Universal Library Components for Verification IDE Development

Bachelor's Thesis

Valentin Racine

23. OCTOBER 2020

Supervisors: Arshavir Ter-Gabrielyan, Prof. Dr. Peter Müller

Department of Computer Science, ETH Zürich

Abstract

In recent years, verification tools have begun to be integrated into development environments. Verification tools have greatly profited from this, as it drastically improves the useability of the tools in question. In this thesis, we want to continue this trend by facilitating the development of such tools. A key step to this consists of refactoring existing verification tools built atop the Viper verification infrastructure. In doing so, we aim to make the underlying infrastructure more modular, reusable, and maintainable. This should in turn facilitate the development of more interactive and user-friendly verification tools. In particular, it should lead to the development of tools that can be used from within popular code editors, such as VS Code.

Table of Content

1. Introduction 1
  1.1 Tools 1
    1.1.1 Verification Infrastructures 1
    1.1.2 Viper 1
    1.1.3 Viper Frontends 2
    1.1.4 Verification Servers 2
    1.1.5 Verification IDE 3
    1.1.6 ViperServer 3
    1.1.7 Viper IDE 4
  1.2 Modularity 6
    1.2.1 Verifying Viper Programs 6
    1.2.2 Developing Verification Servers 7
    1.2.3 Language Server 7
  1.3 Content 8
    1.3.1 Outline 8
    1.3.2 Definitions 8
2. ViperCoreServer 9
  2.1 Introduction 9
    2.1.1 Proposition 9
    2.1.2 Overall Approach 9
    2.1.3 Order of Refactoring 9
  2.2 Design 10
    2.2.1 Prerequisites 10
    2.2.2 API 10
    2.2.3 Options, Configurations 12
    2.2.4 How does VCS work internally? 14
    2.2.5 Refactoring ViperServer 14
  2.3 VCS in Practice 16
    2.3.1 Example use cases 16

2.3.2 Erroneous use of the VCS API ... 16
2.3.3 Use Cases ... 17
2.4 Evaluation ... 19
2.4.1 Unit tests: VCS ... 19
2.4.2 Unit Tests: AstGenerator ... 22
2.4.3 Integration Testing ... 23
2.4.4 Test Scenario: Viper IDE ... 23
2.4.5 Test Scenario: Gobra ... 24
2.4.6 Achieved Goals ... 24
2.5 Discussion ... 25
2.5.1 Summary ... 25
2.5.2 Challenges ... 25
3. Verification Server Interface ... 26
3.1 Introduction ... 26
3.1.1 Proposition ... 26
3.1.2 Approach ... 26
3.1.3 Chronology ... 27
3.2 Process Management ... 28
3.2.1 How is Process Management implemented? ... 28
3.2.2 Identifying language-(in.-)dependent features ... 30
3.2.3 Design ... 30
3.2.4 Refactoring of VCS ... 33
3.3 HTTP ... 34
3.3.1 Identifying common, language-dependent and language-independent parts ... 34
3.3.2 Design ... 34
3.3.3 Refactoring of ViperServer ... 36
3.4 Cache ... 38
3.4.1 Caching in Verification Servers ... 38
3.4.2 Design: Overview ... 39
3.4.3 Design: Implementation ... 41
3.4.4 Refactoring of ViperCoreServer ... 42
3.5 VSI in Practice ... 44
3.5.1 General Use Cases ... 44
3.5.2 Specific Use Cases ... 44
3.6 Evaluation ... 45

3.6.1 Unit Tests 45
3.6.2 Test Scenarios: Viper(Core)Server 45
3.6.3 Achieved Goals 45
3.6.4 Further Work 46
3.7 Discussion 47
3.7.1 Summary 47
3.7.2 Challenges 47
4. Language Server 48
4.1 Introduction 48
4.1.1 Proposition 48
4.2 Technologies 50
4.2.1 JSON-RPC 2.0 50
4.2.2 LSP 50
4.2.3 LSP4J 51
4.3 Redesigning the VS Code extension 53
4.3.1 Client 53
4.3.2 Language Server Sender/Receiver 54
4.3.3 Server State 54
4.3.4 Verification in the Language Server 55
4.3.5 Consumer Actor 55
4.4 Evaluation 56
4.4.1 End-to-End Tests 56
4.4.2 Achieved Goals 56
4.4.3 Further Work 57
4.5 Discussion 58
4.5.1 Summary 58
4.5.2 Challenges 58
5. Conclusion 59
5.1.1 ViperCoreServer 59
5.1.2 Viper IDE 59
5.1.3 VSI 60
6. References 62

1. Introduction

This thesis is based on several tools and concepts related to the Viper project. In this chapter, we will start by introducing these tools and concepts. We will then give an introduction to the problems that this thesis sets out to solve.

1.1 Tools

1.1.1 Verification Infrastructures

Modern verification infrastructures usually follow a similar structure. Traversing it top-down (or front-to-back), looks like this. We start with a program in some programming language, along with the specifications that are to be verified. First, both the program and specifications are translated to an intermediate language. A popular example of such a language is Boogie [1]. This part of the toolchain is typically referred to as a verifier's frontend. Next, the program's intermediate representation is passed to a verifier's backend to generate verification conditions. Contrary to the original and intermediate representations, these conditions are usually expressed as logical formulas. Ultimately, the un-/satisfiability of these formulas tell us whether or not a program and its specifications hold true. Evaluating these formulas is done by a family of tools called satisfiability modulo theories (SMT) solvers. A typical example of such a tool is Z3 [2].

The idea behind an architecture with an intermediate language is to provide common ground for verification tools. Ideally, by using the intermediate language, any programming language should be able to make use of the backends and SMT solvers. Doing so comes at the sole cost of implementing a translation from that language to the intermediate language.

1.1.2 Viper

This thesis is based on a stack of verification tools, ranging from SMT solvers to IDE plug-ins, that follows a similar architecture as previously described. At its core lies the infrastructure called Viper [3]. Viper comprises an eponymous intermediate language as well as two backends called Silicon and Carbon, as can be seen in Figure 1.

Contrary to Boogie, which is based on first-order logic, Viper is designed to support permission logics such as separation logic. As such, Viper allows to specify permissions to, and ownership of memory locations, including memory on the heap. This can be used to reason about programs that have a mutable state as well as programs with concurrent threads of execution. Along with its support for other types of logics, an important feature of the Viper intermediate language its design. On the one hand, the language offers lots of high-level concepts found in other programming languages. This makes it convenient to use as a programming language and express verifications in Viper. On the other hand, it provides enough low-level features to facilitate the translation of other programming languages to it.

As mentioned, Viper is powered by the two backends Silicon and. The backends are, broadly speaking, responsible for turning a Viper program into a set of logic formulas that are ultimately solved

by Z3. Carbon works by encoding a Viper program into Boogie, thereby generating the verification conditions necessary for the SMT solver. Silicon takes another approach. It uses Symbolic Execution to directly generate the conditions that are ultimately passed to Z3.

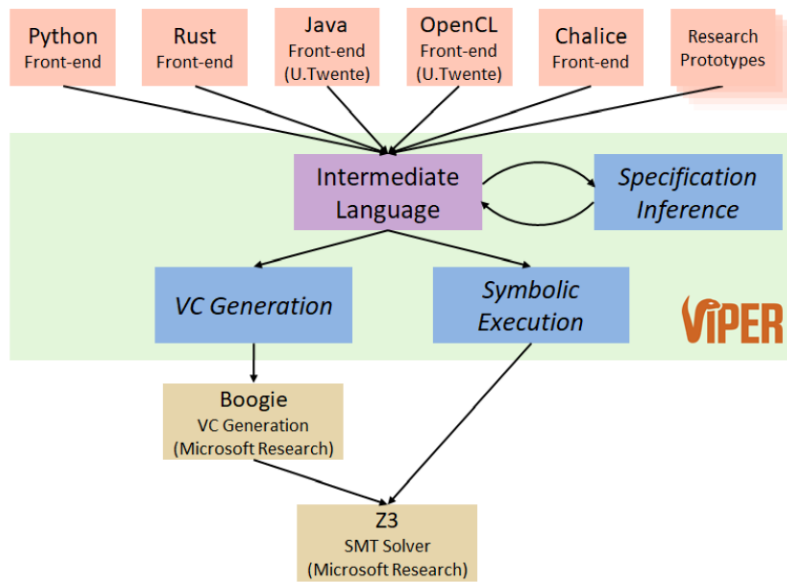

Figure 1 Viper Toolstack.

1.1.3 Viper Frontends

In general, an intermediate language allows for a modular composition of frontends and backends. Since Viper is an intermediate language, this principle applies here, too. Concretely, this is done by implementing a frontend that provides a principled translation from an arbitrary programming language to Viper. Given such a mechanism, any program in that language can be translated into a Viper program. In turn, such a program can be verified by the Viper infrastructure, using any of the available backends. As of today, there have been numerous frontends developed for various languages on top of Viper. Some examples are of Viper frontends are given in Figure 1.

1.1.4 Verification Servers

In the context of Viper, verification servers are servers that embed Viper frontends and provide access to Viper's verification tools. The aim of bundling these tools into a server is to improve both usability and performance.

Running verification as part of a server also allows us to add additional features. For example, most servers provide an HTTP frontend. This means that the server can be attached to other processes. These processes can range from simple HTTP clients to IDE plug-ins that provide support for automatic program verification. Another feature commonly found in verification servers is a caching mechanism. This allows storing verification results, to potentially avoid reverifying future programs. Successfully implementing a caching mechanism can improve performance, as verification is a costly operation.

1.1.5 Verification IDE

Verification servers allow Viper frontends and the Viper verification tools to be attached to other processes. In particular, they may be attached to IDEs. We refer to IDE plug-ins (or "extensions") that do so as Verification IDEs.

Verification IDEs effectively allow integrating verification tools into a code editor. This can greatly improve the user experience of working with verification tools. The reasons for this are two-fold. Firstly, it provides a solid user interface. This eliminates the need to deal with CLIs and the manual input of options, file names, etc. Secondly, verification can be automated. I.e., verification requests can be triggered on file changes, on opening files, etc.

Currently, some verification IDEs have been developed for various Viper frontends. Gobra IDE [4] and Prusti assistant [5], for example, are verification IDEs developed for the frontends Gobra [6] and Prusti [7].

1.1.6 ViperServer

ViperServer was designed to make the Viper toolstack available to clients via HTTP. It is an application written in Scala and therefore runs in a JRE. As such, ViperServer provides a non-terminating application with a persistent state. This allows it to incorporate additional features that complement the Viper toolstack, as can be seen in Figure 2. Aside from communication via HTTP, ViperServer also offers a process management functionality and a cache.

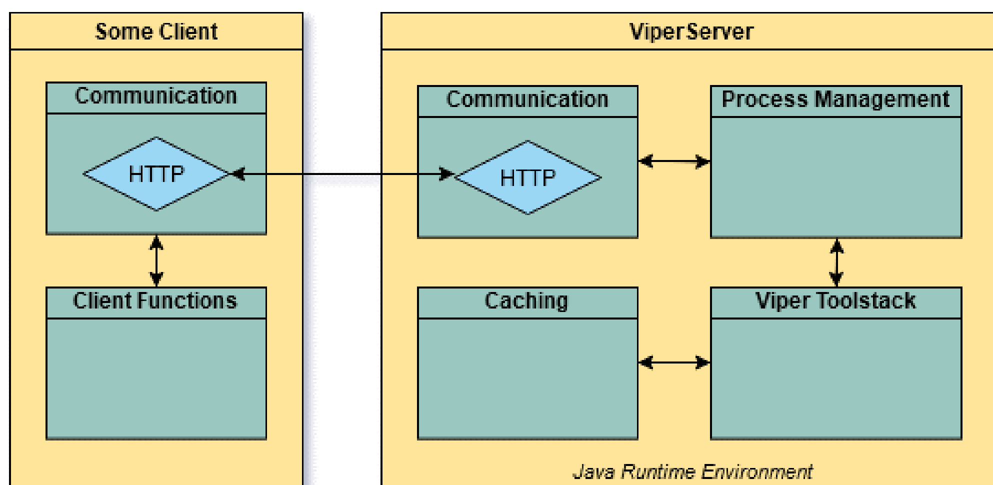

Figure 2 Overview of the functional components that make up ViperServer. ViperServer itself connects to a generic client using HTTP.

ViperServer is an asynchronous application. While this does not strictly require concurrency, ViperServer nonetheless profits from being a concurrent application. Concurrency allows verification requests to be executed as independent and parallel processes. The current process management system is based on an implementation of the Actor model provided by Akka's Actor Library.

This yields a flow of operations as depicted in Figure 3. Upon receiving a verification request, the server creates a dedicated process that runs the verification. Inside the process, the file in question is passed to either of the backends provided by the Viper infrastructure. Currently, the backends take care of parsing, type-checking, and verification. The results are then retrieved from the backends and reported using dedicated messages.

If caching is involved, the pipeline is interlaced with two more operations. After parsing a file in the backend, the corresponding abstract syntax tree (“AST”) is retrieved before verification. Then, all the methods in the AST are checked against the cache. If the cache contains valid results for any of these methods, the results will immediately be returned. Methods that hit the cache are subsequently modified and inserted back into the AST. This is done to indicate to the verifier that these particular methods do not need to be reverified. The modified AST is then verified. Both the cached and new results are ultimately sent back to the client.

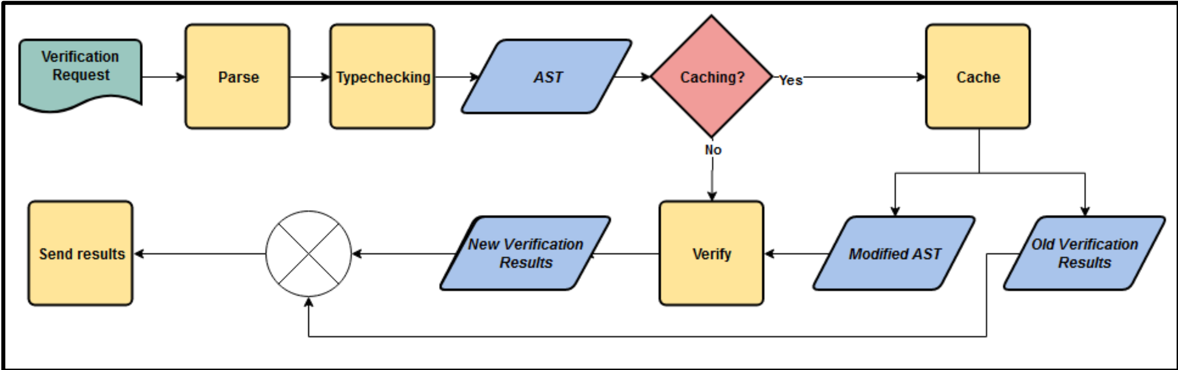

Figure 3 Pipeline of operation used when verifying a Viper.

1.1.7 Viper IDE

To further improve the usability of Viper and the underlying tools, a plugin for VS Code has been developed [8]. This plugin effectively includes ViperServer into a development environment. Therefore, it enables it to unfold its full potential by further automating processes and providing an improved interface for the user. E.g., a user will no longer have to manually issue a verification request after editing source code. Instead, the plugin will automatically detect changes, request verification, and display results.

As it is a VS Code plugin, it is written in Typescript. It is comprised of two distinct parts, as is common for such plugins. The first, called the client, is where most of the interaction with the user is specified. These are typically light-weight tasks such as taking commands from the user, detecting changes in the files, or displaying output. The second, called the language server, is where the heavy-weight tasks are performed. Examples for such tasks are code completion, code look-ups, syntax analysis, etc. The two parts of the plugin communicate using the LSP defined by Microsoft. This division of labor can be seen in Figure 4.

Naturally, a workload such as static code verification is the responsibility of the server-part of the plugin. But since we already have an application that performs the verification with ViperServer, there is no need to re-implemented it as part of the plugin's language server. Instead, the language server only acts as a sender/receiver for messages to/from ViperServer, to which the actual work is outsourced. This is done by starting an instance of ViperServer from within the plugin. Then, ViperServer's HTTP communication capabilities can be used to send verification requests and receive the corresponding results. Figure 4 also shows how ViperServer is connected to Viper IDE.

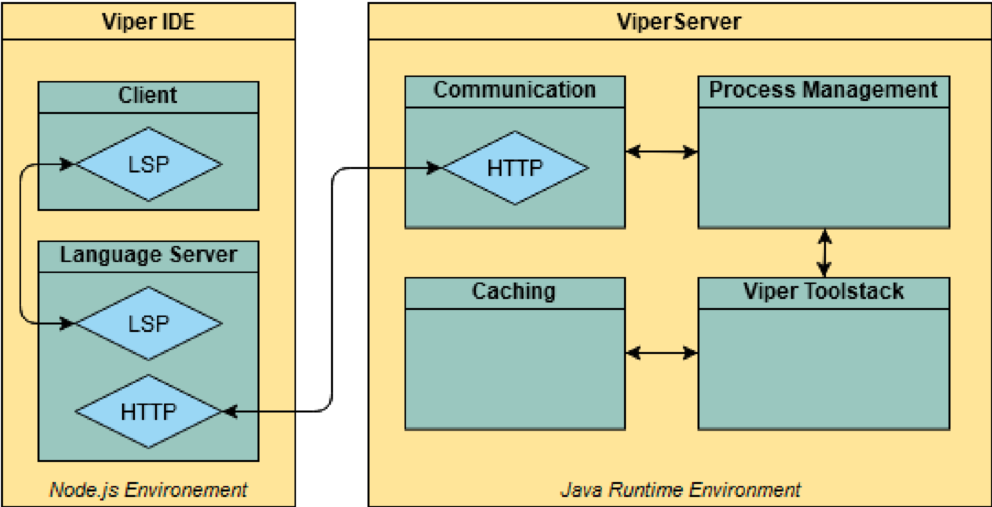

Figure 4 Viper IDE and its division into client/language server hooked up to ViperServer.

1.2 Modularity

This thesis goal is to facilitate the development of verification servers and verification IDEs. One of the ways this can be done is by making the Viper tools more modular. This should in turn make the components more reusable and maintainable, resulting in a simpler development process.

Note that an intermediate language is an inherently modular concept. This pattern has been popular in other fields, such as compiler design, for quite some time. Having an intermediate language allows implementing M frontends that can have access to N backends. This reduces the amount of work from building $M * N$ separate applications to $M + N$. Since Viper is itself an intermediate language, the same principle applies. We have already seen how Viper frontends take advantage of this. However, there is still room for improvement.

1.2.1 Verifying Viper Programs

Viper's backends come with an API that allows them to programmatically invoke them. At first glance, it might seem intuitive to let each frontend make individual calls to the backends. However, this goes against the idea of modularity provided by an intermediate language. To illustrate this, consider verification servers $S_1 \dots S_n$. If each of the frontends makes individual calls to the backends, we end up with the left diagram depicted in Figure 5.

Ideally, the situation in the right diagram of Figure 5 should be achieved. Each of the frontends $S_1 \dots S_n$ should be able to access the backend via a common API. This would allow taking full advantage of the modularity provided by the fact that Viper is an intermediate language. It would also simplify the design and the development of verification servers that are based on Viper.

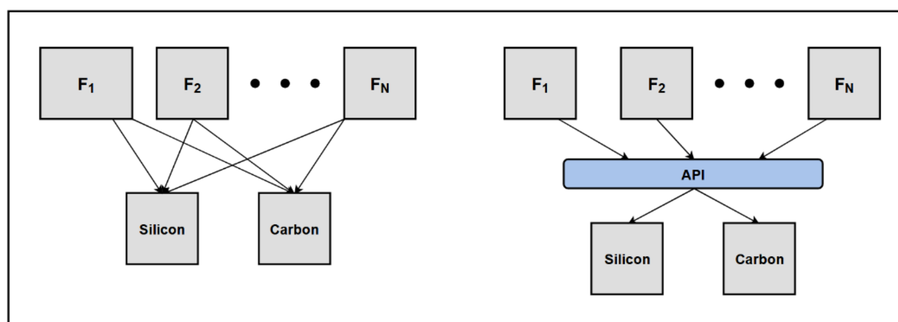

Figure 5 Additional layer of abstraction by hiding backends behind an API.

Unfortunately, such an API does not yet exist. Currently, each verification server manages its calls to the backends individually. One of the milestones of this thesis is to extract these calls to the backends and hide them behind a simple API. This will create an independent module called ViperCoreServer. With this, verification servers will be able to verify Viper programs without dealing with backends directly.

1.2.2 Developing Verification Servers

We have seen that, aside from translating programs to Viper, verification servers may implement a range of other features. For example, we have presented ViperServer, the verification server for Viper itself. Aside from verification, ViperServer provides a few additional features like communication, process management, and caching.

It stands to reason that the features in ViperServer are desirable for other frontends as well. Consequently, we should identify features ViperServer that could be used by other frontends of the Viper infrastructure. Based on these features, a generic version of a verification server could be created. Such a generic server could then be extended by the specific verification server as they need.

Take ViperServer's HTTP functionality as a concrete example. Currently, it implements a set of GET and POST methods, that know how to react to various requests. If other verification servers also want to provide HTTP functionality, they're likely to use implement the same set of GET and POST methods. Obviously, the content of messages will not be the same. This is because every verification server is built for a different programming language. Nevertheless, we should be able to provide some generic HTTP methods that verification servers can then extend for their specific use case.

Another goal of this thesis is to create a library called VerificationServerInterface ("VSI") containing generic code that implements features common among all verification servers. This should lower the cost of implementation and maintainability and therefore facilitate the development of verification servers

1.2.3 Language Server

Recall that at the moment, ViperIDE has a language server that communicates with

- The client-part of the plugin via LSP
- ViperServer via HTTP

Also, note that relaying messages between the two other parts is its only job. This suggests that we could get rid of it if we managed to make the of the plugin's client communicate directly with ViperServer. This is indeed feasible, by implementing the LSP directly as part of ViperServer.

Bypassing the current language server that acts as a relay between two other components is the last goal of this thesis. Removing this middle man will allow the plugin to be simplified down to one code base, making it easier to understand and maintain.

1.3 Content

1.3.1 Outline

In this thesis, each chapter is dedicated to one of the milestones. Each of these milestones is based on one of the problems outlined in the previous Section 1.2. In each chapter, we will

- Recap what the problem is.
- Propose a solution to it.
- Analyze prerequisites that had to be taken into account for that milestone.
- How we designed the solution to the problem it should solve.
- How it was instantiated into the existing tools.
- How it is evaluated.
- Discuss what future work this milestone could inspire.

1.3.2 Definitions

Throughout the text, we will frequently abbreviate long words and names to keep the text more readable. Table 1 gives an overview of the most frequently used abbreviations.

Table 1 Frequently used abbreviations and their meaning

Abbreviation	Meaning
VCS	ViperCoreServer
VSI	Verification Server Interface
IDE	Integrated Development Environment
JRE	Java Runtime Environment
JVM	Java Virtual Machine
API	Application Programming Interfaces
JAR	Java Archive Files.
LSP	Language Server Protocol
HTTP	Hypertext Transfer Protocol

In this thesis, we will often refer to code. In places where we have to refer to code (like the name of classes or methods) in the text, we will use the following `inline notation` as a visual indicator.

2. ViperCoreServer

2.1 Introduction

2.1.1 Proposition

ViperCoreServer (VCS) will provide the bare essentials of verifying a Viper program while still functioning as a server. The idea behind this was explained in Section 0. In a few words, every frontend and corresponding verification server needs to ultimately verify a Viper program. Letting all of them connect to the backends individually creates unnecessary costs in terms of implementation and makes maintainability more difficult. Instead, we propose to extract this common code and pack it into an independent server. Moreover, we propose to include a process management system. This will allow us to execute verifications concurrently and asynchronously. Last but not least, ViperCoreServer should include the caching infrastructure currently in place in ViperServer.

2.1.2 Overall Approach

Note that the three functional parts we want to be present in ViperCoreServer are already present in ViperServer. Hence, the first step towards VCS is to isolate and separate these three parts from ViperServer. The second step is to unite them behind an API. Afterward, the last step will be to rewire ViperServer such that it uses VCS. A diagram of this refactoring process is shown in Figure 6.

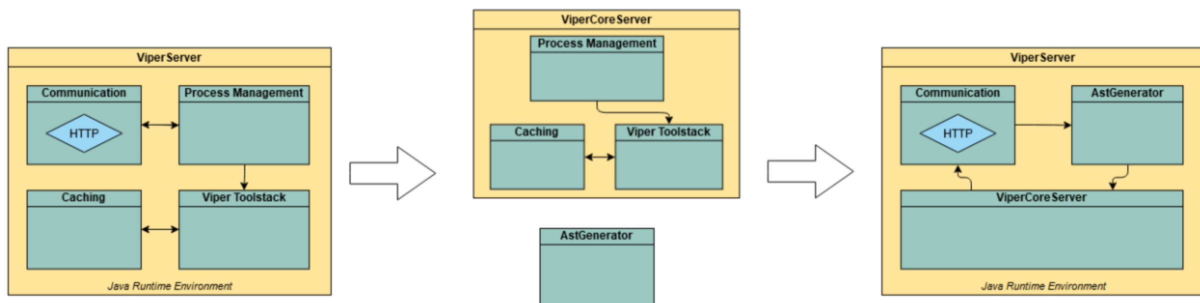

Figure 6 Overview of ViperCoreServer and ViperServer refactorization process.

2.1.3 Order of Refactoring

The reason for tackling this task first is how fundamental the verification of Viper programs is. Not only to ViperServer but also to other frontends. In particular to the Gobra team, who is currently working on a frontend and verification server for Go.

2.2 Design

2.2.1 Prerequisites

An important part of this milestone was the design of the API. Its design was governed by two basic ideas. One was to create a small and abstract API, to make the use of ViperCoreServer as simple and intuitive as possible. The other was to reflect the idea that ViperCoreServer should be able to verify a program and return verification results.

A prerequisite is that ViperServer currently is modeled to be a reactive system[9]. This is done using Akka[10], a Scala library that provides all needed primitives (actors, reactive message streams, and more). Hence, we decided to build ViperCoreServer based on Akka as well. Aside from carrying out their computations, the actors can also communicate by sending messages. Using the second library, such message passing can be implemented with reactive streams. As will become evident, this makes for an elegant, efficient, and reactive solution. However, it also poses the question of how to expose this system to a client.

2.2.2 API

From the prerequisites, five methods were identified that should constitute the interface, as can be seen in Listing 1. Table 2 describes the methods and their signatures.

Table 2 Description of VCS's API.

Method	Description
`start`	Initializes the server. This includes configuring the server, setting up a logging system, and initializing the caching system. This method must be called first.
`stop`	Stops the server. This interrupts currently running verifications. If the interruption of a process times out, the server will forcibly shut down.
`verify`	Verifies a program with the specified backend. The program must be provided as an instance of the Viper `Program` class. The `BackendConfig` specifies which backend is to be used and carries options concerning the backend. The method returns a unique integer for each new verification process that is started or -1, if such a process could not be started.

`streamMessages`	Streams the verification results reported by the backends for a particular job to an actor. The `jid` must be one of the integers previously returned by verify. Also, each `jid` can be used exactly once to retrieve results. Note that the method does not return actual verification results, as these are directly "streamed on" to the actor passed as an argument. Instead, the method returns information about whether or not the process exists (indicated by the `Option`) and whether the process is completed yet (indicated by the `Future[Unit]`).
`FlushCache`	Resets the cache's state.

Arguably, the most interesting part of the API is the `streamMessages` method. Recall that VCS relies on Actors and Streams to manage concurrent and asynchronous verification processes. While this is very useful within VCS, it is unclear how it translates outside VCS. In particular, there's a trade-off to be considered. On the one hand, the API can expose the Akka libraries, thereby passing on their advantages to the client. On the other hand, doing so forces a client to use the Akka libraries to interact with VCS, which reduces the API's flexibility and generality.

```
class ViperCoreServer(args: Array[String]) {

  //...

  def start() = {...}

  def verify(p: Program, b: BackendConfig): JobID = {...}

  def streamMessages(jid: JobID, a: Actor): Option[Future[Unit]] = {...}

  def flushCache() = {...}

  def stop() = {...}
}
```

Listing 1 API of ViperCoreServer, consisting of 5 methods.

As can be taken from Table 2, we decided that the API should expose the Actor framework to the client. There are several reasons for this.

Firstly, the Akka library offers several advantages. Namely, the Akka Streams library is reactive. This means that the flow of information between a source and a sink is dynamic. To illustrate this, consider a stream of messages between two actors. If an actor receives more messages than it can handle, it will signal the source actor to slow down its output. Conversely, the flow of messages is accelerated if the sink has more capacity. This makes for a very stable environment as it ensures that messages are being held back instead of being dropped. This is very convenient for VCS and other

verification servers, as it is crucial that messages not be lost. Exposing a stream to the client allows them to attach their own reactive streams.

Secondly, the key disadvantages of exposing an actor in the API can be mitigated. This can be seen in Listing 2. If the client wants nothing to do with Actors and Streams, it can use either of the methods provided by `ViperCoreServerUtils` to retrieve information from VCS. This will provide the client with either a list of *all* messages produced by the backends or just the verification results. Note that these methods are internally still dependent on `streamMessages`. Since `streamMessages` can only be called once for each `jid`, the same is true for either of the methods in VcsUtils.

Finally, we can expose the `actor` in the API, which is beneficial for any client who also works with the Akka libraries. Moreover, we can do so without actually forcing clients to deal with Akka specific concepts, such as actors. This is done by providing utility functions that extract the information and return them as common data structures.

```
1 class ViperCoreServerUtils() {
2
3   //...
4
5   def getMessagesFuture(core: ViperCoreServer, jid: JobID):
       List[Message] = {...}
6
7   def getResultsFuture(core: ViperCoreServer, jid: JobID):
       VerificationResult = {...}
8 }
```

Listing 2 Additional utility API to avoid the use of Akka libraries.

2.2.3 Options, Configurations

Most of the components throughout the tool stack have global configurations that are set for the entire run time of the tool. These options come in the form of command-line arguments and are set at launch. VCS is no exception in that matter, as can be seen in Table 3. Since VCS is only meant to be launched programmatically, a `List[String]` containing the options can be passed as an instantiation argument.

Table 3 Description of setting options for ViperCoreServer.

Option	Description
`LogLevel`	Sets how much of the logs the server prints to the log file. The accepted values are: "ALL", "TRACE", "DEBUG", "INFO", "WARN", "ERROR", "OFF". The default value is "ERROR"
`LogFile`	Sets the log file. By default, one will be created.
`BackendSpecificCache`	Sets whether or not to have different caches for different backends. By default, a general cache is used
`Port`	Sets the port of the server, if run ViperServer. By default, a free port is found.
`MaximumActiveJobs`	Sets the maximum number of verification processes that can be active at the same time By default, this is set to 3.

Another set of configurations must be passed when calling `verify`. These configurations specify which backend should be used and with what options. VCS comes with an object, called `ViperBackendConfigs`, which contains three case classes, as can be seen in Listing 3. Each specifies (through its type) which backend should be used. Additionally, each takes a `partialCommandLine`. This is a `List[String]` that contains the options for the backends. The Strings passed are configurations in the form of command-line arguments, following the same format as the options for VCS.

```
1 object ViperBackendConfigs {
2
3   case class SiliconConfig(partialCommandLine: List[String]) extends
       ViperBackendConfig
4   case class CarbonConfig(partialCommandLine: List[String]) extends
       ViperBackendConfig
5   case class CustomConfig(partialCommandLine: List[String]) extends
       ViperBackendConfig
6
7 }
```

Listing 3 Configuration classes for the backends that need to be passed to verify.

Most of these options are specific to the backend in use. One that is not is called `disableCaching`. This option disables the use of cache for that run. Note that in VCS, by default, the cache is always on. The decision not to use caching therefore has to be made for every new verification process. Disabling that cache means that there will be no comparison with the cache and no update of the cache.

2.2.4 How does VCS work internally?

In this second chapter, we want to focus on explaining VCS's API and its use in different contexts. We will therefore defer the explanation of implementation details to the next chapter.

2.2.5 Refactoring ViperServer

After settling on an API for VCS and refactoring the relevant parts from ViperServer into VCS, our final task is refactoring ViperServer to use VCS. Concretely, this means complementing VCS with HTTP and a module that parses Viper files.

Why is a parsing module necessary? Recall that the whole point of VCS is to verify a program that is passed as an AST. In particular, an AST passed as a Scala object at runtime, not passed as a file. A frontend, for instance, gets this program by translating a program from a source language to Viper. However, ViperServer does not translate a program from another language to Viper. Therefore, it won't have a Viper program in memory that it could pass to VCS. Instead, it will receive a file as part of a verification request. This file will have to be read into memory first and then parsed into a Viper program. Only then can this program be passed as an object to VCS.

Currently, parsing is implemented as part of the backends. Each backend must implement an interface that requires it to define a series of methods. These methods are executed as a verification pipeline. The pipeline consists of the steps depicted in Figure 7.

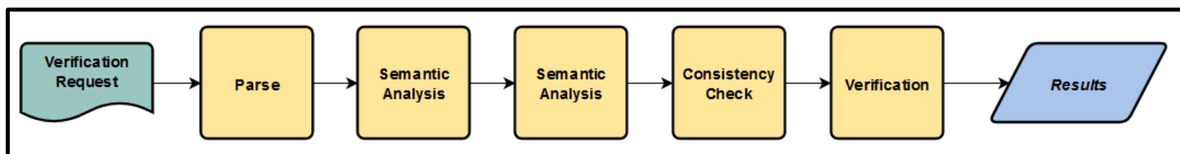

Figure 7 Verification pipeline for a Viper file within the backends.

In order to get a verifiable Viper program from a Viper file, the first four steps of the pipeline must be executed. This is exactly what the AstGenerator module, whose API can be seen in Listing 4, does. It first creates an instance of a Silicon backend, to which the file is passed. Then, the backend executes all the steps up to (and including) consistency check. After these four steps are executed, a verifiable AST program can be extracted from Silicon. In case any of the steps generates errors (e.g., syntax errors in the program), the pipeline is interrupted. After instantiation, the instance of the parsing module can be reused to parse other files.

```
1 class AstGenerator (private val _logger: ViperLogger){
2
3   //...
4
5   def generateViperAst(vpr_file_path: String): Option[Program] = {...}
6 }
```

Listing 4 API of parsing module AstGenerator.

HTTP capabilities will be added by extending VCS by a class called `ViperServerHttp`. This class will use Akka's HTTP library to implement the HTTP requests and responses. Using this library comes with the major advantage, that Akka's Stream library can be seamlessly integrated into it. The implemented requests are straightforward, as can be taken from Table 4.

Table 4 Overview of various GET and POST requests implemented in ViperServer.

Path	Get	Post
/exit	Tells the server to shutdown.	-
/verify	Gets the verification results for the job ID specified after verifying in the path.	Starts a verification process on the specified file passed as sub-resource. Returns the job ID assigned to that verification process. If starting the process fails, -1 is returned.
/discard	If necessary, interrupts, and deletes the verification process for the job ID specified after discard in the path.	-
/cache	Resets the cache completely.	Resets the cache for a particular file. Reports success or failure.
/alloy	-	Generates an instance of the Alloy model Returns this instance or a failure message.

2.3 VCS in Practice

2.3.1 Example use cases

A working minimal example of VCS is given in Listing 5. It shows how to set up the server, perform one verification, and shut it down.

```
object CoreServerRunner {
   def main(args: Array[String]): Unit = {
      implicit val actor_system: ActorSystem = ActorSystem("Example")
      implicit val ec: ExecutionContext = ExecutionContext.global

      val core = new ViperCoreServer(Array("--maximumActiveJobs=5",
        "--backendSpecificCache"))
      core.start()

      val file: String = "path/to/file.vpr"
      val parser = new AstGenerator(SilentLogger())
      val prog = parser.generateViperAst(file).get
      val backend = SiliconConfig(List("--disableCaching"))

      val myActor: ActorRef = actor_system.actorOf(...)
      val handler = core.verify("p1", backend, prog)
      val completionState = core.streamMessages(handler.id, myActor)

      completionState.get.onComplete({
        case _ => println("Verification Done!")
      })

      core.stop()
   }
}
```

Listing 5 Minimal example of VCS.

2.3.2 Erroneous use of the VCS API

There is really only one source of erroneous behavior when using VCS's API. Namely, calling methods in some wrong order.

A few of these erroneous orders cause actual exceptions. They all revolve around the `start` and `stop` methods. One way to cause an exception is by either calling any method before `start` or calling any method after `stop`. The other is calling either `start` or `stop` more than once.

Other erroneous calls are usually indicated by the returned value of a method. These concern the `verify` and `streamMessages` methods. `Verify` will return a `JobID` with the value `-1` if the number of active jobs exceeds the number specified by the corresponding configuration. To perform further verifications, `streamMessages` has to be called. This renders the process attached to the `JobID` passed to `streamMessages` inactive. On the other hand, calling `streamMessages` on an inactive `JobID` or on a `JobID` that doesn't yet exist will cause it to return `None`.

Listing 6 shows examples of erroneous calls. This includes calling the method `verify` before `start` and after `stop`, which throws an exception. It also includes calling `verify` when the number

of active jobs is reached. Furthermore, the listing shows what happens when `streamMessages` is called with non-existing `JobID` or with the same `JobID` repeatedly.

```
def main(args: Array[String]): Unit = {

  // set-up ...

  val core = new ViperCoreServer(Array("--maximumActiveJobs=2"))
  core.verify("p1", backend, prog) //Error!
  core.start()

  core.verify("p1", backend, prog) //returns 0
  core.verify("p1", backend, prog) //returns 1
  core.verify("p1", backend, prog) //returns -1!

  core.streamMessages(0, myActor)
  core.streamMessages(1, myActor)
  core.streamMessages(1, myActor) //returns None!
  core.streamMessages(-1, myActor) //returns None!
  core.streamMessages(42, myActor) //returns None!

  core.stop()
  core.verify("p1", backend, prog) //Error!
}
```

Listing 6 Examples of error-causing behavior.

2.3.3 Use Cases

The use cases for VCS can be divided into two broader categories. In the first category, VCS is used as a part of ViperServer. As such, it orchestrates the verification of Viper programs, that a client requests via HTTP. However, note that a client sends requests for file, rather than programs. Consequently, an AstGenerator is necessary to parse the files into a Viper program. A diagram of this category is shown in Figure 8.

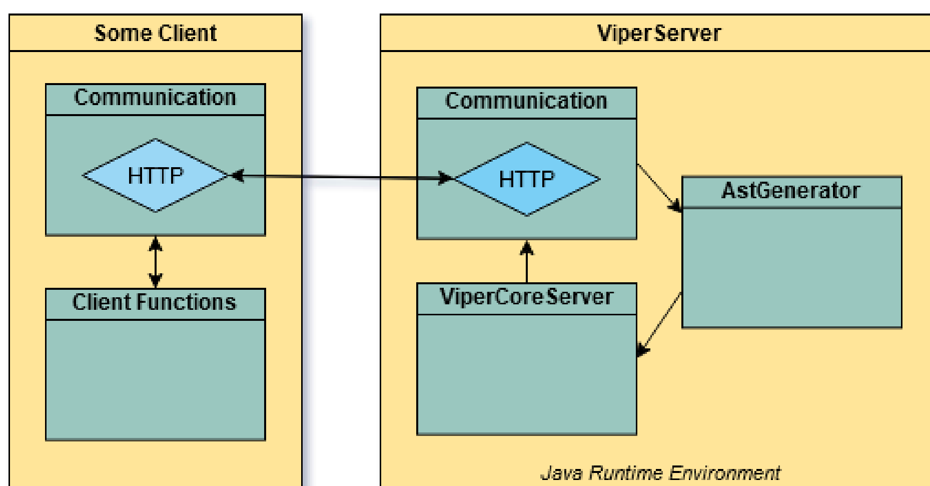

Figure 8 Use cases where VCS is a component of ViperServer.

Inside that first category, VCS is used in two concrete use cases. The first is when ViperServer is connected to the Viper IDE plugin. I.e. has the plugin as a client. In this case, ViperServer will be started and stopped by the plugin. ViperServer will also be tasked to verify files and send back results when the plugin requests them. Additionally, the client can send requests, like canceling a running verification or resetting the cache. All of these requests are sent to ViperServer, who then passes them on to the AstGenerator and VCS. The second concrete use case is to verify files and collect statistics of Viper programs from outside VS Code. This can be done utilizing a simple HTTP client. An example of such a client is given by a Python script called Viper_client[1]. It can be used from a CLI to issue verification requests with basic configuration options to ViperServer.

The other category of use cases for VCS is as part of verification servers for frontends to the Viper infrastructure. In this category, the tasks VCS performs are much the same as in the previous category. It orchestrates the verification of Viper programs. The difference is who the client is and how they interact with VCS. In this case, the client is a verification server, who uses VCS as a part of its verification backend. In this case, there is no need for the AstGenerator, as the client already has a Viper program it can pass to VCS.

At the moment, this also branches into two concrete use cases. The first use case is Gobra Server, a verification server for the frontend Gobra. The second is Prusti Server, a verification server for the frontend Prusti. The first case has already been implemented. Figure 9 shows how the Gobra Server uses VCS to verify programs that have been translated by the frontend Gobra from Go to Viper. It also shows the bigger picture of how Gobra Server is further connected to Gobra IDE, a plugin for VS Code similar to Viper IDE. In the case of Prusti Server, the integration of VCS has not yet been implemented in practice.

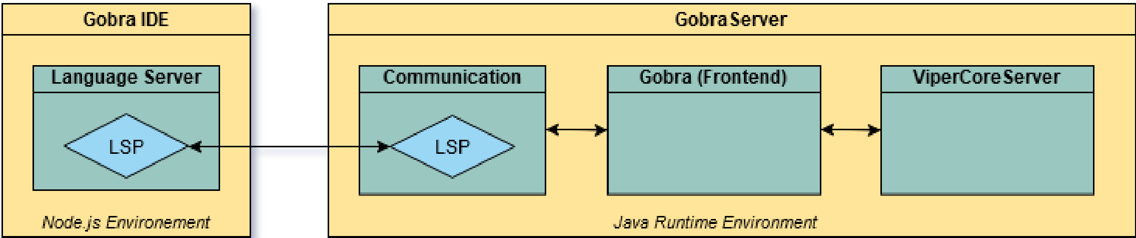

Figure 9 VCS as a component of Gobra Server.

[1] https://github.com/viperproject/viper_client

2.4 Evaluation

The project has been evaluated both from a quantitative and from a qualitative perspective. For the quantitative evaluation, a collection of unit tests have been written. Subjects of the tests are the new parsing module AstGenerator and VCS. Each received its own test suite. These suites contain numerous unit tests that check both the general aspects of the modules as well as corner cases. The tests can be found in `CoreServerTests.scala` and `ParsingTests.scala`.

Qualitative evaluation comes in form of integration of the implemented components into existing projects. Concretely, these are ViperServer and the Gobra frontend.

2.4.1 Unit tests: VCS

The tests suite for VCS was designed to test the API's components under different conditions. In general, the test cycles go through some variation of starting the server, verifying, retrieving results, and stopping the server. Scalatest, the framework used for these tests, allows reflecting this notion of cycles neatly by structuring and annotating the tests. Listing 7 shows the overall structure of the test suite for VCS. Each cycle of tests on the API is contained inside one of the `should` clauses. As can be taken from the annotations, each cycle sets different conditions for the API to be tested under. The conditions can be divided into three dimensions. These are the number of verifications performed, whether or not caching is enabled, and whether the program's verification results should be a success or a failure.

```
1  "An instance of ViperCoreServer" when {
2
3    "verifying a single program with caching disabled" should {...}
4
5    "verifying a single program with caching enabled" should {...}
6
7    "verifying multiple programs with caching disabled and retrieving
       results via 'getMessagesFuture()'" should {...}
8
9    "verifying multiple programs with caching enabled and retrieving
       results via 'getMessagesFuture()'" should {...}
10
11   "verifying multiple programs with caching disabled and retrieving
       results via 'streamMessages()" should {...}
12
13   "verifying an incorrect viper program several times with caching
       enabled" should {...}
14
15   "maximum capacity of verification jobs is exceeded" should {...}
16 }
```

Listing 7 Overall structure of unit tests for VCS.

The first two cycles each perform one verification. The focus lies on testing that the API does not crash and that none of the results are undefined. Listing 8 shows the annotations for the individual test cases, defined inside the `in` clauses. Each cycle also contains a few tests that are meant to cover corner cases or check that improper uses of the API are caught. For example, the last tests in each cycle

make sure that calling a method after `stop` throws an appropriate exception. The test on line 30 makes sure that verifying a non-existent JobID returns a `Future` that completes in a `Failure`.

```
"verifying a single program with caching disabled" should {

  "be able to execute 'start()' without exceptions" in {...}

  "be able to execute 'verify()' without exceptions" in {...}

  "be able to have 'verify()' return a JobHandler with non-negative id"
     in {...}

  "be able to have 'getMessagesFuture()' return a future of a sequence of
     Viper messages." in {...}

  "eventually see the future returned from 'getMessagesFuture()'
     completed successfully" in {...}

  "be able to execute 'stop()' without exceptions" in {...}

  "not be able to execute 'verify()' after 'stop()' without exceptions"
     in {...}
}

"verifying a single program with caching enabled" should {

  "be able to execute 'start()' without exceptions" in {...}

  "be able to execute 'verify()' without exceptions" in {...}

  "be able to have 'verify()' return a JobHandler with non-negative id"
     in {...}

  "be able to have 'getMessagesFuture()' return a future of a sequence of
     Viper messages" in {...}

  "see the future returned from 'getMessagesFuture()' eventually
     completed successfully" in {...}

  "see the future returned by 'getMessagesFuture()' eventually complete
     unsuccessfully for an inexistent job" in {...}

  "be able to execute 'stop()' without exceptions" in {...}

  "not be able to execute 'verify()' after 'stop()' without exceptions"
     in {...}
}
```

Listing 8 First two cycles of testing the VCS API.

The next two cycles, shown in Listing 9, each perform three verification attempts. Contrary to the first two cycles, the results returned by the individual methods are checked. For example, line 5 checks that `verify` returns the `JobID` with values 0, 1, and 2 for the three verification processes. Lines 7 and 16 check that the expected verification results are returned. The files used for testing include an empty file and two files that contain a method each, one of which contains an error. Therefore, the tests should detect two successes and one failure. Listing 10 shows the erroneous method contained used for testing.

```
"verifying multiple programs with caching disabled and retrieving
   results via 'getMessagesFuture()'" should {

  "be able to have 'verify()' executed repeatedly without exceptions" in
     {...}

  "be able to have 'verify()' return JobHandlers with unique non-negative
     ids" in {...}

  "be able to have 'getMessagesFuture()' return a future of a sequence of
     Viper messages containing the expected verification result" in {...}

  "be able to execute 'stop()' without exceptions" in {...}
}

"verifying multiple programs with caching enabled and retrieving results
   via 'getMessagesFuture()'" should {

    "be able to have 'verify()' executed repeatedly without exceptions"
       in {...}

    "be able to have 'getMessagesFuture()' return a future of a sequence
       of Viper messages containing the expected verification result"
       in {...}

    "be able to execute 'stop()' without exceptions" in {...}
  }
```

Listing 9 Testing cycles for verifying multiple files.

```
method sum(n: Int) returns (res: Int)
  requires true
  ensures res == n * (n + 1) / 2
{
  res := 0
  var i: Int := 0;
  while(i <= n)
    invariant i <= (n + 1)
    invariant res == (i - 1) * i / 2
  {
    res := res + i
    i := i + 1
  }
}
```

Listing 10 Erroneous Viper Method used for testing.

The fifth cycle focuses on testing the `streamMessages` method. All other tests use the `getMessagesFuture` method from `ViperCoreServerUtils`. The reason for this is that `getMessagesFuture` is implemented using `streamMessages`. Testing `getMessagesFuture`, therefore, tests both methods in terms of verification results. However, testing `streamMessages` on its own remains necessary, in order to test the `Option[Future[Unit]]` that the method returns, as well as the verification results themselves.

The sixth cycle, shown in Listing 11, focuses on testing the caching functionality. It does so by verifying the method shown in Listing 10 several times. It checks that the second verification attempt

returns the correct results from the cache, instead of reverifying them. Then, `flushCache` is tested. After calling this method, the same verification should no longer be returned from the cache.

```
1  "verifying an incorrect viper program several times with caching
      enabled" should {
2
3    "produce an OverallFailure Message with a non-empty error list upon
        first verification." in {...}
4
5    "produce an OverallFailure Message with an empty error list when re-
        verified." in {...}
6
7    "be able to execute 'flushCache()' without exceptions after several
        verifications" in {...}
8
9    "produce an OverallFailure Message with an empty error list when re-
        verified after flushing the cache." in {...}
10
11   "be able to execute 'stop()' without exceptions" in {...}
12 }
```

Listing 11 Cycle testing cache functionality.

The last cycle checks that VCS behaves correctly when it reaches the maximum number of active jobs. This is done by performing several verifications in successions, to reach the limit of accepted jobs. Afterward, the first test checks that a `JobID` with value -1 is returned. The next test checks that after requesting results for a verification attempt, a new verification attempt is accepted.

```
1  "maximum capacity of verification jobs is exceeded" should {
2
3    "have 'verify()' return a VerificationJobHandler with negative id" in
        {...}
4
5    "have 'verify()' return a non-negative id upon freeing up a
        verification request by calling 'getMessagesFuture()'" in {...}
6
7    "be able to execute 'stop()' without exceptions" in {...}
8  }
```

Listing 12 Cycle testing the maximum number of active verification processes.

2.4.2 Unit Tests: AstGenerator

Since AstGenerator has only one Method in its API, the test structure is considerably simpler, as can be seen in Listing 13. The test suite starts by testing that instantiating an AstGenerator works. Then, it uses this instance to test parsing several files. These parsing tests are the same but vary on the input file. These files were written to cover successful and unsuccessful parsing. The former is indicated by a defined `Option[Program]`, the latter by an undefined Option or an Exception. Files that should fail are files containing either syntax or type errors. An inexistent file should result in an exception.

```
"AstGenerator" should {

  s"should be instantiated without errors" in {...}

  s"have 'generateViperAst()' return an defined option for the file
     'sum_method.vpr'" in {...}

  s"have 'generateViperAst()' return an defined option for the file
     'let.vpr'" in {...}

  s"have 'generateViperAst()' return an defined option for the file
     'empty.vpr'" in {...}

  s"have 'generateViperAst()' return an empty option for the file
     'type_error.vpr'" in {...}

  s"have 'generateViperAst()' return an empty option for the file
     'parse_error.vpr'" in {...}

  s"have 'generateViperAst()' throw an exception for a non-existing
     file." in {...}
  }
}
```

Listing 13 AstGenerator test suite.

2.4.3 Integration Testing

We expanded and used the existing suite found in `ViperServerTests.scala` for this evaluation. Originally designed for ViperServer, we used it to check that the refactoring of ViperServer with VCS did not break anything.

Currently, the suite comprises six tests, whose description can be seen in Listing 14. Tests 1 and 3 send a POST request to ViperServer, each requesting the verification of a file. The first file should verify without errors, the other should verify with errors. Test 2 and 4 send GET requests to retrieve the results produced in tests 1 and 3. Test 5 sends a POST request to verify an inexistent file, to which the server should respond with a rejection message. The last test shuts the server down.

2.4.4 Test Scenario: Viper IDE

We also tested VCS and the refactoring of ViperServer in conjunction with Viper IDE. In this end-to-end test, we tested all the main interactions that a user can have with ViperServer from withing the editor.

```
1  "ViperServer" should {
2    s"start a verification process using ‘$tool‘ over a small Viper
        program" in {...}
3
4    "respond with the result for process #0" in {...}
5
6    s"start another verification process using ‘$tool on an empty file‘ "
        in {...}
7
8    "respond with the result for process #1" in {...}
9
10   s"start another verification process using ‘$tool‘ on an inexistent
        file" in {...}
11
12   "stop all running executions and terminate self" in {...}
13 }
```

Listing 14 Test suite for ViperServer.

2.4.5 Test Scenario: Gobra

VCS has been successfully implemented as a part of Gobra Server, according to one of the intended use cases. This provides valuable feedback about the correctness of the system. More importantly, though, it proves qualitative feedback. It proves that the API that was designed for VCS provides all the required functionality needed by a verification server other than ViperServer. It also shows that, in practice, it is indeed possible and convenient to integrate it into other systems.

On a more abstract level, it shows that the general concept of making components in the Viper infrastructure more modular is sensible. The fact that VCS is already in use in two systems, serves as practical evidence for successful code reuse.

2.4.6 Achieved Goals

The following objectives and were successfully met.

1. Provide unified access to the Viper backends via a server called VCS. The server has implemented the following features
    - Verification of Viper ASTs using either Silicon, Carbon, or a custom backend.
    - Caching of verification results.
    - Asynchronous and concurrent management of ongoing verification attempts.
2. Provide an abstract and intuitive API to VCS that allows to
    - Instantiate it with options.
    - Verify Viper ASTs.
    - Retrieve verification results for performed verification.
    - Reset the cache.
3. Provide a separate module called AstGenerator that can parse a Viper file into a Viper AST.
4. Extract the following components from ViperServer and implemented them as a part of VCS.

2.5 Discussion

2.5.1 Summary

In this milestone, we extracted components from the existing codebase of ViperServer to create VCS. These components include the interface that's used to invoke the verification backends, a process management mechanism that allows us to perform asynchronous and concurrent verification, and a caching mechanism for verification results.

We designed an API for VCS that allows us to use the aforementioned components extracted from ViperServer. This in turn allows other verification servers built atop the Viper infrastructure to use these components programmatically. In particular, having VCS accept AST rather than files, means that other verification servers no longer need to store Viper programs to files in order to verify them.

To verify actual Viper files, we created a parsing module. This module takes as input and parses it into a Viper AST that can then be passed to VCS for verification. The module uses an instance of Silicon to parse and type-check a Viper file.

Finally, we refactored ViperServer to make use of VCS. Doing so effectively adds an HTTP frontend to VCS. Consequently, this would allow communication with other processes such as verification IDEs. Since HTTP communication is based on files, the frontend includes the AstGenerator is used to parse incoming files.

2.5.2 Challenges

For most parts, the realization of this milestone was straight forward. We want to highlight one design choice we made. It concern how verification messages (e.g. results, program statistics, etc.) are retrieved from the backends.

Initially, when discussing the API with the Gobra team, we envisioned the use of *reporters* to retrieve messages from the backend. Reporters are the mechanism that is in place within VCS to retrieve the messages from the backends. The idea was for users of VCS to pass their own reporters to retrieve verification backends.

While this is technically feasible, using reporters to retrieve verifications from the backends was conceptually unsound. One problem was that reporters are a Silver-specific concept used by the backends. With the encapsulation of the backends in VCS, exposing this concept outside VCS was no longer necessary. Another problem was that passing a reporter into VCS would circumvent parts of its process management. Messages from the backends would no longer be buffered into a queue, negating the system's reactive architecture.

Because of these two problems we decided to change the API. Instead of a reporter, we decided that users were to pass an Akka actor to retrieve messages from VCS. With actors, we have exposed a more generic concept that is not Silver-specific. Also, exposing actors extends VCS's reactive architecture to the client.

3. Verification Server Interface

3.1 Introduction

The basis for this chapter is laid by the observation that there exist common features in verification servers. While they are common in the sense that they are used by every verification server in some form, they are usually not entirely generalizable. Most of these features, at some point, depend on parameters that are specific to the source language they support. A simple example of this is verification itself. Obviously, every verification server will eventually execute some verification. Since servers verify different languages, the engines and backends use for verification will vary depending. In this chapter, we will use the terms *language-agnostic* and *language-specific,* or *language-(in-)dependent*, to refer to the parts of these common features that are either generic or specific to a language.

3.1.1 Proposition

We propose to extract features common to all verification servers and to bundle them into a separate module called **Verification Server Interface** (VSI). This module will contain a set of traits and abstract classes that will implement the language-agnostic parts of common features while leaving the language-specific parts for the clients to implement. We identified three groups of features that are common to most verification servers and that will therefore be part of VSI. These are process management, communication, and caching.

Ultimately, we plan to make VSI available for developing novel verification servers, in particular, those atop the Viper infrastructure. With this, we aspire to lower the cost of designing, implementing, and maintaining verification servers.

3.1.2 Approach

The sequence of steps in this chapter is set by the three features that are to be extracted. We will isolate and generalize them in the following order:

1. Process management
2. HTTP Capabilities
3. Caching

To demonstrate the practical usefulness of the library, this will be followed by refactoring VCS and, by extension, ViperServer. Practically, however, this process will be done stepwise and in parallel. I.e., after each of the above steps, we will refactor the corresponding parts of VCS and ViperServer using the newly extracted features. This makes it easier to detect errors. It also allows reflecting on the progress in shorter intervals.

3.1.3 Chronology

This part of the project that may affect other ongoing projects. Completing this milestone second will increase the likelihood of it being used by another project within the duration of this thesis. This would in turn provide an independent form of feedback and evaluation.

3.2 Process Management

At the heart of any verification servers lies some form of process management. Verification requests have to be admitted and executed, verification results have to be returned. We will therefore start by generalizing process management.

The implementation of the process management in VSI is inspired by how process management is currently implemented in VCS. As was argued in the last chapter, this process management system has several important advantages. It is both asynchronous and concurrent and it is built on a library that integrates streaming capabilities.

So far, however, process management in VCS has mostly been treated as a black box. Since we plan on implementing a generalized version of it, we will start by explaining how process management works in VCS. We will then go through the individual parts of the process management system and decide which should be included in VSI. Of those, we will decide which are generic and which language-specific. We will then present the design we came up with and explain how VCS was refactored using VSI's process management.

3.2.1 How is Process Management implemented?

The process management functionality is based on Akka's Actor and Stream libraries. These allow us to express concurrent computations on an abstract level, by defining actors that run code independently and communicate by exchanging messages. The actors are defined by extending the `Actor` trait, which requires the implementation of a partial function called `receive`. This function effectively defines the actor's behavior in terms of the messages it receives. In a typical scenario, an actor would receive a message, perform a computation, and then possibly send a message to some other actor. All actors are registered in an `ActorSystem` which manages the concurrent computations and the flow of messages between the actors.

In VCS, three actors are defined, each concerned with an aspect of process management. Table 5 gives an overview of what they're responsible for.

Table 5 Overview of Actors defined in VCS

Actor	**Task**
JobActor (JA)	Is responsible for executing a verification task. This includes setting up the verification and potentially interrupting it.
QueueActor (QA)	Is responsible for a queue that aggregate messages produced by the backend for a given verification task.
TerminatorActor (TA)	Responsible for terminating individual verification processes as well as terminating the server.

The first actor ever to be spawned is the TA. This happens exactly once when `start` is called. Afterward, a JA is spawned with each call of `verify`. As part of setting up the verification task, the JA will spawn a QA. That QA will be equipped with a queue that will hold messages reported by the

backend. To that end, the QA will be passed as a reference to the backend. Note that since the amount of active verification processes is limited by some number n, there will never be more than $2n + 1$ actors active.

Next, let us look at the termination of actors. The simplest is the TA's, as it is terminated by the call to `stop`. This call makes the TA stop the `ActorSystem` (and, in ViperServer, unbind from any port used for HTTP communication). The termination for the QA is tied to the verification task. When the backend finishes its work, it reports one final message to the QA. This message indicates that the verification task has been (unsuccessfully) executed and that the actor is no longer needed.

The termination of the JA is a bit more complicated. The reason for this is that we do not want to impose on clients when they have to retrieve verification results. Instead, we want the client to control this by calling `streamMessages`. This means that we have to keep some reference to the queue holding the messages until `streamMessages` is called. Even when verification has been completed for a request. Since the QA was terminated at the end of verification, this must be done by keeping the JA alive, as it is the only other actor who holds a reference to the queue. Letting the client decide when it can retrieve the message also means that the client may ask for them when verification is still ongoing. Conversely, this means that we can not simply terminate the JA when `streamMessages` is called. Doing so would mean we can no longer interrupt the verification, as the JA responsible for the job would be gone.

To reconcile these two scenarios, we have put in place two conditions for the termination of a JA. The first condition is that `streamMessages` must have been called. The second is that the queue must have been completed. Meeting the first condition sends a message to the TA, along with a future containing the queue completion state. This future is a feature provided by Akka's Queue implementation. In our case, its completion indicates to the TA that the second condition is met. At that point, the TA can safely delete the JA.

Figure 10 provides an illustrative summary of the processes described in this subsection. It shows the dependencies that trigger the instantiation or termination of actors described in this subsection.

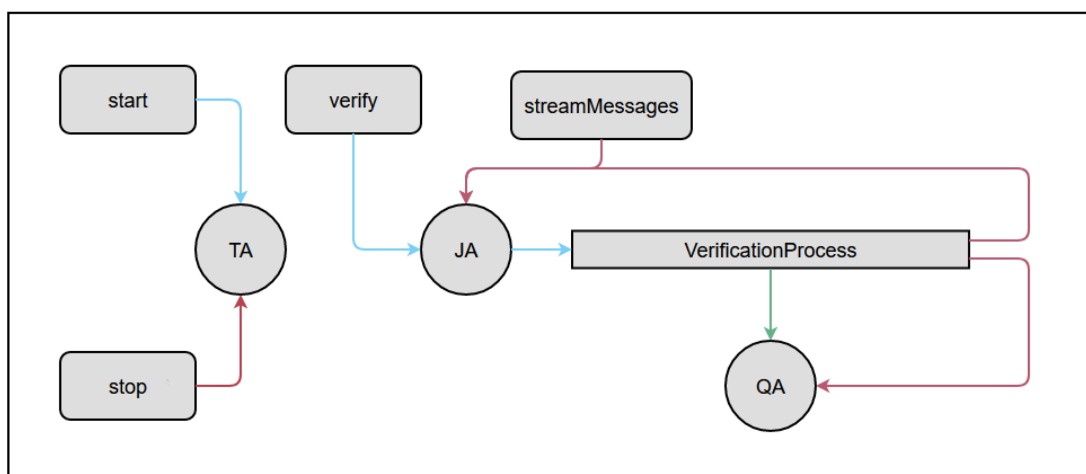

Figure 10 Dependencies graph of the actors and events in VCS's process management. Blue arrows indicate spawning events, red arrows indicate termination conditions. The green arrows indicate messages sent from the backends to QA.

The last mechanism we will explain is the queue used by the queue actor. The queue is a buffer provided by the Akka Streams library. In addition to being a FIFO data structure, it can be used as a source for a reactive stream. This allows us to dynamically decide the number of messages that are streamed per unit time. If the stream's sink is filling up, the queue will apply backpressure and tell the source to slow down. In our case, however, the queue is fed messages from the backends, which is a non-reactive source. I.e., the queue will not be able to send these signals back any further and the backends will not slow down in case the sink fills too quickly. To counteract this, we introduced a blocking mechanism in the backends. This mechanism is based on the fact that each message has to be offered to the queue. This offers can fail, if the queue is overflowing. In that case, the backends will block until the queue's sink is no longer overflowing.

3.2.2 Identifying language-(in.-)dependent features

As we've mentioned previously, process management in VSI is heavily based on VCS's process management. This, because we believe that the features it provides should be common to all verification server. However, there are features that, while common, are not directly generalizable, as they are specific to the language the server is used for. We will now examine this more closely.

The organization of the actors in VCS is independent of any language. The responsibilities of actors, such as terminating jobs, enqueueing messages, and interrupting running jobs, are agnostic to the verification server's source language.

The concept of a `start` and `stop` method is, at its core, language-independent. The function's main responsibilities are to set up and terminate the actor system. This can be provided without knowledge of the verification server's language. Note that it is likely that a `start` and `stop` method do not cover everything a verification server might want to initialize or terminate. For example, a concrete verification server might want to also initialize a logging system. However, this additional behavior is not considered common to all verification servers. Therefore, it is not part of what VSI. That being said, a client can always extend these methods and complement them with additional behavior.

Most of what is happening inside the `verify` method is language-specific. For example, `verify` takes a Viper AST as input. Obviously, a different verification server will want to input an AST for a different language. Also, the verification task that is run on said AST is going to be different from one verification server to another.

Having different ASTs and verification tasks also means that the messages reported during verification (results, errors, etc.) will be different. This implies that the message queue and the `streamMessages` method will have to be adaptable to a specific language.

3.2.3 Design

As can be taken from Listing 15, we introduced a trait called `VerificationServer`. Extending this trait provides the client with common behavior related to the process management of a verification server. This includes a `start` and `stop` method that instantiates/terminates the server. It also includes the definition of the three actors (`Job`-, `Queue`-, and `TerminatorActor`). These provide the concurrent and asynchronous process management described in Section 3.2.1.

`VerificationServer` also provides the client with an `initializeVerificationProcess` method. This can be seen as a generic version of the `verify` method in VCS. However, as a consequence of this generalization, it should no longer be understood as a function that does verification. Instead, it should be understood as the method that sets up a process in which verification will happen. More concretely, it takes as input a `verificationTask`, creates the necessary actors, and passes the task along to them. Then, it returns a corresponding JobID as a reference to that process.

```
1  trait VerificationServer extends Unpacker {
2
3    // Fields and Actor definitions
4
5    def start(active_jobs: Int): Unit = {...}
6
7    protected def initializeVerificationProcess(task:VerificationTask):
       JobID = {...}
8
9    protected def streamResults(jid: JobID, clientActor: ActorRef):
       Option[Future[Unit]] = {...}
10
11   def stop(): Unit = {...}
12
13   // ...
14 }
```

Listing 15 VerificationServer trait.

To allow for language-specific verification tasks, we introduced a corresponding abstract class, as can be seen in Listing 16. Notably, this class extends `Runnable`. This means that a client extending it can specify the language-dependent verification task inside that `run` method. Encapsulating the verification task this way is what allows both the JA and `initialize- VerificationProcess` to be implemented in a generic manner. The method no longer needs to deal with language-specific input (such as an AST) and the JA can simply start the verification task described in `run`.

```
1  abstract class VerificationTask()(implicit val executionContext:
     ExecutionContext) extends Runnable with Packer {
2
3    private var q_actor: ActorRef = _
4
5    final def setQueueActor(actor: ActorRef): Unit = {...}
6
7    protected def enqueueMessages(msg: A): Unit = {...}
8
9    protected def registerTaskEnd(success: Boolean): Unit = {...}
10 }
```

Listing 16 Abstract class VerificationTask extending Runnable.

Furthermore, `VerificationTask` provides methods necessary for the backends to communicate with the QA. To report messages from within a verification task, the client can use the

`enqueueMessages` method. This method will take that message and pass it to the QA. Note that this includes the blocking mechanism explained in Section 3.2.1 to ensure the queue does not overflow. To indicate that the task is over, the client must use the `registerTaskEnd` method. This will send a special message to the QueueActor, indicating that the verification has ended (un-) expectedly.

Contrary to the previous two actors, the `setQueueActor` method is neither meant for the client to use nor modify. Instead, it is used during `initializeVerificationProcess` to pass the QA created for that process to the verification task.

Finally, we will explain the design that allows `VerificationServer` to deal with language-specific messages. As can be taken from the signature of `enqueueMessages` in Listing 16, we wanted the client to be able to enqueue messages of their language-specific type. Analogously, we wanted `streamMessages` to stream language-specific messages. Internally, however, there must exist a more generic representation of messages. To reconcile these requirements, we first introduced an empty trait called `Envelope`. This trait would act as a generic supertype with which messages are represented internally in `VerificationServer`. As the name Envelope suggests, the trait is meant to be wrapped around a language-specific message by the client.

```
1  trait Envelope {}
2
3  trait Post {
4    type A
5  }
6
7  trait Packer extends Post {
8
9    def pack(m: A): Envelope
10 }
11
12 trait Unpacker extends Post{
13
14   def unpack(m: Envelope): A
15 }
```

Listing 17 Traits involved in generalizing the messages produced by a server's backends.

Getting messages into and out of an `Envelope` should be a matter of providing two simple mappings. Given such a mapping, `enqueueMessages` and `streamMessages` can both simply map messages from their language-specific type to the generic type and vice-versa. To enforce this implementation from the client, we introduced two more interfaces. As can be seen in Listing 17, each contains one method. The `Packer` contains a `pack`, the `Unpacker` an `unpack` method. Both also contain a type parameter `A`, which is the type of the language-specific messages. To make sure that this type parameter is the same in both interfaces, it was extracted into a separate interface called `Post`.

We then extended `VerificationServer` and `VerificationTask` with `Unpacker` and `Packer`, accordingly. Doing so gives `enqueueMessages` access to the `pack` and `streamMessages` access to the `unpack` method.

3.2.4 Refactoring of VCS

Refactoring VCS with VSI comes down to implementing the various traits and to change the API to make use of the methods they provide. In this subsection, we will briefly describe what the necessary steps are to accomplish this.

First, VCS has to extend `VerificationServer`. This requires the implementation of the `pack` method, as well as wrapping Viper-specific type `Message` into an object extending `Envelope`. This is done by introducing a simple case class called `ViperEnvelope`, that extends `Envelope` and holds a `Message`.

In a second step, the `VerificationTask` trait must be implemented. This means that the `run` and `unpack` method inherited by their respective traits must be implemented. In the case of VCS, this can be done by extending the `VerificationWorker` class with `VerificationTask`, since `VerificationWorker` already implements a `run` method. Then, the `VerificationWorker` only needs to be complemented with the implantation of the `unpack` method.

In a third step, the VCS methods must be refactored to make use of the ones defined by the `VerificationServer` trait. This is true for all methods in the VCS API except for the `flushCache` method, as it is not connected to the process management. For the others, the general pattern is to override the method, complement it with Viper specific code, and call the corresponding method in `super`. A typical example of this is the `start` method, as shown in Listing 18. In VCS, `start` needs to perform additional initializations like instantiating a logger and checking configurations. For `verify`, the same pattern applies. However, the corresponding method is called `initializeVerification Process`.

```
1  def start(): Unit = {
2
3    _config = new ViperConfig(_args)
4    config.verify()
5
6    _logger = ViperLogger("ViperServerLogger",
         config.getLogFileWithGuarantee, config.logLevel())
7    println(s"Writing [level:${config.logLevel()}] logs into " +
8      s"${if (!config.logFile.isSupplied) "(default) " else ""}journal:
           ${logger.file.get}")
9
10   ViperCache.initialize(logger.get, config.backendSpecificCache())
11
12   super.start(config.maximumActiveJobs())
13   println(s"ViperCoreServer started.")
14 }
```

Listing 18 Overriding the generic start method in VCS to perform Viper-specific initialization.

3.3 HTTP

HTTP functionality is a feature useful to any verification server. In this section we will look at the existing HTTP module present in ViperServer, extract common features and decide whether they are directly generalizable or whether they are language-specific.

To that end, we will quickly recap how that module works now (see Section 2.2.5 for a more detailed explanation). In ViperServer, HTTP functionality is implemented using Akka's HTTP library. This library allows defining *routes* which hold code that defines how the server should behave when receiving HTTP request such as `GET` and `POST` on particular paths. Defining the behavior to such a request includes parsing any information transmitted by the requests and providing a response in the correct format. This is currently done by various protocols that are part of ViperServer.

3.3.1 Identifying common, language-dependent, and language-independent parts

In the current state, not all parts of ViperServer's HTTP functionality are common to all verification servers. For example, we do not assume that all verification servers will be equipped with a caching mechanism. Therefore, the `GET` request for flushing cache found in ViperServer should not be considered common functionality. Another example is given by the `GET` request that returns an Alloy model corresponding to a verification. The generation of such a model is not behavior that can be expected from any verification server.

HTTP functionality that is expected to be common in verification servers, is functionality that relates to the functionality VerificationServer offers. E.g., HTTP requests that start a verification or retrieve that results, requests that cancel a running verification, etc.

Yet again, these common functionalities have both language-dependent and language-independent aspects. For example, setting up the HTTP routes and requests solely relies on the Akka libraries, and is language independent. On the other hand, information that is sent to and from the server is, in some cases, language-specific. An example of this is currently given when streaming verification results from ViperServer to some HTTP client. At the time of streaming, the verification results are Viper-specific Scala objects. For the results to be communicated to a client via HTTP, they need to be transformed. This transformation must be implemented by the client.

3.3.2 Design

This part of the milestone saw the introduction of a new trait, called `Http`. This trait holds four routes common to all verification servers. A description of these routes is given in Table 6.

Table 6 Common routes among verification servers

Path	**Method**	**Behavior**
/exit	GET	Requests the server to shutdown.
/verify	POST	Starts a verification process on the specified file passed as a resource.
/verify	GET	Sets up a stream for the verification results corresponding to the verification process for the specified job ID.

/Discard	GET	Interrupts and deletes the verification process for the specified job ID.

As was established in the previous subsection, the behavior of common HTTP routes across verification servers is linked to the functionality provided by VerificationServer. This makes sense, as VerificationServer is supposed to provide functionality common to all verification servers. To reflect this logic in the code, we decided to have `HTTP extends VerificationServer`, as can be seen in Listing 19. This forms a contract with a user of `Http`, saying that any verification server where `Http` will be used, will be based on `VerificationServer`. A direct consequence of this is that it allows `Http` to make use of the code provided by `VerificationServer`. This is advantageous, as it leads to code reuse. Another consequence is that the `Http` trait is now tied to the `Verification Server` trait. In particular, this means that a user of `Http` will have to implement `VerificationServer` as well.

To enable the clients to specify language-dependent aspects, we declared abstract methods corresponding to the various routes. These methods are typically meant to let a client specify how to respond to a request. Listing 19 shows an example of this for two routes. In the route for the `GET` method on the `discard` path, two abstract methods are used, called `discardJobRejection` and `discardJobConfirmation`. Each allows the client to specify how to respond in case of failure or success of discarding a job, respectively. Note that in this route, most of the logic for deleting a job is already provided. The two abstract methods in this route merely specify the form of the response. Contrary to this, the logic for starting a verification process is entirely left to the client. The reason for this is that `Http` can not make assumptions about what data is transmitted alongside the `POST` request for verification. Therefore, the `onVerifyPost` method must provide the code to parse that data and to initialize a verification process with it.

It stands to reason that an `Http` user might not want to be limited to the four routes provided by `Http`. To accommodate for this, we introduced an additional trait called `CustomizableHttp`. This trait contains the method `addRoute`, that knows how to combine routes. This allows a user of `Http` to define additional routes in an identical manner to the ones already defined in `Http`. Then, using the `addRoute` method, the routes can be merged with the existing routes of `Http`.

```
trait VerificationServerHTTP extends VerificationServer with
    CustomizableHttp {

  // ...

  def onVerifyPost(vr: Requests.VerificationRequest):
      ToResponseMarshallable

  def discardJobConfirmation(jid: Int, msg: String):
      ToResponseMarshallable

  def discardJobRejection(jid: Int): ToResponseMarshallable

override final def routes(): Route = {
  path("verify") {
    post {
      entity(as[Requests.VerificationRequest]) { r =>
        complete(onVerifyPost(r))
      }
    }
  } ~ path("discard" / IntNumber) { jid =>
    get {
      jobs.lookupJob(JobID(jid)) match {
        case Some(handle_future) =>
          onComplete(handle_future) {
            case Success(handle) =>
              implicit val askTimeout: Timeout = Timeout(5000 milliseconds)
              val interrupt_done: Future[String] = (handle.job_actor ?
                  Stop).mapTo[String]
              onSuccess(interrupt_done) { msg =>
                handle.job_actor ! PoisonPill
                complete(discardJobConfirmation(jid, msg))
              }
            case Failure(_) =>
              complete(discardJobRejection(jid))
          }
        case _ =>
          complete(discardJobRejection(jid))
      }
    }
  } ~ // further routes
}
```

Listing 19 Extract of the VerificationServerHttp trait showing two routes and the corresponding abstract methods.

3.3.3 Refactoring of ViperServer

Recall that at the end of the first milestone, we had a class called `ViperHttpServer` that extended the class `ViperCoreServer`. This made sense, as `ViperHttpServer` was meant to be a verification server with added HTTP capabilities. Also, note that after the first part of VSI, we additionally have `ViperCoreServer extend VerificationServer`. This makes sense too, as VCS is a specific instance of VerificationServer. The fact that `Http` now also extends `Verification Server`, and that `ViperHttpServer` needs to extend `HTTP` leads to an interesting situation, illustrated in Figure 11. Should `ViperHttpServer` still extend VCS?

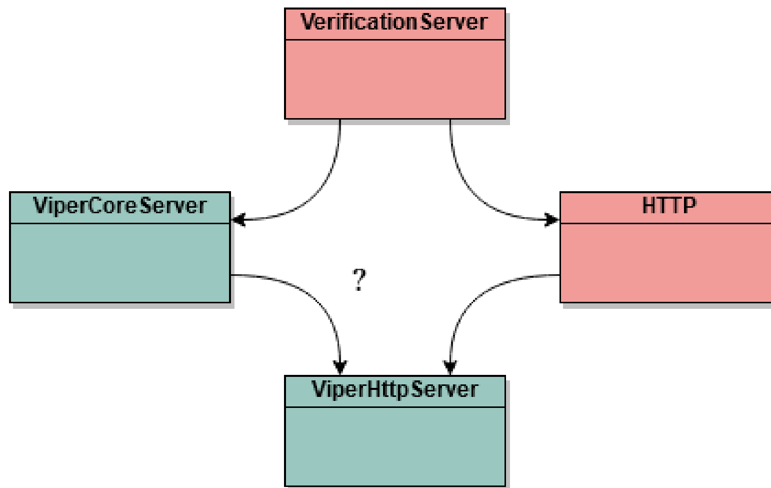

Figure 11 Diamond structure in the class hierarchy around ViperHttpServer

The answer to that question is yes. While this effectively creates a diamond structure in the class hierarchy, this is not a problem in Scala. As can be seen in Figure 11, only VCS is a concrete class. All other parts of the hierarchy are traits, which means the structure is valid. In fact, it is a welcome side-effect of the design. If we could not extend from VCS, we would have needed to essentially re-implement a verification server to use Http. This would have meant that we would effectively have implemented two verification servers: VCS and `ViperHttpServer`. By extending `ViperHttpServer` we can avoid this reimplementation by making use of the implementations of `VerificationServer` provided by VCS. Concretely, this means we can reuse the implementation of the wrapping mechanism for messages and verification tasks.

The rest of the refactoring process consists of rearranging the code in `ViperHttpServer` to conform to the `HTTP` trait.

3.4 Cache

Due to its potential improvement in performance, caching is an important feature to have in any verification server. It is also a feature that is closely tied to the specific language for which a verification server is implemented. Although ViperServer has provided valuable inspiration for what a caching feature should do, it could not be extracted and generalized as well as the process management or HTTP features.

We will start by explaining what a caching feature is in the context of a verification server. Then, we will explain the general idea behind the design of caching in VSI. Finally, we will look at how we translated this design into an actual implementation.

3.4.1 Caching in Verification Servers

The goal of a caching mechanism is to improve performance. In simplified terms, this is achieved by storing and retrieving data that result from a computation, instead of recomputing it. If the cost of computing that data is larger than the cost of looking up, retrieving, and updating the cache, performance is improved.

Applied to verification servers, the computation is the verification performed by the verification backends. The data that is stored and retrieved is the verification results. Hence, a cache system in a verification server should check if verification results exist for a given program. If that's the case, these results are immediately returned. Otherwise, the results must be computed first and consecutively be added to the cache. An illustration of this process is given in Figure 12.

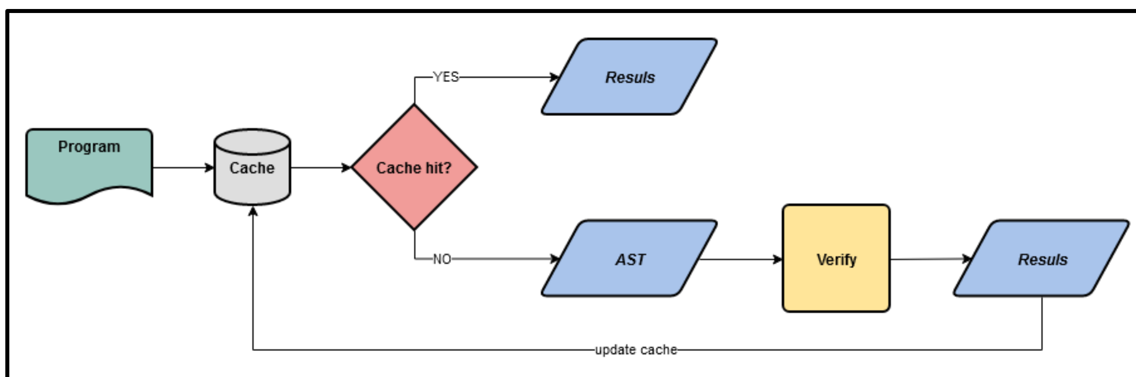

Figure 12 Caching process in a verification server.

One detail that has been left out is how to decide whether a program has corresponding verification results stored in the cache. In other words: If verification results are stored for a program p, will these results still be valid for a modified program p'? We will refer to the function that can answer this question as the cache's *decision function*. Ideally, we would want a decision function that exactly predicts if a change in a program would lead to different verification results. However, this is hard to do short of actually performing verification for large changes. Needless to say, if it takes a full verification to find out whether the cached results are valid for a different program, no performance was gained.

Another detail that was left out is that verification backends typically verify programs modularly. In other words, the verification results are produced for individual parts of the program, such as methods. This means that results could potentially be retrieved from the cache in a more fine-grained manner. Instead of retrieving a set of results only if the entire program hits the cache, the caching mechanism could retrieve results for parts of the program. Imagine a program p consists of two completely independent parts, q_1 and q_2. In this case, changing something in q_2 should not affect the verification results returned for q_1. I.e. results stored for q_1 could be returned from the cache. But what happens with the rest of the program, q_2, if it was changed in such a way that the cached results are no longer valid? Clearly, the answer is that q_2 needs to be reverified. But performing verification on p as a whole would negate the performance gained by retrieving results for q_1 from the cache. Hence, there must be a way for the backends to tell if an individual program component needs to be (re-) verified.

3.4.2 Design: Overview

Note that in VSI, the caching mechanism is laid out to work on the level of program components. From now on we will refer to these program components as *members* of a program.

Generally, an instance of a caching mechanism in VSI should store the following information for each of the cached members. The first is the verification result generated by verifying that member. The second piece of information it should store for a given member is a list of other members. This list is called the *dependency list* for a member. It contains the member that can influence the outcome of verification for a (cached) member if they are changed. The reason why this list is necessary is that even though an individual member may stay the same over time (for some notion of equivalence with respect to verification), other members of a program may not.

Take methods m_1 and m_2, two cacheable members of some hypothetical programming language, as an example. In that language, the verification result of member m_1 depends on m_2. In other words, changing member m_2 will lead to different verification results for m_1. This makes m_2 part of m_1's dependency list. Even though m_1 itself may never change, its cached verification results are no longer valid if m_2 changes.

In terms of behavior, the caching mechanism should have two methods called `retrieve` and `update`. `Retrieve` should take an AST p as input and decide for each of its members, whether or not verification results may be taken from the cache. Additionally, `retrieve` should return a (potentially) modified version p' of the input AST. The version p' should thereby be modified in such a way that verifying it is faster than verifying p. More precisely, it should be modified in such a way, that the backends only verify the parts of p whose results could not be retrieved from cache. The `update` method should take three parameters as input, corresponding to the information stored per cache entry. These are a member, the corresponding verification results, and the dependency list. A visualization of how `retrieve` and `update` are used in the verification process is given in Figure 13.

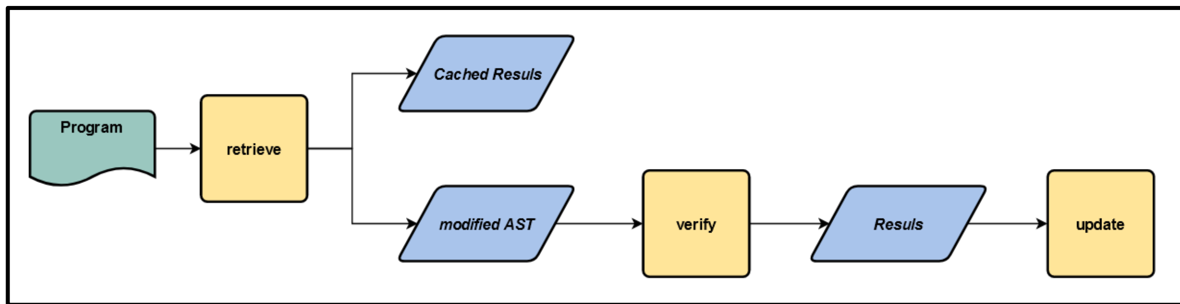

Figure 13 Verification process using a cache's retrieve and update methods.

In a typical verification cycle, caching should behave as follows. Before sending a program p to the verification backend, `retrieve(p)` is called. Inside the method, each member of p is checked against the cache. For each member that hits the cache, the current dependency list is checked against the one stored in the cache. I.e., a dependency list is computed for the member in question with respect to the current program p. If the list corresponds to the dependency list for a member of a previous program, the corresponding results are still valid. This means they can be returned to the user. In parallel, a new AST p' is built. This AST contains a transformed version of each member that has hit the cache. For members that have not hit the cache, or whose dependency list did not match the one stored, nothing is returned. Instead, they are simply added to p' as they are. Afterward, p' is verified. Verification generates a result for each non-transformed member. These members m, their corresponding result r and dependency list d are then added back into the cache by calling `update(m, r, d)`

The mechanism described embodies a functioning cache. However, this mechanism performs less than optimally on programs that are equivalent in terms of verification results. For two such programs, where each member hits the cache, each member's dependency list has to be checked. This means that for each member, the list has to be computed in the context of the current program. This is unnecessary as the programs, and therefore the lists, are identical.

While computing this list is less expensive than performing verification, it is still a costly operation. This is particularly true if there is a computationally less expensive way to compare the equality of two programs. A rather simple example of this situation is given if the exact same program was passed on for verification. In this case, a simple comparison of the program's textual representation could save a lot of computation. Note that, since verification is likely to be used in an automated environment (for example, in an IDE), this situation is indeed likely to occur. For instance, if an IDE triggers verification automatically every time the active editor is changed, a verification on an unchanged program is triggered.

Hence, In addition to the previously described caching mechanism, we added another layer of caching. This program-level cache stores members and their dependency lists for a given program. In `retrieve`, before going through all the members, the method checks the program-level cache. If that cache decides a program in the cache is equivalent to the one passed as input, it will not compare the dependency lists. Instead, it will just retrieve the result associated with the members and return them.

3.4.3 Design: Implementation

On a more concrete level, VSI's cache was implemented as a trait, as shown in Listing 20. Note that the cache structure extended to include files. This way, a verification server can keep several, distinct caches when verifying multiple files at the same time.

```
abstract class Cache {

  protected val _cache = MutableMap[String, FileCash]()
  type FileCash = MutableMap[String, List[CacheEntry]]
  protected val program_cache = MutableMap[AST,
     MutableMap[CacheableMember, List[Member]]]()

  def retrieve(
       file_key: String,
       input_prog: AST): (AST, List[CacheEntry]) = {...}

  final def get(
             file_key: String,
             key: CacheableMember,
             dependencies: List[Member]): Option[CacheEntry] = {...}

  def update(
          file_key: String,
          key: CacheableMember,
          dependencies: List[Member],
          content: CacheContent): List[CacheEntry] = {...}

  def forgetFile(file_key: String): Option[String] = {...}

  def resetCache(): Unit = {...}
}
```

Listing 20 Generic Cache implementation in VSI

In addition to the `retrieve` and `update` methods shown in Listing 20 explained in the previous subsection, the cache also comes with methods to reset the cache: `forgetFile` and `reset`. These four methods provide any VSI client with an abstract API for the caching mechanism.

Aside from the cache trait, several other traits have been added to allow for a generic implementation of the caching mechanism. Firstly, there's the `Ast` trait, that represents a generic version of ASTs. Secondly, there's the `Member` trait, that represents a generic version of members of an AST. Thirdly, there's the `CacheableMember` trait.

As can be seen in Listing 21, `CacheableMember extends Member`. This reflects the notion that not all members of a program are necessarily cacheable. In Viper, for instance, caching is only supported for methods. Predicates, Fields, and other members can not be cached. Accordingly, it reflects the fact that a cacheable member must provide more functionality than a `Member` must. In the code, this is reflected by the two abstract methods declared in the trait: `transform` and `getDependencies`. The `transform` method is what is used to transform program members when they hit the cache in `retrieve`. The `getDependencies` method computes the dependency list for a given member and AST. This is used either when putting a cacheable member into the cache or when checking if the results for a cached member are still valid.

```
trait CacheableMember extends Member {

  def transform: CacheableMember

  def getDependencies(ast: AST): List[Member]
}
```

Listing 21 CacheableMember trait extending Member trait.

In order for the caching system to differentiate between cacheable and non-cacheable members, the client must provide a `decompose` method for the AST. This is shown in Listing 22. With that method, `retrieve` can take an entire AST as input and distinguish between the two types of members by itself.

```
trait Ast {

  def compose(cs: List[CacheableMember]): AST

  def decompose(): List[CacheableMember]

  def equals(other: AST): Boolean
}
```

Listing 22 Ast trait.

After going through the cacheable members to find out whether or not the cache holds results for them, retrieve needs to output a (modified) AST. This requires the to put cacheable members and normal members back into an AST. This is why the client must also implement a `compose` method for their AST, which is shown in Listing 22.

The equal method in the Ast trait requires the client to implement a notion of equality between entire ASTs. This is used to determine if the computation of dependency lists may be avoided.

3.4.4 Refactoring of ViperCoreServer

In a first step, we implemented all traits to conform with the language-specific requirements of Viper. This meant wrapping the Viper AST into VSI's generic `Ast` trait and defining the two types of members. Recall that in Viper, methods are the only cacheable members. This means that the `Ast`'s `decompose` method when implemented for Viper, simply returns all the methods from a Viper AST.

In `CacheableMember`, the `getDependencies` method is inherited from Viper AST. As for the `transform` method, its implementation in Viper is quite simple. It consists of removing a method's body. This works in Viper, as the backends will not verify bodyless methods. An AST transformed that way will therefore take less time to verify.

The `Cache` trait was implemented by letting an object called `ViperCache` extend it. We decided to implement caching as an object, rather than an instantiable class because there should be only one cache instance in VCS at any given time. We concluded that having one global caching instance would simplify the management of the cache's state. Note that since `Cache` provides per-file caching, it is possible to manage caches for multiple files and different backends using only one cache instance.

In a second step, we incorporate code from the previous implementation of caching into ViperCache. This code's purpose is to update the position of errors that were stored in the cache. In Viper, the verification errors returned by the backends (and subsequently stored in the cache), come with positional information. If parts of a program change, the position of an error is likely to change with it. Thus, if an error is cached and then later retrieved, the position that was cached is likely to be outdated and wrong. Again, this is true even if the changes did not directly concern the code where the error was located. It is therefore necessary to update error positions when retrieving them from the cache.

3.5 VSI in Practice

3.5.1 General Use Cases

As was touched on in the introduction, VSI is meant to be used to simplify the development of verification servers written in Scala. Its three components (`VerificationServer`, `Http`, and `Cache`) are meant to provide generic tools that cover the three common features that might be found in a verification server. These are process management, HTTP, and caching.

These features are partly dependent on the specific language for which the verification server is implemented. Due to that fact, the components are usually implemented as traits containing abstract methods, that the developers of a verification server need to implement.

The modularity of the three components allows us to use them in several combinations. Table 7 shows all possible combinations.

Table 7 Possible combinations of VSI modules in a verification server.

Process Management	HTTP	Caching
✓		
✓	✓	
✓	✓	✓
✓		✓
		✓

As can be taken from Table 7, the only constraint on possible combinations is that the `Http` component must be implemented in tandem with the `VerificationServer` trait. Hence, both the `VerificationServer` and the `Cache` components can be used independently. This is useful for clients who may already have implemented some components, and do not wish to refactor them just to add one of the components from VSI.

3.5.2 Specific Use Cases

Currently, there are three specific use cases envisioned for VSI. The first is as part of VCS. In this case, both the `VerificationServer` and the `Cache` component would be used. The `VerificationServer` in this case would manage verification requests for Viper programs. The `Cache` would stores results generated by the verification backends used for the verification of said Viper program.

The second is as part of ViperServer. There, all three components would be used. However, the `VerificationServer` and `Cache` component would be used indirectly by extending VCS. The communication component would be used to issue commands to the server and to stream verification results over an HTTP connection.

The third is as part of GobraServer. In theory, all three parts may be used in GobraServer. In practice, this is yet to be determined.

3.6 Evaluation

The evaluation for this part of the project is less obvious than for other parts of the project. The major reason for this is that VSI is a collection of code that should facilitate the development of verification servers, rather than executable code. Any criterion for evaluation, be it qualitative or quantitative, requires some implementation by a client. However, the only clients that have implemented VSI so far, are VCS and ViperServer.

3.6.1 Unit Tests

Since VCS and ViperServer have implemented VSI, it can be checked for correctness within these two systems.

This was done using the unit tests presented in Section 2.4. The tests in `CoreServerTests` cover the `VerificationServer` and `Cache` components implemented by VCS, while the `ViperServer Tests` mainly cover the `Http` component, as implemented by ViperServer.

3.6.2 Test Scenarios: Viper(Core)Server

Some form of qualitative evaluation can certainly be given by the successful refactoring of VCS and ViperServer using VSI. It shows that the generic code provided by the various components of VSI can indeed be implemented by verification servers. It also shows that the set of features is rich enough to support the development of a specific verification server.

3.6.3 Achieved Goals

The following objectives were successfully met.

1. Providing a library of code implementing common features among verification servers while letting the client implement language-specific features. This code covers the following three functional components of a verification server:
    - Process management and verification
    - HTTP communication frontend
    - Caching
2. The code providing process management and verification offers the following capabilities:
    - Asynchronously and concurrently execute verification tasks and retrieve the corresponding verification messages.
    - Lets the client implement what constitutes a specific verification task.
    - Lets the client implement what constitutes a specific verification message.
    - A system that reactively manages messages produced by executing the verification task.

3. The code providing the HTTP based communication frontend offers the following capabilities:

    - GET and POST methods to initialize a verification process, retrieve results of a process, interrupt a process, and to shut down the server.
    - The possibility to add further, client-specific HTTP methods.
    - The possibility to define a client-specific protocol that specifies the format of the response sent back to the client.

4. The code providing caching offers the following capabilities:

    - A retrieve that takes an AST as input and returns verification results for cacheable members and a modified AST, that takes less (or equal) time to verify.
    - An update method that allows adding verification results for cacheable members to the cache.
    - Lets the client implement client-specific notions of an AST and its (cacheable) members.
    - A layered caching system. The cache is checked both for entire programs and individual members.
    - Let's the client implement client-specific methods necessary for the cache to function.

3.6.4 Further Work

The next steps for this milestone are unclear until other projects instantiate their verification servers based on VSI. This is necessary to get both the qualitative and quantitative feedback required to inform possible improvements.

We want to propose some criteria for a qualitative evaluation of VSI. We expect the criteria described in Table 8 to provide the necessary insight to determine the next steps for this milestone.

Table 8 Possible criteria for a qualitative evaluation of VSI

Criteria	Description
FEATURES	- Are there features common among verification servers that are not covered in VSI? - Conversely, are there features in VSI that should not be expected to be implemented by all verification servers?
ORGANIZATION	- Are the components that hold the features adequately chosen? - Should there be more (or less) modularity?
COMPREHENSIBILITY	- How easy is it for other developers to understand the idea, the organization of the components? - How comprehensive is the code?
EASE OF USE	- Does the code facilitate the implementation of a verification server? - Is the code too generic or too restrictive for it to be useful?

3.7 Discussion

3.7.1 Summary

In this milestone, we identified features that we consider common among verification servers. We then further differentiated these features into language-agnostic and language-specific features.

Based on this, we designed a library called VIS, containing a generic implementation of common features in verification servers. This code can be complemented by a library client to introduce language-specific behavior.

The code was divided into three modules, corresponding to three functional components of a verification server (process management, HTTP frontend, and caching). These modules can be used in different constellations and need not all be implemented together.

Finally, we refactored both VCS and ViperServer to make use of VSI. VCS inherited both the process management and the caching module from VSI, while ViperServer HTTP frontend was refactored using the one provided by VSI.

3.7.2 Challenges

Implementing a generic version of a verification server proved to be a challenging task both conceptually and technically. Conceptually, we found it challenging to strike the right balance between offering generic code and asking the client to complement it with language-specific code. Also, estimating how much we could assume about a client's wishes and needs when implementing a verification server was not always simple. Technically, finding the class- or trait hierarchies that would feel most for a client to implement was often the main challenge.

Wrapping messages generated by the frontend-specific verifiers into generic messages, for example, underwent several changes. In the beginning, we had the client wrap their messages on their own, before passing them to the generic part of the code that manages them. Later, we refactored this into a separate trait. This trait would ask for mappings that would both wrap and unwrap the messages. In doing so, the client would only ever have to deal with their language-specific messages, once they implemented the mappings.

Knowing the balance offering generic code and asking the client to complement it with language-specific code was a challenge when implementing VSI's cache. There, the library code could have been designed to be much simpler to use. In other words, we could have provided generic code that left less abstract methods for the client to define and fewer types to extends. The trade-off, though, would have been that the library would have had fewer features. For example, the client would have had to essentially figure out the logic implemented in the cache's `retrieve` method on their own. We decided that making the library more demanding to implement would result in a cleaner and more abstract caching feature for the client, once implemented. I.e., the client would be rewarded with a `retrieve` method that hides most of the cache's logic behind a level of abstraction.

4. Language Server

4.1 Introduction

In this chapter, we will focus on one of ViperServer's clients, the VS Code extension called Viper IDE. Currently, the Viper IDE extension consists of two parts, both of which are written in Typescript ("TS"). As was explained in more detail in Section 1.1.7, one part is called the extension's *client*, the other the extension's *language server*. The idea of this separation is the division of labor. Computationally intensive work such as code look-up, code completion, analysis, and verification is handled by the server. Less demanding work, such as user interaction is handled by the client. The client and the language server communicate via a dedicated protocol, called **Language Server Protocol** ("LSP").

Because both parts are written in Typescript, they run in a node.js environment. ViperServer, however, runs in the JRE. Consequently, the extension has to make use of the Viper infrastructure by communicating with ViperServer via HTTP. This effectively splits the Viper IDE extension into three parts, across two runtime environments, three codebases, and uses two protocols to communicate.

At the time the extension was introduced, there was no easy way around this architecture. Thanks to the introduction of new technology and the modularization of ViperServer, the architecture could now be improved on. On the one hand, a library called LSP4J [11] now makes it possible to use the LSP in Java (and, by extension, Scala) and run a language server in a JRE. This in turn means that a language server can now be written in Scala. On the other hand, ViperServer is no longer a monolithic HTTP server. The extraction of VCS and allow it to be used with frontends other than HTTP.

4.1.1 Proposition

In this milestone, we propose to rework Viper IDE's architecture and refactor it accordingly. Since the language server can now be implemented in Scala and communicate directly with the client, we propose to remove the existing language server written in Typescript and reimplement it as a part of ViperServer. An illustration of this new architecture is given in Figure 14.

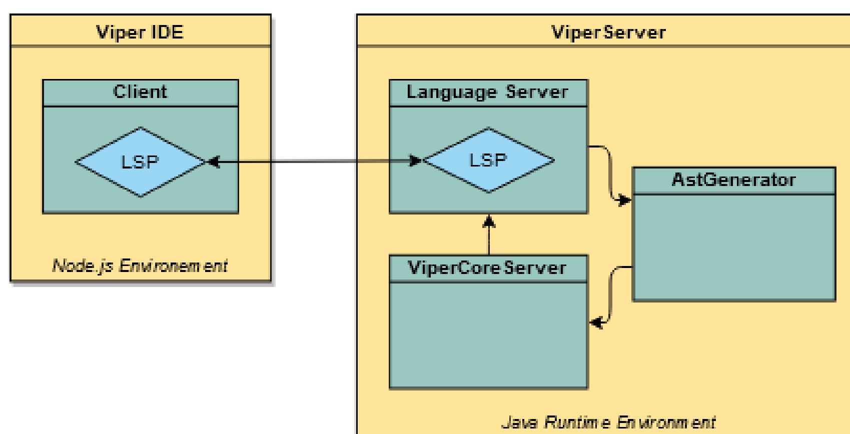

Figure 14 New Architecture of Viper IDE using refactored ViperServer with built-in language server.

Although Refactoring the language server part into ViperServer will require some changes to the client, we propose to keep and reuse as much of the client's code as possible. The client's code presently covers a large amount of functionality such as orchestrating user interaction and the corresponding requests to the language server. Also, the implementation of this functionality is largely independent of the concrete implementation of the language server. Hence, we propose to reuse as much of this code as possible and design the new language server around it.

4.2 Technologies

Before we explain how the language server was designed and structured as part of ViperServer, we want to briefly introduce the LSP.

4.2.1 JSON-RPC 2.0

LSP is based on yet another protocol called JSON-RPC [12]. “RPC” stands for **remote procedure call**. It is a stateless, asynchronous, and transport agnostic protocol. It is used to communicate that a (remotely defined) method should be executed on some data that is passed along. It is also used to respond to such a procedure call. I.e., it allows communicating results that were computed by the method or how to respond with errors. The data format used by the protocol is JSON.

To give a small example, Listing 23 shows how a typical request and response object might look like. The request asks that a method `getAddress` is executed on a JSON object representing a `Person` object. Note that an `id` parameter is sent as part of the request. The same `id` parameter is used in the response to indicate which call the response corresponds to. Finally, the `Address` object computed by the method is sent back.

```
1  {
2    "jsonrpc": "2.0",
3    "method": "getAddress",
4    "params": {
5      "name": "Sherlock Holmes",
6      "age": 60,
7      "female": false
8    },
9    "id": 42
10 }
11
12 {
13   "jsonrpc": "2.0",
14   "results": {
15     "street": "221B Baker Street",
16     "city": "London"
17   },
18   "id": 42
19 }
```

Listing 23 RCP request and response pair.

4.2.2 LSP

An LSP message is divided into two parts, a header, and a content part. The header contains two fields, describing the content’s length and type, respectively. The content part simply contains a JSON-RPC object. LSP also specifies different kinds of messages. The most prominent of which are the request and the notification messages. In LSP, the former expects a response, while the latter does not.

Moreover, LSP specifies a set of notifications and requests that may be used when implementing a language server. These range from general messages like initializing and terminating a language

server, to more specific messages like informing the server about changes to a text file. Alongside this, LSP also specifies corresponding methods, as well as the data they take as input or return as output.

For example, Figure 15 shows the messages that perform the initialization handshake. The client sends an initialize request, which asks the server to execute its `initialize` method. The protocol specifies that the method may expect as arguments an `initializeParams` object. Since it is a request, the protocol also specifies that the method must return an `initializeResult` object. The handshake is completed when the client sends the `initialized` notification

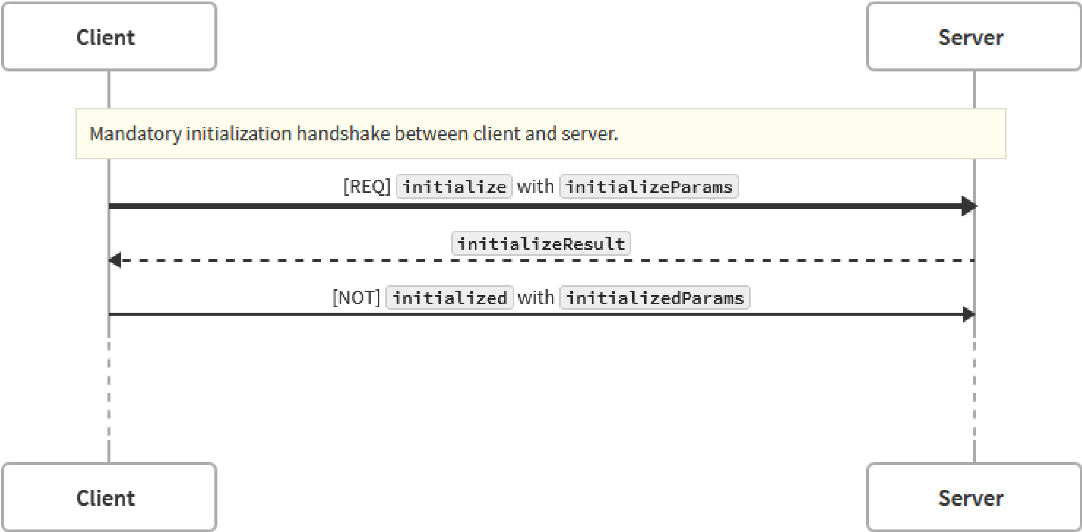

Figure 15 Initialization handshake between client and language server.

The protocol makes very few assumptions about either the client or the server. An important one it does make, though, is that the client is the one who is in charge of the server's lifetime. I.e., the client is the one who starts/ terminates the server and initiates the corresponding handshakes.

4.2.3 LSP4J

This Java library contains packages that provide Java bindings for LSP. As such, it provides bindings to all the methods specified by the protocol, along with the corresponding data structures.

Due to LSP4J being implanted in Java, the library comes with a few idiosyncrasies. For example, an LSP message will not actually call the Java method whose identifier corresponds to the method name provided in the message. Instead, it calls the method that is *annotated* with the method name sent in the message. Hence, the actual identifier of the Java method doesn't matter in terms of LSP; the annotation is what matters. This also implies that any method that's to be called by a client within a language server must have an appropriate `@JsonNotification` or `@JsonRequest` annotation containing the method's name. An example of this is given in Listing 24, by demonstrating how the handshake shown in Figure 15 would look when implemented with LSP4J. The method name as specified by LSP is `initialize`, as is reflected in the annotation.

```scala
@JsonRequest("initialize")
def onInitialize(params: InitializeParams):
   CompletableFuture[InitializeResult] = {...}

@JsonNotification("initialized")
def onInitialized(params: InitializedParams): Unit = {...}
```

Listing 24 Scala implementation of the LSP initialization handshake using LSP4J.

Another LSP4J-specific behavior of LSP concerns the return type that is expected from a method execution. For a message of type notification, the method is expected to return `Void`. In the case of a request, a method is expected to return a `CompletableFuture` of the object that is going to be transported to the server. This can also be seen in Listing 24, where `onInitialize` does not return `initialize Result`. Rather, it returns a `CompletableFuture[initializeResult]`.

4.3 Redesigning the VS Code extension

4.3.1 Client

Due to the separation of the VS Code extensions into a client and language server, the client runs independently from the server. While the two components must be able to communicate and understand each other, the technology used for one is independent of the other. In the case of Viper IDE, this means that the client can use a server written in any programming language that runs in any environment. The client and server need only to be are aware of their respective capabilities and to agree on the messages that are being sent. In conclusion, we should be able to reuse most of the client's functions when attached to a new language server.

The only major change introduced to the client is the starting procedure of the language server. Whereas the client previously had to start a server in a node.js environment, it must now start a server run in a JRE. This means that both launching the server process and setting up inter-process communication have to be done explicitly when activating the client. As is shown in Listing 25, the server is launched by spawning a child process that executes the language server's assembled JAR file. As can also be taken from Listing 25, sockets are used for inter-process communication. Consequently, information about address and port are passed as arguments when executing the server's JAR file.

```
1  function startViperServer(): Promise<StreamInfo> {
2      return new Promise((resolve, reject) => {
3          let server = net.createServer((socket) => {
4              console.log("Creating server");
5              resolve({
6                  reader: socket,
7                  writer: socket
8              });
9              socket.on('end', () => console.log("Disconnected"));
10         }).on('error', (err) => {throw err;});
11
12         // grab a random port.
13         server.listen(() => {
14             // Start the child java process
15             let serverBin =
                   "C:\\Users\\Valentin\\Desktop\\viperTools\\viperserver\\target\\
                   scala-2.12\\viper.jar"
16
17             let args = [
18                 '-jar',
19                 serverBin,
20                 (server.address() as net.AddressInfo).port.toString()
21             ]
22
23             let process = child_process.spawn("java", args);
24
25             // Send raw output to a file
26             let logFile = context.asAbsolutePath('languageServerExample.log');
27             let logStream = fs.createWriteStream(logFile, { flags: 'w' });
28
29             process.stdout.pipe(logStream);
30             process.stderr.pipe(logStream);
31
32             console.log(`Storing log in '${logFile}'`);
33         });
34     });
35 }
```

Listing 25 Client-side code to Initialize ViperServer in language server mode.

4.3.2 Language Server Sender/Receiver

The new language server can be thought of as ViperServer with a different frontend. I.e., instead of using HTTP to communicate with an HTTP client, ViperServer now uses LSP to communicate with the client-side of the Viper IDE extension.

To conform to LSP, a few new classes have been introduced to ViperServer. In the `receiver.scala` file, the `StandardReceiver,` and the `ExtendedReceiver` class deal with receiving messages from the client. I.e., they implement the methods that a JSON-RPC object carries when sent to the server. An example of two such methods can be seen in Listing 24. Note that `StandardReceiver extends LanguageClientAware`, which inherits the `connect` method. This method must be Implemented to register a client in the LSP4J library. The reason for distinguishing between `StandardReceiver` and `ExtendedReceiver`, is that some of the methods we want our language server to execute are not specified by LSP.

To illustrate why we need additional methods, consider the following example. LSP specifies a `didChange` method that should be executed by the server when the client detects changes in a text document. In our case, this method can be used to automatically trigger verifications in ViperServer. Despite that, we also want the user to be able to manually request a verification, without having to change the text. In order to do so, the client currently provides a "run verification" command. When used, it should notify the language server that the `verify` method should be executer with the corresponding parameters. LSP, though, does not specify a `verify` method and corresponding data structures. However, since LSP is based on JSON-RPC, it can easily be extended. In our example, a message conforming to the LSP will be sent, containing a custom JSON-RPC object.

We designed a similar structure for messages leaving the server. In the `IdeLanguageClient` trait, we specified custom notifications and requests that the server wants the client to execute. Additionally, `IdeLanguageClient` inherits from the `LanguageClient` trait. `LanguageClient` is defined by LSP4J and contains all notifications and requests specified by LSP that go from the server to the client.

In order to support custom notifications and requests, the protocol also needs to be extended in terms of data structures. To that end, both the server and the client contain a description of all the data structures that are sent between them. In the server codebase, these data structures are defined in `DataProtocol.scala`.

4.3.3 Server State

The state requires to keep track of a few variables during its lifetime. A typical example of this is the reference to the client, information about network address and port, a reference to the backend, etc. This is all stored in a global object called `Coordinator`.

More importantly, though, the server needs to keep track of the state of the files currently active in the editor. To that end, the `Coordinator` holds an instance of a `FileManager` for each Viper file active in the editor. The `FileManager` stores information about the file itself (name, type, etc.), as well as the state of verification and results of previous verifications. E.g., whether verification is running, aborting, stopped, or done and whether or not it resulted in errors.

4.3.4 Verification in the Language Server

The `ViperServerService` (VSS) class is where the language server integrates the Viper verification tools. This is done by extending `ViperCoreServer`.

Aside from the methods provided by the VCS API, VSS also extends VCS with a few additional methods that help to manage the extension's state. Recall that the client runs an event-driven state-machine. This machine's state not only depends on the client's state but also on the server's. For example, when starting the backend (in our case, ViperCoreServer), the machine expects a notification from the server, confirming that the backend is ready to use. VCS has a similar method called `start`, which has to be called to initialize an instance of VCS. VSS, therefore, extends this `start` method to a `setReady` method that additionally notifies the client that the instance of VCS is now ready to use.

4.3.5 Consumer Actor

To extract messages from the verification backends in VCS, the `streamMessages` method must be used. This method takes an actor, that receives these messages and deals with them, as an argument. In the case of the language server, this actor is called `ConsumerActor`.

Depending on the type and content of the message, its job is to either send notifications to the client and/or to update the state of the `FileManager` for the file being verified. When receiving information about a Viper program's definitions, for example, the definitions are stored in the `FileManager`. They will only be sent to the client they request them. In the case of `OverallResult` messages, on the other hand, a notification is sent to the client. That notification indicates to the state machine that the verification is over and communicates the corresponding results.

4.4 Evaluation

4.4.1 End-to-End Tests

We tested the changes to the Viper IDE's client and the incorporation of the language server into ViperServer using an end-to-end approach. We tested the implemented functionality by starting an instance of Viper IDE and using the features. The following list shows the tested features as well as the expected behavior.

- **Automatically starting Viper IDE.** Viper IDE should automatically start when an active Viper or Silver file is detected (these are files ending with *.vpr or *.sil) in the editor.
- **Automatically launching ViperServer in language server mode.** When the ViperIDE extension is started, it should automatically spawn a child process that executes the assembled JAR files of ViperServer in language server mode.
- **Automatically verify a file when starting the extension.** After starting up, Viper IDE should automatically start verifying the file in the active editor tab. We tested this on the files, `sum_method.vpr,` and `verification_error.vpr`, both taken from the previous chapter's unit tests. One is expected to verify with success, the other is expected to return a verification error. The verification should be displayed as a squiggly line below the first invariant of the while loop contained in the method.
- **Manually verify a file.** Using the corresponding command in VS Code's menu or pressing F5 should trigger a verification request on the file in the active editor tab.
- **Automatically reverify file when text is changed.** Changing the text in the editor should automatically trigger a verification request on the file in the active editor tab.
- **Caching.** Reverifying a previously verified file should return cached results. In the case of a verification error, this information should be displayed when hovering over the error.
- **Flushing the cache.** Using the corresponding command in VS Code's menu should reset the cache's state. This should manifest itself by removing the cache indication on a verification error when verifying it a third time.
- **Swapping backends.** Using the corresponding command in VS Code's menu should swap the backend used for verification. Currently, the menu should propose two backends: Silicon and Carbon. This should be indicated by a label on VS Code's status bar.

4.4.2 Achieved Goals

The following objectives were successfully met.

1. Providing an LSP-based frontend to ViperServer that replaces the language server previously implemented in Typescript. As such it allows to:
    - Communicate with the client-side of the Viper IDE extension.
    - That can make use of the verification features provided by VCS

2. Providing an LSP based client that allows to:
    - Start and terminate an instance of ViperServer in language server mode.
    - Communicate with the ViperServer.
3. Providing a refactored version of the existing Viper IDE based on points 1) and 2). This new version currently offers the following features:
    - Starting/Stopping a verification backend.
    - (Automatically) Executing verification requests.
    - Managing the verification state and results across multiple files.
    - Using VCS's cache when performing verification.
    - Displaying verification errors for multiple.
    - Triggering a verification request manually.
    - Flushing the cache.
    - Swapping between the verification backends Silicon and Carbon.

4.4.3 Further Work

While the new language server implements all essential features that should be present in Viper IDE, further work is necessary for it to implement the same set of features as the old language server did. Concretely, the following has to be done.

- **Checking/ updating tool versions.** Viper IDE was able to check the versions of tools (like Java, ViperServer, etc.) on start-up and, if necessary, update them.
- **Simplifying and including settings.** Since Viper IDE could previously connect to various verification backends, settings were necessary to keep track of user preferences. Due to the language server being embedded into ViperServer, this is no longer the case. The settings should therefore be simplified.
- **Include Debugging and counterexamples.** Features around Visual Debugging for Symbolic Execution [13] have not yet been ported to ViperServer.
- **Testing and debugging**. Not all features behave exactly like the previous implementation of Viper IDE did. Further testing might be necessary to ensure that the same set of features is available as were previously available.

4.5 Discussion

4.5.1 Summary

In this milestone, we have complemented ViperServer with an additional frontend based on LSP. This frontend has effectively allowed us to embed a language server for verification IDEs into ViperServer. Consequently, we have able to simplify and port the code from the previous implementation of the language server.

We have made changes to the existing client of Viper IDE that allows us to connect it directly to ViperServer. Doing so has consolidated the architecture of Viper IDE by eliminating the previous implementation of the language server. It has also reduced the number of interfaces necessary between the codebases, as the client and ViperServer can now communicate directly.

4.5.2 Challenges

The main point of contention in this milestone was whether the client should be reused or whether a new client should have been built alongside the new language server. As has been stated in Section 4.3.1 we have decided to keep the client as was and to adapt it to start ViperServer as a language server. However, we have not given a reason for this. Instead, we want to discuss this design choice here.

Reusing the client has several benefits. Firstly, it allows keeping the same set of features that the client provides. For example, the user interface in the editor, the displaying message, as well as the commands that the editor makes available to a user can be reused. This is advantageous for users familiar with the IDE. Keeping the client the same means users will be faced with essentially an identical IDE. Another benefit is that the client is currently operated by a state-machine that orchestrates user interaction and language server services. This feature is particularly helpful in the asynchronous and event-driven environment that the editor is. It keeps track of commands issued by the user, sets up their processing, and reports their results when the server responds.

Conversely, one could also argue that redesigning a client from scratch might have been a better solution. Consider, for example, that LSP has been significantly expanded compared to when the IDE was originally designed. Also, Viper IDE now relies on a single verification engine that is directly embedded into the server. Both of these technical improvements might lead to a simpler and cleaner implementation of the client.

5. Conclusion

The common theme throughout this thesis was to modularize and generalize existing verification tools to facilitate the development and maintainability of other projects. In this chapter, we want to take a look back and see what this thesis has achieved. We will contrast the previous and current state of the various projects it has affected. Finally, we will summarize what this has enabled us to do and what it might enable others to do in the future.

5.1.1 ViperCoreServer

The idea of modularity is inherently present in Viper. As an intermediate language, it allows other languages to reuse the verification backends built for it. This can be done by building a frontend translates it a particular programming language to Viper.

So far, the projects built atop Viper had not been able to take full advantage of this idea. While the backends could be reused, each frontend had to invoke the Viper backends and manage verification of Viper programs on their own. Ideally, there should be an additional layer of abstraction between the backends and the frontends. I.e., the details of using verification backends should be hidden from the frontend by some API that each frontend can use.

This was done by modularizing parts of ViperServer that were responsible for verification. I.e., we extracted verification-related functionality from ViperServer and modularized it into a separate server called VCS. This includes additional features like process management (allowing asynchronous and concurrent verification) and a caching mechanism. We made these features accessible through an intuitive and abstract API.

By extracting VCS, we enabled to reuse of the verification backends for Viper in other projects. Since it would be programmatically accessible, it could be embedded into other existing verification servers or future verification servers built atop the Viper infrastructure. In fact, the Gobra verification server has already successfully incorporated VCS.

5.1.2 Viper IDE

In recent years, several projects have recognized the importance of incorporating verification software into IDE. Examples of this include projects like Dafny [14] or jStar-eclipse [15]. Dafny, a programming language and verifier for static program verification, has been made available as an extension to Visual Studio. jStar-eclipse, a tool that verifies Java programs, has been made available as a plugin to the Eclipse IDE. Integrating these tools into IDEs allows to fully automate the verification process. Using an IDE as a user interface for verification tools makes the verification faster and increases usability. Users no longer need to manually issue verification requests through CLI and read through console output. Instead, they can edit code and get visual feedback on the code they wrote.

Viper-related projects have also recognized the importance of integrating their tools into IDE. Several verification IDEs like Viper IDE, Gobra IDE, and Prusti assistant have worked on integrating their respective verifiers into the editor VS Code.

Currently, Viper IDE is split into 3 distinct parts. A client and a language server, both written in Typescript, make up the VS Code extension itself. Additionally, ViperServer is used as a verification engine to power the extension. In this structure, the client acts as the interface to the user and ViperServer executes the verification requests. The language server acts as a middleman between the two. Since ViperServer only has an HTTP frontend and the extension's client only understands LSP, the language server is needed to orchestrate communication between the two other components.

The structure described for Viper IDE is not ideal. If the client and ViperServer could understand each other, the language server could be removed. This would make the codebase smaller, more maintainable, and easier to expand. At the time, this was unfortunately not easily feasible. A similar problem was also experienced by the VS Code extension for Dafny [16]. The Dafny extension was also built as a three-part structure, analogous to Viper IDE's. The language server used for verification communicated with the extension's client via an intermediate language server. In the case of the Dafny extension, the verification server has been redesigned to communicate with the client directly, thereby removing the intermediate language server.

With the extraction of VCS, we can now refactor ViperServer to support additional frontends beside HTTP. In particular, it means that it can now be complemented with an LSP frontend. This was made possible by the LSP4J library, which offers Java bindings for LSP. Consequently, we refactored ViperServer to contain a second frontend. This additional frontend allowed us to turn ViperServer into Viper IDE's language server. In turn, this allowed us to remove the Typescript language server, as ViperServer can now directly communicate with the client.

Embedding the language server directly into the verification server also facilitates the development of further extensions for other editors. Presently, most editors follow the same principle for the development of extensions. I.e., they also divide extensions into clients and language servers that communicate via LSP. This should allow developing extensions for other editors by creating only a client for the specific editor. The refactored version of ViperServer can then be reused as the extension's language server.

5.1.3 VSI

We have argued that the modularity found in the Viper infrastructure facilitates the development of verifiers by making the verification backends reusable to other programming languages. We have also shown the importance of integrating verifiers into IDEs. To be used in IDEs, the frontends built atop Viper must be embedded into a verification server. These servers typically provide additional features besides verification. For example, management of verification request and verification processes, providing a communication frontend such as HTTP or caching.

With VSI, we provide a new Scala library that contains code that implements features common among verification servers. Since each verification server is based on a specific language, we designed the code in such a way that a client can complement it to fit their specific use case. The library is divided into three distinct parts. The first part provides functionality to manage the server and orchestrate verification processes. It allows us to concurrently execute verification tasks specified by the user. The second part is an HTTP frontend that allows clients to interact with the server. The last part provides a generic caching mechanism.

Caching verification results is not a new concept. In Boogie, for example, it was shown that fine-grained caching can have a significant impact on the verification time [17]. In VSI, we focus on a more

coarse-grained caching mechanism. In fact, we provide two levels of caching that can be used independently. On the lower level, we allow the client to cache results for individual program members. On a higher level, we allow the client to cache entire programs. The idea behind this mechanism is to only use the lower level cache if the higher-level cache misses. Since a cache hit on the program-level implies that each member will also hit the cache, a lot of effort can be avoided.

The caching mechanism also poses interesting questions to any client who implements it. For example, how do they compare entire programs efficiently? For instance, comparing the textual representation of two programs to determine equivalence will only catch a small subset of equivalent programs (equivalent w.r.t. verification). An improvement on this could be to compare the AST representation of these programs. This might help to extend the subset to include programs where whitespaces were altered. Currently, this is how ViperCache implements this equality method. It generates pretty-printed strings of each AST, which it then compares.

Also, for which of their AST members will the clients store verification results? In ViperCache, for example, only methods are cached. However, this is not set in stone. For instance, it might be interesting to try to expand the number of cacheable members. Maybe the ViperCache implementation of VSI could be changed to also cache functions or predicates?

As a concluding remark, we want to suggest a possible addition to VSI. This suggestion has is based on the fact that language servers can now be embedded into verification servers. As summarized in the last chapter, this has been shown in the case of ViperServer and Viper IDE. However, it stands to reason that other Scala-based verification projects will also implement corresponding verification IDEs. This is already the case for Gobra, who has released Gobra IDE. We know also know from Gobra IDE that they use a similar architecture for their project as Viper IDE. I.e., using LSP to connect a Typescript client to a Scala language server. Therefore, we could argue that an LSP frontend could become just as common in verification servers as an HTTP frontend. This addition would then allow other verification servers to be turned into language servers. This in turn would allow these verification servers to be used to implement verification IDEs in several code editors.

6. References

[1] K. Leino, „This is Boogie 2“, 2016.

[2] L. de Moura und N. Bjørner, „Z3: An Efficient SMT Solver“, in *Tools and Algorithms for the Construction and Analysis of Systems*, Bd. 4963, C. R. Ramakrishnan und J. Rehof, Hrsg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 337–340.

[3] P. Müller, M. Schwerhoff, und A. J. Summers, „Viper: A Verification Infrastructure for Permission-Based Reasoning“, in *Verification, Model Checking, and Abstract Interpretation*, Bd. 9583, B. Jobstmann und K. R. M. Leino, Hrsg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, S. 41–62.

[4] S. Walker, „IDE Support for a Golang Verifier“, S. 53, Sep. 2020.

[5] J. Dunskus, „Developing IDE Support for a Rust Verifier“. Aug. 2020.

[6] „Gobra“. https://github.com/viperproject/gobra.

[7] V. Astrauskas, P. Müller, F. Poli, und A. J. Summers, „Leveraging rust types for modular specification and verification“, *Proc. ACM Program. Lang.*, Bd. 3, Nr. OOPSLA, S. 1–30, Okt. 2019, doi: 10.1145/3360573.

[8] R. Kälin, „Advanced Features for an Integrated Verification Environment“, S. 121.

[9] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson, „The Reactive Manifesto, v2.0“. Sep. 16, 2014, [Online]. Verfügbar unter: https://www.reactivemanifesto.org/.

[10] Lightbend, „Akka Libraries“. https://akka.io/docs/.

[11] Eclipse Foundation, „Eclipse LSP4J“. https://github.com/eclipse/lsp4j.

[12] „JSON-RPC 2.0“, *JSON-RPC 2.0 Specification*. https://www.jsonrpc.org/specification.

[13] A. Aurecchia, „Visual Debugging for Symbolic Execution“, S. 109.

[14] K. R. M. Leino, „Dafny: An Automatic Program Verifier for Functional Correctness“, in *Logic for Programming, Artificial Intelligence, and Reasoning*, Bd. 6355, E. M. Clarke und A. Voronkov, Hrsg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 348–370.

[15] D. Naudziuniene, M. Botincan, D. Distefano, M. Dodds, R. Grigore, und M. J. Parkinson, „jStar-eclipse: an IDE for automated verification of Java programs“, in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE ’11*, Szeged, Hungary, 2011, S. 428, doi: 10.1145/2025113.2025182.

[16] Kistler und Hess, „Dafny VSCode Server Redesign“. Dez. 2019.

[17] K. R. M. Leino und V. Wüstholz, „Fine-Grained Caching of Verification Results“, in *Computer Aided Verification*, Bd. 9206, D. Kroening und C. S. Păsăreanu, Hrsg. Cham: Springer International Publishing, 2015, S. 380–397.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Universal Library Components for Verification IDE Development

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Racine	Valentin

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
23.10.2020	V. Racine

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.