $\mathsf{J} = \mathsf{Job}(p, \mathsf{ViperServer})$. With this, we formally define verification equivalence as follows:

$$
\begin{aligned}
&\forall n \in \mathbb{N}\colon P(n) \equiv \\
&[\mathsf{jq}_{\mathsf{init}}, \mathsf{dc}_{\mathsf{init}}, \mathsf{d}_{\mathsf{init}}, \mathsf{vss}_{\mathsf{init}}] \xrightarrow{s_n} [\mathsf{J} :: \mathsf{jq}, \mathsf{dc}, \mathsf{d}, \mathsf{vss}] \xrightarrow{\mathsf{verifyProgram}} [\_, \_, \mathsf{d}_1, \_] \wedge \\
&[\mathsf{Job}(\mathsf{p}, \mathsf{SingleRun}), \emptyset, \emptyset, \mathsf{vss}(\emptyset)] \xrightarrow{\mathsf{verifyProgram}} [\_, \_, \mathsf{d}_2, \_] \\
&\quad \Longrightarrow \\
&\mathsf{d}_1 = \mathsf{d}_2
\end{aligned}
$$

In the above execution sequences, we use the subscript init to denote arbitrary initial values. We assume that this initial state does not contain any information about the program p. In particular, the initial Viper Server state $\mathsf{vss}_{\mathsf{init}}$ does not contain any verification results of program p.
Since the errors generated in the preprocessing of Gobra do not depend on any previous runs of Gobra, the verification equivalence will always hold when p fails preprocessing. Thus, for the further discussion we can assume that preprocessing on p succeeded.
To show that $P(n)$ holds for all $n \in \mathbb{N}$, we make an induction on n. In the base case $n = 0$, the sequence s_0 is empty and thus d_1 follows from the execution sequence:

$$[\mathsf{J} :: \mathsf{jq}_{\mathsf{init}}, \mathsf{dc}_{\mathsf{init}}, \mathsf{d}_{\mathsf{init}}, \mathsf{vss}_{\mathsf{init}}] \xrightarrow{\mathsf{verifyProgram}} [\_, \_, \mathsf{d}_1, \_]$$

Since the initial Viper Server state does not contain any information about p, all returned verification errors have the cached flag set false. This means that all errors originate from the backend verifier of Viper Server directly. In particular, this is equivalent to using the backend verifier directly for the verification, since the cache is not utilized. Thus, the errors returned by Viper Server are the same as the ones returned by the single run backend verifier. As the construction of diagnostics is deterministic, the equivalence of the returned errors implies the equivalence of the diagnostics. This concludes the base case of the induction.

For the step case $n \to n + 1$, the event sequence of d_1 is written as:

$$[\mathsf{jq}_{\mathsf{init}}, \mathsf{dc}_{\mathsf{init}}, \mathsf{d}_{\mathsf{init}}, \mathsf{vss}_{\mathsf{init}}] \xrightarrow{s_{n+1}} [\mathsf{J} :: \mathsf{jq}', \mathsf{dc}', \mathsf{d}', \mathsf{vss}'] \xrightarrow{\mathsf{verifyProgram}} [\_, \_, \mathsf{d}_1, \_]$$

In order to apply the induction hypothesis for a sequence of length n, the event sequence of length $n + 1$ is expanded to:

$$\begin{aligned}
[\mathsf{jq}_{\mathsf{init}}, \mathsf{dc}_{\mathsf{init}}, \mathsf{d}_{\mathsf{init}}, \mathsf{vss}_{\mathsf{init}}] &\xrightarrow{s_n} [\mathsf{jq}, \mathsf{dc}, \mathsf{d}, \mathsf{vss}] \\
&\xrightarrow{\mathsf{EVENT}} [\mathsf{J} :: \mathsf{jq}', \mathsf{dc}', \mathsf{d}', \mathsf{vss}'] \\
&\xrightarrow{\mathsf{verifyProgram}} [_, _, \mathsf{d}_1, _]
\end{aligned}$$

where EVENT denotes an arbitrary file change event, a verify event, or a job enqueue event. In the following, we make a case distinction on EVENT.

If EVENT is a verification event, two sub cases have to be distinguished: One for the verification with *SingleRun* as a backend verifier and one with *ViperServer* as backend verifier.
First, we consider when *SingleRun* is used as backend verifier. A verification

using the single run backend verifier updates neither the Viper Server state nor the diagnostics cache. Thus, for an arbitrary Gobra program p', we obtain the following computation sequence for d_1:

$$\begin{aligned}[\mathsf{jq}_{\mathsf{init}}, \mathsf{dc}_{\mathsf{init}}, \mathsf{d}_{\mathsf{init}}, \mathsf{vss}_{\mathsf{init}}] &\xrightarrow{s_n} [\mathsf{Job}(p', \mathsf{SingleRun}) :: \mathsf{jq}, \mathsf{dc}, \mathsf{d}, \mathsf{vss}] \\ &\xrightarrow{\mathsf{verifyProgram}} [\mathsf{J} :: \mathsf{jq}', \mathsf{dc}, \mathsf{d}', \mathsf{vss}] \\ &\xrightarrow{\mathsf{verifyProgram}} [\_, \_, \mathsf{d}_1, \_]\end{aligned}$$

We observe that the verify function described in Algorithm 1 does not depend on the job queue and the currently published diagnostics. Therefore, the previous computation sequence yields the same results for d_1 as the following sequence of length n:

$$\begin{aligned}[\mathsf{jq}_{\mathsf{init}}, \mathsf{dc}_{\mathsf{init}}, \mathsf{d}_{\mathsf{init}}, \mathsf{vss}_{\mathsf{init}}] &\xrightarrow{s_n} [\mathsf{J} :: \mathsf{jq}, \mathsf{dc}, \mathsf{d}, \mathsf{vss}] \\ &\xrightarrow{\mathsf{verifyProgram}} [\_, \_, \mathsf{d}_1, \_]\end{aligned}$$

To conclude the first sub case, we apply the induction hypothesis to the sequence of length n and obtain $\mathsf{d}_1 = \mathsf{d}_2$ directly.
Let again p' denote an arbitrary Gobra program. If *ViperServer* is used for the verification event, we obtain the following computation sequence for d_1:

$$\begin{aligned}[\mathsf{jq}_{\mathsf{init}}, \mathsf{dc}_{\mathsf{init}}, \mathsf{d}_{\mathsf{init}}, \mathsf{vss}_{\mathsf{init}}] &\xrightarrow{s_n} [\mathsf{Job}(p', \mathsf{ViperServer}) :: \mathsf{jq}, \mathsf{dc}, \mathsf{d}, \mathsf{vss}] \\ &\xrightarrow{\mathsf{verifyProgram}} [\mathsf{J} :: \mathsf{jq}', \mathsf{dc}', \mathsf{d}', \mathsf{vss}'] \\ &\xrightarrow{\mathsf{verifyProgram}} [\_, \_, \mathsf{d}_1, \_]\end{aligned}$$

We make a case distinction on the program p', i.e. $p' = p$ and $p' \neq p$. If $p' = p$, we can apply the induction hypothesis to obtain $\mathsf{d}' = \mathsf{d}_2$. Additionally, we know that after the first verification event, all errors are contained in Viper Server state vss'. Therefore, the cached flag is set true for all errors returned by Viper Server in the second verification. Additionally, according to the cache consistency lemma, all these errors have the corresponding up to date diagnostics stored in the diagnostics cache dc'. As stated in Algorithm 2, these diagnostics are directly retrieved to construct the diagnostics d_1. Thus, we have $\mathsf{d}_1 = \mathsf{d}'$ from which we obtain $\mathsf{d}_1 = \mathsf{d}_2$. This concludes the case $p' = p$.
If $p' \neq p$, the program identifier of the two programs differs as well. Thus, the entries in the Viper Server cache and the diagnostics cache corresponding to program p do not depend on the verification of p'. Therefore, we can rewrite the calculation sequence of d_1 as a computation sequence of length n:

$$\begin{aligned}[\mathsf{jq}_{\mathsf{init}}, \mathsf{dc}_{\mathsf{init}}, \mathsf{d}_{\mathsf{init}}, \mathsf{vss}_{\mathsf{init}}] &\xrightarrow{s_n} [\mathsf{J} :: \mathsf{jq}, \mathsf{dc}, \mathsf{d}, \mathsf{vss}] \\ &\xrightarrow{\mathsf{verifyProgram}} [\_, \_, \mathsf{d}_1, \_]\end{aligned}$$

Applying the induction hypothesis to this sequence yields $\mathsf{d}_1 = \mathsf{d}_2$ directly. This concludes the case $p' \neq p$.

If EVENT denotes a job enqueue event, the verification equivalence holds trivially since the enqueueing of a job does not change any of the state used for verification. Thus, the constructed diagnostics do not change if an enqueue job event is inserted into the sequence s_n.

In the last case, EVENT is a file change event containing some file changes. In this case, p is obtained by applying the file changes to some Gobra program p'. As defined previously, the diagnostics cache and the currently calculated diagnostics are updated whenever a file change arrives. These file changes are applied to the diagnostics according to the translation rules defined in Section 4.1. Since the diagnostics are translated for every file change accordingly, applying a file change and verifying a file is associative. Meaning that applying a file change first and then verifying the file will result in the same diagnostics as verifying the file and then applying the file change. Therefore, we can reformulate the calculation of d_1 in our goal $P(n+1)$ as:

$$
\begin{aligned}
[\mathsf{jq}_{\mathsf{init}}, \mathsf{dc}_{\mathsf{init}}, \mathsf{d}_{\mathsf{init}}, \mathsf{vss}_{\mathsf{init}}] &\xrightarrow{s_n} [\mathsf{Job}(p', \mathsf{ViperServer}) :: \mathsf{jq}, \mathsf{dc}, \mathsf{d}, \mathsf{vss}] \\
&\xrightarrow{\mathsf{verifyProgram}} [\mathsf{jq}', \mathsf{dc}', \mathsf{d}', \mathsf{vss}'] \\
&\xrightarrow{\mathsf{fileChange(pos,text)}} [\_, \_, \mathsf{d}_1, \_]
\end{aligned}
$$

where we first verify the program p' and then apply the file changes to obtain program p.
Additionally, we can reformulate the computation of d_2 in our goal $P(n+1)$ by first inserting an intermediate file change event before the verification event. Then, we apply the associativity of file changes and verifications to obtain:

$$
\begin{aligned}
[\mathsf{Job}(p', \mathsf{SingleRun}), \emptyset, \emptyset, \mathsf{vss}_{\emptyset}] &\xrightarrow{\mathsf{verifyProgram}} [\_, \_, \mathsf{d}'_2, \_] \\
&\xrightarrow{\mathsf{fileChange(pos,text)}} [\_, \_, \mathsf{d}_2, \_]
\end{aligned}
$$

With this reformulation, we can apply the induction hypothesis and get that d' is equal to d'_2. Since in both cases the same file changes get applied to the diagnostics, it holds that $\mathsf{d}_1 = \mathsf{d}_2$ concluding the last case and thus the induction step.

Chapter 5

Advanced IDE Features

In order to facilitate the development and verification of Go programs, the following features are offered in Gobra IDE. Some of these features simply improve the usability of the IDE whereas others add new functionality to Gobra IDE.

5.1 Automatic Gobra Plugin Installation

The installation of Gobra IDE with all its dependencies is split into two parts: Firstly, the Gobra IDE client is installed using the Visual Studio Code Extension Marketplace [11]. Gobra IDE can now be activated by simply opening a Gobra or Go file. On activation, Gobra IDE checks whether the dependencies are already installed or not. Whenever the tools are not installed, Gobra IDE performs the second installation part and installs the Gobra dependencies. These utility tools include the Z3 theorem prover, the Boogie program verifier, and the Java Archive (JAR) containing the compiled Gobra Server. The utility tools are assumed to be downloadable from a server specified by some uniform resource locator (URL). The URL and installation folder of the dependencies is specified in the settings of Gobra IDE.
After the installation, a reinstallation of the Gobra utility tools can manually be initiated by the user via the command palette. During reinstallation the installed tools are overwritten by the current version of the Gobra utility tools available at the URL.

Since Gobra is actively under development, new features are added continuously. These newly implemented features are included in nightly builds of the Gobra utility tools. Gobra IDE allows switching between stable and nightly builds in the settings. Upon switching, Gobra IDE downloads the corresponding version. Afterwards, the IDE has to be restarted and the selected build of the Gobra utility tools will be used.

5.2 Syntax Highlighting

Modern IDEs provide syntax highlighting for common programming languages. This aids the development and readability of programs. In Visual Studio Code, syntax highlighting is implemented using TextMate grammars [10], which tokenizes the source code. Each token gets assigned a scope defining the context of this token. Visual Studio Code then maps the scopes to colors and assigns each token the corresponding color.
Since Gobra is an extension of the Go language, an extension of the grammar for Go [9] is used in Gobra IDE. The Gobra grammar includes Gobra specific keywords.

5.3 Verification Process Display

During a verification, Gobra IDE displays the verification progress in the Visual Studio Code toolbar. As shown in Figure 5.1, the progress is displayed in percent and as visual progress bar.
To calculate the overall progress of the verification in Gobra, the verification steps are split into preprocessing and Viper verification. Preprocessing is composed of parsing, type checking, desugaring, and Viper encoding. Each of these steps contribute an equal amount to the verification progress. Resulting in the following computation of the preprocessing progress:

$$\begin{aligned}\mathsf{progress}_{\mathsf{preprocessing}} = {} & 0.25 \cdot \mathsf{didParse} + 0.25 \cdot \mathsf{didTypeCheck} \\ & + 0.25 \cdot \mathsf{didDesugar} + 0.25 \cdot \mathsf{didViperEncode}\end{aligned}$$

where didParse, didTypeCheck, didDesugar, and didViperEncode are either 0 or 1.

The progress during the Viper verification is calculated as the fraction of completed Viper entities divided by the number of total entities. A Viper entity is either a method, function, or predicate.

$$\mathsf{progress}_{\mathsf{verification}} = \frac{\mathsf{verifiedMethods} + \mathsf{verifiedFunctions} + \mathsf{verifiedPredicates}}{\mathsf{totalMethods} + \mathsf{totalFunctions} + \mathsf{totalPredicates}}$$

Gobra IDE then calculates the overall progress as a weighted sum of the preprocessing progress and the Viper verification progress. In the current implementation of Gobra IDE, preprocessing and the Viper verification account 25% and 75% to the overall progress, respectively.

$$\mathsf{progress}_{\mathsf{overall}} = 0.25 \cdot \mathsf{progress}_{\mathsf{preprocessing}} + 0.75 \cdot \mathsf{progress}_{\mathsf{verification}}$$

Figure 5.1: Verification Progress Bar

The completion of the different steps in the verification is tracked using result streaming from Gobra. With this information and the above formula, a rough estimate of the actual overall progress is calculated. This progress estimate is then displayed to the user.

5.4 Goifying and Gobrafying

Since Gobra programs include annotations, tools for normal Go programs such as a Visual Studio Code plugin for Go, and the Go compiler cannot be used. To tackle this issue, Gobra IDE includes Goifying and Gobrafying of Gobra and Go programs, respectively. In Goifying, a syntactically correct Gobra program gets translated into a Go program. This translation is performed by commenting out the Gobra annotations. Contrary, the translation of a Go program to a Gobra program is called Gobrafying. Hereby, the Gobra annotations are recovered from the commented out annotations in the Go program.
Gobra annotation comments are distinguished from normal comments in the Go program by adding an @. In particular, single line comments start with //@. Conversely, multiline annotation comments start with /*@ and end with @*/. Figure 5.2 shows an example of a Gobra program on the left with the program resulting from Goifying on the right.

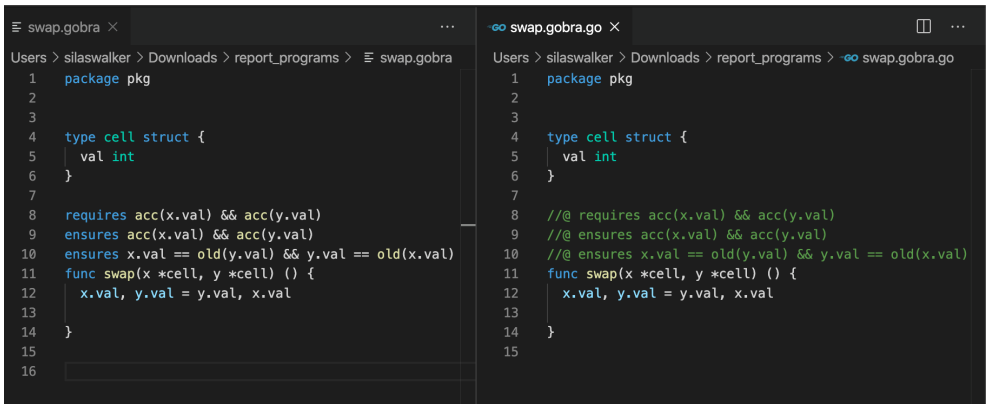

Figure 5.2: Gobra Program (left) with result from Goifying (right)

Both of these actions are invoked manually using the command palette. On completion of the Goifying or Gobrafying, the translated program is opened by Visual Studio Code automatically.

Internally, the Goifying is done using a pretty printer. This pretty printer is invoked on the AST representation of the Gobra program. The pretty printer classifies parts of the program either as Gobra annotations or actual code. Actual code is printed as usual, whereas annotations are printed inside commented out sections. The Gobrafying is done using a regex to detect the commented out annotations. These annotations are then put back into the program to recover the Gobra program.

5.4.1 Verification of Go Files

The aforementioned translation between Gobra and Go programs enables the verification of Go programs with commented out ghost code. In order to verify this Go program, the user initiates the Gobrafying. On completion of the Gobrafying, the Gobra program gets opened and is automatically verified.
The Gobrafying of a Go program can be skipped as Gobra IDE allows to manually invoke a verification on a Go program via the command palette. Gobra IDE internally invokes a Gobrafying of the Go program. Afterwards, the program is verified. Gobra IDE then displays the verification errors in the Go program.
Gobra IDE over approximates error locations to avoid complex translations of error locations: The character position of each error is extended to the whole line. All transformations that Gobrafying performs preserve the line numbering as neither lines are inserted nor deleted. Hence, errors point to the correct line after applying this approximation. Further information helping the programmers to locate the issue within the indicated line are provided in the error description.

5.5 Cross-Language Feature

As explained in Section 2.1, Gobra internally translates the Gobra program to different intermediate representations. Particularly interesting are the internal representation and the Viper encoding. The Cross-Language feature allows the display of the internal representation or the Viper encoding of a Gobra program snippet. With this feature, the detection of bugs during the translation of Gobra is simplified. The Cross-Language feature is invoked by selecting some Gobra program text and choosing which of the intermediate representations should be displayed. The intermediate representation is then shown in a read-only mode in Visual Studio Code. The parts of the intermediate representation corresponding to the selection in the Gobra program are highlighted. With this, relevant parts in the intermediate representation can be located quickly.

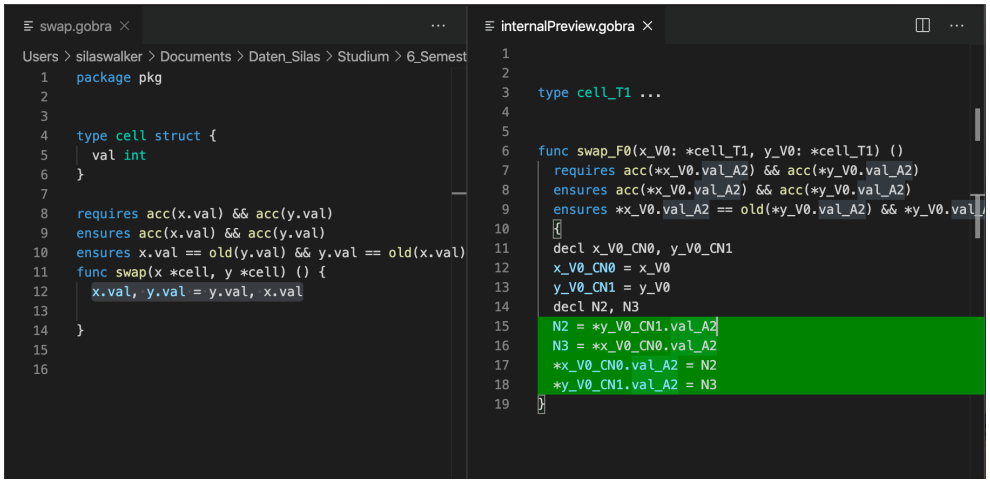

Figure 5.3: Gobra IDE's Cross-Language Feature in action: A Gobra program (left) with its internal representation (right) are shown side-by-side. Highlighted in green is the translation of the selected Gobra code (line 14 in the left panel).

5.6 Performance Improvement

Section 2.1 describes the verification in Gobra. The two main phases are denoted as preprocessing and Viper verification. During preprocessing, the Gobra program is encoded into a Viper AST. This AST is then verified in the Viper verification. In order to improve the performance of Gobra IDE, the verification pipeline is split into two pipelines. This allows Gobra IDE to concurrently preprocess verification requests. The resulting Viper ASTs are then enqueued into an intermediate job queue. These Viper ASTs in the job queue are then verified by Gobra Server sequentially.
For further performance improvements, multiple backend verifiers could be run in parallel. However, this is currently not possible since Silicon has some globally shared internal state.

Chapter 6

Evaluation

To evaluate Gobra IDE, it is compared against the already existent Viper IDE and the command line interface of Gobra. The comparison against Viper IDE is qualitatively performed on the offered features. In contrast, the comparison against the command line interface, takes usability and performance into account. The usability analysis is focused on the workflow for the verification in both Gobra IDE and Gobra as command line tool.

6.1 Comparison of Visual Features to Viper IDE

In this visual comparison of Viper IDE [14] and Gobra IDE, the visual features present in both IDEs are compared. The compared features are the following: (1) The verification error display, (2) the verification progress bar, (3) the settings interface, and (4) the debugging mode.
The first two visual features are similar in both IDEs. Verification errors are displayed by both IDEs directly in the code. As additional information, the IDEs use the toolbar of Visual Studio Code to display the overall verification result. During the verification, Gobra IDE and Viper IDE display the verification progress in a similar fashion with a progress bar. However, Gobra IDE does not take into account the verification time of previous verifications as it is done in Viper IDE.

The visual features (3) and (4) differ more between the two IDEs. Let's first consider the settings interface in Visual Studio Code. Figures 6.1 and 6.2 show the respective interfaces of Gobra and Viper IDE. The settings for Viper IDE have to be made exclusively in the *settings.json* file. Contrary, Gobra IDE allows the configuration of most settings directly over a visual interface. This allows a user to configure the IDE without any knowledge of the format in which *settings.json* has to be.
Lastly, the debugging mode of the IDEs is compared. Viper IDE supports

a debugging mode allowing the user to see the internal state leading to verification errors. This allows the user to identify the reason of verification errors by displaying for example the execution trace leading to the problem. This feature is currently not supported in Gobra IDE.

6.2 Comparison to Command Line Interface

6.2.1 Usability

To evaluate the usability, the verification workflow between the command line interface and the IDE integration is compared. In both cases a user wants to develop a functionally correct Go program using Gobra. The workflow using the command line interface looks as follows: Firstly, the specifications are added to the Go program. The resulting Gobra program is then verified using the command line interface. This interface returns the verification results. Whenever the verification is not successful, the errors are displayed on the command line. Then, the user has to manually locate the errors in the program text and fix them according to the information given in the error messages.
Conversely, the worklow of the IDE integration is the following: The program is opened in Visual Studio Code triggering the activation of Gobra IDE. After the activation, the program is automatically verified. When specification is added to the program, Gobra IDE starts a new verification. Hence, the user sees without any button press whether verification of the currently opened program succeeds or not. On failure of the verification, the errors are highlighted directly in the program text simplifying the localization of errors.

6.2.2 Performance Benchmark

As discussed in section 4.2, Gobra IDE caches verification results. Additionally, Gobra IDE is expected to have some performance improvement compared to the CLI. This performance improvement results from Gobra IDE reusing the java virtual machine (JVM). Conversely, the CLI restarts the JVM for each verification invocation. To quantify the impact of these improvements, Gobra IDE and Gobra on the command line are benchmarked in three different scenarios: In the first scenario single verifications of files with increasing program size are performed. Thus, the performance impact parameterized on the file size is measured. As second scenario, the performance parameterized on the amount of errors is measured. This is done by single verifications with increasing amounts of errors. The last scenario is used to model the actual use case of the IDE. In this scenario, a file is verified. Then, different change sets are applied to the file. After each ap-

plication of a change set, the changed file is reverified. In particular, four different change sets are applied in this benchmark. In the first two scenarios, the time of a single verification is measured. Conversely in scenario three, the times of all five verifications are measured and summed up.
For the evaluation of the first two scenarios and third scenario, four and three different file sets have been used, respectively. The file sets, denoted as F1, F2, F3, and F4, used in the scenarios are the following: File sets F1 and F4 consist of many small functions and predicates. The complexity of the functions is comparable with the swap function introduced and used throughout this report. File set F2 contains few methods which each consist of nested if statements. The remaining file set, namely F3, consists of numerical functions.
As described in the first paragraph, each scenario has one parameter which is varied in the benchmark. Each of these parameters is evaluated for three different values, namely small, medium, and large. The meaning of the values in each scenario are: In the first scenario, the values of the parameters approximately correspond to the verification time. For the second scenario, the parameters correspond to the number of errors contained in the programs. The parameters in the third scenario correspond to the number of functions being changed in a file change. Since the total number of methods varies between the four file sets, the exact values of the parameters vary between the file sets as well.
All experiments were performed with Silicon as backend verifier on a 3.4 GHz machine with six cores and 32 GB of memory running Windows 10 Professional 64-bit. Each experiment was executed five times and the medians are shown in table 6.1.

It can be seen that scenarios one and three yield a performance gain of Gobra IDE compared to the command line interface. Contrary, in scenario two the IDE has a significant performance overhead compared to the CLI. From the evaluations with different file sizes, one can conclude that the performance gain received from the reuse of the JVM is substantial. In particular, the verification time is decreased by roughly five seconds when using the IDE.
A significant overhead of the IDE is seen when the amount of errors is increased. In order to find the reason of this overhead, the verification time and time to cache the results was measured individually. This led to the observation, that the actual verification time is similar to the verification time of the CLI. The overhead resulted exclusively from the caching of many verification errors. Additionally, our observation is supported by the fact that the overhead is not as significant for file sets F2 and F3. This is because those file sets do not contain as many functions as the other ones.
The performance gain resulting from Viper Server cache is seen in the third scenario. In this scenario, the observation from the first scenario has to be taken into account, i.e. each verification in Gobra IDE is roughly five

		Small		Medium		Large	
		LOC		LOC		LOC	
Size	F1	290	4.27 / 9.64	570	8.78 / 14.29	710	11.29 / 16.46
	F2	215	5.14 / 9.27	777	11.61 / 17.39	1031	16.43 / 22.16
	F3	24	1.52 / 5.71	49	9.83 / 14.31	77	10.64 / 15.43
	F4	57	10.02 / 14.12	458	15.71 / 19.86	931	26.15 / 30.61
Errors	F1	711	23.64 / 16.36	711	49.78 / 16.89	711	65.1 / 16.39
	F2	1031	17.01 / 22.35	1031	18.28 / 21.7	1031	25.88 / 22.35
	F3	77	13.05 / 15.35	77	11.19 / 15.42	77	4.26 / 7.37
	F4	928	45.01 / 22.53	928	78.41 / 22.38	928	107.93 / 22.96
Changes	F1	710	52.32 / 85.69	710	58.66 / 86.14	710	61.83 / 89.87
	F2	1031	72.68 / 112.09	1031	84.51 / 111.02	1031	83.36 / 109.83
	F3	77	20.34 / 77.44	77	46.73 / 79.78	77	40.6 / 81.18

Table 6.1: Results from performance benchmark of Gobra IDE and Gobra on the command line in three scenarios denoted by Size, Errors, and Changes. The different file sets are denoted by F1, F2, F3, and F4. Verification times resulting from the evaluations are displayed in seconds (Gobra IDE / Gobra CLI).

seconds faster than with the command line tool. Hence, the performance gain resulting from caching verification results is roughly 25 seconds less than one might think solely based on the results of the third scenario. Taking this into account, one can see that in the case of small file changes, Gobra IDE benefits from result caching. Additionally, the file set F4, containing numerical functions, benefits from caching for each file size.

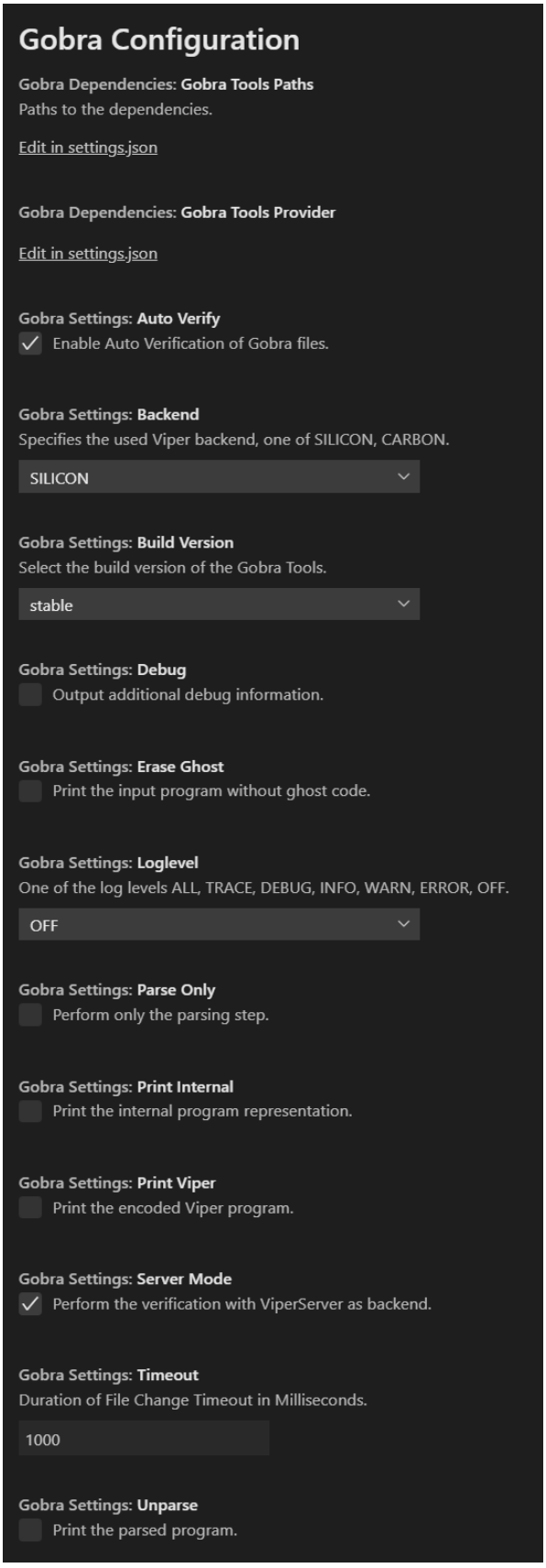

Figure 6.1: Settings Interface for Gobra IDE

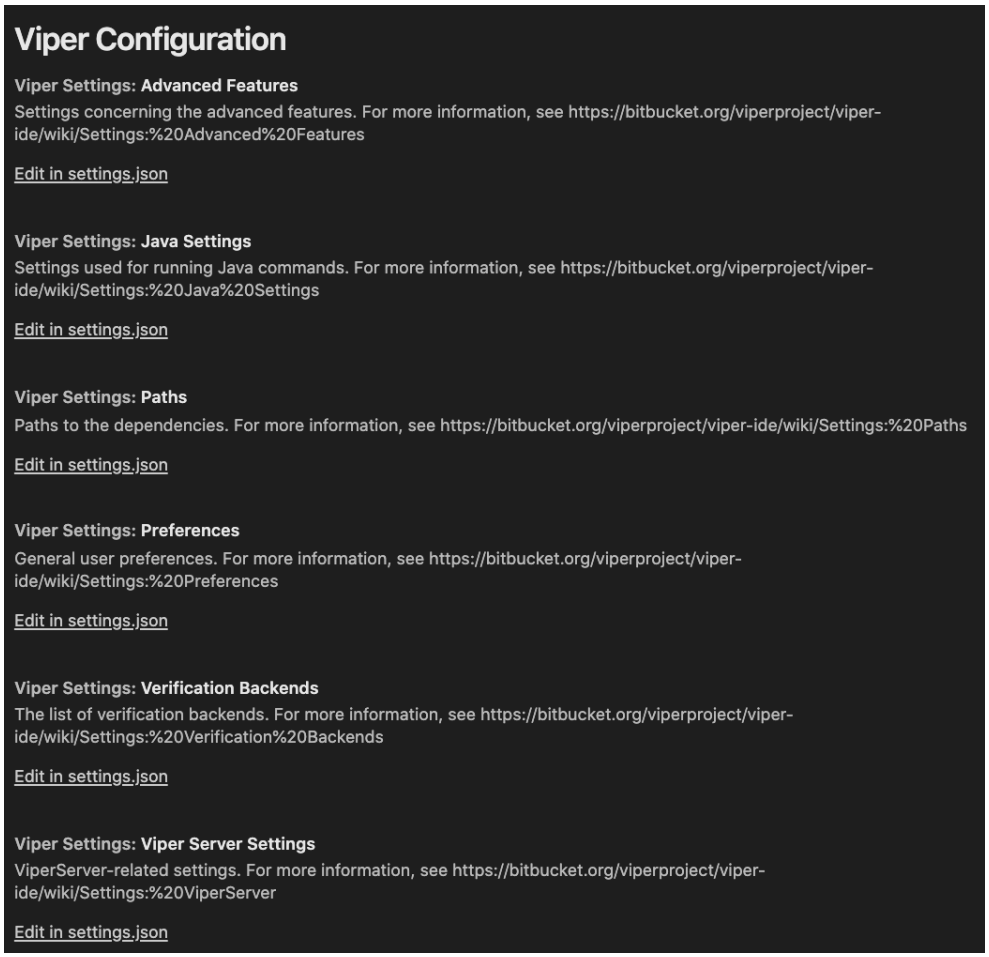

Figure 6.2: Settings Interface for Viper IDE

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The main focus of this project was to develop a Visual Studio Code plugin for Gobra. This plugin aids the development and verification of Go programs with proof annotations. During development, the IDE invokes verifications automatically. Verification results obtained by verifications are displayed conveniently to the user in the editor.
The usability of Gobra IDE was improved by adding caching of intermediate verification results with Viper Server. This caching of verification results led to the IDE displaying outdated positions of errors. To overcome this issue, a mapping from errors to diagnostics was introduced. To assure the caching does not change the behavior of Gobra, a reasoning about the soundness of verifications in Gobra IDE with caching is provided. Further performance improvements, exhibiting the potential parallelism during the preprocessing in Gobra, were added to Gobra IDE.
Additional advanced features were added to Gobra IDE. These features improve the usability of the plugin. These features are the following: The automatic plugin installation, syntax highlighting, the verification progress bar, Goifying and Gobrafying, and the Cross-Language feature. Some of these features improve the usability of the plugin when it is used in an everyday setting. Conversely, other features support the further development of Gobra itself.

7.2 Future Work

Many of the planned features were included in Gobra IDE. Some of features were not covered by this thesis. The following are some examples of features possibly improving the user experience of Gobra IDE:

- Code navigation possibly improves the productivity while working with Gobra IDE. This navigation includes features such as going to a definition or implementation. Additionally, Gobra IDE could provide further code analysis by showing lemmas which can be derived from the current program.
- Gobra IDE currently provides no possibility to intercept verification jobs. This is critical, since the backend verification is performed sequentially. In order to avoid long running verification jobs blocking short verification jobs solutions include: (1) Remove the shared memory state from Silicon allowing verification jobs running completely in parallel. (2) Enable aborting verification jobs in Gobra IDE.
- Verification of larger Gobra programs leads to a significant overhead caused by parsing, type checking, desugaring, and Viper encoding. This overhead can be reduced by introducing a cache on the Gobra level.

Bibliography

[1] Viper: A verification infrastructure for permission-based reasoning. pages 41–62.

[2] *Carbon*, 2020 (accessed August 3, 2020). https://github.com/viperproject/carbon.

[3] *Go Programming Language*, 2020 (accessed August 3, 2020). https://golang.org.

[4] *Silicon*, 2020 (accessed August 3, 2020). https://github.com/viperproject/silicon.

[5] *Visual Studio Code*, 2020 (accessed August 3, 2020). https://code.visualstudio.com.

[6] *Language Server Protocol*, 2020 (accessed July 15, 2020). https://microsoft.github.io/language-server-protocol/.

[7] *VS Code API*, 2020 (accessed July 15, 2020). https://code.visualstudio.com/api/references/vscode-api.

[8] *Boogie Program Verifier*, 2020 (accessed July 30, 2020). https://github.com/boogie-org/boogie.

[9] *Go TextMate Language*, 2020 (accessed July 30, 2020). https://github.com/microsoft/vscode/blob/master/extensions/go/syntaxes/go.tmLanguage.json.

[10] *Language Grammars*, 2020 (accessed July 30, 2020). https://macromates.com/manual/en/language_grammars.

[11] *Visual Studio Code Marketplace*, 2020 (accessed July 30, 2020). https://marketplace.visualstudio.com/vscode.

[12] *Z3 Theorem Prover*, 2020 (accessed July 30, 2020). https://github.com/Z3Prover/z3.

[13] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 1–16, Cham, 2014. Springer International Publishing.

[14] Ruben Kälin. Advanced features for an integrated verification environment. 2016.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

IDE Support for a Gdang Verifier

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Walker	Silas

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Schattdorf, 01.09.2020	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.