Advanced Features for an Integrated Verification Environment

Ruben Kälin
rukaelin@student.ethz.ch

Master Thesis
November 2016
https://bitbucket.org/viperproject/viper-ide/

Supervisor
Arshavir Ter-Gabrielyan

Abstract

Currently, in software verification, the IDE support is insufficient, especially for symbolic execution. Many verification technologies are being developed, but only few IDEs target software verification. For example, users of the Viper framework had only little assistance in writing, and no assistance in debugging their software so far. The common way of invoking the verifier is through the command line, as a sufficient IDE support has been missing. An IDE capable of handling more than one language is desirable, as there are many small verification languages.

Without proper IDE support writing, verifying, and debugging software is cumbersome. Many of the tasks normally handled by an IDE need to be performed manually. A capable IDE solution could reduce the overhead of verifying software and thereby improve the programmer's productivity.

In this Master's thesis, we built a cross-platform IDE for creating Viper programs. The IDE assists the user in the process of writing, verifying, and debugging Viper source code. This support ranges from automatically performing tasks at the right time to visualizing the symbolic states and counterexamples. We provide intuitive debugging support for traversing all states of symbolic execution and allow the user to visually compare states. The Viper IDE is based on Microsoft Visual Studio Code. We use the Viper toolchain as a verification engine.

We provide a solution for symbolic execution, whereas the existing solutions focus on a different verification approach. We manage to visualize the internal state of the verification in a usable way. The implemented solution is highly configurable and allows for an easy integration of additional tools, for example, for specification inference. These novel features allow the user to explore failed verification attempts and thereby understand the cause of the problem. Moreover, the implemented solution generally improves the usability of the Viper framework.

Acknowledgment

First and foremost, I would like to thank my thesis advisor Arshavir Ter-Gabrielyan of the Chair of programming methodology at ETH Zürich. He guided the project and provided invaluable feedback. His door was always open whenever I had a question, and he took the time for testing the system and proofreading the thesis.

My thanks go to Prof. Dr. Peter Müller, head of the Chair of programming methodology at ETH Zürich for providing the big picture and valuable feedback.

During the course of this project many people were involved in answering technical and conceptual questions and testing the system. I am grateful for Malte Schwerhoff's help with all verification backend-related issues. He also assisted the project planning with insightful ideas. I would like to thank Dimitar Asenov for his support in taking the right decisions for the user interface of the project. My thanks go to Marco Eilers, Dr. Alex Summers, and all other testers for detecting and reporting issues.

Finally, I want to express my gratitude to my family and friends, especially Amanda Jenny for their support and continuous encouragement throughout the thesis. Thank you for taking your time to proofread the thesis. Without you this thesis would not have been possible.

Contents

1

Introduction

Tools for verifying program correctness are becoming more and more powerful and widely used. However, it is still difficult for a non-expert in program verification to debug verification errors. Current verification environments provide helpful support for verifying properties of software, such as syntax highlighting, auto-completion, and visualization of syntax errors, but lack assistance for fixing errors in proofs.

1.1. Problem Statement

Modern verification tools are complicated. In case of a verification failure, the warnings and error messages shown to the user are often cryptic due to being generated from the satisfiability modulo theories (SMT) solver at the bottom of the program verification toolchain. Figure 1.2 shows the Viper framework [Müller et al. 2016]. Moreover, it is often already difficult to distinguish spurious errors, e.g., those originating from an incompleteness of the SMT solver from problems in the specification or in the code to be verified. For a spurious error the specification might need to be reformulated in order to enable a successful proof. Another possible reason for the verification to fail is a timeout, in which case simply increasing the timeout might solve the issue. There could be non-terminating verifications such as those caused by matching loops. If the problem is due to a wrong specification, it is unclear where to apply the fixes, because the error message often only points to the assertion it failed to prove. In case the SMT solver found a counterexample, verification experts can track down the source of the problem in a tedious and manual process.

The current state of tools requires considerable expertise and leaks abstractions, e.g., SMT solvers. This places a barrier to adoption for many employed and hobbyist developers and

verification remains in the realm of experts. It is desirable to automation this process to improve the productivity of verification experts and enable software developers to more easily localize and diagnose the root cause of verification errors.

1.2. Goals

In the current project, we address the problem of simplifying the analysis of failed verification attempts by building a debugger for symbolic execution (SE). Our project is embedded in the context of the Viper program verification infrastructure [Müller et al. 2016]. The debugger enables software developers to diagnose verification errors they were previously not able to track down. Additionally, we want to increase the productivity of verification experts.

The project can be structured into three parts as shown in Figure 1.1. Each part is important and therefore, a good verification IDE should address all of them.

Automation concerns inference, relieving the programmer from manually specifying the entire program. Moreover, performing simple and repetitive tasks should be automated. Those include automatic saving, formatting, and verification invocation. Also, the user benefits from a simplified workflow thanks to features like syntax highlighting and code completion.

Integration covers the use of external tools. The integrated verification backends are the most prominent examples for such tools in the IDE. Others include inference tools such as Viper's specification inference tool [Ferrara and Müller 2012]. A good verification IDE is flexible enough to be able to handle different external tools, ideally even supporting tools not known when implementing the IDE.

In the feedback part, we are concerned with showing the necessary information to programmers in order to assist them in finding the reasons for failed verification attempts. The previous IDE already provided integration of the verification backends and simple support for automation-related tasks. Our main focus during the project was on the feedback part. Moreover, when comparing our IDE to an IDE for imperative programs, the largest discrepancy between their desired capabilities lies in the feedback part. While automation and integration have similar requirements, the suitable mechanisms for providing useful feedback differ considerably more. For example, a verification IDE needs to visualize the states of the verification, whereas in a traditional IDE there are no such states.

There already exist tools for debugging verification condition generation [Le Goues et al. 2011]. Therefore, we chose to implement an IDE solution primarily targeting symbolic execution. The goal is to provide debugging support for SE while designing the system such that a future extension of the debugging functionality to VCG would also be possible. In Chapter 3 we analyze the requirements of this project in more detail and explain the planned tasks one by one.

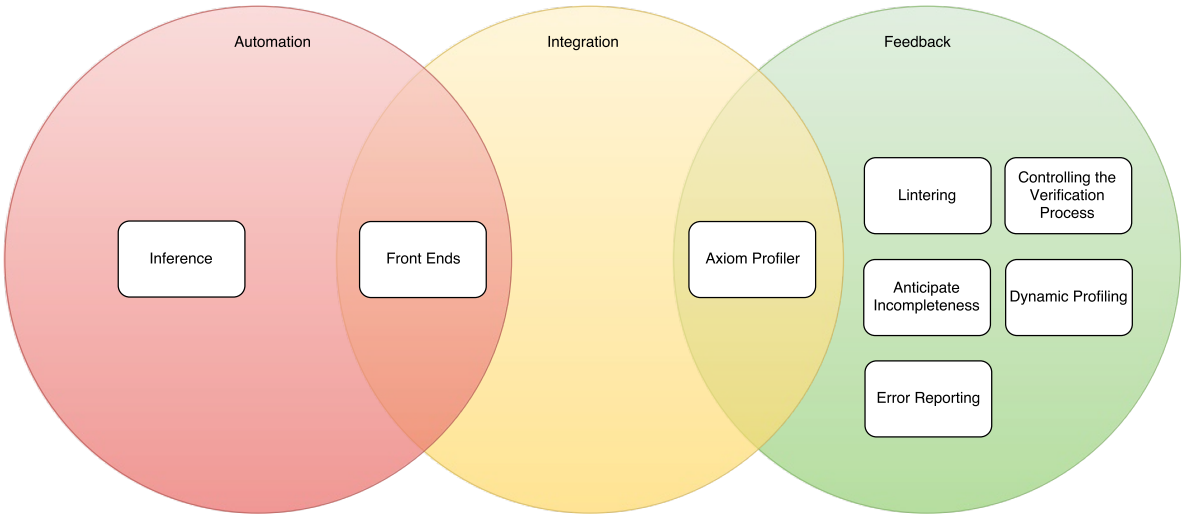

Figure 1.1. *Overview of the structure of the project.*

1.3. Previous Solution

Currently, there exists a lightweight IDE solution for Viper based on the text editor Sublime Text 3. Section 2.3 introduces the Viper Sublime IDE. It can be used to edit Viper intermediate language source code and provides basic support such as syntax highlighting. Moreover, it allows verifying programs using either the VCG backend or the SE backend from the IDE. As a result of the verification, the IDE displays the time spent and either a success or an error message directly within the editor. In case of a verification error, possible assertion violations are marked in the source code.

1.4. Viper

The Viper IDE implemented in this project is specifically designed to target the Viper toolchain, although the general principles are applicable more generally. The verification capabilities of the IDE are based on the verification backends of the Viper infrastructure.

Figure 1.2 gives an overview of the tools that make up the Viper framework. The Viper toolchain includes front ends for the languages Chalice, Python, Java, and OpenCL that allow an automated translation to the Viper intermediate language. A specification inference tool enhances a Viper by completing missing parts of the specification.

Viper features two verification backends for verifying programs: a verification condition generation (VCG) [Heule et al. 2013b] backend and a backend that proofs the correctness of programs using symbolic execution (SE) [Heule et al. 2013a] [Jacobs et al. 2010]. Section 2.1.2 introduces the VCG backend of Viper. In Section 2.1.3 we discuss the SE backend in more details. Both backends are based on the statement modulo theories (SMT) solver Z3 [Moura and Bjørner 2008].

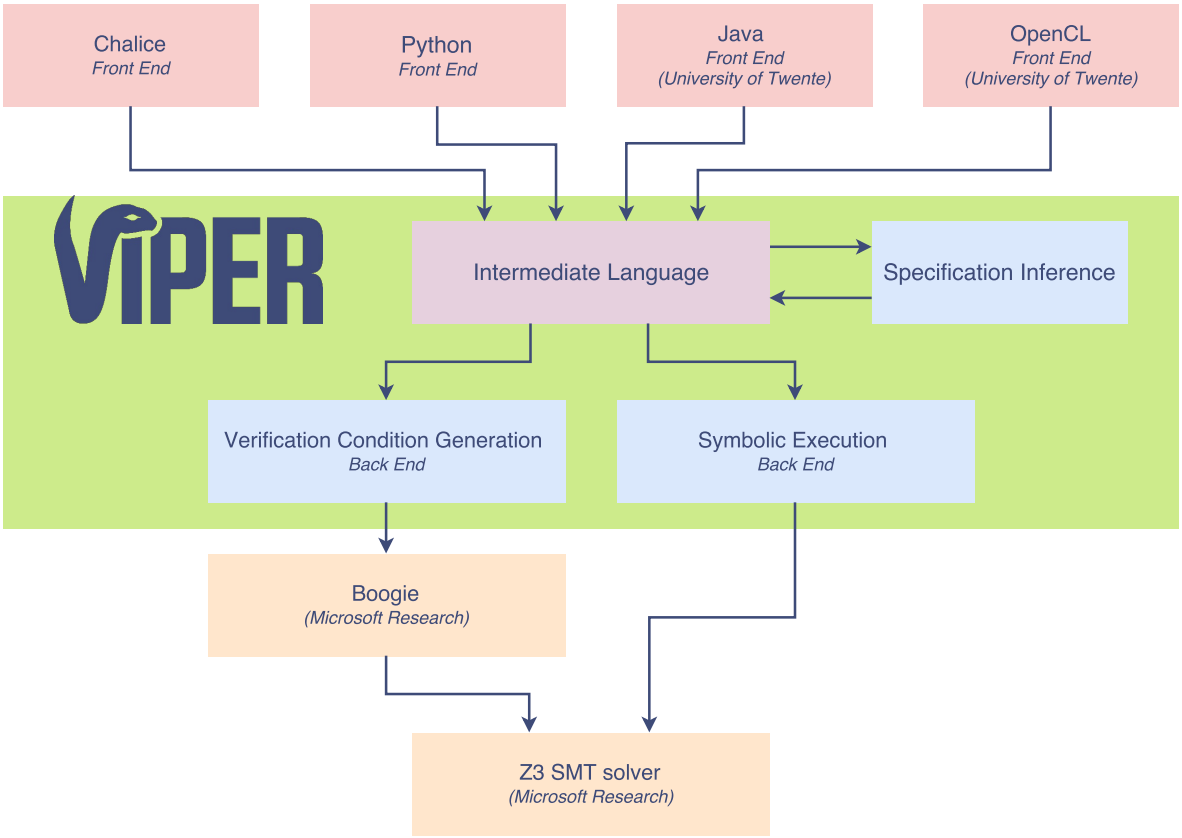

Figure 1.2. *Diagram of the individual tools involved in the Viper toolchain.*

1.5. Abbreviations

In this section, we list all abbreviations used throughout the document. Additionally, we introduce each abbreviation when first used. Table 1.1 simplifies the lookup of abbreviations.

Abbreviation	Definition
API	Application Programming Interface
AST	Abstract Syntax Tree
DOM	Document Object Model
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IVL	Intermediate Verification Language
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
OS	Operating System
SE	Symbolic Execution
SMT	Satisfiability Modulo Theories
VCG	Verification Condition Generation
VS Code	Visual Studio Code

Table 1.1. *Abbreviations used in this document.*

1.6. Structure of this Document

In this section, we present a short overview of the structure of this document. Chapter 1 introduces the project by explaining the problem statement, touching on previous work in the field, and presenting the Viper framework, for which the IDE implemented in this project is designed. In Chapter 2, we present the state-of-the-art of verification technologies, in particular, in relation to verification IDEs. Then, we cover the major design choices, such as the selection of the text editor this project builds upon. An introduction to the goals for this project is given in Chapter 3.

In Chapter 4 we design the Viper IDE starting from the visual aspects. The chapter is divided into two parts: the first part covers basic GUI features and is concerned with the visualization of basic information such as error messages. The second part elaborates on the user interaction during an advanced debugging session. Chapter 5 discusses the implementation details. It both

serves as a documentation (as it contains a detailed description of the functions provided by the IDE) and describes the foundation for assisting a potential follow-up project. The latter is possible due to the fact that it elaborates the ideas behind the structure of the code, points out potential limitations, and mentions places for extending the IDE.

In Chapter 6 we evaluate the project by comparing it to the state-of-the-art of verification IDEs, and mention known limitations. Finally, in Chapter 7 we draw conclusions and point out interesting potential continuations of the project.

2

State-of-the-Art

Automatic program verification is an active research area. Many new verification tools are being implemented and papers written about automatic software verification. There is a wide range of languages dedicated to verification. Dafny [Leino and Wüstholz 2014] and Boogie [Leino 2008] are two prominent examples. Even though the variety of verification tools for these languages is manifold, the research community agrees on some of the technologies which are used in most of these tools. For example, almost all automatic verification tools use and an SMT solver internally. In Section 2.1, we describe the established verification technologies.

In order to conveniently use the numerous developed verification tools, IDEs have been proposed which tie the individual components into a unified environment. Section 2.2 introduces the most advanced verification IDEs and names their key features. Section 2.3 discusses the Viper Sublime IDE, the lightweight predecessor of the Viper IDE. In Section 2.4, we compare different advanced text editors and determine the best contender for this project.

2.1. Verification Technologies

Software verification requires a formal representation of a program. Verification tools often integrate several technologies for solving formalized programs. There are several methods that were proposed during the last decades. Here, we briefly introduce the most common technologies and highlight their importance to the verification process. Their flexibility and generality make them widely applicable.

2.1.1. Satisfiability Modulo Theories Solver

Given a formula in first-order logic, a satisfiability modulo theories (SMT) solver is determines whether the formula is satisfiable or not and provides ana assignment for each variable, satisfying the formula. State-of-the-art SMT solvers can discharge even large formulas quickly. Therefore, these solvers are the basis of many modern verification systems, allowing them to automatically assess the proofs formulated during the verification process. Unfortunately, even advanced SMT solvers such as Z3 [Moura and Bjørner 2008] are not complete, i.e., there are valid formulae in first-order logic that Z3 fails to prove. This discrepancy explains one source of spurious errors. Fortunately, Z3 manages to correctly determine the satisfiability of most formulas.

2.1.2. Verification Condition Generation

A verification system using verification condition generation (VCG) first formalizes the semantics of the analyzed program in a higher order logic. In multiple steps, this logic representation is subsequently translated to simpler logics such as the separation logic. When the semantics are broken down to first-order logic, an SMT solver can evaluate it, as described in the previous section.

2.1.3. Symbolic Execution

In SE, all possible states inside the program under verification are obtained. This can be done by having a deduction rule for each statement, transforming the pre-state, a representation of all possible states before the execution of the current statement, into a post-state. This approach leads to a state description at all points in the program. If the desired properties can be proven for all possible states, we know that the program cannot violate the given property at the current location. All proofs involved are encoded in first order logic and discharged using an SMT solver, see Section 2.1.1.

2.2. Integrated Development Environments for Verification

Verifying programs includes writing imperative code as well as coming up with the right specifications. Ideally, the specification enriches the program with enough information to allow an automated verification, while also defining the properties to be proven. There are many IDEs supporting the development of imperative programs. However, an IDE perfectly suitable for writing imperative code does not provide the right assistance for writing specifications. Therefore, it is necessary to devise IDEs especially designed for software verification.

2.2.1. Design Concepts for Debugging Symbolic Execution

Colombo et al. envisioned a debugging tool for symbolic exectution [Colombo et al. 2012]. They presented helpful ideas on visualizing individual states within the verification. The visualizations aim at conveying the information in an easily accessible and orderly fashion without cluttering the view.

In the current project, we justify the design decisions regarding the visual representations of the debugging features by a comparison to the envisioned design concepts [Colombo et al. 2012]. Section 6.2 discusses this comparison in detail.

2.2.2. Dafny Integrated Development Environment

Most existing work is focused on including debugging functionality into a Verification Condition Generation (VCG)-based toolchain [Leino and Wüstholz 2014] and [Leino 2008]. However, although VCG and SE are fundamentally different ideas, we can still draw inspiration from those projects, because ultimately both approaches are based on an SMT solver.

We provide similar functionality for the Viper toolchain as Dafny does. Therefore, some of the visual aspects of this project were inspired by the Dafny debugger. Section 6.3 contains a detailed comparison of the visual features implemented in this project that are also part of the Dafny IDE.

2.3. Previous Solution

The predecessor of the current project is based on the Sublime Text 3 editor, and thus, it is called Viper Sublime Text plugin. This lightweight plugin can be used for editing Viper source code files. It provides the basic functionality for assisting the user in verifying programs. The Viper Sublime Text plugin is capable of highlighting the syntax and performing simple code completion. The user can start a verification of the opened Viper file and upon verification completion, the resulting errors are marked in the source code. Any output from the verification backend is forwarded to the user. Verification using either the VCG backend or the SE backend are supported. The Viper Sublime plugin runs on Windows, Mac, and Linux.

2.4. Text Editor Comparison

We allow the user to write, verify, and debug the program, all in one tool. Moreover, the planned debugging support for SE is most useful when integrated into an IDE as this allows to augment the source code with debugging-related information. Many of the simple features are common and well supported by existing software. Therefore, we refrained from implementing an IDE from scratch.

We incorporated the verification functionality as an extension into the VS Code text editor. The

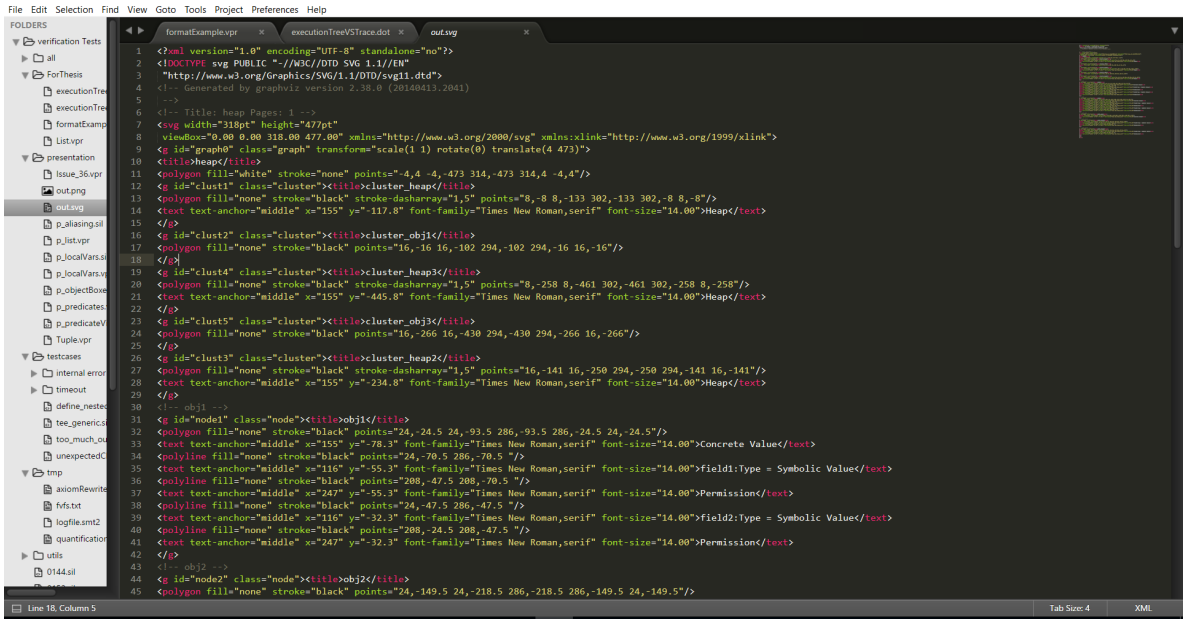

Figure 2.1. *The Sublime Text 3 editor.*

resulting IDE supports the basic functionality as the Viper Sublime Text IDE. Additionally, the implemented system provides debugging support for SE and other new features such as progress reporting.

When deciding to use VS Code as a basis for the Viper IDE we had to choose from a multitude of contenders. There are many capable text editors available that provide excellent support for programming, while still being lightweight compared to traditional IDEs. The Viper toolchain is supporting the three largest operating system, Windows, OS X, and Linux. Therefore, also the Viper IDE is designed to be cross-platform. Adding the cross-platform requirement and choosing the most extensible text editors, leaves the three candidates Sublime Text 3, Atom, and Visual Studio Code.

2.4.1. Sublime Text

The Sublime Text editor [sub 2007] was an important milestone in the development of modern text editors. When published in 2007 it was the first lightweight, cross-platform, extensible text editor for developers. Moreover, Sublime Text allows the integration of a compiler, and since version 2, it is running on Windows, OS X and Linux. Since the third version, Sublime Text 3 is based on Python 3. Sublime Text has a large user base and is a powerful text editor. One of the advantages of using a modern text editor such as Sublime is the command palette. This useful input mechanism allows the user to quickly open files and execute other commands using a command line like input mechanism which can be conveniently opened any time using a shortcut. However, the proprietary licensing and closed sources make Sublime less attractive as a foundation for an IDE. Figure 2.1 shows a screenshot of the Sublime Text 3 editor.

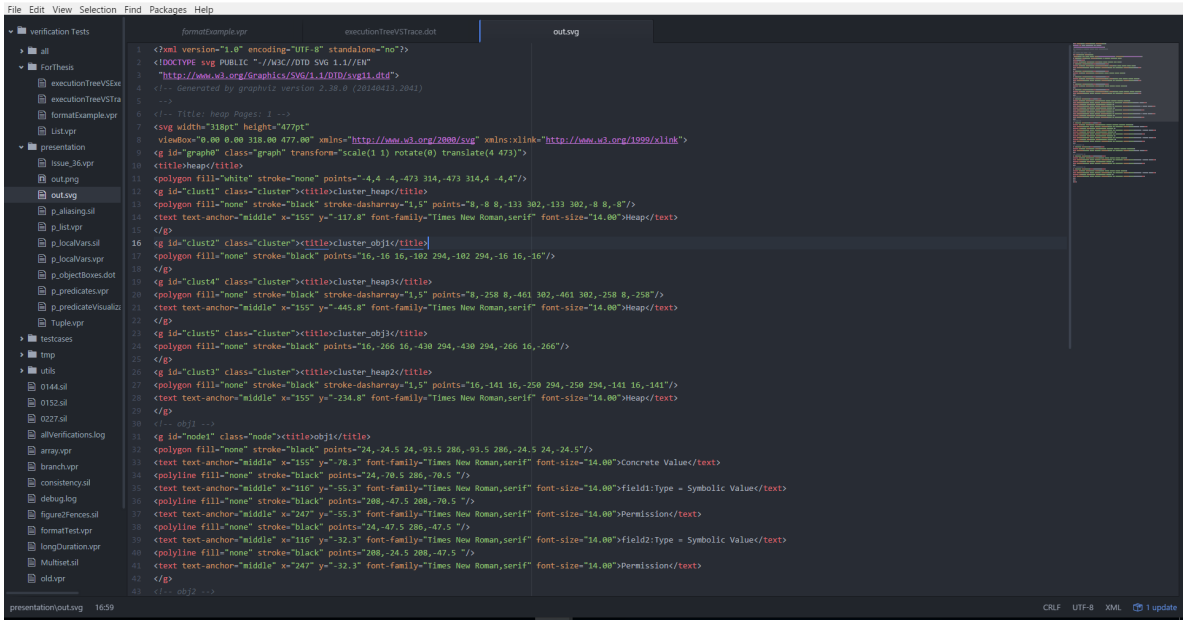

Figure 2.2. *The Atom text editor.*

2.4.2. Atom

Atom is a free and open-source text editor, was first released in February 2014, and inspired by Sublime Text. This editor was created by GitHub, and is being developed and maintained by the open-source community. It is based on Electron, a framework for building cross-platform desktop applications using web-technologies. As Atom was developed in CoffeeScript, extensions can be written in CoffeeScript likewise. The user can directly access the DOM when developing an extension, making Atom very flexible. However, this comes at the price of reduced performance. The main web page of Atom contains a more detailed description of the text editor and provides downloads for the most common platforms [GitHub 2014]. In Figure 2.2, we show the visual appearance of the Atom text editor.

2.4.3. Visual Studio Code

Visual Studio Code was created in the beginning of 2015 and is an open-source text editor developed by Microsoft. The more recent release date enabled Microsoft to learn from the experiences of Atom and fix their most important flaws in the design phase. Much like Atom it is based on the Electron framework. But Microsoft laid the focus on performance rather than flexibility. Therefore, all extensions are obliged to communicate to VS Code through the VS Code extensibility API, whereas the direct access to the DOM is forbidden. On the other hand, VS Codes performance is astonishing. Due to the highly multi-threaded design of VS Code even larger actions, such as searching a regular expression in each file in a directory, complete quickly. VS Code has a large community of open-source developers contributing to the project and is moderated by Microsoft, resulting in a thorough documentation, a consistent software, and fast progress. More details about VS Code can be found on their web page [Microsoft 2015]. Fig-

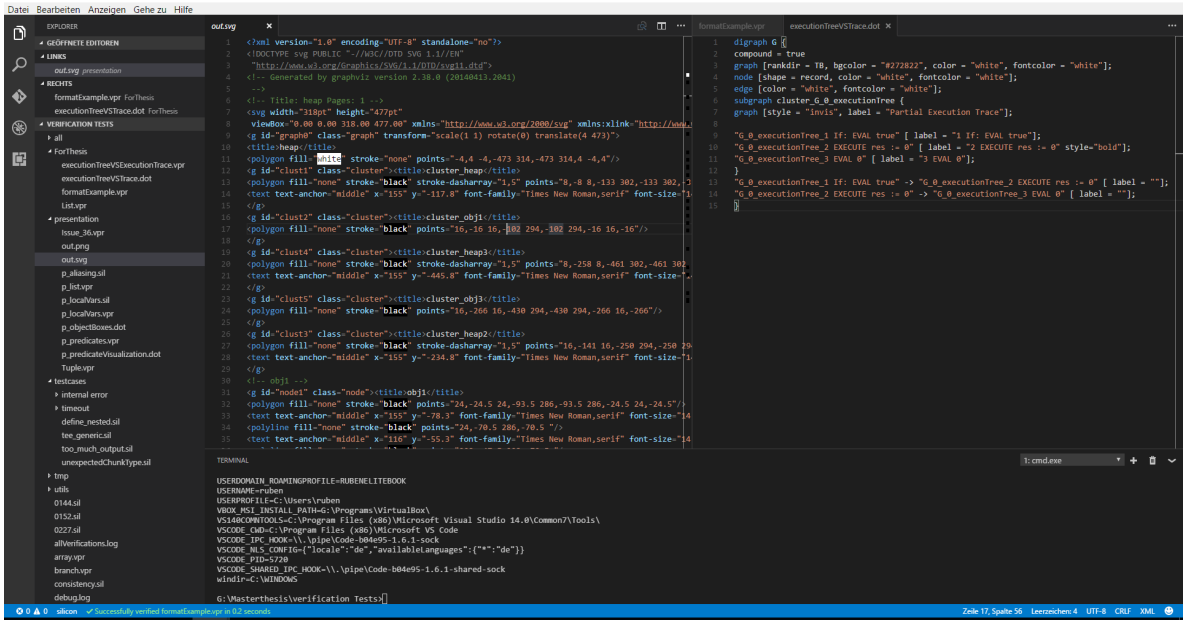

Figure 2.3. *The VS Code text editor.*

ure 2.3 presents a screenshot of the VS Code text editor.

Figure 2.4 shows the extensions installation infrastructure of VS Code. The user browses and installs extensions using the marketplace GUI. Internally, the VS Code queries the marketplace, which provides a centralized repository of all available extensions, to show the user the available extensions. The user selects an extension for installation, VS Code downloads and locally installs the extension. VS Code activates installed extensions if needed and communicates with them using the extensibility protocol.

We found Microsoft's Visual Studio Code (VS Code) to be the text editor best suited for our purposes [Microsoft 2015]. This extensible open-source project shows better performance than its contestant Atom [GitHub 2014]. Moreover, the structure of VS Code allows it to be enhanced either by implementing an extension using VS Code's extensibility API or modifying the API itself. Most of the graphical features needed for this project are supported by VS Code, and the possibility of extending the API allows for further customization.

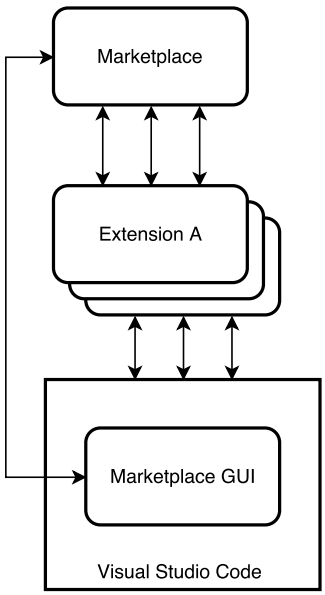

Figure 2.4. *Extension installation infrastructure of VS Code.*

3

Requirement Analysis and Overview of Planned Features

In this chapter, we introduce all features relevant to a verification IDE. They are divided into four verification phases. Figure 3.1 shows a typical interaction scenario with the IDE, including all four phases and possible transitions between them.

In the pre-verification, phase 1, the programmer is assisted in editing the source code and its specifications. Phase 2 deals with information reporting. It takes place when the user requires semantic information. For example, at the beginning of the information reporting phase, inferred properties such as triggers and ghost operations are shown. Phase 3 is concerned with control and feedback of a running verification. Progress reporting is a feature of this phase. After a failed verification, the IDE reports the detected errors and assists the process of finding and fixing the cause of the problem. For example, the state of a counterexample, i.e., an execution violating the specification, is presented to, and can be explored by the user.

This project is expected to tackle a subset of the mentioned tasks. In Section 6.1, we assess the importance of each task for this project and cluster them into core tasks and extensions. Later, we compare the planned tasks to the achievements. We came up with this classification of tasks prior to the project when planning its scope. We use the unchanged original plan as a reference for the comparison with the achieved implementation.

3.1. Pre-Verification Phase

Before the verification is started, the programmer is concerned with editing the code. Therefore, a good IDE provides support for editing, such as code completion and dynamic syntax checking.

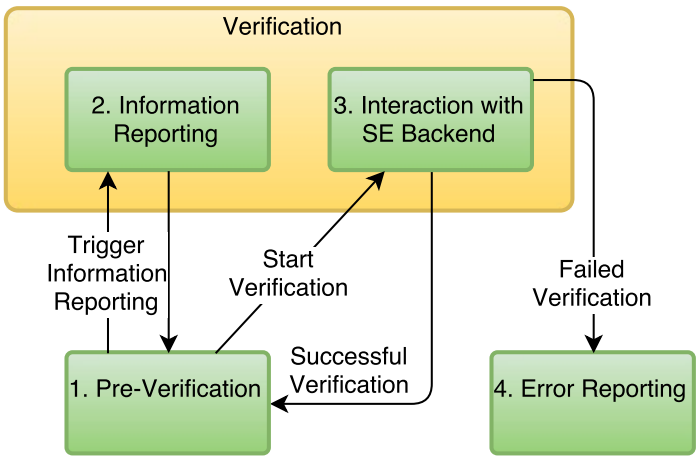

Figure 3.1. *Four verification phases and possible transitions between them.*

To help the programmer in annotating a program with contracts, some of the specifications can be generated using inference techniques. Concretely, we consider the following features:

- **Code completion and syntax highlighting.** The predecessor of the proposed system already provided basic assistance, such as syntax highlighting and code completion. We reimplement this functionality for the Viper IDE. The existing verification support is using a different parser than the Viper compiler. Ideally, we could use the same incremental parser for the Viper IDE as in the Viper framework. The implemented syntax highlighting and code completion are described in Sections 4.1.1, 4.1.2, respectively.
- **Dynamic syntax checking.** Parsing errors can be shown directly in the code by underlining the erroneous parts of the program. A tooltip conveys more information, explaining the problem or suggesting a possible fix. Section 4.3.1 explains how the Viper IDE reports errors.
- **Specification inference.** A separate tool for specification inference already exeists in the Viper toolchain [Ferrara and Müller 2012]. Including this inference into the Viper IDE would improve the programming experience.

3.2. Pre-Verification Information Reporting

Information reporting takes place after phase 1 of the verification as it depends on the output of the verification pre-processing. Since it is computationally intensive to verify a program, information reporting can save time. The displayed inferred triggers can be checked by the programmer. Any detected incompleteness issues can be reported without the need to run the verifier. Here, we list the features relevant for information reporting phase:

- **Incompleteness anticipation.** Due to the undecidability of the underlying proof problems, all SMT solvers are incomplete. Anticipating this incompleteness and informing the programmer about detected issues in the form of warnings facilitates the verification.

While linear arithmetic is supported well, most SMT solvers sometimes struggle to resolve even simple non-linear formulas. When anticipating such an incompleteness, we could warn a user formulating non-linear equations.

- **Trigger inference.** In order to verify programs, the SMT solver needs to be provided with triggers for quantifiers [Leino and Pit-Claudel 2016]. Coming up with the right triggers is difficult [Heule et al. 2013a]. However, triggers are always inferred (unless explicitly specified), and displaying them allows programmers to check if they are appropriate. Showing the inferred triggers in the IDE might improve usability, although the programmer still needs to manually assess each of them.
- **Inference of ghost operations.** Ghost operations facilitate the verification. These modifications on the written source code are generated automatically in the verification preprocessing phase and are thus invisible to programmers. In order for them to learn about the ghost operations, we should make them visible in a tooltip upon request. For instance, the statement `x :=new(*)` is expanded differently depending on the context as illustrated below:

```
field f: Int
field g: Int
...
var x: Ref
x := new(*)  //expands to
//inhale acc(x.f) && acc(x.g)
```

3.3. Interaction with the SE Verification Backend

During the verification, some gathered information could be shown to the programmer. For example, the progress can be shown via a progress bar. We list the considered features below.

- **Progress reporting.** Currently the Viper system is only informing the programmer about the outcome of the verification once it has finished. Therefore, the programmer cannot decide whether a slow verification only takes longer than expected or whether it will not terminate at all, e.g., due to a matching loop [Heule et al. 2013a]. Adding a visual progress bar to the IDE displaying the progress of the verification could alleviate this problem. The SMT solver Z3 provides feedback about running verifications that can be used to implement such functionality [Moura and Bjørner 2008]. In Section 4.3.2 we discuss the implemented progress reporting in detail.
- **Controlling the verification process.** The programmer can start the verification using either the VCG or the SE backend. Moreover, it is possible to abort a running verification from within the IDE. Upon completion, the verification displays a success message or mark the detected problems in the source code. Section 4.4 introduces the commands that allow the user to control the verification in the Viper IDE.

3.4. Error Reporting

After a failed verification attempt, the IDE helps to pinpoint the source of the problem. The error reporting system shows any information relevant for the unsuccessful verification. For example, the user is able to debug the verification by inspecting the verification states one by one. Here, we list the most important features of the error reporting phase:

- **Counterexample visualization.** The debugging system imports the counterexamples provided by the SMT solver (Z3) [Moura and Bjørner 2008]. The concrete values of all variables relevant to the detected error are shown to the user. The information is visually displayed in a dedicated frame. The frame shows similar imformation as the one developed for Dafny [Leino and Wüstholz 2014]. Section 4.2.4 discusses how the counterexample model is used in the Viper IDE.
- **Stepwise debugging.** For an unsuccessful verification it is interesting to step through the code. For each step, the associated verification actions can be shown in a separate frame and executed one by one. It should be possible to step forward as well as backwards through the code. In Section 4.2 we discuss the implemented debugging mechanisms and elaborate on the controls allowing the user to step in an active debugging session.
- **Triggering traces.** In order to use a quantifier, the SMT solver needs to instantiate its body with concrete terms that match the quantifier's trigger. Displaying the triggers for which the quantifiers have been instantiated during verification can be helpful. E.g., determining the reasons for non-terminating or slow verification is facilitated by knowing the instantiation graph.
- **Verification path visualization.** During a debugging session the path taken to reach the current state is visually displayed in the source code. E.g., in an if-then-else statement, the branch that was taken can be marked, and the one that was skipped can be grayed out as described in [Colombo et al. 2012]. The code with the visualization makes it clear which execution path has been taken. For example, in XCode, the execution paths are visually shown in the code when analyzing for memory leaks. Figure 3.2 demonstrates the execution path visualization in XCode [Apple 2011]. The arrows mark the control flow directly in the source code. Unfortunately, such a visual feature is not available for VS Code through the native API. It can only be implemented by adding it to the extensibility API of VS Code. During an active debugging session, the Viper IDE shows the execution trace leading to the current state. Section 4.2.2 explains how the execution trace is displayed.
- **Heap visualization.** Verifications of imperative programs reason about the state of the heap. Therefore, it may be helpful for debugging to have a visualization of the heap at certain points in the symbolic execution. The heap can be represented as a graph where the nodes are objects and the links are pointers. For example, such visualization for heaps has been envisioned in [Colombo et al. 2012]. The implemented heap visualization is discussed in Section 4.2.4.

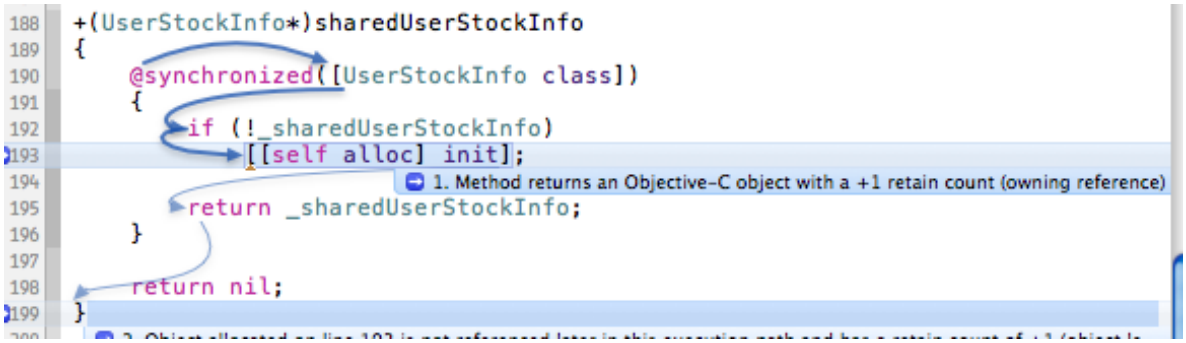

Figure 3.2. *Visualization of the control flow leading to a memory leak in XCode.*

3.5. Supported Operating Systems

Microsoft Windows, Mac OS X, and GNU/Linux are the three most widely used operating systems for desktop and laptop computers. A survey conducted in 2016 on StackOverflow suggests that 52% of developers use Windows, 26% work on a Mac and 22% use a Linux kernel-based operating system [stackoverflow 2016]. More than 45 thousand programmers from all over the world participated in this survey. To ensure the usability of the Viper IDE, it is important for the IDE to support Windows, Mac and Linux.

4

Graphical User Interface

For an IDE, the visual appearance is crucial. The graphical user interface (GUI) exposed to the user, informs about the state of the IDE, and reads the users input. The Viper IDE is based on VS Code, an extensible text editor. Section 2.4.3 introduces VS Code and explains why it is a good basis for the Viper IDE. We implemented the IDE as an extension of VS Code which adds the verification-related functionality and extends the GUI.

In this chapter, we introduce the graphical aspects of the project. The provided features are divided into basic GUI features, covered in Section 4.1, and features for debugging the verification. The basic features include tasks that are provided by traditional IDEs as well as the most basic verification-related tasks. In Section 4.2 we elaborate on the visual aspects of the verification debugging. We use the VS Code status bar for a number of different tasks. Section 4.3 covers those features in detail. Finally, in Section 4.4, we discuss how the user can control the IDE using commands and introduce the predefined shortcuts.

4.1. Basic GUI Features

In this section, we elaborate on the basic GUI features provided by the Viper IDE. On one hand, these include features that are commonly provided by traditional IDEs. On the other hand, the most basic verification-related features such as verification invocation are considered simple features.

Section 4.1.1 introduces syntax highlighting and explains how it integrates into the VS Code framework. In Section 4.1.2 we talk about the code completion support provided to the user. Section 4.1.3 covers both the auto-save mechanism and the automatic verification invocation.

4.1.1. Syntax Highlighting

Syntax highlighting is a basic functionality for enhancing the programming experience by appropriately coloring the program code. This technique makes the code more readable and helps in writing syntactically correct software. Most modern text editors provide syntax highlighting. Its support in VS Code works in two stages. First, the code is tokenized and categorized, and then the tokens are colored according to their category. VS Code allows an extension to include custom syntax definitions, which are used for tokenization and categorization, enabling custom coloring themes. The syntax highlighting implemented in the Viper IDE is visible in all screenshots showing the source code. For example, Figure 4.2 shows highlighted Viper code.

The syntax provided by the Viper IDE is based on the syntax used in the previous IDE solution. The performed modifications fitted the syntax to VS Code requirements and reflected changes in the definition of the Viper intermediate language. The syntax definition allows the tokenization to be done in a context-specific manner. This is necessary because the same combination of characters can have an entirely different meaning depending on the surrounding context. The Viper theme provides the colors associated with each of the token categories, resulting in a context-specific highlighting. The user can choose between a dark and a light theme for the Viper intermediate language, as we include both in the IDE. In case the user prefers an alternative theme, a variety of themes are available in the VS Code marketplace [Microsoft 2015]. In Section 2.4.3, we provide more details about VS Code and its marketplace.

Output Coloring

The Output panel provides useful information about the state of the IDE to the user. Similarly to the way it works with syntax highlighting of the source code, coloring helps to emphasize and structure the important information and makes the log more readable. To achieve the coloring for the output panel, we defined a syntax and a color theme specifically for this purpose. When opening the log file, we provide the same coloring as in the output panel. VS Code is able to detect the content of the output panel as well as the log file to be of the same language, based on IDE's output-language specification and the *.log* file extension of the log file. Figure 4.1 shows a screenshot of the colored output messages.

4.1.2. Code Completion

Assistance in the form of code completion is a common practice and is provided out-of-the-box by most modern text editors. VS Code also supports code completion, not only considering the token already used in the file but also using the language definition as a dictionary. Therefore, we can achieve a good code completion by simply providing the language syntax definition. However, this only includes the automatic completion of single tokens.

Additionally, to support more complex structural autocompletions, we provide a set of snippets allowing for entire structures (such as functions or methods) to be autocompleted. For example, when starting to type `method` the IDE suggests to expand the method snippet, creating a complete method signature including pre-, postcondition, and an empty method body.

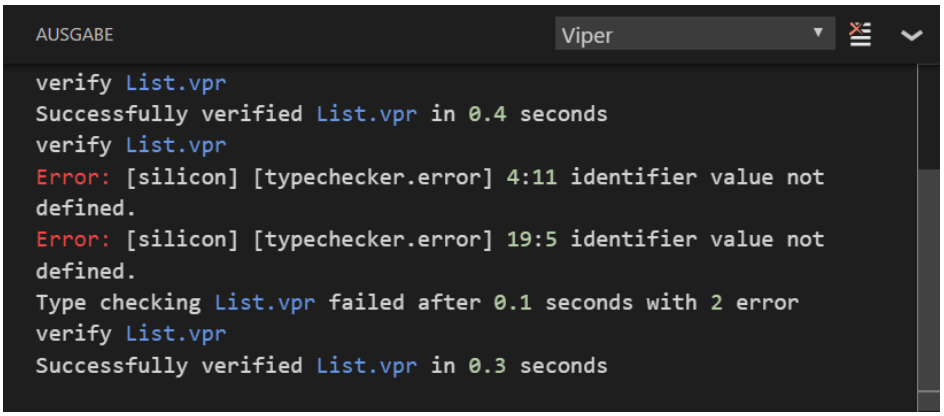

Figure 4.1. *Colored output panel.*

Moreover, snippets can provide aliases, assisting the programmer in choosing the right synonym, e.g., when starting to type `precondition`, the IDE immediately suggests the keyword `requires`. As the different verification tools all use their own terminology, such assistance can be useful. Below, we list the included snippets for the most commonly used structures as well as some aliases. If the user requires more snippets they can add their custom ones, by locally creating a *snippets* directory inside the workspace containing a *viper.json* file defining the custom snippets.

- **method**
- **predicate**
- **function**
- **domain**
- **axiom**
- **if**
- **while**
- **acc**
- **precondition:** alias for requires
- **postcondition:** alias for ensures

4.1.3. Automatic Verification Invocation

When editing a Viper file in the IDE, the start verification command (`verify`) is used commonly. Usually, the user performs a small modification on the source code and then reverifies the program to check whether the change fixed the problem. When the verification action has to be triggered manually each time, it is not used as often as would be beneficial. Therefore, we added the support for automatic verification invocation to the IDE relieving the user from having to manually start verifications. There are multiple events that trigger an automatic verification

invocation. For example, when the user opens a file, the verification is invoked. The IDE only ever shows the user up-to-date verification results. Consequently, modifying the opened file results in reverification. As restarting the verification at every keystroke might be too fine-grained, we only reverify once the file is saved. Finally, when the user changes the backend, the opened file is reverified as soon as the new backend is ready. We allow the automatic verification to be disabled using the `toggleAutoVerify` command.

Additionally, the automatic verification invocation serves as a dynamic syntax check. The verification backend first parses and type checks the analyzed file. Only after confirming its syntactic correctness, the backend starts the actual verification. Otherwise, the syntax errors are reported and the IDE presents them to the user as discussed in Section 4.3.1.

In addition to the automatic verification invocation, the Viper IDE provides an automatic saving mechanism, which saves the modified Viper file every second. The auto-save mechanism provided by VS Code affects all opened files, and saves files once the focus is changed to another file. We implemented our own auto-save mechanism, as we only target Viper files and need the save to happen even if the focus remains on the same file. By setting the configuration entry `viperSettings.preferences.autoSave` to `false` the automatic saving of Viper files can be disabled. In Section 5.2.4 we give an overview over the internal states of the Viper IDE and explain the control flow.

4.2. Features for Debugging Symbolic Execution Faliures

The error messages generated by the verification backend often do not provide enough information to understand and fix the problems. Expert knowledge and experience in software verification help to pinpoint the problem, but software engineers might get stuck. The verification result conveys useful information about the verifiability of the analyzed program. It points out the source code location where the verification failed, together with a description of the failure. However, these messages often only reveal the symptoms, rather than pointing out the cause of the problem. For example, the message `assertion x might fail` merely states that the assertion checking was unsuccessful instead of stating the underlying reason.

In case the user struggles to pinpoint the problem, we assist them by providing access to the states that lead to the problem. The simplified debugging mode is tailored for such a scenario. It focuses on one verification error at a time and provides all relevant information available, e.g., the execution trace leading to the error. Each state on the execution trace is easily accessible. The user can pick a state which is then visualized and drawn next to the focused error state. The simple debugging mode allows the user to get a quick grasp of why the verification failed and offers the possibility to trace the source of the problem. See Section 4.2.2 for more details on the simplified debugging mode.

The developers of verification tools might require more information about the state of the verification than users of the tools. To obtain this data, a verification tool developer has to consult lengthy execution traces and debug the verification tool. For example, the SE backend of Viper can output a trace of the entire verification containing descriptions of all states. Even for small

examples this trace is rather large, together with it not showing the execution structure and its formatting, it is cumbersome to handle manually. The SE backend is inherently recursive, which makes it difficult to debug. When looking at stack traces, we can see that the recursion depth is usually several hundred levels deep.

To provide assistance for this usecase of diagnosing the verification tools, we created the advanced debugging mode. Section 4.2.3 discusses the advanced mode, which is targeting developers of verification tools. By giving access to all states during the verification and showing more of the internal details about the state than the simplified mode, programmers can compare their expectations of the states to the actual states and thereby evaluate the verification tools. We offer debugging support that visualizes the states while still providing all details that previously had to be extracted out of the trace of the verification backend. Moreover, all states are easily accessible and can be explored based on the structure of the verification. This support allows developers to quickly identify wrong states.

4.2.1. Marking Symbolic Execution States

We display the states of SE using a source code annotation for each state. These annotations are called state markers and are implemented using VS Code's text decoration functionality. In Section 5.6.2 we discuss the implementation of the state markers. Even for small examples there may be many states. E.g., the *List.vpr* file implementing an *add* operation for a linked list in 28 lines of Viper code results in 99 verification states. The *List.vpr* can be found in Appendix C. Thus, showing all states is problematic, as it drastically clutters the source code. In order to resolve this issue, we prioritize the states by introducing four colors and three sizes of state markers. A state marker can be either expanded, collapsed, or hidden. Hidden states are invisible, whereas collapsed states are visible but reduced to minimal size. Expanded states contain the state number in their label. The state number is the index of the state in execution order. In order not to use unnecessarily large numbers, the state numbers start from one in each method, predicate, and function. In addition to the size of the marker, we use different colors to distinguish between the states. In Table 4.1 we describe all state marker types. Figure 4.2 shows a screenshot of a Viper program. The code has been annotated with state markers because a debugging session is currently running in the advanced mode.

Marker Type	Description
Error State/ Selected State	The *selected state* (in advance mode) or *error state* (in simplified mode) is shown in red. In Figure 4.2 state 21 is a *selected state*, (A). In addition, the debug marker, (D), highlight the line of the *selected state*.
Reference State/ Previous State	A green state number marks the *previous state* (advanced mode) or *reference state* (simplified mode). State 4, (B) in Figure 4.2 is marked as a *previous state*.
Interesting State	All states considered interesting are marked in yellow. Figure 4.2 shows an *interesting state*, (C).
Uninteresting State	A grey state number is used to show *uninteresting states*. The grey color does not distract the user from the interesting states. For example, state 22 in Figure 4.2 is an *uninteresting state*, (E).
Expanded State	When a state marker is expanded, it contains the state number in its label. In Figure 4.2, e.g., state 4 is expanded, (B).
Collapsed State	A *collapsed state* is only shown as a dot instead of as a state number. Figure 4.2 shows a *collapsed state*, (C).
Hidden State	Hidden states are not visible at all.

Table 4.1. *State marker types used for marking symbolic execution states.*

The state markers not only convey the location of the state, but also reveal the execution order. The verification of a program can visit a code location multiple times. For example, every method invocation results in a verification of the invoked methods pre- and postcondition. Each application of a predicate triggers a verification of the used predicate. Moreover, control flow structures such as `if` clauses can cause states to be verified multiple times. Therefore, we include the state numbers in the label of the state markers, because the source code location alone fails to convey the order of the shown states. Details about how the state markers are implemented can be found in Section 5.6.2.

Additionally, the user might be interested in the structure of the execution rather than only in the order. The complete execution tree contains all verification states plus some injected structural nodes required for forming a tree structure out of the execution trace. Any path from the root to a leaf of the execution tree corresponds to one chosen path through the verified program. For each branch in the program the execution tree contains two sub-trees representing either choice at the branch. In Section 4.2.4, we introduce the partial execution tree that shows the part of the execution tree around the inspected state as a graph. This tree visually shows the structure of the execution, while still fitting on the screen.

Figure 4.2. *Screenshot showing Viper source code annotated with state markers.*

4.2.2. Simplified Debugging Mode

The main purpose of the simplified mode is to pinpoint the root cause of a verification error and to help the user to come up with a solution. Therefore, in simplified mode, debugging can only be started when there is a verification error. Otherwise there are no errors to fix and thus, debugging the verified program does not serve any purpose.

Real world examples involve many states. One additional line of code usually results in about 2.5 additional verification states. A reasonably sized Viper file of about 500 lines of code is thus expected to yield about 1250 verification states. The amount of states per line of code varies strongly with the formatting and the complexity of the source code, values between 0.5 and 5 are commonly observed. Coping with this large number of states requires careful filtering. When analyzing a specific verification error, only the states belonging to the execution trace that leads to the error are important and thus, we hide all other states. Additionally, we filter out the children of the states in the execution trace because they do not help in understanding SE and rarely change the state, thus, only cluttering the view. The remaining top level states are marked as interesting states, and all other states are hidden. Only the non-selected error states are shown as collapsed interesting states, as we do not need to know the state number of the other error states. In Section 4.2.1 we introduced the state markers and explain the different kinds.

When the debugging session is started, the first error state is in focus, and the state visualization window opens up, showing details about the error state. The state visualization is discussed in detail in Section 4.2.4. By clicking on the interesting states the user can select a reference

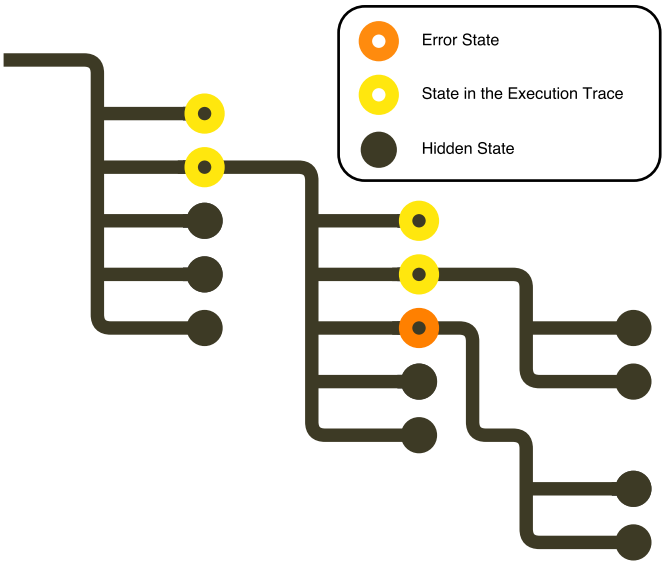

Figure 4.3. *Symbolic execution tree with states marked in the simplified debugging mode.*

state that should be visualized next to the error state. Table 4.1 lists all kinds of state markers. The state visualization is automatically updated and shows the reference state and the error state side-by-side. Clicking on another state in the execution trace leading to the error state in focus replaces the referenced state with the one selected. The reference state can be removed from the state visualization by clicking on it again, or by selecting the error state. Figure 4.3 schematically shows which states are shown in the simplified mode. Figure 4.4 shows the simplified mode for debugging SE in action.

For traversing the execution trace in order, the traditional debugging controls can be used. The semantics of the controls in simplified mode are listed in Table 4.2.

Control	**Resulting Action**
Continue	Focus on the next error state; do nothing if there only is one state.
Step and Step In	Select the next state on the execution trace as a reference state
Back and Step Out	Select the previous state on the execution trace as a reference state

Table 4.2. *Description of the debug controls in simplified mode.*

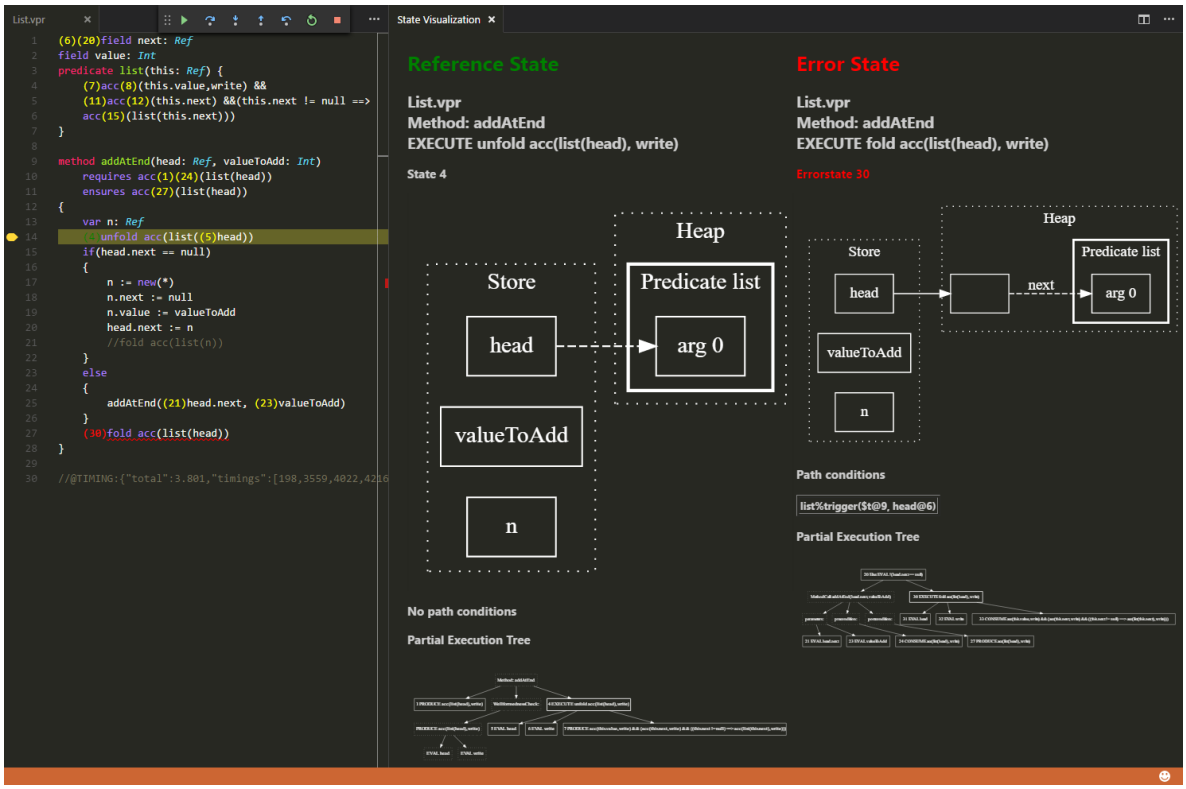

Figure 4.4. *Active debugging session in the simplified mode.*

4.2.3. Advanced Debugging Mode

The advanced mode is tailored for the use of verification tool developers. Therefore, it exposes more of the verification's internal state than the simplified mode. Figure 4.5 shows a screenshot of an active SE debugging session in the advanced Mode.

In case the analyzed program has a verification error, the debugging starts in the first error state. Otherwise, the entry point is the very first state in the program. Initially, the state visualization shows only the starting state as the selected state. When selecting another state by either clicking on its state marker or using the debug control buttons, the state visualization shows both the just selected state and the previously selected one. This mechanism allows for comparing two arbitrary states. To focus on the selected state and remove the previous state form the state visualization, the selected state has to be clicked on again. We discuss state visualization in detail in Section 4.2.4.

Unlike in simplified mode, all states are explorable in the advanced mode, not just the states on an execution trace leading to an error. To allow such an exploration, the selected states are expanded, exposing their children in the execution tree as uninteresting states. All top level states, of the method, predicate, or function the current state belongs to, are marked as interesting states. A definition of all state marker types can be found in Table 4.1. Figure 4.6 illustrates the states marked in the advanced debugging mode.

In addition to navigating using the state markers, the user can navigate via the debug controls.

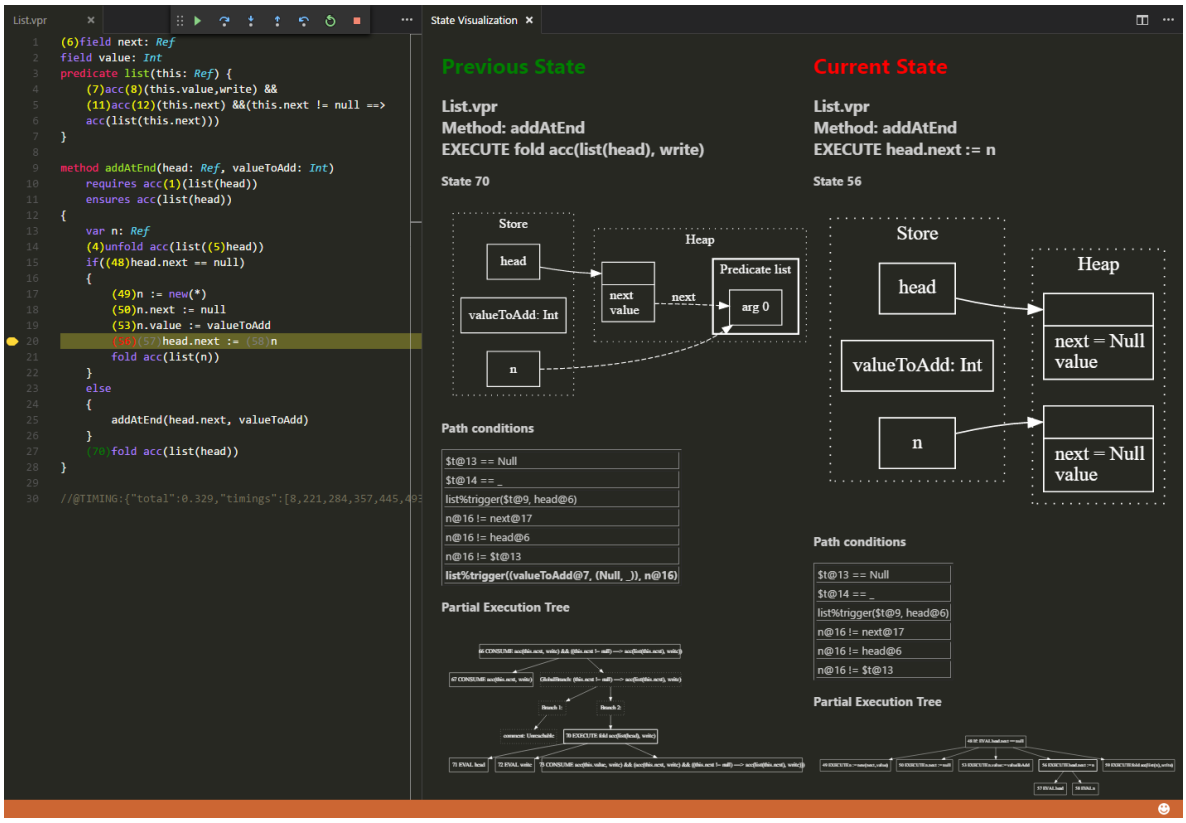

Figure 4.5. *Active debugging session in the advanced mode.*

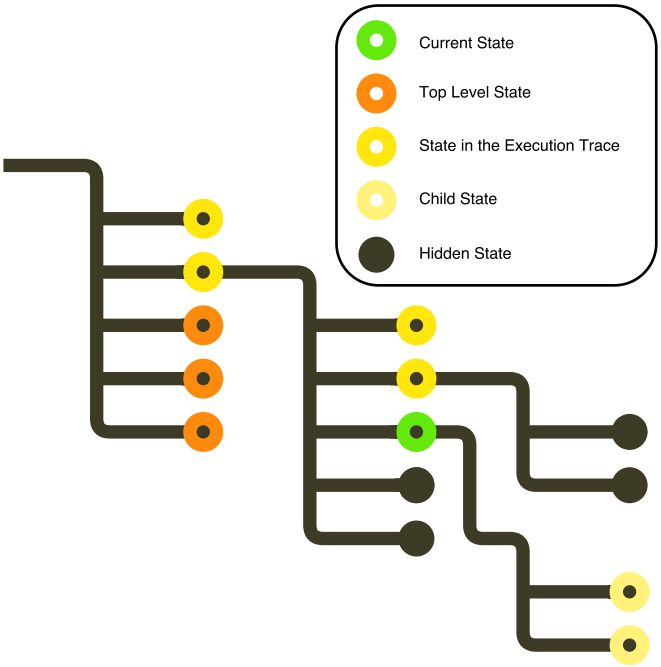

Figure 4.6. *Symbolic execution tree with states marked in the advanced debugging mode.*

The controls are similar to the controls used for debugging imperative program errors. In addition to continuing, stepping over, stepping in and out, we provide the functionality to step back. Post-mortem debugging allows stepping back to be easily implemented. Switching between methods cannot be done using the state markers. However, using the debug controls the user can step from one verifiable structure into another. We describe the semantics of each of the provided controls in Table 4.3. Figure 6.2 visually demonstrates the controls.

Control	**Resulting Action**
Continue	Go to the next error state; if there is only one state do nothing.
Step Over	Stop at the next state in the execution tree that is not a child of the current state; if there is no such state, end the debugging session.
Step In	Select the very next step according to the execution order; if there is no such state, end the debugging session.
Step Out	Stop at the next state in the execution tree that is no child or sibling of the current state, if there is no such state, end the debugging session.
Step Back	Inverse of *Step In*; select previous state in execution order, if there is no such state, end the debugging session.

Table 4.3. *Description of the debug controls in advanced mode.*

4.2.4. State Visualization

The state visualization contains useful information about a state of SE. Up to two states can be shown at the same time in the IDE. Each shown state consists of a heap visualization, a path conditions table, an old heap visualization, and a partial execution tree. The old heap and the partial execution tree can be disabled using the settings `viperSettings.advancedFeatures.showOldHeap` and `viperSettings.advancedFeatures.showPartialExecutionTree`, respectively. The visualization of the old heap works exactly like the normal heap visualization, explained in Section 4.2.4. Section 4.2.4 discusses how the partial execution tree is created and what states are included.

Heap Visualization

Heap visualization is responsible for creating a graph based on known facts about the heap. We render heap graphs using Graphviz (See Section 5.1.5 for details on external tool dependencies).

The heap visualization includes the SE store with all local variables and function invocations, and the heap, which consists of all memory chunks that we have access to in the current state. For visualizing the known values of the variables, we draw solid arrows to denote a *points to* relation, and dashed arrows to signify an *argument of* relation. Figure 4.7 shows a heap visualization.

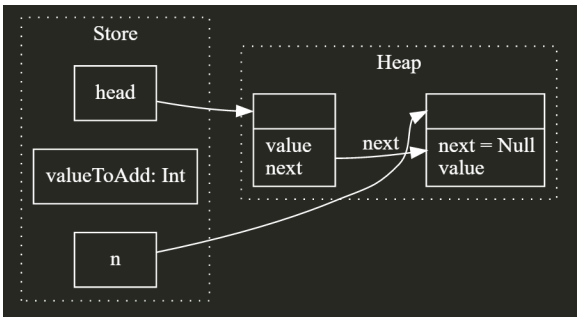

Figure 4.7. *Heap visualization of a state during the addition of an object into a linked list.*

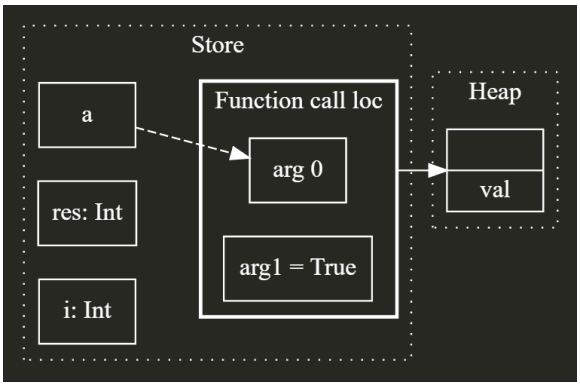

Figure 4.8. *Function application node in the heap visualization.*

Store.
All local variables are contained in the store. If the value of a variable is known, e.g., when it has an outgoing edge, we visualize its value by showing a pointer for `Ref` type variables or add the value into the label of the node for primitive types. In case the value is unknown, we enrich the variable by adding the type information into the node. In Figure 4.7, the variable head has an outgoing pointer and thus, it is of type `Ref`. We show the type of the `valueToAdd` variable, as we do not know its value, whereas `head.next.next` is known to be `null`.

A function application with known arguments can be seen as a parametrized variable. Therefore, it makes sense to add function applications to the store. We visualize function application as a node with a child node for each argument. A dashed arrow is pointing from the values used as arguments to the respective argument node. For arguments whose value was specified directly in the function parameters, we add the corresponding value directly into the argument node. A solid arrow is pointing from the function invocation node to its result. Figure 4.8 shows a heap visualization containing a function application.

Heap.
For each heap chunk to which we have access, we add a node to the heap. A heap chunk node consists of an empty header and a body containing a list of all fields we know to have access to in the current state. Any outgoing pointer of a heap chunk node is labeled with the name of the field. Incoming arrows point to the header of the node. In Figure 4.8 the heap contains a heap chunk node for which we know to have access to its `val` field.

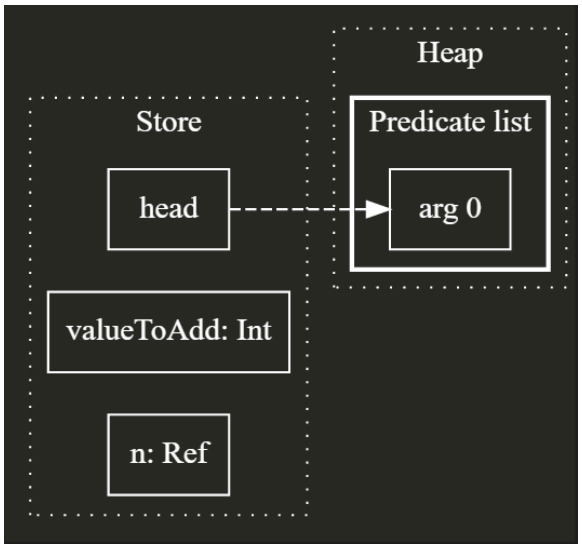

Figure 4.9. *Predicate node in the heap visualization.*

A pointer is drawn from all nodes that have a heap chunk as a value. Whenever we know that a predicate holds, we add a predicate node to the heap. Predicate nodes have a bold border and contain argument nodes, much like the function application nodes in the store. The argument nodes are pointed to with a dashed arrow by the heap chunks or variables holding the value for which the predicate holds. Similar to the function application nodes, we also add the primitive arguments directly into the argument nodes. Figure 4.9 shows a node for the predicate `list`. The semantics of this node is that the variable `head` which is an argument of the predicate is a list.

Counterexample Model

When inspecting an error state, the counterexample model generated by Z3 can be used to increase the amount of information conveyed by the heap visualization [Moura and Bjørner 2008]. The model contains concrete values for all relevant variables, thus, describing a state that could occur in a valid execution at the location of the state. In the state visualization, we can augment all variables and heap chunks with their concrete values, by adding these values to the label of the respective nodes. The counterexample resolves any potential aliasing relations, by assigning a value to all variables. In a concrete situation, two variables either are aliases or not. Knowing this information simplifies the debugging, as sometimes, problems only emerge when some unexpected aliasing constellation occurs. However, Z3 can only generate a counterexample model for error states. Therefore, we can show more detail when visualizing an error state.

Path Conditions

The path conditions are another source of information about the current state. They contain all conditions that are known to hold, based on the path chosen so far. For example, when the current state is in the `then` part of an `if` statement, we know that the if-condition must hold.

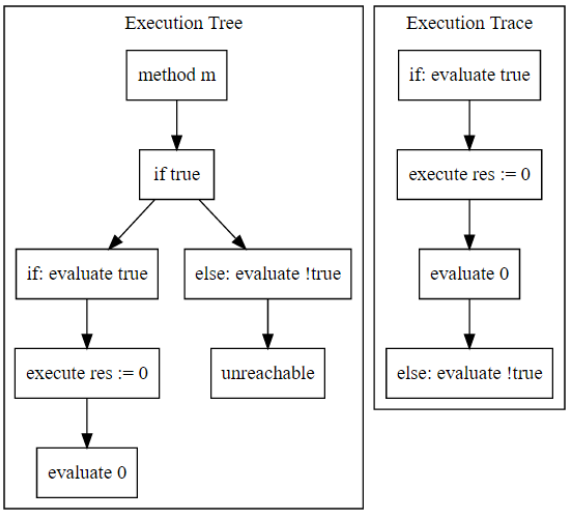

Figure 4.10. *Comparison between an execution tree and the equivalent execution trace.*

The Viper IDE uses information about the nullity of variables or heap chunks to enhance the heap visualization. A variable of heap chunk field whose value is known to be null is marked as such. Additionally, information about possible aliases could be used in visualizing the state. As we cannot include all information contained in the path conditions into the heap visualization, we present all path conditions in a table to the user. When comparing two states, the path conditions that differ from the other shown state are highlighted to assist the user in comparing the states.

Partial Execution Tree

There are two different views of the verification execution. On one hand, the execution trace is the order in which the states were produced by the verification backend. As the backends are highly optimized, this order can differ from how the program would be expected to be executed. On the other hand, the execution tree adds a hierarchy to the execution trace, defining parent and child states. In order to represent the structure of the execution, a number of structural nodes with no associated state are included in the execution tree. Method invocation nodes or if-then-else nodes are examples of such nodes. However, the execution tree shows all possible execution orders and does not capture the real execution trace. Figure 4.10 shows a comparison between an execution tree and the execution trace belonging to the same Viper program. For debugging purposes, the execution tree is useless, as only for very trivial programs, it has a manageable size. For common programs, the complete execution tree is unusable. Figure 4.11 shows the execution tree of the *List.vpr* file. We provide this Viper program containing 28 lines of code in Appendix C.

To enhance the debugging experience based on the execution ordering, we show the part of the execution tree around the selected state to visually inform the user about the execution structure. Displaying the partial execution tree instead of the complete tree results in reasonably small tree while still showing the structure of the execution. For example, this allows the user to anticipate

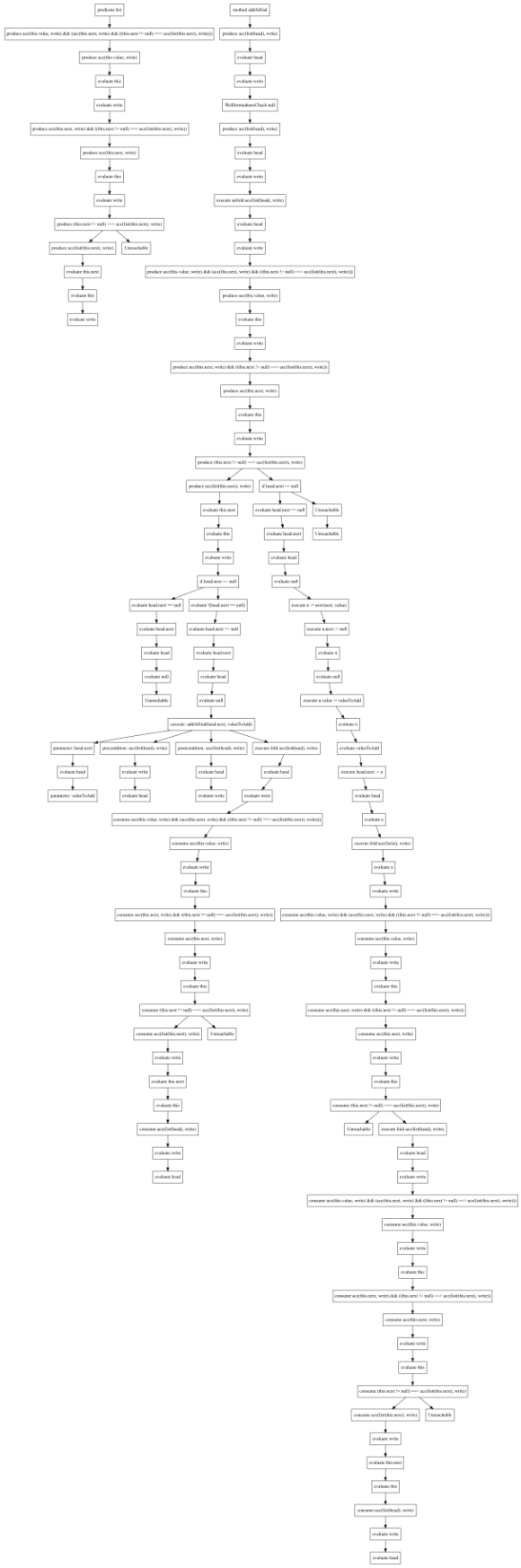

Figure 4.11. *Complete execution tree of List.vpr with 28 lines.*

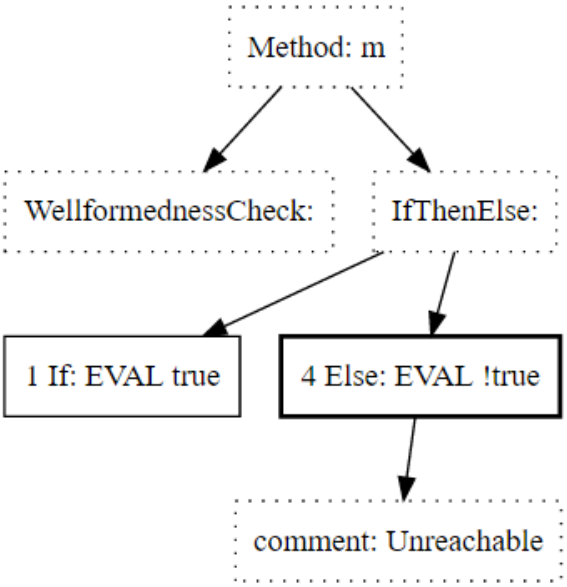

Figure 4.12. *Partial execution tree.*

branching instructions before stepping.

Figure 4.12 shows a partial execution tree for the same Viper file used for illustrating the difference between an execution tree and an execution trace in Figure 4.10 The node marked with a bold border is the current state, structural nodes are shown with a dashed border, and normal nodes, nodes with an associated execution state, are drawn with a solid border. To create the partial execution tree, we start at the parent node of the selected state. As we are looking for a normal node, we walk up the execution tree until we find a normal state or its root. The found state is the root of the partial execution tree and we add all its children, once again only stopping at normal nodes. Then, we add all children of the current node. In the execution tree, we show both the normal nodes and the structural nodes that do not have an associated state. Additionally, we add all children of the structural nodes already included in the graph, in order to get an overview of the states marked in the source code. The finished partial execution tree now only has structural nodes as leaves if these nodes do not have any children. The label of every normal state starts with the number of its associated state, corresponding to the one shown in the state markers. This analogy simplifies the lookup of the structure of the execution in the partial execution tree. Moreover, by adding the state numbers to the partial execution tree we manage to both capture all possible execution orders while still marking the real order of execution.

4.2.5. Comparing States

During an active debugging session, the state visualization can show up to two states simultaneously. This functionality can be helpful for finding the state in which some problematic configuration first occurred, as the visualization quickly conveys the structural changes in the state. The state visualization draws the compared states close to each other and aligns the heap

graphs for an optimized comparison. In the path conditions, also included in the heap visualization, we highlight the entries that differ from the compared state. This is particularly useful as manually comparing the potentially long formulas is cumbersome.

4.3. Status Bar

The status bar is the bar at the bottom of the VS Code window. It is useful to write the current status of the IDE into the status bar, as it is persistently visible. It allows to non-intrusively give additional information to the user, such as the progress of a running verification introduced in Section 4.3.2, or the result of the last verification described in Section 4.3.1.

4.3.1. Error Reporting

An essential task of the IDE is to report the verification errors. The information about whether the verification succeeded or failed is written to the status bar. The displayed message contains the verification outcome, the name of the file that has been verified, and the duration of the verification. We distinguish between the following verification outcomes.

Verification Outcome	**Description**
Success	The verification succeeded; no errors were found. In case the advanced mode is enabled, debugging is now possible.
Parsing failed	The program contains at least one parsing error.
Type checking failed	Parsing succeeded, but the type checker detected problems in the analyzed program.
Verification failed	Parsing and type checking succeeded but there are verification errors. In this case debugging can be started.
Aborted	The verification has been manually aborted.
Timeout	The verification was automatically aborted because it was taking too long.
Internal Error	Either the verification backend, the IDE, or the communication between the two failed. An internal error message contains a reason shown together with the error message.

Table 4.4. *Verification outcomes.*

When the verification has been manually triggered using the `verify` command, the user is additionally informed by an information message about the outcome of the verification. Internal

Figure 4.13. *Information message notifying about a successful verification.*

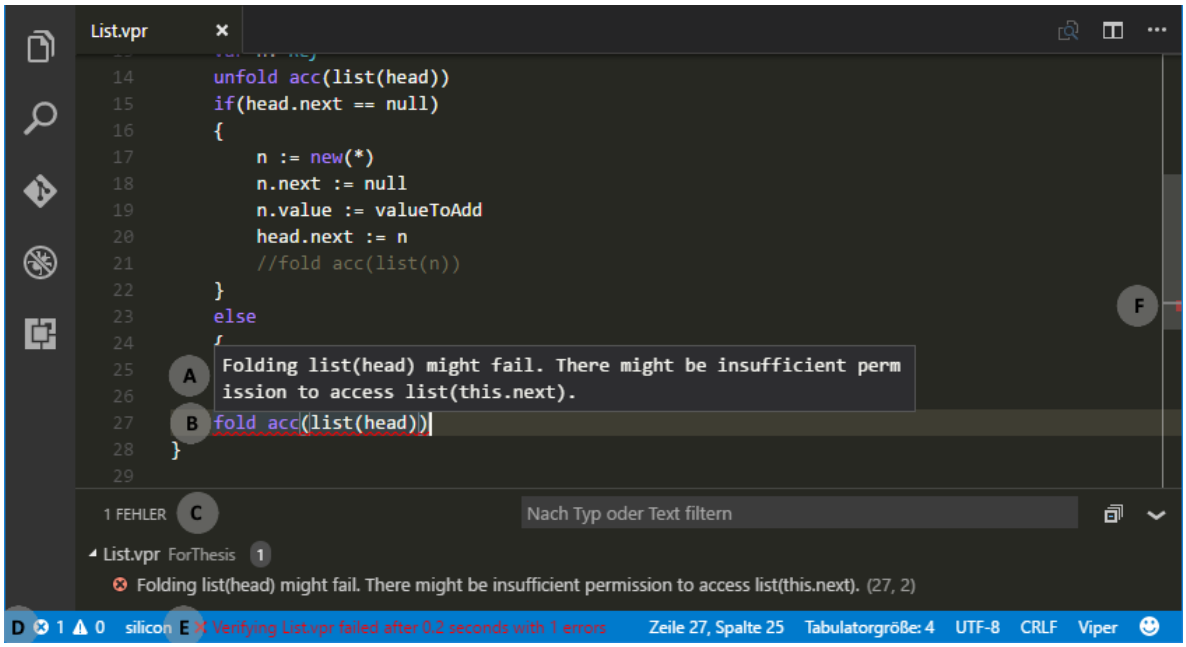

Figure 4.14. *Error reporting in the Viper IDE.*

errors are brought to the user's attention in the same way. Figure 4.13 shows an information message.

The error messages reported by the verification backend are presented to the user in multiple ways. Figure 4.14 shows the error reporting in action. Whenever an error message contains a source code location, the error is directly marked in the program using a red squiggly underline (B). The corresponding line is marked in the scroll bar (F) for an easy access to the errors in a large file. A tool tip (A) presents the errors description to the user when hovering with the mouse over the source code location of the error. VS Code provides a panel especially designed for errors (C). The reported errors also show up in this error panel. The panel can be opened by clicking on the indication about the number of errors and warnings in lower left corner (D). When working with large or multiple files, the error panel greatly improves the accessibility of the detected errors, as the error panel stores errors across multiple files and provides a hyperlink to the error location. The verification outcome is presented in the status bar (E). The Viper IDE informs the user about the error type, as we distinguish between parsing-, type checking-, and verification errors. Additionally, all detected error messages are written to the output panel. This logging allows access to old error messages if desired. Section 5.4 introduces the output panel and its purpose in the Viper IDE.

Figure 4.15. *Status bar during the verification of longDuration.vpr.*

4.3.2. Progress Reporting

Verifying programs sometimes takes a long time, or shows non-terminating behavior. We provide assistance for both cases. In the latter case, we abort the verification after a timeout. By default, we set the timeout to 100 seconds. The user can adjust this verification timeout in the settings. It is possible to specify a timeout individually for each backend. A verification that takes longer than the timeout is expected to be non-terminating and thus, yielding no conclusive result about the programs correctness, is aborted. In case of a long-running verification, we inform the user about the progress. Therefore, we add a progress indication to the status bar. We display the progress both visually, as a progress bar, and textually as a percentage. Figure 4.15 shows the progress bar in action.

For determining the current progress, we use the information about the active verification and the previous verifications of the same file. When we do not have information about earlier verifications, we have to base the computation of the progress solely on the active verification. In this case, we assume that each of the verifications has the same duration. In practice the duration of the individual verifications is varying strongly, leading to a non-linear behavior of the progress bar.

$$\texttt{progress} = \frac{\texttt{verifiedMethods + verifiedPredicates + verifiedFunctions}}{\texttt{totalMethods + totalPredicated + totalFunctions}} \tag{4.1}$$

For each completed verification, we store the total time spent, and a list of intermediate timings. A timing is taken when the verification is started in the backend, when an internal verification finishes, when the verification is done, and when the IDE is done loading the verification results. To get this information, we rely on the verification backend implementing the Viper protocol. Section 5.2.3 discusses the involved messages and their expected format. Using the timing information about earlier executions of the program allows for a more exact estimation of the progress:

$$\texttt{progress} = \frac{\texttt{timeSpent}}{\texttt{expectedTotalTime}} \tag{4.2}$$

The recorded timings effectively split the verification into a sequence of steps. For computing the `expectedTotalTime`, we distinguish two cases. We call the first case *in time*, meaning that the recorded duration of the active step is larger than the time already spent on this step in the running verification. In Figure 4.16 we visualize the progress estimation of an *in time* verification.

$$\texttt{expectedTotalTime} = \texttt{timeSpentUntilLastStep} + \texttt{timeLeft} \tag{4.3}$$

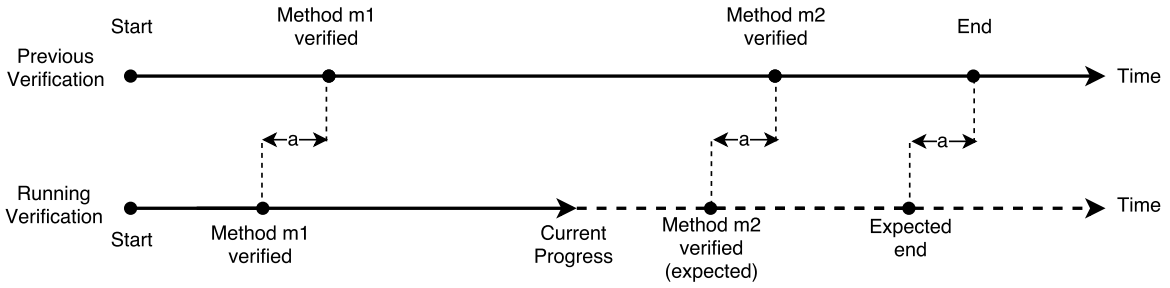

Figure 4.16. *Progress estimation when current verification is as fast as the previous one (in time).*

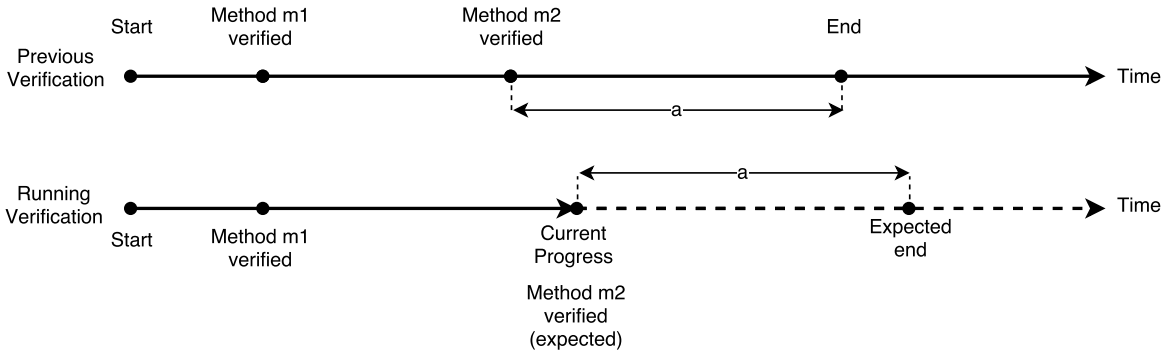

Figure 4.17. *Progress estimation when the current execution is taking longer than the previous one (behind time).*

$$\texttt{timeLeft} = \texttt{recordedTotalTime} - \texttt{recordedTimeSpentUntilLastStep} \tag{4.4}$$

In the second case, *behind time*, we already spent more time on the current step than in the recorded execution. Figure 4.17 illustrates the progress estimation in this case.

$$\texttt{expectedTotalTime} = \texttt{timeSpent} + \texttt{timeLeft} \tag{4.5}$$

$$\texttt{timeLeft} = \texttt{recordedTotalTime} - \texttt{recordedTimeSpentUntilCurrentStep} \tag{4.6}$$

We assume that the current step is about to be finished, and thus, `timeLeft` does not include the current step any more.

In case `timeSpent` is larger than `expectedTotalTime`, we set `progres = 1`. This only occurs, when the last stage takes longer than expected, in which case, we predict the verification to finish any moment. Also if there are more steps than expected, `timeSpent` can be larger than the expected time.

When using the estimated `progress` in the IDE, we only show 100% when the verification is known to be finished, otherwise, we display a `progress` of 1 as 99%. Even though the

estimated value is 100% showing 99% is preferable. Commonly, one associates 100% with an already finished process. Therefore, we only show a progress of 100% if the verification is known to have finished.

4.4. Commands and Shortcuts

The user can invoke commands using the VS Code's command palette. In order to increase the usability of the IDE, we invoke most actions automatically without the user having to worry about when they should be executed. The most important actions are still available as commands, in case a user prefers to manually trigger some action. For the most common actions, we provide a predefined keyboard shortcut. However, VS Code allows the shortcuts to be customized by the user.

Command	**Shortcut**	**Description**
`verify`	$f5$	By running the `verify` command of pressing $f5$, a verification of the opened Viper file is triggered. If the active file is not a Viper file, the system starts the verification of the last active Viper file. A running verification is only restarted, when the content of the file has changed or when the already running verification starts analyzing another file.
`selectBackend`	Windows: $ctrl + l$ Linux: $ctrl + l$ Mac: $cmd + l$	To select an alternative backend the user can press $ctrl + l$ on Windows and Linux, or $cmd + l$ of Mac to start the `selectBackend` action. In case there are more than one backend defined in the configuration, the user can choose the desired backend from a drop-down menu. Selecting the already running backend does not trigger a restart. This is not needed, as any change in the settings affecting the running backend is automatically triggering a restart.
`stopVerification`	Windows: $ctrl + h$ Linux: $ctrl + h$ Mac: $cmd + h$	A long-running verification can be stopped using the `stopVerification` command with the associated shortcut being $ctrl + h$ for Windows and Linux and $cmd + h$ for Mac. Upon receiving this command the IDE stops the verification and prepares the IDE for the next one. In case no verification is running, a warning is shown.

`startDebugging`	Windows: $ctrl + d$ Linux: $ctrl + d$ Mac: $cmd + d$	If the advanced features are enabled in the configuration and the active Viper file is verified, a debugging session can be started. Otherwise, the user is informed about the reason for the command not starting a debugging session. Section 4.2 is explaining the debugging sessions in detail. The `startDebugging` command has the shortcut $ctrl + d$ on Windows and Linux and $cmd + d$ on Mac.
`toggleAutoVerify`		In case the user would like to manually decide when to start a verification, e.g., because they want to do a large modification on the file and verification is not needed for some time, the automatic verification can be deactivated using the `toggleAutoVerify` command.
`openLogFile`		The log file is located in the default temp directory of the operating system. However, as this might not be a convenient place to go to, the `openLogFile` command provides a quick way of accessing the log file. As with the contents of the Viper output, also the log file is colored when opened in VS Code. Section 4.1.1 elaborates on this coloring.
`verifyAllFiles-InWorkspace`		It is convenient to have a mechanism for checking multiple files without having to manually open up all of them. Such functionality is provided by the `verifyAllFilesInWorkspace` command. When triggered, all Viper files in the workspace are opened and verified one by one, using the active backend. The user has the possibility of stopping the process by stopping one of the verifications, using either the `stopVerification` command or the stop button displayed during an active verification. When the workspace verification finishes, the total time used is logged together with all processed files and their respective verification result, allowing the user to observe the correctness of all their files at a glance.
`format code`	$alt + shift + f$	Having an automatic formatting mechanism can be useful. For example, when copying source code, some indentations might be lost. For a quick recovery the `format code` command can be run either through the command palette or using the shortcut: $alt + shift + f$. Section 5.8 discusses this formatting mechanism.

Table 4.5. *IDE commands together with their associated keyboard shortcuts.*

5

Implementation

In this chapter, we explain the most important aspects concerning the implementation of the Viper IDE. In Section 5.1, we discuss the infrastructure of the Viper IDE. Section 5.2 introduces the Viper protocol, defining all interfaces used for communication both within the IDE and between the IDE and external tools. In Section 5.3 we explain the implementation details of the configuration system of the Viper IDE and motivate the choices that led to it. Then, in Section 5.4, we discuss the logging system of the Viper IDE. Section 5.5 covers the changes we had to apply to the verification backends in order to enable debugging support for Viper's symbolic execution verification backend. In Section 5.6 we mention important corner cases that had to be considered during the implementation. The supported syntax highlighting is discussed in Section 5.7. Automatic formatting of Viper source code is introduced in Section 5.8. The Viper IDE manages a multitude of external tools. The implemented timing mechanisms to prevent concurrency issues, are discussed in Section 5.9. Finally, in Section 5.10, we introduce the shutdown sequence that is executed when the IDE is closed.

5.1. Infrastructure

There are different kinds of extensions for VS Code. The first one only consists of a language client and is able to provide simple lexical analysis and other computationally lightweight tasks. For example, counting the number of words in a document. For tasks such as parsing a document and marking errors in the source code, it is recommended to use a two tier architecture consisting of both a language client and language server. The server is responsible for all heavy computations, whereas the client only coordinates the communication between VS Code's graphical user interface (GUI) and the server. This separation helps preserving the responsiveness of the GUI under heavy load. For extensions featuring a debugger, VS Code pro-

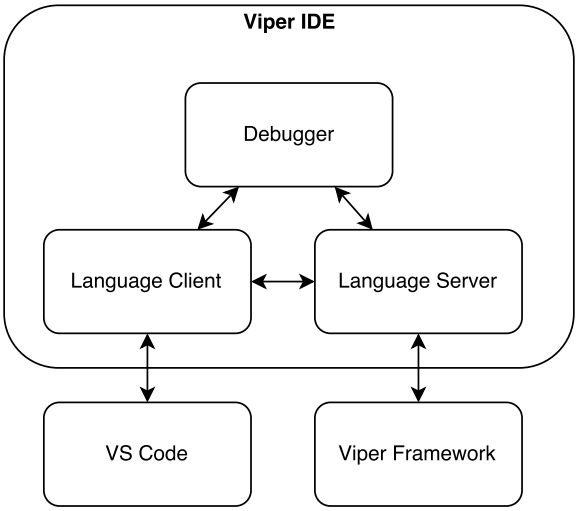

Figure 5.1. *Overview of the infrastructure involved in the IDE.*

vides an infrastructure that integrates debugging functionality into the GUI. VS Code's debug protocol thereby handles the communication between the language client and the debugger.

Figure 5.1 shows an overview of the parts of the Viper IDE. For this project, we use a three tiered system combining the debugger extension with the language client/language server variant. Therefore, the Viper IDE consists of a language client, a language server and a debugger. The client communicates with VS Code in order to use VS Code's functionality, whereas the server manages the Viper framework to allow verifications. The protocol used for the communication within the Viper IDE is called Viper protocol and discussed in Section 5.2. For a more detailed diagram see Figure A.1 in the appendix. In this section, we describe the tasks of the individual parts of the Viper IDE and introduce the external tools used by the IDE.

5.1.1. Language Client

The main job of the language client is to orchestrate the communication between the different parts of the system, to handle all user input and produce all output. Below, we list specific tasks handled by the language client.

- **Commands and shortcuts.** Commands activated by the user need to be detected, and corresponding actions need to be started.
- **Reacting to user interaction.** For example, in case the user opens or modifies a file the client needs to initiate a verification invocation.
- **Logging.** The client handles the log file, all other parts need to send messages to the client for adding a log entry.
- **Informing the user about the running processes.** The client is in charge of notifying the user about active actions using the output panel and information messages.

- **Progress reporting.** The client estimates the progress of a running verification and handles the progress bar.
- **Visualizing the state of the IDE.** The visualization of the current state of the extension needs to be shown using the VS Code extensibility API. For example, the debugging visuals are activated and deactivated by the language client.
- **Requesting data from server.** For instance, during an active debugging session, the client needs to request information about the SE states from the server.
- **Controlling the debugger.** The language client is in charge of handling the debug controls.
- **Keeping track of files.** The language client keeps track of the used files and their verification status.
- **Automatic verification invocation.** The client decides about the necessity of a verification invocation and initiates it if needed, by requesting the file verifications from the language server.
- **Automatic saving.** The auto-save mechanism is run by the client.
- **Converting graphs.** The conversion of Graphviz *.dot* graphs to their *.svg* counterparts is orchestrated by the language client.

5.1.2. Language Server

The language server is mainly responsible for handling external tools, providing information to the client, and handling all computationally intense tasks. The following list contains specific tasks controlled by the language server.

- **Controlling the Nailgun server.** The language server starts, monitors and closes the Nailgun server.
- **Starting file verification upon request.** When requested by the client, the server must perform the requested verification and deliver the verification outcome to the client.
- **Parsing backend output.** The server parses the output of the verification backend.
- **Prepares the errors for displaying.** The server transforms the errors output by the verification backend into a format suitable for displaying.
- **Providing stepping functionality.** The server contains the logic deciding where to step to in an active debugging session.
- **Validating the backend output.** The server is responsible for checking whether the backend's output corresponds to the Viper protocol.
- **Configuration validation.** The server is responsible for validating the configuration and producing error messages and warnings about the detected problems.
- **Store verification states.** The server stores all states of the last verification and makes them available for the client and the debugger.

- **Verification states.** The language server is in charge of loading and parsing the verification states from the output file of the verification backend.
- **Heap visualization.** Generating the heap visualization graph descriptions upon request is done by the server.
- **Structure of execution.** Upon completed verification, the server provides the state markers to the client.
- **Partial execution tree.** The language server creates the partial execution trees and sends them to the client.

5.1.3. Debugger

The debugger is a much smaller part than the client and the server and only holds the state of the active debugging session and handles move requests from both the client and the server. In order to determine the next state to move to, the debugger queries the language server, which holds the stepping logic.

5.1.4. External Tools

Since the main use of the IDE is to assist program verification, it is crucial to integrate the verification into the IDE and thereby, relieving the user from manually having to run the verification tools. In this section, we list the external tools supported by default and shipped together with the IDE. It is possible to set up other tools in the configuration, as long as they adhere to the Viper protocol, introduced in Section 5.2. Section 5.3.4 explains how to configure additional external tools.

Verification Backends

The Viper framework provides a SE-based verification backend [Müller et al. 2016] that is capable of proving the correctness of Viper source code programs. This backend supports fractional permissions, recursive predicates, and heap-dependent functions.

The Viper framework provides a second verification backend based on VCG, which verifies Viper source code programs using a translation to the intermediate verification language (IVL) Boogie [Leino 2008] [Le Goues et al. 2011].

Specification Inference

Viper's specification inference tool can statically analyze Viper source code. Its capabilities include inferring the required access permissions of methods, functions, and loops. The specification inference integrates seamlessly into the Viper IDE as it replaces the file lacking permissions by a version containing the inferred permissions.

5.1.5. Third-Party Tools

Graphviz

We use Graphviz for rendering all graphs shown during debugging [AT&T and Bell-Labs 2000]. For example, when debugging, we display the heap visualizations discussed in Section 4.2.4. Graphviz is capable of automatically aligning the nodes in the graph. The visual features required by the IDE are supported and exporting to *.svg* format is possible. Moreover, Graphviz is running on Windows, Mac, and Linux, thereby satisfying all requirements of the Viper IDE. As an input Graphviz expects the *.dot* format, a graph description language.

Nailgun

Nailgun is an open-source tool that repeatedly starts Java application using the same Java virtual machine (JVM) [martylamb 2004]. This is desirable because starting up the JVM takes around 3 seconds. Having this delay once during the startup of the Viper IDE is acceptable. However, if each verification is taking 3 seconds longer it would negatively affect the user experience. The Nailgun server hosts the JVM. For each verification, we start a fresh instance of the Nailgun client that connects to the Nailgun server instead of starting its own instance of the JVM.

5.2. The Viper Protocol

The Viper protocol defines interfaces used for the communication between the three tiers of the system, i.e., the language client, the language server, and the debugger. Figure A.1 presents an overview of the system design and highlights which communication channels use the Viper protocol. In this section, we give a detailed explanation of the interfaces that make up the API of the Viper protocol. The complete protocol specifications can be found in the *ViperProtocol.ts* file.

5.2.1. Client-Server Communication

In Table 5.1, we present notifications and requests sent from the language server to the client. Many of the messages are required to communicate the state of the Viper IDE among the tiers. Figure 5.3 shows an overview of the control flow and the internal states of the Viper IDE. Some messages are used for logging, while others serve a specific purpose. For example, the `HeapGraph` message transfers the heap graph description, used for visualizing the selected state during an active debugging session. More details about debugging SE can be found in Section 4.2.

Message	Explanation
`SettingsChecked`	The server notifies the client about the result of the settings check. The server must send the detected configuration errors to the client, in order for them to be shown to the user.
`StateChange`	Whenever the internal state of the active verification backend changes, the client needs to be informed in order to update the GUI accordingly. Section 5.3.3 introduces the backend, i.e., a flexible configuration describing how an external verification backend can be integrated into the Viper IDE.

`Log`
`Error`
`ToLogFile`
	Log a message or an error to the output panel or to the log file, the centralized design of the logging infrastructure demands all log messages to be sent to the client. Section 5.4 discusses the logging system in depth.

`Hint`	The server tells the client to show an information message to the user. This message is needed, because only the client can access the GUI and show information messages. Figure 4.13 shows an information message.
`BackendChange` `BackendReady`	The server informs client about an ongoing backend change by sending a `BackendChange` message, in case the configuration is modified. The client must learn about the resulting backend restart in order to reject requested verifications until the `BackendReady` message, notifying about the completed backend startup, is received
`FileOpened` `FileClosed`	The server informs client about opened or closed file. These events are detected by the server, but the client also needs to know about these events in order to determine whether an automatic verification invocation is required. Section 4.1.3 introduces the automatic verification invocation mechanism provided by the Viper IDE.
`VerificationNot-` `Started`	It is possible that the client determines all requirements for a verification to be fulfilled but the server still cannot start the verification, e.g., when the backend is restarting and the client is not informed in time. In such cases, the server notifies the client that the verification could not be started.
`StopDebugging`	The server requests the active debugging session to be stopped. More details about the life cycle of a verification debugging session can be found in Section 5.2.2.

`StepsAsDecoration-Options`	The server sends the steps loaded from the SE log in the form of decoration options to the client. The client can use this decoration options for annotating the source code with state markers. Section 4.2.1 introduces state markers and explains their use of simplifying navigation during an active debugging session.
`HeapGraph`	The server sends the graph descriptions of the heap graph, the old heap graph, and the partial execution tree to the client which assembles this information to complete the state visualization. Section 4.2.4 discusses this mechanism in detail.

Table 5.1. *Commands sent from the language server to the client.*

Table 5.2 contains the messages sent from the client to the language server.

Message	**Explanation**
`RequestBackend-Names`	When the user requests the use of another verification backend, the client needs to know about all backends that are set up in the configuration and therefore, asks the server for a list of all backend names. Section 5.3 introduces the configuration of the Viper IDE.
`Dispose`	In case VS Code is shut down, the client needs to clean up all used resources. Therefore, the client tells the server to terminate its external tools before deactivating itself. Details about the shout-down sequence can be found in Section 5.10
`Verify`	The client sends a `Verify` message to the server when requesting a verification of the opened Viper file. The server performs the verification using the running verification backend.
`StopVerification`	The client tells server to stop the running verification, e.g., when the stop was requested by the user.
`ShowHeap`	For visualizing the selected state during an active debugging session, the client requests the heap graph description of the selected state from the server.
`StopDebugging`	The client requests the active debugging session to be stopped, e.g., when the user requests the end of the debugging session using the `stopDebugging` command.
`StartBackend`	The client tells the server to start the verification backend. Most of the time the server manages to startup the backend autonomously. However, e.g, if the backend startup timed out, the client can trigger a retry by sending the `StartBackend` message.

`GetExecutionTrace`	The client requests a list of all states on the execution trace leading to the current state. The execution trace includes the states that happened prior to the current state in the execution order. More details on the execution trace and the order of execution can be found in Section 4.2.4.
`GetDotExecutable`	The client is in charge of generating the vector graphics from the graph descriptions. The server, on the other hand, loads and validates the configuration. Therefore, the path to the external tool capable of translating *.dot* graph descriptions to a *.svg* format is unknown to the client and has to be requested from the server.

Table 5.2. *Commands sent from the client to the language server.*

5.2.2. Communication of the Debugger with Client and Server

The user must be able to start and stop a debugging session for a verified file. Moreover, during an active debugging session the Viper IDE should provide an intuitive navigation within the source code. For the client-debugger communication, the VS Code debug protocol has proven to suffice, as it defines all interfaces required for the navigation in the source code. Figure A.1 introduces the three tiers of the Viper IDE and shows the used communication protocols within the Viper IDE. The user can start debugging by invoking the `startDebugging` command. VS Code provides the debug controls, allowing the user to navigate within an active debugging session using a standardized set of controls. There are six controls: *continue*, *step over*, *step in*, *step out*, and *step back*. The sixth control, *stop debugging*, allows the user to quit the running debugging session. Figure 5.2 shows a screenshot of the debug controls. VS Code notifies the debugger when the user clicks on one of the debug controls. Then, the debugger informs the language server where the user wants to step to. Alternatively, the user can navigate by selecting one of the shown state markers. In this case, the client informs the debugger indirectly about the detected user input by first notifying the language server, which then determines if a move action is necessary, and informs the debugger accordingly.

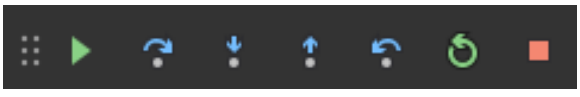

Figure 5.2. *Debug controls provided by VS Code.*

The client is responsible for starting up a debugging session when requested by the user. VS Code requires a launch configuration to be specified when starting a debugging session. This configuration contains information about the used debugging engine, the file to debug, and additional settings required by the debugging engine. We include the used launch configuration in Appendix B. When the `startDebugging` command is invoked, the client automatically determines the right configuration and launches the debugging session, in order to relieve the user from specifying such configurations. See Section 4.4 for more details on the commands pro-

vided by the IDE. After the startup, the newly initialized debugger connects to the language server, thereby informing it about the active debugging session. The language server acknowledges the connection by returning the starting position of the debugger. The starting position is the state of the first detected verification error, or the first state in case the is no error state.

There are multiple reasons why a debugging session could be ended. First, in case the user manually stops the debugging session using the debug controls or by stepping beyond the last state of the program, the debugger disconnects. The language server detects the disconnected debugger and reacts by informing the client. Second, if the user invokes the `stopDebugging` command, modifies the debugged file, or triggers the verification of another Viper file, the client detects the need to end the active debugging session. Consequently, the client informs the server about the required stop. Upon receiving the `StopDebugging` request, the server prepares for termination before ultimately asking the debugger to shut down. In both scenarios, the tier that first detects the need for termination of the debugging session informs the other two tiers.

5.2.3. Communication between Backends and the IDE

Apart from defining the communication within the IDE, the Viper protocol also specifies the structure of messages expected from a verification backend. Those messages must be output on the standard output channel. The IDE parses the messages using a JSON parser and thus expects them to be in a valid JSON format. Only the format of the message containing the counterexample model should not be in JSON format; instead it should comply with the output format of *SMT2* [Moura and Bjørner 2008]. Each JSON message must have a `type` field containing the type of the message. The IDE uses the `type` field to distinguish the messages from each other. Below, we list all allowed messages and their expected format.

- **Start Message.** The backend is expected to send a start message to acknowledge that the verification session has started. The start message must provide the type of the backend. For example, `Verification` is a typical value for the backend type. In the current implementation of the Viper IDE the backend type is not used. However, to simplify treating some backends specially in the future, we added this field to the Viper protocol.

```
{
  "type": "Start",
  "backendType": "Verification"
}
```

- **Verification Started.** Once the verification has started, a verification start message must be sent. This message informs the IDE about the end of the pre-verification phase and the start of the actual verification. It ought to contain the number of methods, functions, and predicates that will be verified. As we use this information for progress reporting, the shown progress bar might be imprecise when the counts are not correct. Section 4.3.2 contains details about the implemented progress reporting mechanisms.

```
{
  "type": "VerificationStart",
  "nofMethods": 1,
```

```
  "nofFunctions": 2,
  "nofPredicates": 0
}
```

- **Verified Message.** Upon completed verification of a predicate, a method, or a function, an appropriate message informing about this event must be output by the verification backend in order to comply with the Viper protocol. The verified message has to include the name of the verified structure. There are three types of verified messages, one for each of the verifiable structures: method, function, and predicate.

```
{
  "type": "MethodVerified",
  "name": "exampleMethod"
}
{
  "type": "FunctionVerified",
  "name": "exampleFunction"
}
{
  "type": "PredicateVerified",
  "name": "examplePrediate"
}
```

- **End Message.** When the backend completes the verification, the IDE expects a corresponding message. The end message should contain the total duration of the verification in seconds. We use the following regular expression for parsing the timing.

```
^.*?(\d+)([\.,](\d+))?.*$
```

  Integers as well as decimals are permitted. To support different localities, we allow the separation character of the decimal to be either a `.` or a `,`. In case another separation character is used, we only miss the fractional part, but still retain the integer part of the duration. For example, the strings `"4.101"`, `"the verification took 4,101 seconds"`, or `"4.101s"` are valid time strings.

```
{
  "type": "End",
  "time": "4.101 seconds"
}
```

- **Error Message.** The backend ought to package all detected errors and send them in one message. The error message comprises the name of the verified file, e.g., *verified-file.vpr*, and a list of all individual detected errors. A single error should provide both a start and an end location in the format of `line:column`. In case the location is unknown the value `start` or `end` can be specified as `<unknown line>:<unknown column>`. Additionally, each error must include an error tag, providing a hierarchical categorization of the reason for the error. The tag `typechecker.error`, for example, specifies that the problem was detected during type checking and prevented the verification from continuing. Finally, each error message is expected to include a verbose and human readable

description of the error. The IDE uses this message both as a tooltip for the errors marked in the source code and for logging the errors to the output. Moreover, the Viper IDE adds the errors to VS Code's internal error list in order to enable an easy lookup. Section 4.3.1 explains the usage of this error message, e.g., in the internal error list, for annotating the source code with the errors, and for providing tooltips.

```
{
  "type": "Error",
  "file": "verified-file.vpr",
  "errors":[
    {
      "start": "27:2",
      "end": "27:22",
      "tag": "example.error.tag",
      "message": "A message explaining the error."
    }
  ]
}
```

Handling of invalid Messages

We developed the Viper IDE using TypeScript, an extended version of JavaScript. TypeScript was designed as a web programming language and is mainly used for that purpose. As websites often consist of many different individual parts, even parts that are unknown to the programmer of the website such as advertisements, it is crucial for ensuring a working system that a runtime exception does not crash the system. Therefore, TypeScript silently ignores errors and the rest of the system carries on without the user noticing.

However, in VS Code TypeScript is used to develop a desktop application. For desktop applications it is easier to ensure correctness, and thus, it is a common practice to let them fail on runtime exceptions. We improved on the default behavior of TypeScript and catch and report runtime exceptions instead of silently ignoring them. As our system is a desktop application that supports potentially unknown external tools, we are on a middle ground. For example, the Viper IDE should be able to handle crashing tools, or tools that do not perfectly implement the Viper protocol.

Before using the information contained in the messages, we first check all messages received from the verification backend for compliance with the Viper protocol. The IDE rejects messages that do not satisfy the specified requirements, and reports an error. To still provide the best possible support to the user, we do not stop the system in case an invalid message is received. For example, if a verification backend is sending invalid *verified* messages, we still provide the progress bar, even if the estimated progress will be less accurate.

5.2.4. Control Flow of the Viper IDE

Figure 5.3 shows the control flow inside the Viper IDE. The green nodes in the diagram are states where the IDE is waiting for user input. For example, after a verification, the IDE is in such a state. The yellow nodes are decision nodes. A decision involves internal computation and its decision alters the chosen control flow path. However, as the computation involved in decision nodes does not require external tools, it is quickly reaching a conclusion. The blue nodes correspond to active states in which the IDE is performing some computation involving external tools. An active node can only be left when the computation is finished or if a user interferes with the action. In the diagram, we model the interference as arrows pointing from boxes instead of nodes. This denotes that no matter where inside the box the execution currently is, it stops and the control flow follows the interruption arrow.

Initially the system is in the stopped state at the top of the diagram. When the user opens a Viper file, the IDE starts up and immediately checks the configuration. In case the configuration is not valid, it shows the detected problems to the user and we return to the stopped state. In case the user opens another file, presses $f5$, requests a change in the active backend, or modifies the configuration, we repeat the configuration validation. When there is a valid configuration, we start the backend resulting in a running system.

After the start of the backend, we automatically start a verification on the opened file. However, before starting the actual verification, we first have to determine whether all necessary condition for the verification are met. For example,we check the type of the opened file, as it only makes sense to verify Viper files. In case the verification is interrupted by a timeout or by the user requesting a verification stop, we stop the running verification and wait for user action. After a completed verification, the IDE presents the results to the user and waits for user action.

The user can continue verifying by modifying the opened file or opening a new Viper file, or request a debugging session using the `startDebugging` command. When all requirements are met, a debugging session is started for the last verified file. For instance, the debugging requirements include the availability of execution states. When the debugging session started the state visualization window opens up, showing a visualization of the initial state. In Section 4.2.4 we introduce the state visualization and Section 4.2 discusses debugging in general. Whenever the user changes the current state, either using the debug controls or by selecting the state markers, the state visualization window is updated to include the newly selected state. An active debugging session can be ended using the debugging controls or by stepping out of the verification states. The IDE then returns to waiting for further user action. Moreover, any action causing a verification stops an active debugging session. For example, modifying the opened file, changing the focus to another opened Viper file, or pressing $f5$, are actions causing a verification.

No matter in what state of the control flow graph the Viper IDE currently is, a backend change request or a modification of the configuration stops all running actions and returns to checking the validity of the configuration. Similarly, in case the user shuts down VS Code, the Viper IDE terminates all external processes and cleans up the state.

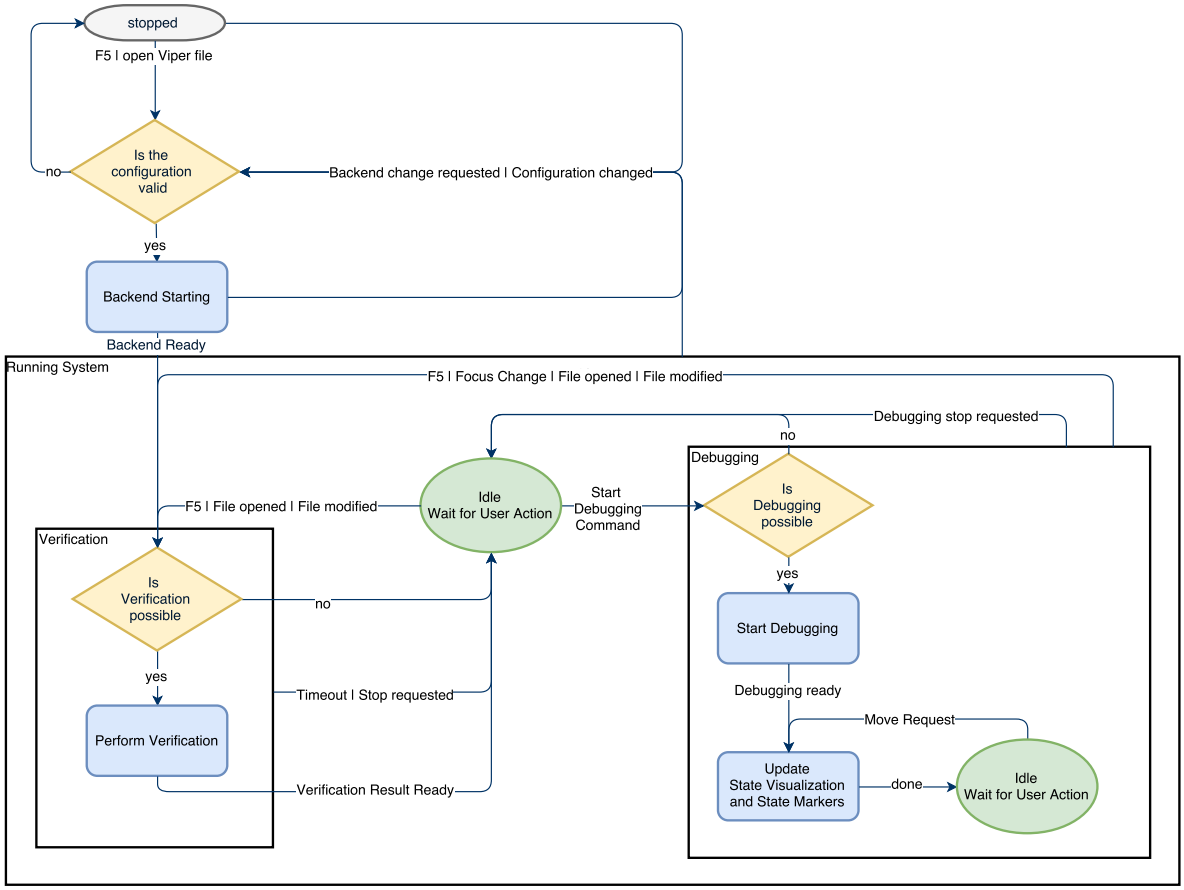

Figure 5.3. *Control flow of the Viper IDE visualized as a state diagram.*

5.3. Configuration

The Viper IDE is designed to work out-of-the-box on Windows, Mac OS X, and GNU/Linux. The default configuration is supposed to work for standard scenarios. For example, a user has just completed the installation of the Viper IDE, by following the instructions in Section 5.6.3. When starting VS Code and opening a folder, the Viper IDE is still inactive and only once a Viper file is opened the IDE is activated. The backend is started and the opened file is verified automatically. The user then sees the result of the verification and changes a line in the file. The IDE saves the file and invokes an automatic reverification. Then, the user requests a debugging session, as the file still contains a verification error. The IDE starts the debugging session in the simplified mode and initially visualizes the error state. The user then stops the debugging session and closes VS Code.

If a user prefers to refrain from using the default settings, they can configure the behavior of the Viper IDE by modifying the configuration. In VS Code there are user settings and workspace settings. The user settings affect all instances of VS Code run by the active user, whereas the workspace settings only hold for the opened folder. Therefore, the workspace settings are stored in a *.vscode* subfolder located in the opened folder. The user settings file is located in the user files directory. The exact locations depend on the operating system and are listed below:

- **Windows:** *%APPDATA%\Code\User\settings.json*
- **Mac:** *$HOME/Library/Application Support/Code/User/settings.json*
- **Linux:** *$HOME/.config/Code/User/settings.json*

When retrieving an entry from the settings, VS Code first checks if the entry is specified in the workspace settings, followed by a lookup in the user settings. In case the entry is still not found, its value is taken from the default settings. The complete default settings of the Viper IDE can be found in Appendix E.

5.3.1. Configuration Validation

Since the settings allow the user to extensively configure the behavior of the IDE, it is important to provide a settings validation mechanism to assist the user in specifying sound settings.

Settings Check	**Description**
Existence check	The IDE checks that all required settings entries have a value.
Version check	Each top level Viper setting JSON object contains a version that is checked against the required version. All default settings always have the version of the installed extension. The required version corresponds to the version in which backwards-incompatible settings changes have been made. In case the version is older than the required version for a Viper setting, this part of the settings has to be renewed by replacing it with the newest version from the default settings.

File existence check	For each path in the settings, the IDE checks whether the file referenced by the resolved path exists. Section 5.3.2 explains path resolution in detail.
Backend Validation	The configuration needs to specify least one backend, each backend is required to have a unique name, and needs to have at least one verification stage. Verification stages describe the actions executed during the verification. Each stage must contain a `mainMethod` and `customArguments`. In the `customArguments` the `--ideModeAdvanced` can only be set if also the `--ideMode` is set. The stages can be arbitrarily ordered and executed depending on the outcome of the previous stage. This order can be specified, by configuring a follow-up action in the stages description. For example, we can use a stage performing a verification as the initial stage. In case the verification fails, an inference tool should be run followed by another run of the verification, to determine whether the inference managed to fix the program. In the backend validation, we check the follow-up actions. For each specified follow-up action there has to be a stage with the denoted name. However, as the follow-up actions are not mandatory, it is allowed to omit them. Since debugging SE only works when the `customArguments` contain `--ideModeAdvanced`, a warning is issued if the `customArguments` do not correspond to the advanced features settings, i.e., the part of the configuration that contains all debugging-related settings. Moreover, each stage is required to specify at least one path to a Java executable, either directly referring to a *.jar* file or to a folder containing any number of *.jar* files. In the latter case, all of them are included to the Java environment.
Java settings check	The Java settings must contain a `customArguments` property. We check the version of the installed Java environment for compatibility with the IDE. However, this check happens only when the backend is started.

Table 5.3. *Implemented settings checks.*

After the IDE checks its configuration, the user is presented with a list of all checks that failed. Both errors and warnings are shown to the user. In case only one check failed, the message is included into the displayed information message, otherwise only the count is shown in the pop-up, and all messages are written to the output. See Section 5.4 for details on the logging mechanisms and the output. The settings are considered valid if no settings errors are detected.

5.3.2. Path Resolution

For all settings specifying the location of a file the user can describe the file location in several ways. We present the accepted path kinds below. When the settings are checked, each path is resolved to an absolute path, which is subsequently used by the IDE.

Resolution Action	Description
Absolute path	All absolute paths in Windows, OS X, and Linux notation are permitted.
Relative path	Relative paths are completed by prepending the currently opened workspace folder to complete the relative path.
Operating system-dependent path	In addition to specifying paths as a single string, they can also be chosen depending on the operating system (OS). As the path usage among the supported platforms differs, we need to allow the paths to be specified for each OS individually. Path resolution only regards the path specified for the currently active operating system. The format of an OS-dependent path is expected to be of type `{"windows":string, "mac":string, "linux":string}`. Section 5.6.4 elucidates the necessity of OS-dependent paths to the Viper IDE.
Environment variable	All paths are allowed to contain environment variable values. For example, the `Viper tools directory` is referenced by an environment variable in the default configuration.
Viper tools directory	The paths can be specified relative to the `Viper tools directory` by including `$viperTools$` as a prefix. By default, all paths are specified relative to the Viper tools directory, allowing the user to easily setup the system, by placing Viper tools into the expected location, or, alternatively, adjusting the Viper tools location in the settings.
Canonization	After resolving custom mechanisms, such as environment variables or OS-dependent paths, the path is normalized and formatted to suit the needs of the active operating system. Due to this normalization, paths like */folder/file.jar* are valid for all supported operating systems, (even though on Windows, paths are expected to use backslash symbols instead of the normal slashes). Moreover, paths with multiple trailing slashes and indirections are correctly simplified. For example, canonization transforms the path */folder///subfolder/../file.jar* into */folder/file.jar*.
Inference of the file ending	On Windows the *.exe* extension can be omitted for all executables. If there is no file without the extension, *.exe* is appended, and the existence is checked again. This mechanism is only active on Windows.

Table 5.4. *Mechanisms used for resolving paths.*

5.3.3. Verification Backend Settings

All information necessary for invoking the verification backend is configurable in the backend settings.

```
{
  "v": "0.3.5",
  "name": "silicon",
  "paths": [
    "$viperTools$/backends/silicon.jar"
  ],
  "useNailgun": true,
  "timeout": 100000,
  "stages": [...]
}
```

The above backend description is one of the backends included in the default configuration. The `v` field is used for the version check discussed in Section 5.3.1. This mechanism allows the IDE to detect old settings and warn the user accordingly. To uniquely identify the backend, we use its `name`. The paths linking all Java executables used by the backend are included in the `paths`. The provided path resolution mechanism, introduced in Section 5.3.2, can be used to specify a path. In case the path is pointing to a directory instead of a single file, all *.jar* file inside this folder are added to the paths. The field `useNailgun` allows a usage of the Viper IDE without Nailgun. Nailgun is introduces in Section 5.1.5. However, setting `useNailgun` is not recommended, because without Nailgun the Viper IDE has to start a fresh instance of the JVM for each verification, resulting in slower verification. The `timeout` field can be used to adjust the time after which a verification is aborted because it is expected to be non-terminating. The `timeout` is specified in milliseconds. A verification backend consists of one or more `stages`. A stage describes the action taken when it is executed. Moreover, it can define follow-up stages to continue with depending on the outcome of the stage. In Section 5.3.4 we elaborate on the verification stages and explain their importance for extending the IDE with additional external tools.

5.3.4. Extensibility

The description of a verification stage allows for invoking any *.jar* file with customizable arguments. For example, in the default configuration, we have the following verification stage, describing the action taken upon verification:

```
{
  "name": "verify",
  "isVerification": true,
  "mainMethod": "viper.silicon.SiliconRunner",
  "customArguments": "--ideMode --ideModeAdvanced --z3Exe
     $z3Exe$ $fileToVerify$"
}
```

The Viper IDE assembles the information specified in this stage, into a command:

```
"/path/to/ng" viper.silicon.SiliconRunner --nailgun-port 7654
   --ideMode --ideModeAdvanced --z3Exe "/path/to/z3" "/path/to/
   verified/file.vpr"
```

In order to make the `customArguments` dependent on the verification, several internal variables are allowed to be used in specifying the custom arguments. Here, we present the predefined internal variables. Any occurrence of an internal variable is replaced by its value before using the custom arguments to invoke the stage.

Internal Variable	**Description**
`$z3Exe$`	The absolute path to the Z3 executable.
`$boogieExe$`	The absolute path to the Boogie executable.
`$mainMethod$`	The main Java method of the active stage.
`$nailgunPort$`	The port used for the communication between the Nailgun client and the Nailgun server. Section 5.1.5 introduces Nailgun.
`$fileToVerify$`	The absolute path to the Viper file under verification.
`$backendPaths$`	A list of all jar dependency paths of the current backend concatenated by semicolon (;) on Windows or colon (:) on Linux and Mac.

Table 5.5. *Internal variables that can be used in the custom argument configuration.*

To integrate an external tool, the user has to extend the backend by adding a new stage. The new stage needs to have a unique name within the backend and `isVerification` can only be set to `true`, when the tool produces valid verification output. The path to the *.jar* file of the added external tool must be included into the paths of the backend. The main method of the tool ought to be specified within the newly added stage. If the tool needs command line arguments, they can be specified in the stage description using the internal variables, (see Table 5.5). If the tool should be executed before the verification, it must be placed in the beginning of the list of stages. Other stages can be linked by setting the `onSuccess` action to the name of the follow-up verification stage. E.g., for inference tools it might be favorable to run them prior to the verification. For tools that need to run after a verification or in between two verifications, the verification stage should go first and link to the stage of the external tool using one of the follow-up actions. Below, we show how the follow-up stages integrate into the stage description:

```
stages:
[{
  "name": "verify",
  ...
  "onSuccess":"",
  "onParsingError":"",
  "onTypeCheckingError":"",
  "onVerificationError":"inference"
```

```
},
{
  "name": "inference",
  ...
  "onSuccess":"verify"
}]
```

As the follow-up actions are not mandatory, the IDE performs no action, in case the follow-up action is specified as an empty string or not specified at all. To avoid a non-terminating sequence of stages, the IDE records the executed stages and runs each stage only once. An exception is made for verification stages: they are allowed to be run twice, before the verification process finishes.

To try out the extensibility of the Viper IDE, we experimented with the support for other verification frameworks. We only needed to adjust the supported file endings and managed to configure the Viper IDE such that it supported both the Dafny and the Python verification toolchain. Even though those frameworks are completely different from the Viper toolchain, our IDE managed to handle them.

5.4. Logging

It is important to have a capable logging system for an IDE, especially when multiple external tools are simultaneously running in the background. On one hand, finding the cause of problems is simplified by a logging system. On the other hand, the user can view the log to monitor the currently active tasks or for having persistent access to verification results. Table 5.6 shows the different supported log levels and describes their semantic verbosity. The log messages are written to the output panel in VS Code. The panel can be opened through the menu located at *View -> Output -> Viper*.

Code	Verbosity	Description
0	`None`	The log is disabled, no messages are written to the output.
1	`Default`	Only verification-specific output is shown.
2	`Info`	Some information about internal state is logged, the critical errors are shown.
3	`Verbose`	Fine-grained information about internal state is presented.
4	`Debug`	Detailed information about internal state, non-critical errors, and the commands used to run the verification are logged.
5	`LowLevelDebug`	Some of the raw output of external tools is logged. Printing all output of external tools to the log file would result in too many messages, lessening the usefulness of the output. However, the complete output is collected in the log file.

Table 5.6. *Available verbosity levels.*

All log messages, even those not printed to the output panel due to the active verbosity level, and all outputs of external tools are written to the log file. The syntax highlighting applied to the log file is explained in Section 4.1.1. We use a single centralized log file for collecting all log messages. The language client handles this log file and the server and the debugger send their log messages to the client instead of logging them autonomously.

In Appendix F we explain the startup process of the Viper IDE by analyzing a typical trace of output messages.

5.5. Adaptation of Tools

The symbolic execution verification backend of the Viper toolchain is designed for a push-button use. Thus, it lacks most of the output required by the IDE and provides output tailored to being read by the end user, rather than in a format easily parsable. Section 5.2.3 defines the output expected from a verification backend. Prior to this project, the SE backend was most commonly invoked using a command line interface or the previous IDE. In both cases, it did not have to create intermediate output, but only the verification outcome (together with the detected errors) after the verification had finished. As the error messages contain the location of the error in the source code, we could use this output to underline the erroneous parts of the program and output the corresponding error messages as a tool tip. Section 4.3.1 discusses the different ways in which the Viper IDE presents the detected errors to the user.

However, for all other functionality, such as the progress reporting or the debugging, we adapted the SE verification backend to provide the necessary output. To improve extensibility of the IDE, we changed the output format of all messages to a JSON format. Section 5.2.3 contains a detailed description of the backend output format expected by the IDE.

5.5.1. Output for Progress Reporting

For progress reporting, the Viper IDE requires verification backends to output the intermediate progress during the verification. As described in Section 4.3.2, we need timed information on the verification of methods, functions, and predicates, as well as their total numbers, in order to determine a precise estimate of the overall progress. To implement the required output for the SE backend, we extended it to output intermediate information during the verification process. We format the progress messages as JSON objects and send them through the standard output channel to the Viper IDE.

5.5.2. Outputting SE States

To allow the user to debug a verified file, the Viper IDE needs to learn about all states of the verification. By debugging SE, the user can comprehend the actions taken during the verification by executing them one by one and analyzing the intermediate states. We refer to states before the action as pre-states and call the states after post-states. Each action taken by the verification backend transforms the pre-state into a post-state. For the IDE it suffices to only consider the pre-states, because all but the very last post-state have an equivalent pre-state. Each state consists of a store, a heap, an old heap, and a set of path conditions that hold when the state is reached. We enrich the pre-states with information about the action following after them. In particular, the states location in the source code, the type of the handled action, either *Execute*, *Evaluate*, *Consume*, or *Produce*, and the actions expression are stored together with the pre-state.

In symbolic execution, a state S consists of a store π, a heap h, an old heap h_o, and path conditions c [Jacobs et al. 2010].

$$S = (\pi, h, h_o, c) \tag{5.1}$$

The store consists of all local variables. The heap and the old heap are sets of heap chunks that each describes a piece of the heap for which we know to have either read or write access to. The old heap describes the old state, e.g., the state at the beginning of the method, whereas the heap characterizes the current state. The path conditions are the accumulated conditions that are known to hold based on the path taken. For example, the loop guard is known to hold, if the current state is inside the loop body.

The existing state output mechanism of the SE verification backend allows to get useful information for debugging the SE backend [Schwerhoff 2016]. However, it is insufficient for the purpose of the Viper IDE, as it does not output the structure of the execution. This output can be activated by specifying required verbosity of the output. When passing `--logLevel=trace` as an argument to the verification backend, it outputs some of the states. However, not all states are output, because this functionality is only expected to be used for debugging the SE backend. Moreover, the output appears in linear order and, thus, does not convey any information about the tree-like structure of the execution. All that can be learned from the execution trace is the order in which the execution tree has been traversed. However, for providing an intuitive stepping functionality, the structure of SE needs to be known. We discuss the implemented stepping mechanisms in Section 4.2.

Fortunately, there has been a Bachlor's project focused on capturing and outputting the symbolic execution tree [Buob 2015]. This SE logger creates the two files listed below.

File Name	Description
executionTreeData.js	The *executionTreeData.js* file contains all states of the execution organized as a JavaScript object in a tree-like structure. All states used when debugging the verification originate from this file.
dot_input.dot	The *dot_input.dot* file contains the execution tree description in Graphviz format. Currently the IDE does not use this file, it only transforms it to *.svg* format, allowing the user to view the complete execution tree outside the Viper IDE.

Table 5.7. *Temporary files output by the SE-backend.*

For the output of the logger to be usable by the IDE, we had to apply some modifications to the SE logger. First, in order for the IDE to be able to parse the *executionTreeData.js* file, the contained JavaScript object has to be in a valid JSON format. The logger used the less restrictive JavaScript format. Second, the description of the pre-states had to be completed, e.g., by adding the path conditions which were missing. Moreover, the location of the states and the information about the SE action type, (*Execute*, *Evaluate*, *Consume*, or *Produce*), had to be added to the output of the SE logger to ensure the IDE has all necessary data. Finally, the SE logger project has been completed in September 2015, but the contributions have never been merged into the main branch of the repository of the SE verification backend. To guarantee the availability of the SE logger in the verification backend, we merged the SE logger to the main branch of the repository.

5.5.3. Outputting the Counterexample

Another output required from the SE verification backend is the counterexample model generated by Z3 [Moura and Bjørner 2008]. The counterexample provides concrete values for all variables in the current state. E.g., in a state consisting of the integers a, b, and c, a counterexample model could look as follows:

```
"a@2 -> 1 b@1 -> 3 c@0 -> 0"
```

In the IDE, the counterexample model is used when debugging the SE. It is considered for creating the heap visualization in case the advanced mode is enabled. The details on how the counterexample is used can be found in Section 4.2.4. The heap visualization, introduced in Section 4.2.4, is concerned about visualizing the heap as a graph. We directly pass the model from Z3 on to the IDE without changing the format of the model. We use the standard output channel to pass the model to the IDE.

5.5.4. Backwards Compatibility

The original command line-based usage of the verification backend must stay unchanged. In order to preserve the user experience when invoking the SE backend, we only output the IDE-

related information when the `--ideMode` flag is set. If the flag is not set, the tools behavior is identical to the original functionality. The integrated SE logger collects information about the running verification. Both the SE log and the Z3 counterexample model can become large, potentially resulting in reduced performance. In Section 6.4, we describe the benchmark conducted to evaluate the impact of the additional functionality on the performance of the backend. For the command line-based usage of the backend (or when debugging is disabled in the configuration), collection and reporting this additional information is not needed. Therefore, we introduced the `--ideModeAdvanced` flag and only produce the additional information if this flag is set.

5.5.5. Validation of Changes

After the merge of the SE logger into the SE verification backend was completed, all regression test cases succeeded, indicating a successful merge. We expected the additional output generated by the SE logger to have an impact on the performance of the backend. We assessed this impact with a benchmark, comparing the performance of the verification backend with a disabled SE logger with the performance when the logger is enabled. Section 6.4 discusses the benchmark in detail.

5.6. Corner Cases

On several occasions during the development of the Viper IDE, the need arose to treat some cases individually. This section mentions the most important of these corner cases and the required changes resulting thereof.

5.6.1. Opening Individual Files Without an Opened Workspace

VS Code allows the user to either open a folder or an individual file. An opened folder defines a workspace, i.e., a directory containing all files relevant to the current project. On the other hand, a single file is treated individually, leaving the workspace undefined. For example, the workspace settings can only be accesses once a folder has been opened.

Both the IDE and the backends create temporary files and folders. A workspace provides a natural place to store these temporary files, being both controllable by the user and storing the files in a centralized location. For each file that is opened individually, without a workspace, the IDE would create new temporary files when naively using the location of the active file as a place for the temporary files. Moreover, e.g., the log file, containing information about the entire execution of the IDE, would not have a well defined location. Therefore, we place all files into the default temporary directory of the OS. Thus, all files are collected in one directory while the user remains in control of its location. See Appendix D for more information about the files created by the IDE.

Apart from opening individual files or using a directory as a workspace, the user could also start multiple instances of VS Code. However, the Viper IDE does not support this scenario.

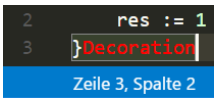

Figure 5.4. *Demonstration of a text decoration.*

When running multiple instances of Viper IDE, the processes slow each other down, as the verification involves computationally intense processes. Moreover, working with multiple files is easy within one instance of VS Code, since VS Code allows a workspace to be opened and provides support for tabs and tab-groups.

5.6.2. State Markers

VS Code supports text decorations, i.e., character sequences that can be inserted at a code location. We denote the line and column position inside the code as *location*. To the user, a decoration can look like usual text. However, text decorations can be formatted independently of the code and are not stored with the file, as they are augmented information. Consequently, decorations are read only. They do not take up space in the text as they are assigned to a character. The location before and directly after a decoration is logically the same location. Figure 5.4 shows a text decoration displaying the text *Decoration* in red color. The decoration is bound to the first character in the third line, as indicated with the border. The cursor is directly after the *n* of *Decoration* which is at the second character in the third row.

Some styles, such as the color of the decoration, can be set for each decoration individually. By design, VS Code only allows one text decoration to be put per location. The text decorations do not come with an on-click mechanism. However, debugging SE requires us to place multiple decorations at the same location, potentially with different colors, and support an on-click mechanism for each of them.

Our solution is to insert zero-width whitespace characters (unicode `\u200B`) at the location of the states in order to have a place to put each of the decorations. The user does not see the inserted characters due to their zero-width property. Detecting whether the user selected a state, in the execution trace of the debugged program, by clicking on the respective state marker is done by tracking the location of the cursor. In case the cursor location matches the location of one of the decorations, we know that the user just selected the state. For more information on debugging SE or the states involved, see Section 4.2.

The zero-width whitespace characters are added to the file when debugging is started and removed when it ends, when the file is closed or when the Viper IDE is shut down.

5.6.3. Installation

We simplified the complete installation process of the Viper IDE, VS Code, and all its other dependencies to improve its usability. The process is identical for all supported platforms, only in step 3 the default location of the Viper tools differs.

The official documentation of the Viper IDE can be found on:
https://bitbucket.org/viperproject/viper-ide/wiki/Home.
In case the installation procedure changes, we will update the documentation on this website.

Below, we list detailed instructions for the installation.

1. **Install VS Code.** The VS Code installer can be downloaded from their official website: https://code.visualstudio.com
2. **Download Viper tools.** The location of the prepackaged version of the Viper tools directory containing all dependencies of the Viper IDE can be obtained form here: https://bitbucket.org/viperproject/viper-ide/
3. **Move Viper tools to the right place.** The Viper tools folder must be put into the default location expected by the Viper IDE. Alternatively, after step 4, the expected location can be changed in the configuration, by modifying `viperSettings.paths.viperToolsPath`.
   - Windows: *C:\Program Files (x86)\Viper*
   - Mac: */usr/local/Viper*
   - Linux: */usr/local/Viper*
4. **Install the Viper IDE within VS Code using the marketplace.** After opening the extensions pane, ($ctrl + shift + x$ on Windows and Linux, or $cmd + shift + x$ on Mac) simply type *Viper* and click install.

5.6.4. Platform-Independent Path Configuration

The default configuration needs to be platform-independent. When running the Viper IDE, the user has the possibility to modify the configuration to be compatible with the operating system at hand. The default configuration, however, needs to be valid out-of-the-box on all supported platforms.

The differences between the supported platforms make it impossible to use the same configuration for all operating systems. For example, the format of paths on Windows differs from that of Linux paths. Windows requires backslashes as delimiters, whereas on Linux forward slashes are expected. Moreover, on Windows, absolute paths start with the drive letter. On Linux, an absolute path starts with a slash, e.g., */absolute/Linux/path*.

To satisfy the requirement of having a platform-independent default configuration, we introduced OS-dependent paths. In addition to permitting `string` type paths, we also allow OS-dependent paths which are JSON objects with the following format:

```
{
  "windows": "C:\\path\\to\\file.exe",
  "mac": "/path/to/file",
  "linux": "/path/to/file"
}
```

For Windows paths, single backslashes need to be escaped by `"\\"` in order to be adhere to the JSON standard. Whenever a platform-dependent path is found, the IDE only regards the entry that belongs to the currently acting operating system. This approach allows us to specify a separate path for each of the supported platforms and enables a platform-independent default configuration.

On Windows, when using the environment variable `%ProgramFiles%` within VS Code, it links to the program files directory VS Code was installed in. Either its value is *C:\Program Files* or *C:\Program Files (x86)*. Section 5.4 explains how internal variables such as `%ProgramFiles%` can be used and discusses other path resolution mechanisms provided by the IDE.

5.6.5. Output Buffer Size

In counterexamples, Z3 produces a concrete value for each of the variables, object fields, and functions that are of importance to the current proof. For long programs, this model can grow larger than the default buffer size of the standard output channel of 200KB, resulting in an interrupted communication. To overcome this problem, we introduced the configuration entry `viperSettings.advancedFeatures.verificationBufferSize`, making it possible for the user to increase the buffer size as required. Additionally, we implemented a detection mechanism, effectively warning the user in case the described problem occurred and pointing out how to change the buffer size.

5.7. Syntax Highlighting

Colored code is much more readable than non-colored code. In Section 4.1.1, we elaborate on the context-dependent syntax highlighting provided by the Viper IDE.

5.8. Automatic Code Formatting

When writing Viper programs, it is useful to have an automatic source code formatting functionality. Therefore, we implemented formatting support in the Viper IDE. The keyboard shortcut $alt + shift + f$ and the command `format code` both trigger the formatting process. The formatting corrects all indentations in the opened Viper file, introduces line breaks next to curly braces such that there is a line break before a closing and after an opening curly bracket. The spacing is corrected, such that type declarations have a space after the colon but none before it. Moreover, all duplicated spaces not used for indentation are replaced by single spaces. The formatting also includes removing all spaces after an opening bracket and before a closing bracket. For instance, Figure 5.5 shows a program with non-formatted Viper code. The same code after the automatic code formatting is shown in Figure 5.6.

The Viper framework includes a pretty printer that is able to format the parsed code. However, the Viper IDE features its own code formatting mechanism, because all code comments are lost during pretty printing. When the user requests code formatting of the opened Viper file by

```
1   method f ( arg :Ref )returns( res : Int ) { res:=1 }
```

Figure 5.5. *Non-formatted Viper code.*

```
1  method f(arg: Ref) returns (res: Int){
2      res := 1
3  }
```

Figure 5.6. *Code from Figure 5.5 after formatting.*

pressing $ctrl + shift + f$, the file is first tokenized. Then, the program is formatted by printing all tokens and the appropriate whitespaces between then.

5.9. Time Management of the Viper IDE

In case of a non-terminating verification, the Viper IDE should not await its termination indefinitely. We introduced a timeout for the verification to avoid this problem. When a verification takes too long, it is aborted automatically, (the default value is 100 seconds). Even a long Viper program with poor specifications, forcing the verification backend to perform extensive computation, manages to finish within 100 seconds. Additionally, the user has the ability to manually abort a long-running verification using the `stop` command, introduced in Section 4.4, or the stop button shown in the status bar during a verification. See Section 4.3, for more information about the status bar.

A more critical situation arises in case the backend startup does not complete. For example, an incompatible version of Nailgun can cause this. Nailgun is introduced in Section 5.1.5. In case the startup fails, the IDE would block, because only when the backend is ready, all functionality of the IDE can be used. Moreover, retrying the startup is not allowed, because the startup is already in progress. To resolve this issue, we implemented a timeout mechanism. If the startup takes longer, it is aborted and started anew. The default value of this timeout is 3 seconds. The duration of the backend startup usually takes less than a second. Therefore, in case the process is not finished within 3 seconds, it is reasonable to assume that it will not finish.

As appropriate values for the timeouts might differ from system to system, both the verification timeout and the backend startup timeout can be customized in the configuration to suit the needs of the user. In anticipation of the potential difference between the backends, we allow the user to individually choose a verification timeout for each of the backends. Section 5.3.3 introduces the notion of a verification backend and mentions how the timeout can be specified. For the backend startup, one timeout is sufficient, because the startup time is dominated by the startup of the Nailgun server.

5.10. The Shutdown Sequence of Viper IDE

When VS Code, hosting the Viper IDE, is shut down, the Viper IDE is deactivated. The deactivation includes the following steps: First, we terminate all external tools such as verification

backends, running instances of Z3, and the Nailgun server. In case the shutdown occurred during an active debugging session, we remove the special characters from the active file to allow the unhindered use of the file with any other program. Finally, we close the log file.

Unfortunately, VS Code limits the time an extension is given for deactivation. After this time limit the extension is forcefully closed, potentially leaving some child processes running. This is one of the known limitations of the Viper IDE.

6

Evaluation

To evaluate the work done in this project, we compare the initial plan to the goals achieved in the final version of the Viper IDE. Details about this comparison can be found in Section 6.1. Moreover, by individually collating the Viper IDE to other projects, we can evaluate the design decisions made during the course of the project. Section 6.2 compares our project to a theoretically envisioned verification IDE, a predecessor work done in the Chair of P.Müller. In Section 6.3 we compare the functionalities of the Viper IDE to the Dafny IDE, the closest related verification IDE project with advanced visual features. The performed benchmark is introduced in Section 6.4 In Section 6.5, we discuss the test coverage of the verification backends.

6.1. Achieved Goals

In this section, we compare the initially planned features to the ones actually implemented. The core tasks were expected to be part of the final system, whereas the extension tasks were seen as additional features that can but not necessarily have to be implemented. The features are ordered according to the priority we assigned them in the beginning of the project.

6.1.1. Core Features

As core tasks, we originally planned the design and implementation of following features for the Viper IDE. Figure 1.1 shows an overview of the planned features. We completed all core tasks. Therefore, for each feature, we link to the corresponding section of this document containing detailed description of its implementation. The following list contains all core tasks.

- **Code completion and syntax highlighting.** Section 4.1.1 discusses how syntax highlighting works in the Viper IDE. In Section 4.1.2 we explain the implemented code completion mechanisms.
- **Dynamic syntax checking.** We introduce our approach to dynamic syntax checking in Section 4.1.3.
- **Controlling the verification process.** Section 4.4 explains the different ways the user can control verification, debugging, and the IDE in general.
- **Counterexample visualization.** Section 4.2.4 describes how we integrate the counterexample model into the heap visualization.
- **Verification path visualization.** We provide two debugging modes that vary in the amount of detail presented and the completeness of information accessible. For example, in the advanced mode, all states are accessible, whereas the simplified mode only shows the states on the verification trace leading the state in which the verification error was detected. We annotate the code using state markers in order to visualize this trace. In Section 4.2.1, we introduce state markers and Section 4.2.2 elaborates on how we determine which states belong to the verification trace.
- **Progress reporting.** In Section 4.3.2 we discuss the progress reporting mechanism for estimating the remaining verification time and present the visualization of the estimate in the IDE.
- **Stepwise debugging.** We describe the implemented mechanisms for stepping through the verification while debugging in Section 4.2.2 for the simplified mode and in Section 4.2.3 for the advanced mode.

6.1.2. Extension Features

In the original project plan, we included tasks that could be but do not necessarily need to be part of the project. For the implemented extension tasks, we link to the section providing details about our solution. We explain the reasons for not including the others in the project. The following list contains tasks planned as extensions.

- **Heap visualization.** The heap visualization is an important feature of the IDE. Details about the heap visualization can be found in Section 4.2.4.
- **Inference of ghost operations.** We did not implement this feature.
- **Specification inference.** We integrated a Viper's specification inference tool into the IDE. Section 5.1.4 introduces the tool and we discuses the implemented mechanism for integrating an arbitrary Java program into the IDE in Section 5.3.4.
- **Trigger inference.** In favor of a more elaborate heap visualization and other tasks, we did not to include trigger inference into the Viper IDE. The IDE can handle Viper files whose verification includes triggers, but there is no special support for debugging them.
- **Triggering traces.** We did not implement this feature.

- **Anticipate incompleteness.** We did not implement this feature.

Trigger inference, anticipation of incompleteness, and inference of ghost operations were not implemented. These three tasks belong to the pre-verification information reporting phase introduced in Section 3.2. This phase follows the pre-verification tasks such as parsing, type checking, and desugaring, but is still *before* the actual verification. Figure 3.1 shows all phases of the verification process. Viper's SE verification backend is not designed to be used in an interactive mode. Changing it to support an interactive mode (which would be necessary for the pre-verification tasks), would require substantial changes in the structure of the verification backends. We did not change the backends to support the pre-verification information reporting phase as this would have exceeded the scope of this project.

6.1.3. Unplanned Features

During the course of the project, we shifted the priorities for the planned features. Partly, because we found some of the initially planned features to be of lower significance than expected and partly because we discovered new features that have higher value to the user. For instance, we implemented the partial execution tree, introduced in Section 4.2.4. This project is not a proof-of-concept work, but should instead result in a usable system. Therefore, we did not implement every planned feature, but rather extended the tasks in order to maximize the usability of the Viper IDE. Below, we list the features not planned originally, but nonetheless implemented in the project.

- **Automatic formatting.** A description of the formatting mechanism can be found in Section 5.8.
- **Output coloring.** Section 4.1.1 introduces the coloring of the output.
- **Extensibility.** The Viper IDE aims at maximizing extensibility. The customizability of the configuration allows the user to control many aspects of the IDE, allowing it to be used in a multitude of ways. For example, we allow the user to choose between two modes for debugging SE. Section 5.3 presents a detailed description of the implemented configuration. In Section 5.3.4 we explain the steps needed to support an additional external tool. As an experiment, we tried to support the Dafny verification language in the Viper IDE. Within a few hours, we were able to setup a new verification toolchain and integrate it into Viper IDE. The changes needed in the source code of the Viper IDE were minor and only amounted in adding the new file extension to the recognized extensions list.
- **Adopting tools.** Originally, we expected the tools to already provide most of the outputs required for the IDE to provide its functionality. However, the necessary changes amounted in substantial modifications on the verification backend including the complete merge of an old project into the SE backend. An explanation on the details of the performed changes and the reasoning of their necessity can be found in Section 5.5.
- **Partial execution tree.** The execution tree can be helpful for understanding the structure of the symbolic execution. However, this tree seldom fits on the screen entirely. Therefore, in addition to the verification path visualization described in Section 4.2.2 we also

implemented the partial execution tree graph. During debugging, this graph shows the part of the execution tree around the current state. We discuss the partial execution tree in Section 4.2.4. The tree is displayed in the state visualization panel.

- **Easy installation.** Section 5.6.4 introduces the platform-independent configuration we had to implement in order to enable an easy installation on all supported platforms. The four steps required for installing the Viper IDE amount in installing VS Code, downloading and extracting the pre-packaged archive containing all dependencies for the Viper IDE to the right location, and installing the Viper IDE extension within VS Code via the marketplace. Section 5.6.3 elaborates on the installation procedure, and states the expected location of the dependencies folder.
- **Navigation using state markers.** The state markers introduced in Section 4.2.1 extend the traditional navigation during an active debugging session in an intuitive way. The user can interactively explore the states of the verification by selecting the state markers drawn directly into the source code. Depending on which marker is selected, the IDE shows other markers, allowing the user to explore all states while not being overwhelmed by all states at once. Section 4.2.3 gives details about this state exploration.

6.1.4. Extensibility

We designed the Viper IDE to be extensible. The resulting modular design allows the IDE to support other verification tools requiring only minor adjustments. This adjustment is only necessary, because the Viper IDE is only activated when needed. Therefore, only when a Viper file is opened the IDE is enabled. Unfortunately, VS Code forces extensions to statically specify the enabling file types. Therefore, we must adjust the IDE when supporting new verification frameworks. So far, the Viper IDE was used as an IDE for verifying Dafny and Python programs. Even though the architecture of the Viper toolchain, is completely different from the one of Dafny or Python, the IDE is able to handle all of them.

6.2. Comparison of Visual Features with Design Concepts

Colombo et al. devised design concepts for debugging SE [Colombo et al. 2012] which are relevant to this thesis. Therefore, it is important to compare the Viper IDE to these design concepts. First, we analyze whether the Viper IDE abides to the design principles for verification IDEs. Then, we determine the compliance of the Viper IDE's debugging mechanisms to the envisioned ones.

The thesis about design concepts for debugging SE contains propositions for features not currently implemented in the Viper IDE. These could be added in a continuation of the current project.

6.2.1. Design Principles

Colombo et al. describe the following four design principles. For each aspect, we analyze the Viper IDE for compliance with the principals.

1. **"Adjust the level of detail."** We allow the user to decide how detailed output they would like to see by providing a flexible configuration options, (see Section 5.3). We offer two modes for debugging SE. The user can use the simplified debugging mode or activate the advanced debugging mode, showing much more details about the state of the verification. In advanced mode the user has the possibility to explore all states and thereby decide on the level of detail of the shown states. Section 4.2.3 discusses the mechanics of the advanced debugging mode in detail. An explanation of the simplified mode can be found in Section 4.2.2.
2. **"Use the user's language."** The IDE learns about the language of the user by looking at the naming of the files in the opened workspace and the content of those files. Whenever applicable, we use this knowledge to formulate the messages shown to the user. For example, the state visualization discussed in Section 4.2.4 contains the name of the verified file, the currently inspected method, the names of variables and snippets of the code. Additionally, we use understandable formulations throughout the IDE. Messages composed of parts with different origins are carefully tested to still yield understandable messages. Moreover, we consistently use the terminology of VS Code to simplify the search for users requiring more information about VS Code-related topics. By annotating the source code with state markers, we can visually show the states to the user in their own words, instead of describing them.
3. **"Reuse known concepts."** Building the Viper IDE as an extension of VS Code is favorable for reusing known concepts. Many of the basic features described in Section 4.1 make use of the tool support of VS Code. For example, code completion already worked reasonably well out-of-the-box. Section 4.1.2 introduces the implemented improvements on the code completion. The more advanced features provided by the Viper IDE did not have an out-of-the-box solution, however, we aimed at making good use of VS Code's infrastructure. Re-implementing existing solutions is problematic for two reasons. First, unnecessary re-implementations are hard to maintain as they clutter the code, make it less usable due to duplicate definitions of the same concepts, and are likely to fail with updates of VS Code. Second, extensions of VS Code can only access the GUI through the extensibility API. Therefore, it is hard to re-implement features with the same performance as the existing solutions. We only re-implemented existing functionality when the existing support was insufficient for our purposes. For example, automated saving discussed in Section 4.1.3 is such a feature.
4. **"Compare before and after."** It is important for the user to have the possibility of comparing the states before an action happened to the state after. The Viper IDE provides state comparison capabilities during the debugging of a verified Viper file. The visualization of two states can be displayed side-by-side. Instead of only being able to compare pre- and post-states, the Viper IDE allows the user to compare any two states. In Section 4.2.5 we present the state comparison in more details.

6.2.2. Comparison of the Debug Controls

The envisioned design concepts include a definition of how the debug controls should look. Figure 6.1 visualizes the semantics of the controls. They include a *step out*, two *step over* commands, and two controls for stepping in one for stepping forwards and one for stepping backwards

The simplified mode only allows users to move along the execution trace leading to the verification error. However, in the advanced debugging mode, the Viper IDE provides a similar set of debug controls. Figure 6.2 shows the semantics of the debug controls of the Viper IDE. The *step over* command has the same semantics as the envisioned *forward step over* command. However, we only provide one *step in* command with the same semantics as the *forward step in* control. We use the *step out* command to continue along the verification order, instead of effectively stepping back as proposed. The *step back* control provided by the Viper IDE allows to jump to the directly predecessing step. Therefore, it is similar to the envisioned *step in backwards*.

The debug controls of the Viper IDE provide support for stepping forward and inwards. They cover all three forward commands and both step in commands. Only for stepping back, we miss two debug controls, however, as we additionally allow the user to navigate using the state marker, stepping back is also covered. Both in the simplified and the advanced debugging mode, the two states not reachable using the debug controls are part of the execution trace leading to the current state and thus, shown as clickable state marker. Stepping back is not commonly used in traditional IDEs. Therefore, the semantics of the step back control may not be obvious to the user. The navigation based on the state markers provides an intuitive mechanism for stepping back and combines well with the debug control-based navigation.

6.3. Comparison to Dafny IDE

The Dafny IDE is implemented as an extension of Microsoft Visual Studio. VS Code and Visual Studio have almost nothing in common, except that both are a product of Microsoft. VS Code is a lightweight text editor, whereas Visual Studio is a full fledged IDE. Figure 6.3 shows an active debugging session in the Dafny IDE. In this section, we discuss and compare several features of the Dafny IDE and the Viper IDE. In Figure 6.4, we show a screenshot a debugging session in the Viper IDE. The layout of the elements is similar in both IDEs: the source code is shown in the left panel, the error list on the bottom panel, and the visualization of the inspected verification states is drawn on the right panel.

6.3.1. State Markers

The Dafny IDE is providing state markers that behave very similar to our implementation introduced in Section 4.2.1. For each state on the execution trace leading to the selected error state a state marker is shown. The simplified debugging mode provided by the Viper IDE does the same thing. Only the fact that the Dafny IDE allows the user to compare any two states

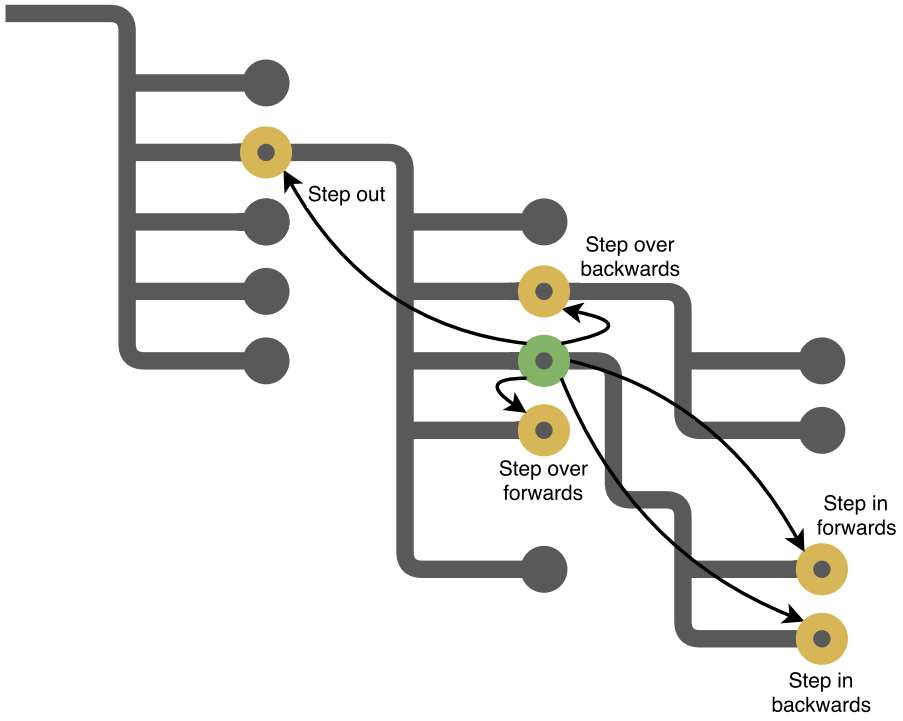

Figure 6.1. *Diagram demonstrating the semantics of the debug controls envisioned by Ivo Colombo.*

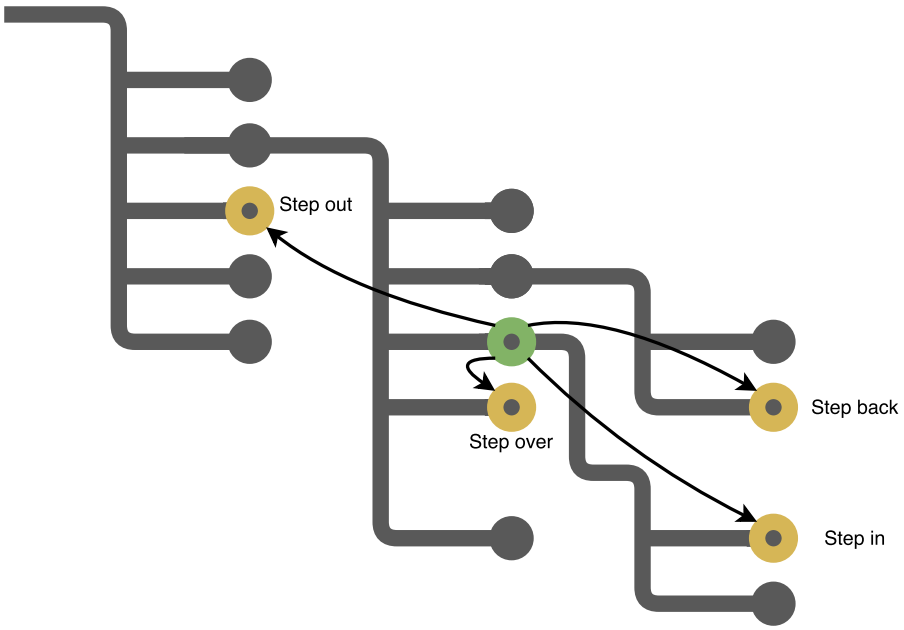

Figure 6.2. *Demonstration of the debug controls of the advanced debugging mode of the Viper IDE.*

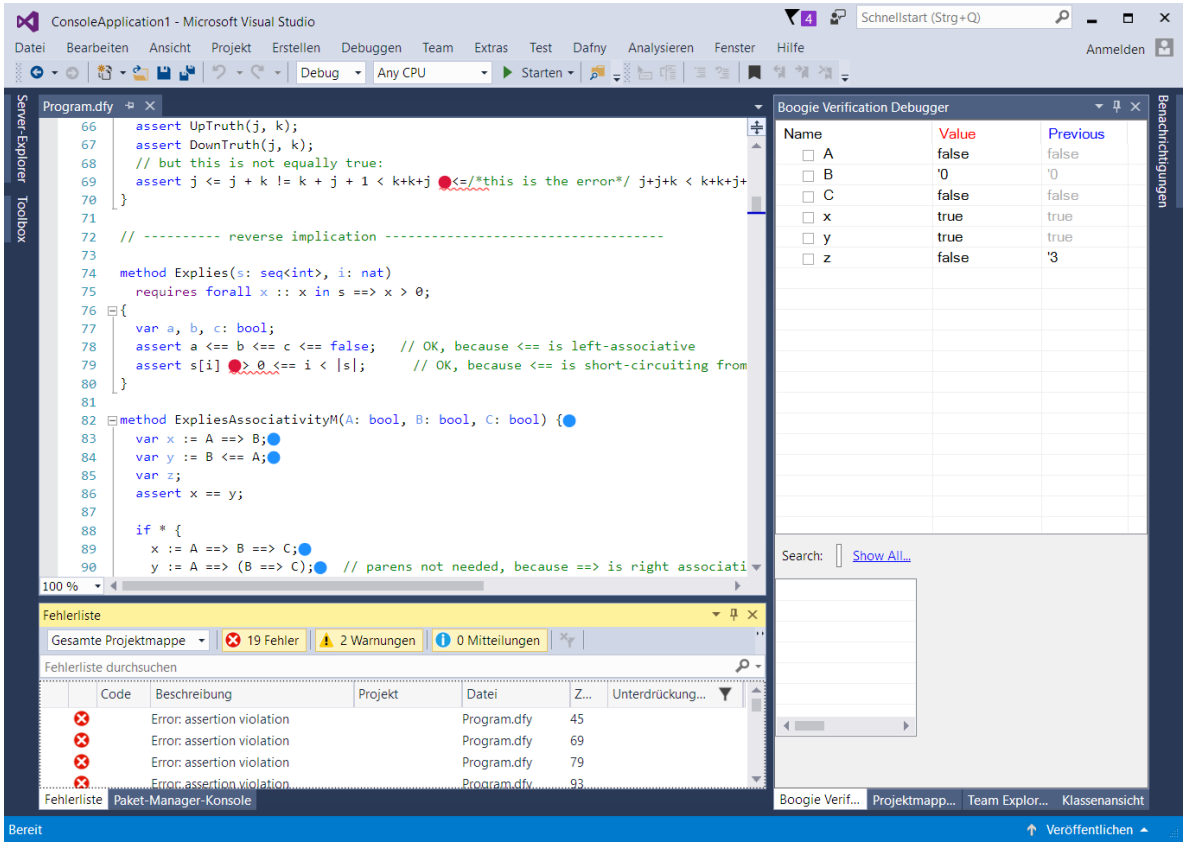

Figure 6.3. *Screenshot of an active debugging session in Dafny IDE.*

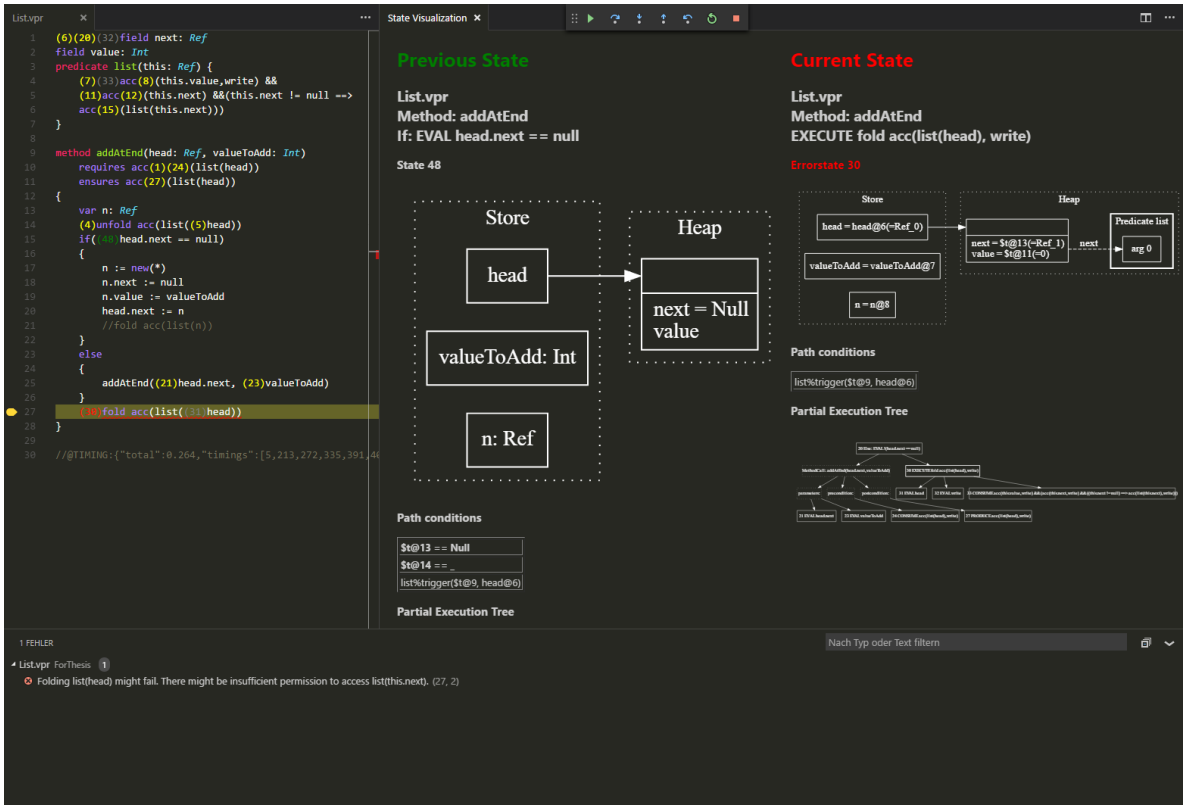

Figure 6.4. *Debugging failed verification in the symbolic execution backend in the Viper IDE.*

on the execution trace whereas we always show the error state is different. In the Viper IDE comparing any two states is only possible in the advanced debugging mode. The Dafny IDE shows only collapsed state markers, whereas the Viper IDE expands the states on the execution trace. Showing the numbers of the states conveys the order of the states during the verification. Moreover, the shown states depend on what states the user inspected before, allowing the user to explore the verification.

6.3.2. Error List

Both IDEs provide an error list with hyperlinks allowing the user to jump to the line where the error is marked. As the error list is a feature provided by VS Code and Microsoft Visual Studio, further comparing the two IDEs essentially results in comparing Visual Studio with VS Code.

6.3.3. State Visualization

The Dafny IDE provides a list of all variables with their values, whereas the Viper IDE shows a graph with a node for each variable. Our approach has the advantage of conveying the structure of the data. For example, a user can visually detect that the memory is containing a linked list, or a tree structure. In addition to the variables, we show the path conditions and the partial execution tree. The path conditions are accumulated facts due to the chosen path such as assumptions about branch conditions in the current state of SE. Section 4.2.4 explains path conditions in detail. The partial execution tree shows the current state in the context of the execution. Decisions taken during the verification are visualized as branches. The partial execution tree assists the user in understanding the structure of the execution which in turn helps to locate the cause of a verification failure.

6.3.4. Comparison of Progress Reporting

Progress reporting implemented in the Dafny IDE is taking a different approach than the mechanism provided by the Viper IDE. As shown in Figure 6.5, the color in the left gutter of the editor is used to show the progress [Leino and Wüstholz 2014] an orange bar shows that the verification on the method has not yet started, whereas a purple bar indicates that the method is currently being processed. For the Viper IDE, we present the progress of the entire verification in the status bar. Our approach does not show the verification progress of individual parts of the program, but instead provides a rather accurate estimate of the verification duration. When working with large files containing more lines of code than fit on the screen, marking the verification progress in the source code can become ineffective. For smaller files (that verify in about a fraction of a second) progress marking is not providing useful information, but the quickly changing colors might distract the user. Figure 4.15 shows the progress bar provided by the Viper IDE. In Section 4.3.2 we discuss the implemented progress reporting in more detail.

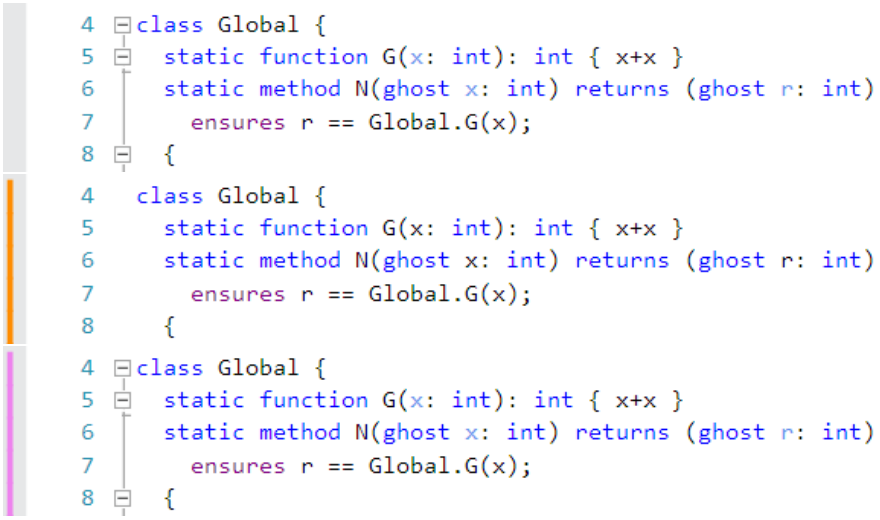

Figure 6.5. *Screenshots demonstrating the progress visualization in the Dafny IDE.*

6.4. Performance Benchmark of the Adopted Backend

In Section 5.5 we discuss the changes performed on the verification backends. The biggest change was the inclusion of the SE logger into the backend. In order to assess the impact of including the SE logger, we created a benchmark. In both scenarios the test cases were executed from the command line. We verified 385 Viper files with the SE backend and the SE logger either disabled or enabled. All experiments were performed on the same system (CPU: 1.73 GHz, quad core, Ram: 8 GB, OS: Windows 10 Professional 64-bit) and averaged across 3 runs. The results of the benchmark can be found in Table 6.1.

For verifying all 385 Viper files the SE backend needed on average 418s when the SE logger was disabled. When adding the SE logger to the system, running the benchmark took 42 seconds, or 10% longer. This slowdown originates in the computation required to create the execution tree description and write all states encountered during the verification to the *executionTreeData.js* file. In the SE logger project the performance of the logger was never assessed. Therefore, the logger could potentially be sped up significantly. We However, the logger is only constructing the output when the `--ideModeAdvanced` flag is specified at the start of the verification. When debugging is disabled (by setting `advancedFeatures` to false), the user is not interested in debugging and the performance not influenced by the logger.

Setup	**Average Total Time**	**%**
SE backend with the SE logger disabled	439s	100
SE backend with the SE logger enabled	481s	110

Table 6.1. *Results of the benchmark assessing the impact of the SE logger on the performance of the SE backend.*

6.5. Test Coverage

The IDE adaptation of the SE backend of the Viper toolchain is described in Section 5.5. In order to validate the performed modifications, we tested Viper's symbolic execution backend with all existing unit tests. After integration of the SE logger has been completed and all other changes in the verification backends have been finalized, we achieved a unit test coverage of 100%. Therefore, we conclude that our changes have not introduced regression. In Section 5.5.5 we discuss other validations of the changes.

6.6. Limitations

During the course of the project, we encountered and fixed many problems. Unfortunately, there are still some issues left that we did not yet manage to resolve. Here, we list the major problems that for some reason remained unsolved.

- **Mutiple instances of VS Code.** We do not support multiple instances of VS Code simultaneously running the Viper IDE. The IDE might crash, as both instances use the same resources, start and stop the same programs, and communicate on the same channels. Section 5.6.1 discusses this issue.
- **Tab groups.** In VS Code, each opened file is shown as a tab. Since version 1.3, VS Code supports up to three tab groups to organize tabs, each of which may contain multiple tabs. Figure 2.3 shows a screenshot of VS Code with three open files, distributed among two tab groups. VS Code displays the tab groups next to each other as editors, showing the top-most tab in this group. Verification works best if the files to verify are opened in the first tab group within VS Code. The reason is that the state markers used during debugging can only be added to the file when it is selected. For example, when the user switches the focus from the verified file, adding the state markers fails. Therefore, we have to programmatically return the focus to the verified file. Unfortunately, we do not know in which tab group the file resides, thus, we have to assume it is the first tab group. Alternatively, we could re-open the file instead of changing the focus. However, this might move the file to another tab group. Moreover, due to a bug in VS Code this operation takes almost a second.
- **Shutdown problem.** The fundamental problem is that the Viper IDE extension is not given enough time to reliably stop all external tools. Section 5.10 discusses the shutdown problem in detail.

In this project, we implemented a software verification IDE that assists to the user in writing, verifying and debugging Viper programs. We completed all 7 core tasks, 2 of the 6 extension tasks, and 7 unplanned tasks. Moreover, we did not only build a proof-of-concept system, but instead, we successfully managed to create a usable IDE. The Viper IDE is actively used by a small group of people and extension was installed on over 500 systems. There are still some issues to be solved, and interesting follow-up projects to be done, but nevertheless, the project can be considered a success. Due to the extensible design of the IDE, we could apply the system with only minor modification to support other verification toolchains. The Viper IDE manages to support both the Dafny and the Python verification framework.

7

Conclusion and Future Work

In this thesis, we built an IDE tailored to the needs of software verification. To the best of our knowledge, this is the first IDE with native support for debugging symbolic execution and represents a signification contribution because it is widely applicable and simplifies the adoption of software verification.

The IDE provides assistance for editing, such as syntax highlighting, code completion, and is able to annotate the source code with the detected errors. Moreover, the IDE includes an automatic formatting mechanism that preserves comments and order of methods, unlike Viper's pretty printer, allowing the user to format the code on a button press. Additionally, the IDE provides automated verification invocation. This mechanism monitors opened files and invokes a verification session, when it detects the need. For example, newly opened Viper files are automatically verified.

The Viper IDE was implemented as an extension of VS Code. We use the functionality provided by VS Code in order to create a suitable environment for developing Viper programs. We use the main source code editing frame, the status bar, the error list, the output panel, and the functionality for previewing HTML to present the user with information about the verification.

We devised an extensive debugging support for analyzing completed SE verifications whether they are successful or fail. The user can choose the simplified mode, tailored for fixing verification errors, or the advanced mode, meeting the needs of verification tool developers. In the advanced mode, we allow the user to explore all verification states and visualize the individual states of the verification in detail. Moreover, the IDE allows to visually compare verification states. During debugging, the user can navigate in the execution state space using both the traditional debug controls and the state markers, embedding the states directly in the source code.

We defined a set of commands through which the user can control the IDE. The commands

include verifying the opened Viper file, debugging the verification, toggling the automatic verification invocation, and opening the log file. We provide a shortcut for the most important commands. VS Code allows the user to customize the shortcuts. The IDE executes many actions autonomously when the need is detected, e.g., saving or verifying the opened Viper file.

The Viper protocol defines all communication interfaces within the IDE and between the IDE and external tools. The Viper IDE has an expressive configuration that enables the user to adjust the behavior of the IDE to their needs. As we designed the configuration with extensibility in mind, external tools can be easily integrated into the IDE by setting them up in the configuration.

To improve the user experience, we implemented a time management system that detects and terminates long-running processes. The shutdown sequence ensures the termination of external tools when the Viper IDE is closed. The logging system records all taken actions and is useful for diagnosing problems, in both external tools and the IDE itself.

The extensible design of the Wiper IDE allows the system to support other verification toolchains. The IDE successfully adopted the Dafny toolchain and the Python verification framework. The expressive configuration mechanism allows the integration of external tools. Moreover, the user can control much of the IDE's behavior by customizing the configuration.

The basic features provided by the Viper IDE increase the ease of writing, and verifying programs. The process of pinpointing verification problems is assisted by the IDE's extensive debugging support. Together with its easy installation, extensibility, and capability of supporting new verification toolchains, the Viper IDE manages to both simplify the adoption of software verification and increase the productivity of verification experts. The source code of the Viper IDE is available online on https://bitbucket.org/viperproject/viper-ide.

7.1. Future Work

In this section, we include all features that are of importance for the IDE in general, but are not part of the current project. It both serves as an inspiration for future projects and limits the current project by stating what has not been done. The tasks considered future work are listed below.

- **Debugging VCG backends.** The Viper IDE provides debugging support for SE, but additionally, it was designed to allow adding support for the VCG verification backend in the future. Implementing this support can be done by modifying the VCG backend to comply with the Viper protocol defined in Section 5.2.
- **Integration with front ends.** The information gathered and shown by the IDE relates to the Viper intermediate source code. However, since programs can be translated from languages such as Chalice, Scala, or Python to the Viper intermediate language, it is interesting to think about mapping the debug information back to the source language [Leino et al. 2009]. Since the current state of the existing Viper front ends is incipient, integration is out of scope of the current project.
- **Extending the heap visualization.** Currently the heap visualization does not support quantifiers. As data structures are often formalized using quantifiers, supporting them

would be a promising project.

- **Integrated static SMT profiling.** Microsoft Research provides an axiom profiler for Z3 that could be integrated into the IDE [Moura and Bjørner 2008]. Currently there is no tool for profiling SE directly. Building such a tool would be valuable and could be the subject of another project.
- **Adding abstract syntax tree.** The Viper IDE currently does not have access to the abstract syntax tree (AST) of the Viper program under verification. Adding AST-awareness to the IDE would enable interesting new features. For example, we currently only store information about what is already verified on the level of files. AST-awareness allows a more fine-grained approach, storing information for each method, function, and predicate and, enables the verification to be controlled more precisely. For instance, in case the user edits a method, we would not need to reverify the entire file, but only the modified method. If the modification affected the contracts of the method all dependent methods, i.e., those invoking the modified method, need to be re-verified, too.
- **Unimplemented extension features.** We did not implement all features planned as extension tasks for the Viper IDE. The anticipation of incompleteness, the trigger inference, the inference of ghost operations, all discussed in Section 3.2, are good candidate tasks for future work.
- **Custom dependency checks.** We perform a series of checks, e.g., the existence of a compatible version of the Java runtime environment. However, to allow the user to easily add checks additional checks, we could include descriptions of these checks into the configuration. Section 5.3 discusses the configuration in detail.
- **Automated installation.** Currently the dependencies for the Viper IDE have to be downloaded manually. All dependencies are already packaged in an archive which needs to be extracted to the right directory. However, an automation of this process is still preferable, and could be done in the future.
- **Regression testing for the Viper IDE.** For faster detection of newly introduced bugs (or fixed bugs that were resurrected), the Viper IDE could provide a unit test API. This would both increase the quality of the Viper IDE and allow for a faster development.

7

List of Figures

7

List of Tables

Bibliography

APPLE, 2011. Memory leak in Objective C. http://stackoverflow.com/questions/6558130/objective-c-memory-leak-issue-in-class-method.

AT&T, AND BELL-LABS, 2000. Graphviz. http://www.graphviz.org/.

BUOB, A., 2015. Recording symbolic executions. https://www.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Andreas_Buob_BA_report.pdf.

COLOMBO, I., SCHERWOFF, M., AND MÜLLER, P. 2012. *Debugging Symbolic Execution*. Master's thesis, ETH Zürich.

FERRARA, P., AND MÜLLER, P., 2012. Automatic inference of access permissions verification, model checking, and abstract interpretation (VMCAI). http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=FerraraMueller12.pdf.

GITHUB, 2014. Atom. https://github.com/atom/atom.

HEULE, S., KASSIOS, I. T., MÜLLER, P., AND SUMMERS, A. J., 2013. Verification condition generation for permission logics with abstract predicates and abstraction functions. http://people.inf.ethz.ch/summersa/wiki/lib/exe/fetch.php?media=papers:predicates.pdf.

HEULE, S., SUMMERS, A. J., AND MÜLLER, P. 2013. *Verification Condition Generation for the Intermediate Verification Language SIL*. mathesis, ETH Zürich.

JACOBS, B., SMANS, J., AND PIESSENS, F., 2010. Verification of imperative programs: The VeriFast approach. a draft course text. https://lirias.kuleuven.be/bitstream/123456789/266053/1/CW578.pdf.

LE GOUES, C., LEINO, K. R. M., AND MOSKAL, M. 2011. The Boogie verification debugger (tool paper). *In Gilles Barthe, Alberto Pardo & Gerardo Schneider, editors: Software Engineering and Formal Methods (SEFM), LNCS 7041, Springer, pp. 407-414*.

LEINO, K. R. M., AND PIT-CLAUDEL, C., 2016. Trigger selection strategies to stabilize program verifiers. http://pit-claudel.fr/clement/papers/dafny-trigger-selection-CAV16.pdf.

LEINO, K. R. M., AND WÜSTHOLZ, V. 2014. The Dafny integrated development environment. *In Formal-IDE*.

LEINO, K. R. M., MÜLLER, P., AND SMANS, J. 2009. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*, Springer-Verlag, A. Aldini, G. Barthe, and R. Gorrieri, Eds., vol. 5705 of *Lecture Notes in Computer Science*, 195–222.

LEINO, K. R. M., 2008. This is Boogie 2. https://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf.

MARTYLAMB, 2004. Nailgun: Insanely fast Java. http://martiansoftware.com/nailgun/.

MICROSOFT, 2015. Visual Studio Code. https://code.visualstudio.com/.

MOURA, L., AND BJØRNER, N., 2008. An efficient SMT solver. http://link.springer.com/chapter/10.1007/978-3-540-78800-3_24.

MÜLLER, P., SCHWERHOFF, M., AND SUMMERS, A. J. 2016. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer-Verlag, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583 of *LNCS*, 41–62.

SCHWERHOFF, M. 2016. *To be published.* PhD thesis, ETH Zürich.

STACKOVERFLOW, 2016. Survey 2016, Desktop Operating System. https://stackoverflow.com/research/developer-survey-2016#technology-desktop-operating-system.

2007. Sublime Text 3. https://www.sublimetext.com/.

A

Infrastructure

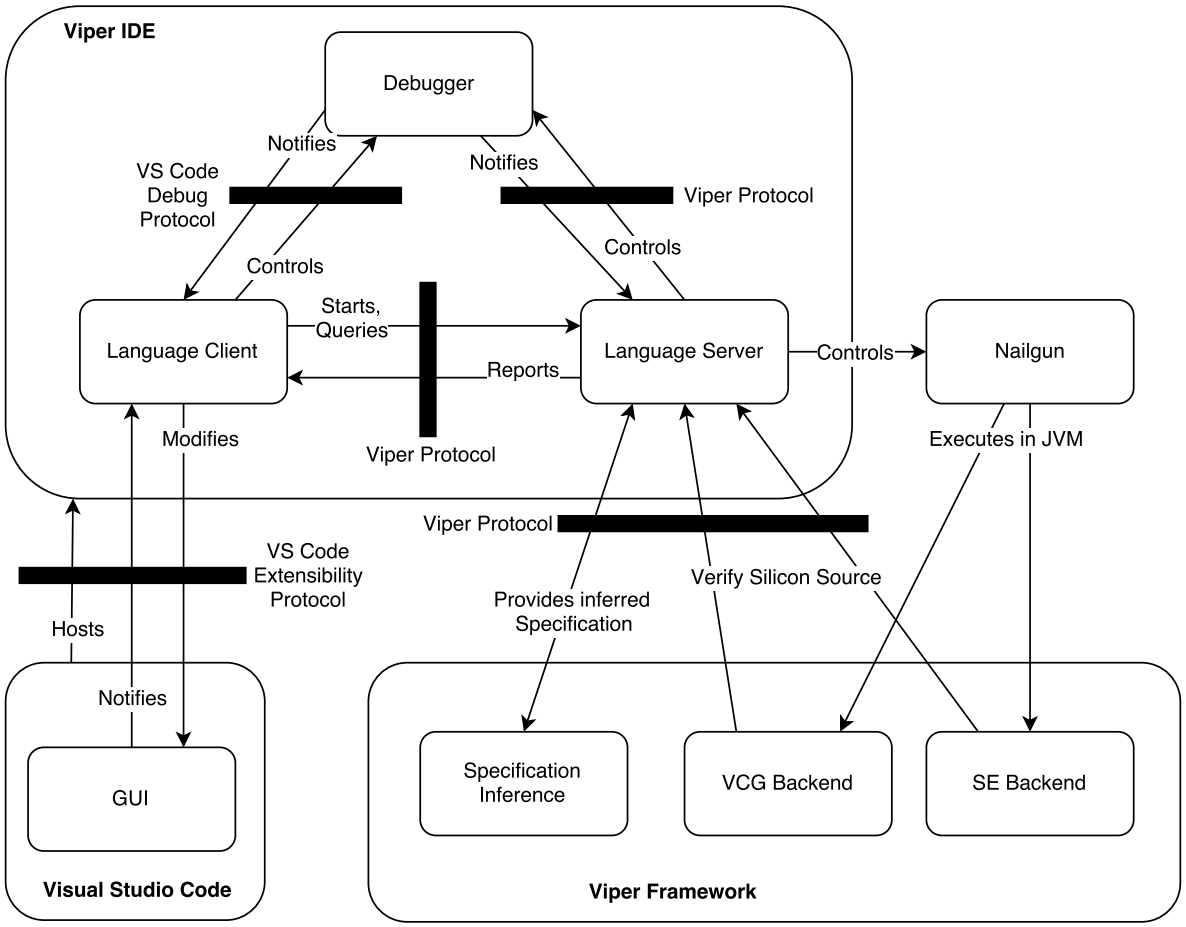

Figure A.1. *Detailed overview of the infrastructure of the IDE.*

B

Debug Launch Configuration

```
{
  name: "Viper Debug",
  type: "viper",
  request: "launch",
  program: "/absolute/path/to/active/file.vpr",
  startInState: 0,
  internalConsoleOptions: "neverOpen"
}
```

C

List.vpr

```
field next: Ref
field value: Int
predicate list(this: Ref) {
   acc(this.value)
   && acc(this.next)
   && (this.next != null ==> acc(list(this.next)))
}
method addAtEnd(head: Ref, valueToAdd: Int)
   requires acc(list(head))
   ensures acc(list(head))
{
   var n: Ref
   unfold acc(list(head))
   if ( head.next == null ) {
      n := new(*)
      n.next := null
      n.value := valueToAdd
      head.next := n
      fold acc(list(n))
   } else {
      addAtEnd(head.next, valueToAdd)
   }
   fold acc(list(head))
}
```

D

Temporary Files

The Viper IDE generates temporary files persistently retaining information. For example, the log file needs to be accessible even after VS Code was closed. The temporary files are stored in the default *Temp/.vscode* directory of the active operating system. Here, we present the generated files and their use.

File	**Example**	**Description**
Log file	*viper.log*	The main log file of the IDE.
Heap files	*heap0.dot* *heap1_old.svg*	For showing the heap and the old heap in the state visualization, it is necessary to create a *.dot* file and a *.svg* file for each shown graph. We need to create two sets of files corresponding to the two states shown side-by-side. Section 4.2.4 introduces the state visualization and emphasizes the importance of the heap.
Partial execution tree files	*partialExecutionTree0.dot* *partialExecutionTree1.svg*	Similarly to the heap file, we also need a *.dot* and a *.svg* file for both shown states.

Execution tree description file	*dot_input.dot*	The SE logger outputs the *dot_input.dot* file which contains a Graphviz description of the complete execution tree.
Execution tree vector graphic	symbExLoggerOutput.svg	The IDE converts the *dot_input.dot* file to an SVG representation stored in the *symbExLoggerOutput.svg* file.
Execution states	executionTreeData.js	The SE logger is written to the *executionTreeData.js* file in *JSON* format.

Table D.1. *Temporary files produced by the IDE.*

E

Default Configuration

```
// Viper Configuration

// All Nailgun-related settings.
// The Nailgun port can be referred to by $nailgunPort$.
"viperSettings.nailgunSettings": {
  "v": "0.3.5",
  "serverJar": "$viperTools$/nailgun/nailgun.jar",
  "clientExecutable": "$viperTools$/nailgun/ng",
  "port": "7654",
  "timeout": 3000
},

// You can add your backends here.
"viperSettings.verificationBackends": [
  {
    "v": "0.3.5",
    "name": "silicon",
    "paths": [
      "$viperTools$/backends/silicon.jar"
    ],
    "useNailgun": true,
    "timeout": 100000,
    "stages": [
      {
        "name": "verify",
        "isVerification": true,
```

```
      "mainMethod": "viper.silicon.SiliconRunner",
      "customArguments": "--ideMode --ideModeAdvanced --z3Exe
         $z3Exe$ $fileToVerify$"
    }
  ]
},
{
  "v": "0.3.5",
  "name": "silicon with infer",
  "paths": [
    "$viperTools$/backends/silicon.jar",
    "$viperTools$/backends/infer.jar"
  ],
  "useNailgun": true,
  "timeout": 100000,
  "stages": [
    {
      "name": "verify",
      "isVerification": true,
      "mainMethod": "viper.silicon.SiliconRunner",
      "customArguments": "--ideMode --ideModeAdvanced --z3Exe
         $z3Exe$ $fileToVerify$",
      "onParsingError": "",
      "onTypeCheckingError": "",
      "onVerificationError": "infer",
      "onSuccess": ""
    },
    {
      "name": "infer",
      "isVerification": false,
      "mainMethod": "
         ch.ethz.inf.pm.sample.permissionanalysis.Main",
      "customArguments": "$fileToVerify$",
      "onParsingError": "",
      "onTypeCheckingError": "",
      "onVerificationError": "",
      "onSuccess": "verify"
    }
  ]
},
{
  "v": "0.3.5",
  "name": "carbon",
  "paths": [
    "$viperTools$/backends/carbon.jar"
  ],
```

```
    "useNailgun": true,
    "timeout": 100000,
    "stages": [
      {
        "name": "verify",
        "isVerification": true,
        "mainMethod": "viper.carbon.Carbon",
        "customArguments": "--z3Exe $z3Exe$ --boogieExe
           $boogieExe$ $fileToVerify$"
      }
    ]
  }
],

// Paths to the dependencies.
"viperSettings.paths": {
  "v": "0.3.5",
  "viperToolsPath": {
    "windows": "%ProgramFiles%\\Viper\\",
    "linux": "/usr/local/Viper",
    "mac": "/usr/local/Viper"
  },
  "z3Executable": {
    "windows": "$viperTools$/z3/bin/z3.exe",
    "linux": "$viperTools$/z3/bin/z3",
    "mac": "$viperTools$/z3/bin/z3"
  },
  "boogieExecutable": {
    "windows": "$viperTools$/boogie/Binaries/Boogie.exe",
    "linux": "$viperTools$/boogie/Binaries/Boogie.exe",
    "mac": "$viperTools$/boogie/Binaries/Boogie"
  },
  "dotExecutable": {
    "windows": "$viperTools$/graphviz/bin/dot.exe",
    "linux": "$viperTools$/graphviz/bin/dot",
    "mac": "$viperTools$/graphviz/bin/dot"
  }
},

// General user preferences.
"viperSettings.preferences": {
  "v": "0.3.5",
  "autoSave": true,
  "logLevel": 1,
  "autoVerifyAfterBackendChange": true,
  "showProgress": true
```

```
},

// Settings used for running Java commands.
"viperSettings.javaSettings": {
  "v": "0.3.5",
  "customArguments": "-Xmx2048m -Xss16m -cp $backendPaths$"
},

// Settings concerning the advanced features.
"viperSettings.advancedFeatures": {
  "v": "0.3.5",
  "enabled": true,
  "showSymbolicState": false,
  "simpleMode": true,
  "darkGraphs": true,
  "showOldState": true,
  "showPartialExecutionTree": true,
  "verificationBufferSize": 102400
}
```

F

Output Messages

When setting the logLevel to verbosity 5 (LowLevelDebug), the output panel contains a detailed description of the internal state of the IDE. Below, we present a sample output generated when starting the IDE and opening the file *List.vpr*.

```
1  LogFilePath is: "C:\Users\ruben\AppData\Local\Temp\.vscode\
      viper.log"
2  Viper-Client is now active!
3  OS: Windows
4  Start Language Server
5  S: Configuration changed
6  File openend: List.vpr
7  S: The settings are ok
8  S: Change Backend: from No Backend to silicon
9  S: Stop all running verificationTasks
10 S: Check if Jre is installed
11 Backend is not ready, wait for backend to start.
12 The new state is: Stopping
13 S: gracefully shutting down nailgun server on port: 7654
14 S: nailgun server is stopped
15 S: Set Stopped
16 The new state is: Stopped
17 S: starting nailgun server
18 The new state is: Starting
19 S: java -Xmx2048m -Xss16m -cp "C:\Program Files (x86)\Viper\
      nailgun\nailgun.jar";"G:\Masterthesis\Viper\siliconFork\
      target\scala-2.11\silicon.jar" -server com.martiansoftware.
```

```
   nailgun.NGServer 127.0.0.1:7654
20 S: [NS]: NGServer 0.9.2-SNAPSHOT started on address /127.0.0.1
   port 7654.
21
22 S: "C:\Program Files (x86)\Viper\nailgun\ng.exe" --nailgun-port
    7654 NOT_USED_CLASS_NAME
23 S: The backend is ready for verification
24 Backend ready: silicon
25 Reset all viper files
26 autoVerify after backend change
27 Verification is expected to take 4.1 seconds
28 Hide states for List.vpr
29 No special chars to remove
30 Hide decorations
31 Request verification for List.vpr
32 S: start or restart verification
33 S: Stop all running verificationTasks
34 S: verify List.vpr
35 S: silicon verification started
36 The new state is: VerificationRunning
37 Progress: 0.13003901170351106
38 S: "C:\Program Files (x86)\Viper\nailgun\ng.exe" viper.silicon.
   SiliconRunner --nailgun-port 7654 --ideMode --
   ideModeAdvanced --z3Exe "C:\Program Files (x86)\Viper\z3\bin
   \z3.exe" "G:\Masterthesis\verification Tests\ForThesis\List.
   vpr"
39 Progress: 10.879930645860425
40 Progress: 21.738188123103598
41 Progress: 32.57477243172952
42 Progress: 43.41135674035544
43 Progress: 54.24794104898136
44 Progress: 65.12787169484179
45 Progress: 75.9644560034677
46 The new state is: VerificationRunning
47 Progress: 81.01208114884886
48 Progress: 91.05394946505805
49 Progress: 91.19053038925563
50 Progress: 96.06587374199451
51 S: Error: [silicon] [fold.failed:insufficient.permission] 27:2
   Folding list(head) might fail. There might be insufficient
   permission to access list(this.next).
52 S: [NS]: NGSession 2: 127.0.0.1: viper.silicon.SiliconRunner
   disconnected
53
54 S: [NS]: NGSession 2: 127.0.0.1: viper.silicon.SiliconRunner
   exited with status 1
```

```
55 S: [NS]:
56
57 S: Child process exited with code 1
58 The new state is: PostProcessing
59 Progress: 98.99336536261725
60 S: Loading The symbexLog from: C:\Users\ruben\AppData\Local\
      Temp\.vscode\executionTreeData.js
61 S: Execution tree successfully loaded: 2 toplevel constructs
      found
62 S: Number of Steps: 87
63 S: Update the decoration options (60)
64 Store new States
65 The new state is: Ready
66 Progress: 99
67 Verifying List.vpr failed after 3.9 seconds with 1 errors
68 Timing information added to List.vpr
```

- **Lines 1-3.** we inform the user about the detected operating system, the location of the log file and that the Viper IDE has been activated.
- **Lines 4-5.** The language client starts the language server, which detects a change in the configuration. All messages starting with a `S:` originate from the server. Messages from the debugger are prefixed with `D:`.
- **Lines 7-10.** The server checks the configuration, which is valid, and prepares the backend startup.
- **Lines 6,11.** The client detected the opened file, but does not start a verification because the backend is not ready yet.
- **Lines 12-16.** In case a conflicting instance of the Nailgun server was already running, we first need to shut it down.
- **Lines 17-19.** The exact command used for starting the Nailgun server is output.
- **Lines 20-21.** The Nailgun server is started and this message is received from its process.
- **Lines 22-23.** Due to a bug we detected in the implementation of Nailgun, we can only be sure it really started, when checking it with a dummy request.
- **Lines 24-25.** Upon notification about the completed startup of the Nailgun client, the language client resets the state of all opened files (when a file was verified with another backend, we need to rerun the verification with the new backend).
- **Lines 26-27.** The client can now prepare the verification of the file List.vpr, since the file already contained timing information about a previous verification, we know how long the verification is expected to take.
- **Lines 28-30.** In case a debugging session has been running, the file needs to be cleaned of the special characters. We also do that even if we just started the IDE, because the file might still contain special characters.

- **Lines 31-38.** The client is now ready and requests the server to invoke the verification. The server makes sure no other verification is running, before forking the verification process.
- **Lines 37,39-50,59,66** The progress reported to the user, is also written to the log, in order to have a chronological correspondence between the observed progress and the log messages.
- **Line 51.** The verification error detected in the file List.vpr is included in the log.
- **Lines 52-57.** The Nailgun server detected the end of the verification process and reports corresponding messages.
- **Lines 58-62.** After the verification, the IDE gathers the information required for debugging SE. In this post processing phase, the IDE manged to load 87 states and 2 top level constructs (methods, functions, or predicates) from the *executionTreeData.js* file.
- **Lines 63-65.** The information required for showing the 60 state markers in the source code, in case the user wishes to debug, are sent to the client.
- **Lines 67-68.** As soon as the end of the verification process is detected, the IDE informs the user about the outcome and updates the timing information in the List.vpr file.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

Advanced Features for an Integrated Verification Environment

Authored by (in block letters):

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

Name(s):	**First name(s):**
Kälin	Ruben

With my signature I confirm that

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

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

Place, date	**Signature(s)**
Zürich, 3.11.2016	

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