Reusable Components for Verification IDEs

Bachelor's Thesis Project Description

Valentin Racine

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

Department of Computer Science, ETH Zurich, Switzerland

March 2019

Current Situation

Currently, Viper IDE is implemented as an extension of Visual Studio Code (VSC). This extension allows to write and (automatically) verify Viper files, as well as having the verification results logged and displayed in VSC. It currently consists of the following three components

1. VSC Extension Language Server
2. VSC Extension Client
3. Server running Viper called ViperServer

The first two make up the VSC extension itself, the last one is where the actual verification is done.

The reason for running Viper on a server is that Viper is a stand-alone tool. To be more precise, it is an application that runs inside a Java Runtime Environment (JRE). This means that each invocation of Viper requires restarting a JRE, which takes a considerable amount of time. This start-up time is avoided by ViperServer as it can manage multiple instances of Viper inside a single JRE. Running Viper as part of a server also allows to integrate it into any environment via HTTP. As such, it can provide functionality like caching results or managing several (concurrent) verification requests. I.e., upon receiving a verification request, the server can process it independently. Programs written in the Viper language can be verified using different verification backends. The two backends in use are

- Silicon
- Carbon

The verification backends are currently library dependencies of the ViperServer implementation. Viper's verification backends have external dependencies, for example Z3 (an SMT solver) and the Boogie verifier. These are invoked as stand-alone applications, outside of the JRE that runs ViperServer.

As was mentioned earlier, the VSC extension consists of two components. The idea behind this split is to outsource workload, thereby increasing performance and modularity of the IDE. That way, the client-side part of the extension can handle lightweight tasks such as displaying results as part of the editor's GUI. The language server part on the other hand can then perform long-running tasks. In the case of Viper IDE however, the language server's main task is to send/receive HTTP messages to/from an instance of ViperServer. While the two servers communicate via HTTP, the client and Language Server communicate using a dedicated protocol called Language Server Protocol (LSP).

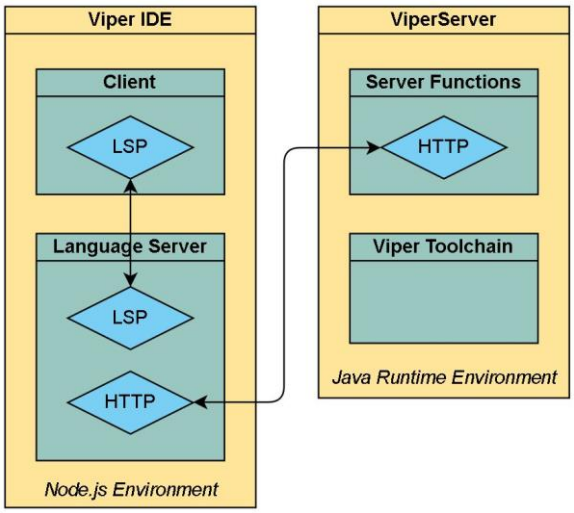

Figure 1 Model of components and their relation currently used in Viper IDE

Clearly, the current version of Viper IDE already has some level of modularity incorporated but it is not fully modular. On the one hand, ViperServer is a self-contained module consisting of modular parts like backends. This allows, for example, to run the verification tasks on a remote server. It also makes it possible to choose which parts of the verification toolchain should be used (e.g. Silicon vs. Carbon verifier). On the other hand, it is currently hard to reuse the codebase of ViperServer for other verification-oriented IDEs built atop Viper. For instance, a typical Viper-based verifier must map Viper-level verification errors back to the source language before reporting them to the IDE, but the verification result streaming functionality of ViperServer is tightly integrated with the load-balancing infrastructure and cannot be easily customized for a new language.

Limitation of the current design

This brings new requirements, challenges and opportunities to continue developing a modular architecture for Viper as a verification IDE.

Currently, a lot of work is done to build Viper-based verifiers for various programming languages. Tools like *Nagini* and *Prusti* already provide verification for the programming languages Python and Rust. A tool called Gobra is planned for the language Go, too. These

tools are called Viper frontends as they translate the source language into Viper. As long as there is a principled translation of a program from its source language to Viper, the verification can be done on that translated Viper program. This in turn means that the main difference between these tools comes from parsing and translating the programming language in question and mapping the verification results of the Viper encoding to the program in the source language. Despite that, the current modules of the Viper IDE project (Viper IDE extension and ViperServer) cannot be easily reused in the IDEs for the frontend languages.

Of course, an IDE could be written for all these tools independently. Let AssistantX be a VSC extension for frontend X. Following the current architecture for Viper IDE one could write a two-part VSC extension that implements AssistantX and have it communicate (via some protocol, possibly HTTP) with a verification server called ServerX. Such a verification server would then process the request by running the corresponding frontend tool and the Viper toolchain. Furthermore, the same could be done for frontend Y, Z, and so on. Looking at the Figure 2 of such models hints at why this is not an efficient approach.

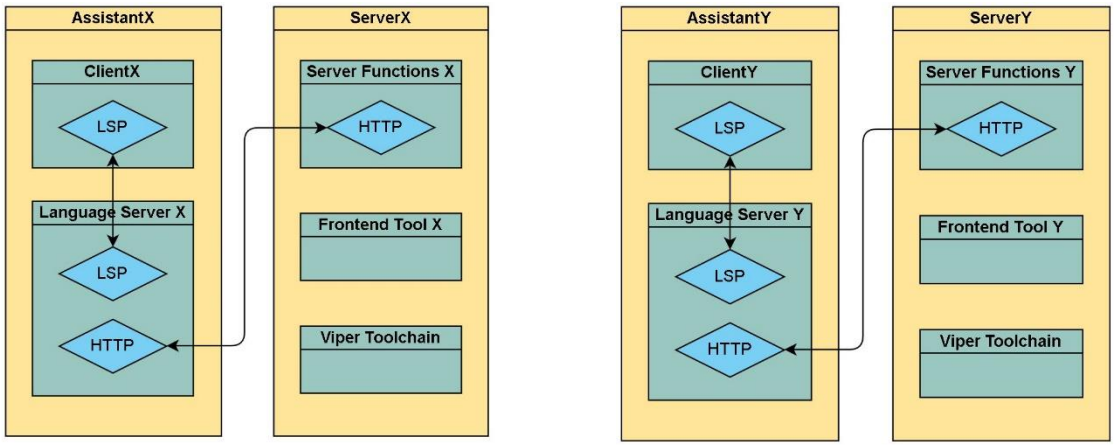

Figure 2 IDE models for several Viper frontends

Improved Architecture

There is a lot of common functionality that could be reused. First, the component responsible for the basic server functions, such as receiving /sending messages via some protocol, could be extracted and reused across all extensions. Correspondingly, the part of the VSC Language Server that is responsible for communication - to both the verification server and the VSC Client - could also be extracted. Second, tasks like managing requests (load balancing, caching, etc.) should be the same for any verification server. This suggests that they can be extracted as well. To some degree, the same is true for client-side part of the extension. An example thereof is displaying success or error messages of the verification process. Given such a message, printing its content to console or displaying it in the editor's GUI is a functionality that can be reused across IDEs for any Viper frontend.

We propose a new, more generalized architecture that meets these demands. At its core are two new modules that help extracting the common functionality mentioned earlier:

1. **VSC Verification Toolbox**
2. **Verification Server Interface** (VSI)

The first module is a typescript library that provides functionality for various verification-oriented VSC extensions. The second module is an abstract class in the sense of Java. The client can extend this component, implement the language-specific features, and reuse the language-independent functionality. For example, deciding when the cache should be invalidated is specific to the language (so the client must implement the method isCacheStillValid), but managing the cache can be done the same way for e.g. Viper and Go, and that should be implemented in VSI. Another important feature provided by VSI is the HTTP server that can stream messages (verification results) to the IDE; having this module standardised allows us to also reuse the HTTP client, which should be provided by the VSC Verification Toolbox

Naturally, building around these two modules affects Viper IDE's architecture as well. We propose a refactoring of the current Viper IDE infrastructure that capitalises on the new, modular architecture. This refactoring will consist of three parts:

1. Add component called **ViperCoreServer**
2. Let ViperServer implement abstract methods of VSI and inherit common methods
3. Rewrite the **Viper Language Server** as a component of ViperServer (in Scala)

The first step is to create a Scala service that can be used by ViperServer as well as any other Viper-based verification project. It will provide functionality related to the verification of Viper programs. In addition to the current implementation it will also be able to take the AST of a Viper program as input, rather than a Viper file.

The second step is to make use of the VSI. As previously explained, this will consist of implementing language-specific features, and reuse the language-independent functionality provided by VSI.In the third step, we will replace the Language Server part of the current

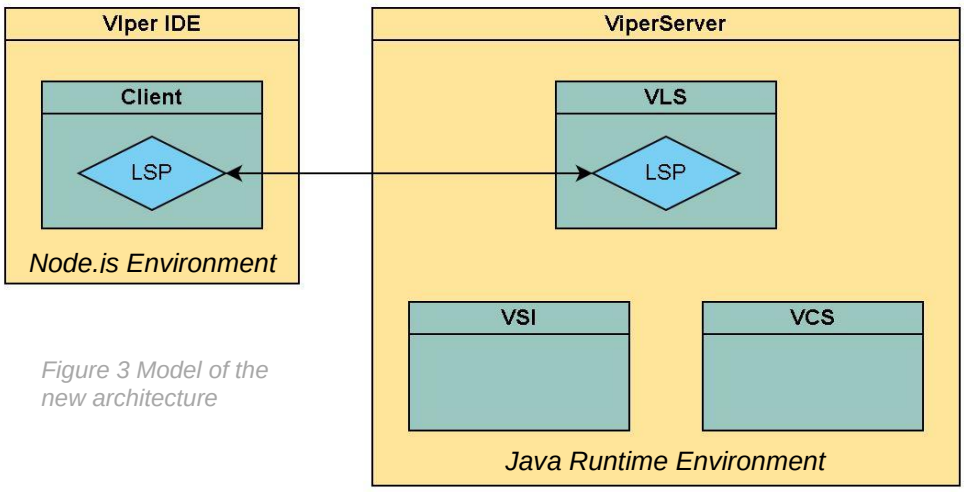

Figure 3 Model of the new architecture

Viper IDE with a server written in Scala. As such it will provide functionality related to viper programs such as parsing and type checking. In order to communicate with Viper IDE, it will also have to comply with the LSP.

The milestones of this thesis are:

- Designing and implementing VSI.
- Refactoring Viper IDE
    - Implementing ViperCoreServer.
    - Implementing Viper Language Server in Scala.

To reach these milestones, we set the following intermediate goals.

Priority	**Component**	**Task**	**Description**	**Workload (est. weeks) Total = 24**
Core	—	Learning	Get a good understanding of current code base for - Viper IDE - ViperServer.	2
	ViperCoreServer	Design, implement, test	- Collect requirements - Identify portions of the current code that already implement the desired functionality - Work on the implementation - Review the implementation/ write tests.	2
		Refactor ViperServer Using ViperCoreServer	- Identify portions of current code that need replacing - Work on the refactorization - Review refactorization/ write tests.	2
	Verification Server Interface	Design, implement, test	- Collect requirements - Identify portions of the current code that already implement the desired functionality - Work on the implementation	3

			- Review the implementation/ write tests.	
		Refactor ViperServer using Verification Server Interface	- Identify portions of current code that need replacing - Work on the refactorization - Review refactorization/ write tests.	2
	Viper Language Server, Viper IDE	Rewrite Viper Language Server in Scala	- Collect requirements - Work on the implementation - Review the implementation/ write tests.	5
Extensions		Enable continuous integration (CI) for testing Viper tools through ViperServer		4
	Viper IDE	Refactor Viper IDE extension to use common functionality provided by VSC Verification Toolbox	- Identify portions of current code that need replacing - Work on the refactorization - Review the refactorization/ write tests.	
		Contribute new features to the VSC Verification Toolbox	- Identify important features - Work on features - Review features/ write tests.	
Core	—	Thesis	- Draft text, discuss scope and structure - 2-3 revisions.	4