ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Developing an Interactive, Web-Based Tutorial for an Intermediate Verification Language

Bachelor Thesis Report

Mathias Birrer

August 16, 2015

Supervised by Malte Schwerhoff
Prof. Dr. Peter Müller

Chair of Programming Methology
Department of Computer Science, ETH Zürich

Abstract

This Bachelor's thesis introduces an interactive, web-based tutorial for an intermediate verification language. The tutorial uses an existing web interface, which allows to use various verification tools in a web-browser. The project has three parts: the improvement of the existing web interface, in a way that the interface can be used in an interactive tutorial, a framework to easily recreate an interactive online tutorial, and a tutorial of the Silver intermediate verification language. The changes to the web interface include improvements to tool provision, design changes and alterations to the communication protocol. Besides the improved web interface, the project introduces a framework for an automated workflow to implement an interactive tutorial. The goal was to minimize the effort needed to create a tutorial for a verification language. This allows researchers to quickly offer access to their developed verification tools. The developed tutorial covers the intermediate verification language Silver and includes examples which the reader can verify right in place. It should be a first example how the described tools can be used to create an interactive tutorial for other verification languages.

Contents

Chapter 1

Introduction

1.1 Motivation

The Viper verification infrastructure [18] is built for permission based reasoning and provides an intermediate verification language (IVL) called Silver [16] , that natively supports an expressive permission model and a set of carefully chosen language constructs that enables an encoding of a wide range of higher-level programming and specification features. The Viper verification infrastructure includes two backend verifiers called Silicon and Carbon.

Since only a technical report [20] of Silver exists, the effort needed to get started with the Silver IVL is very high. A tutorial which explains the basic features and demonstrates how to encode features from high-level languages into Silver would make it more easily accessible to interested people.

In an earlier Bachelor's thesis, a web interface called Tuwin [19], which allows to use various verification tools, has been developed. Such a web-based access simplifies the use of verification tools a lot. A similar tool called Rise4Fun [1] has been developed by Microsoft research.

With the existing web-based access to verification tools, we can create an interactive tutorial for the Silver IVL. Interactive means that the reader of the tutorial is able to write and try some examples right on the tutorial page, which leads to a more hands-on introduction to Silver.

The idea is to provide a framework which allows to easily deploy further tutorials for other tools, which may not be very accessible at the moment. This allows researchers to easily provide access to their tools and improves the visibility and reach among people interested in such tools.

[1] http://rise4fun.com/

1.2 Goals

The core goal of this Bachelor's thesis is to develop an interactive tutorial for the Silver IVL. The following points clarify the tasks related to the core goal:

- Create an interactive tutorial for the Silver IVL. The tutorial is a combination of a normal tutorial that explains the features of Silver and an online verification tool, with which the reader can try examples themselves.
- The tutorial should address an audience that has some general experience in software verification, but not necessarily in permission-based reasoning, and is interested in learning the advantages of the Silver IVL.
- The tutorial should be rich in examples and tasks the reader can try themselves with the online verification tool.
- The tool used for online verification has very high latency in its current version. The performance of this tool has to be improved such that the response time becomes acceptable for an interactive tutorial.

1.3 Outline

The rest of the report is structured as follows: In chapter 2, the report states the initial situation, clarifies what is already in place and which issues need to be addressed. Chapter 3 gives details about the implementation of changes to the server, tool hoster and user interface. Chapter 4 explains some of the important design decisions and shows which aspects have influenced the design of Viper Online. Chapter 5 covers the tutorial part of the project. It first describes the technical implementation of the presented framework and also gives details about design decisions. In the third section of chapter 5, the developed Silver tutorial is discussed in detail. Chapter 6 offers a guide to deploy a new tool hoster and an interactive tutorial. Chapter 7 offers details about documentation, testing and compatibility aspects of this project. In the last chapter, the conclusion gets presented and some future work is proposed.

Chapter 2

Initial Situation

As the thesis builds partially upon an already existing tool, this section intends to make clear which parts already were in place, what the shortcomings were and which improvements needed to be addressed. It also should establish a basic understanding of the architecture and functionality of the tool.

2.1 Viper Online

The online command-line tool was developed as a Bachelor's thesis in 2012 and introduced under the name of "Tuwin" [19]. In the process of further development the tool was renamed to "Viper Online" because it is intended to be used with the tools of the Viper verification infrastructure. In the rest of the report, the tool will be referred to by the name Viper Online.

All described functions and details of Viper Online are specific to this project. This report does not include a full documentation of the initial Viper Online tool. Readers interested in details not mentioned in this report are invited to have a look at the initial report [19].

2.1.1 Functionality

Viper Online has three components: the server, the tool hoster and the user interface.

The server is the main component of Viper Online. The server essentially provides the user interface for the clients, manages the different tool hosters, forwards the user input to the correct tool hoster and redirects the received results back to the user. It basically establishes the connection between the user and the requested tool hoster.

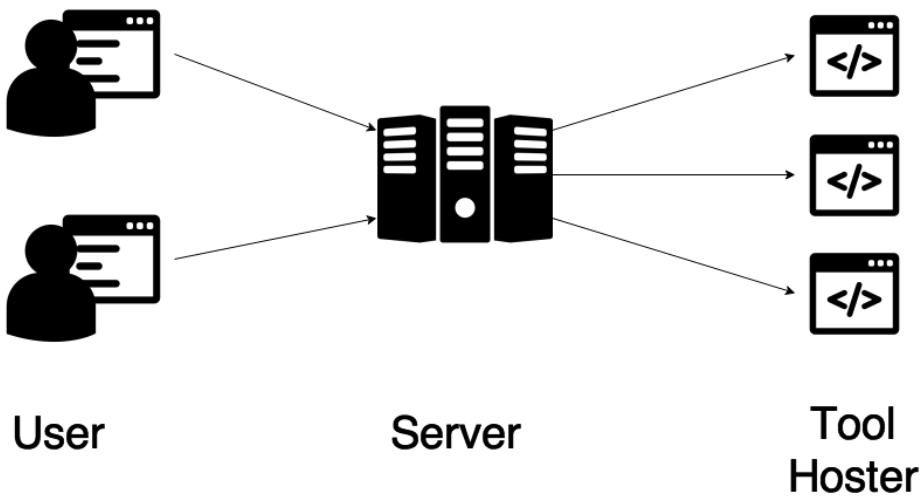

Figure 2.1: Viper Online overview

The tool hoster is a web application a developer installs on a web server and configures it to run the tool they want to make accessible. The tool hoster accepts the input from the server, starts the tool with the received input and sends back the results generated from the tool.

The user interface consists mainly of an online code editor. After selecting one of the available tools, the user can write code to be verified right in the browser. The user interface also offers command-line options, which can be sent along with the input, and provides some examples the user can run.

2.1.2 Tutorial

The tutorial, which is introduced in this project, uses Viper Online to verify code snippets presented in the tutorial's content. The Viper Online user interface can be used to let the user run given examples or serves as an verifier for exercises proposed by the tutorial.

Either way, Viper Online needs to be embedded into the tutorial. Figure 2.2 can be seen as extension of figure 2.1 and illustrates how the Viper Online user interface is used in the interactive tutorial.

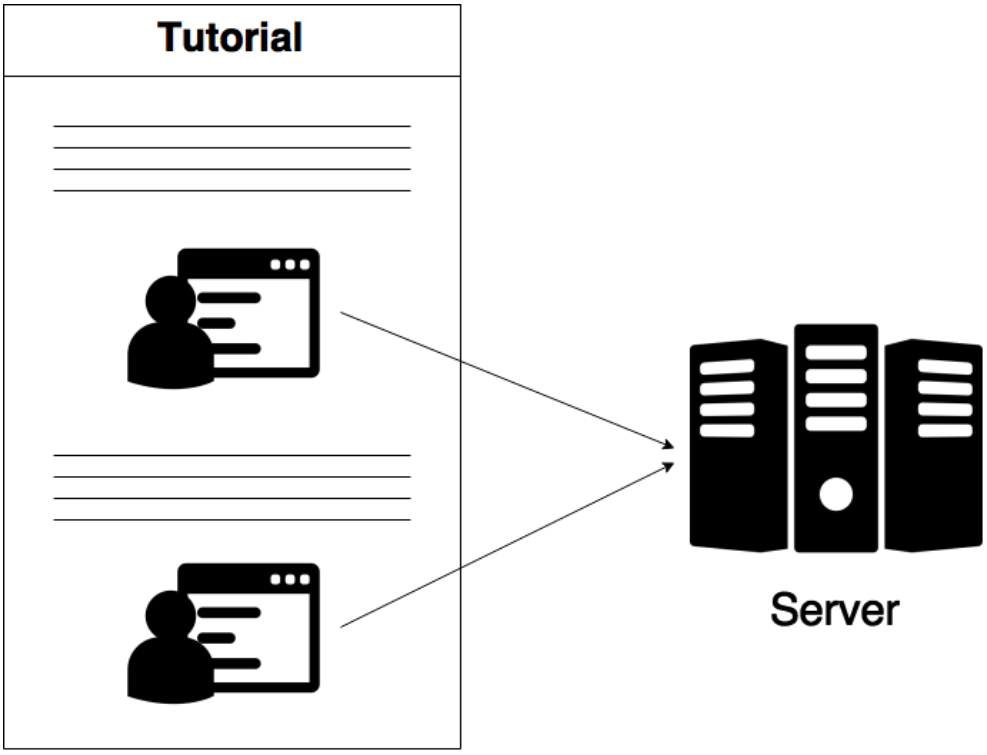

Figure 2.2: The Viper Online user interface embedded in a tutorial

2.1.3 Issues

The tool, as it was, struggled with certain issues or was missing features crucial for developing an interactive tutorial. The following list clarifies what the tool was missing in order to serve as the basis for an interactive tutorial.

Latency

The tool's response time was not satisfying at all. Running simple examples took too long. To verify a simple Chalice [1] code snippet took the tool between 15 and 45 seconds. Deeper analysis of the runtime showed that there were two main reasons for this high latency:

- Two JVMs started for each request
- Communication based on server-side polling with increasing intervals

Figure 2.3 illustrates the response time of some selected Chalice examples with the initial implementation of Viper Online. It is important to mention

[1] Chalice was the first tool to be installed on the initial implementation of Viper Online. Chalice is a language and program verifier for concurrent programs. http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=LeinoMuellerSmans09.pdf

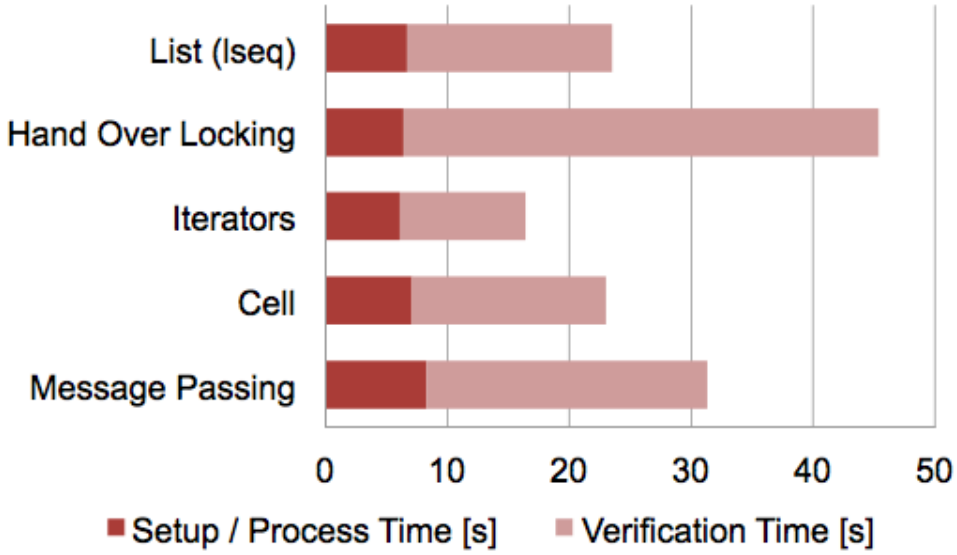

Figure 2.3: Viper Online response time of Chalice examples before project

that the time labeled with "Verification Time" is not the actual verification time, measured by the tool, but the verification time measured by the server. More detailed analyses of the runtime have shown that the actual verification time is much shorter. "Setup / Process Time" includes everything from the user interface to the server and back.

Generally it is obvious that the response time is not optimal. The overall process time, meaning everything which is not actual verification time, should be less than 2 seconds.

Multiple User-Interfaces

A tutorial with code snippets to be verified requires the possibility to have multiple instances of the user-interface on the same web page. The initial tool was not built with this requirement and therefore did not support it.

Cross-Origin Resource Sharing [5]

The user interface was intended to run only on the same web server as the server itself. This would have implicated that every tutorial, based on Viper Online, needs to be hosted under the same URL as Viper Online itself. As the user interface is implemented in Javascript and HTML, it would be easy to move it to another location. Because of lacking support for cross-origin requests, this was initially not possible.

Design

The design of Viper Online was requested to be changed. It needed a cleaner look and should be accessible on all modern web-browsers.

Code Editor

The underlying Javascript code editor is the main aspect of the user interface. CodeMirror [3] was initially used to display the code on the user interface. Extending CodeMirror with additional language specification, in order to get a proper syntax highlighting, is cumbersome. Therefore, the code editor should be updated with the best available alternative.

Chapter 3

Implementation

The implementation of the changes to Viper Online described in the following sections is based on the report of the initial development of Viper Online. This chapter does not cover implementation details that were not changed with respect to the original tool. For the sake of understandability, some overlap with the initial report is intended.

3.1 Server

The server is implemented in Scala and makes use of the Scalatra web-framework [15]. The server contains two servlets: the *GUIServlet* and *CommunicationServlet*. The *GUIServlet* is responsible for the user interface. It provides the start, tool and admin page. The *CommunicationServlet* is responsible for all requests from the user interface and the tool hoster. This servlet is the backbone of the communication protocol.

For each new request to run a tool, a session gets created on the server. The *Session* class assembles the options and the input, logs the request to the database, forwards the request to the tool hoster, waits until results arrive, logs the results to the database and then forwards it to the user interface. As mentioned in section 2.1.3 about response time issues, the communication between server and tool hoster was initially based on polling. As polling always introduces a delay, it does not contribute to an fast overall response time of Viper Online. Furthermore, the tool hoster has to answer every single poll from the server. With multiple run-requests executing at one time, this has an impact on the performance of the tool hoster and the network utilization.

The polling was replaced by an addition to the communication protocol. As soon as results are available at the tool hoster, it sends a HTTP request, as a notification that the results are available, to the server. The server can

now immediately fetch the results from the tool hoster. Polling between the server and the tool hoster is not completely removed. In order to check whether the tool hoster is still alive the server polls the tool hoster at large intervals. This polling has only a minimal impact on performance and network utilization.

The new communication protocol between the server and the tool hoster is illustrated by figure 3.1.

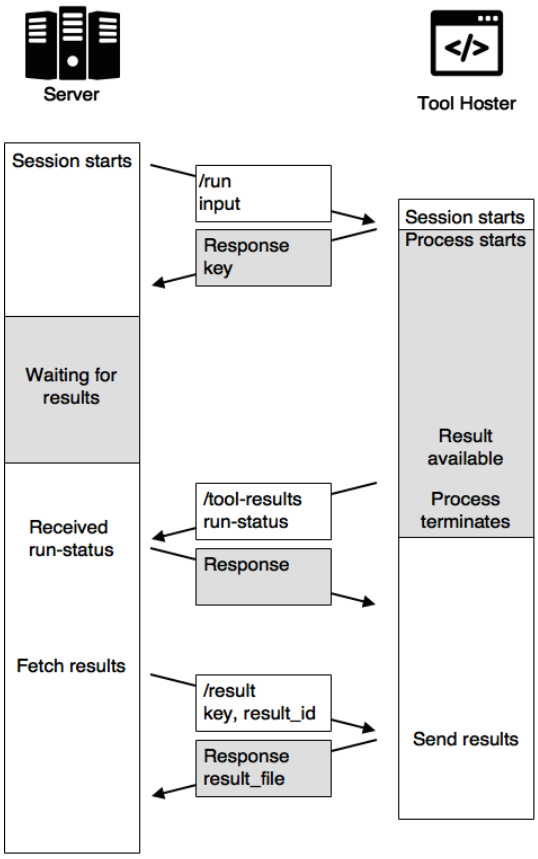

Figure 3.1: Server - Tool hoster communication protocol

3.2 Tool Hoster

The tool hoster is also implemented in Scala and based on the Scalatra web-framework. The servlet *ToolServlet* is the only servlet of the tool hoster and handles all HTTP requests. Like the server, the tool hoster creates a session for each incoming run-request. The *Session* class builds the input to the tool,

runs the tool on the user-specified input in a separate process and notifies the server as soon as the results are available.

Multiple Backends

In the Viper verification infrastructure the Silver language can be verified by two different verifiers, namely, Silicon and Carbon. To make it easy to switch between the verifiers for the same code snippet, the support for multiple backends per tool has been added to Viper Online. Each backend consists of its own command (i.e. tool to be executed), tool options, examples and result files. With each run-request, the tool hoster receives an identifier determining the backend the user has chosen. Based on the identifier, the backend-specific parameters get loaded and the right command gets executed.

Output parsing

In order to provide visual feedback of verification errors to the user in the user interface (e.g. code annotations), there needs to be a standard format of tool output. The initial implementation of Viper Online introduced therefore the *result_type: text/table* which is essentially a JSON-encoded string with the format listed in listing 3.1.

```
{
  "prolog" : "This is the optional prolog message.",
  "epilog" : "This is the optional epilog message.",
  "lines" : [
    {
      "line_nr" : 8,
      "column_nr" : 12,
      "icon" : "w",
      "message" : "The verification error message."
    }
  ]
}

```

Listing 3.1: text/table format

It is the tool hoster's responsibility to provide such a result file. The initial implementation of Viper Online suggested to use an adapter which runs the tool and matches the input against a regular expression. As this adapter was implemented in Scala it had to be started within a JVM for each request. This additional startup of a JVM for each run-request is additional overhead and greatly impacts the runtime. The new implementation transfers the

output parsing into the tool hoster itself. The class *TableBuilder* implements a simple, configureable lexer that produces a result formatted like shown in listing 3.1.

Wakeup Call

The first run-request for a tool often suffers a large delay because the operation system hasn't loaded the tool into memory yet. To reduce this delay, the possibility to specify a wakeup call has been added to the tool hoster. A HTTP request *GET tool/wakeup* to the tool hoster triggers the execution of all *wakeup_commands* defined in the tool hoster's configuration file.

The Viper Online user interface asynchronously performs such a request on initialization. Therefore, the runtime of the first run-request is reduced. Nevertheless, first run-requests can still have a longer runtime as subsequent ones. This is due to further runtime optimization done by the operating system of the tool hoster.

3.2.1 Configuration

As the tool hoster did undergo many changes, also the configuration file had to be adapted. The most obvious change is the support for multiple backends per tool. All backend-specific configurations are now summarized in an element of a list *backends*. As not all properties have changed since the initial development of Viper Online, the description of the properties is partially identical to the first report [19]. Nevertheless they are included for completeness. The configuration file consists of the following properties:

name	The name of the tool as it is displayed to the user.
description	A short description of the tool. It is displayed on the tool selection page as well as above the code input area on the editor page. Restricted HTML markup is allowed.
language	Optional. This is used for syntax highlighting by the Ace javascript editor.
timeout	Optional. Time in seconds after which a running tool is automatically aborted and the run-status' "status" field is set to "timeout". If not set, a default timeout of 5 minutes is used.

max_file_size	Optional. The maximum file size in bytes a result file may have. If a result file exceeds this value it is not sent and a warning is inserted into the run- status. If not set, a default value of 500 MB is used. The default value is also an upper bound for this parameter.
concurrency_limit	Optional. The maximum number of concurrently running sessions. If this limit is reached new requests to execute the tool are rejected until a session terminates.
hidden	Optional. A Boolean value. If set to true the tool will not show up on the server's tool selection page but can otherwise be fully used.
backends	List containing all configurations specific to a backend.
	Backend specific
backend_tag	Unique identifier of the backend.
name	Name of the backend. Gets displayed to the user
description	Short description of the backend.
author	Optional. The name of the developer or organization.
version	Optional. The version number of the tool.

command	The command or path to the tool including command-line parameters. The following placeholders may be used: • $input$: The input file. • $outputX$: The Xth result file (zero-based). • $options$: The parameter string given by the options. • dir: The session directory. • $configdir$: The directory where the configuration file is located. At least $input$ should be included to tell the tool about the user's input.
wakeup_command	Optional. Executable command which will be called as soon as someone loads the tool in the user interface. The wakeup command intends to load the tool executables into memory again and speed up the first run-request significantly.
fetch_output	Optional. A Boolean value. If set to true the tool hoster will redirect the standard output from the tool into the first result file. This should be set to false or left out if the tool handles file output by itself or if pipes are used to redirect output into the file.
options	Optional. A list of options that the user has when executing the tool. See section 5.2.4 of the initial report for more information.
examples	Optional. A list of example inputs from which the user can choose. See section 5.2.3 of the initial report for more information.
results	Optional. A list of result files the tool can produce. The list should contain at least one entry which is used to store the standard output from the tool. See section 5.2.5 of the initial report for more information.
hidden	Optional. A Boolean value. If set to true the backend will not be selectable in the user interface.

Table 3.1: Tool hoster configuration file parameters

Listing 3.2 shows a possible tool hoster configuration file.

```
{
  "name": "Silver",

  "description": "Silver is a permission-based verification language.",

  "backends" : [
  {
    "backend_tag" : "silicon",

    "name" : "Silicon",

    "description" : "This is the Silicon backend.",

    "command": "[path\to\the\tool] $options$ $input$ > $output1$ 2>&1",

    "wakeup_command" : "[path\to\the\tool] [some specific options] [some input file]",

    "fetch_output": false,

    "hidden": false,

    "options":[
    {
          "name":"IDE Mode",
          "tag":"idemode",
          "type":"boolean",
          "true_value":"--ideMode"
    }
    ],

    "results":[
      {
        "name":"Output",
        "result_type":"text/table",
        "file_name":"table-output.txt"
      },{
        "name":"Original Output",
        "result_type":"text/plain",
```

```
39          "file_name":"output.txt",
40        }
41      ],
42
43      "examples":[
44        {"name":"Arithmetic", "file":"..\examples\
    arithmetic.sil"}
45      ]
46    },{
47          //Next backend configuration here
48    }
49    ],
50
51    "directory": "..\\working_dir",
52
53    "language": {
54      "mode" : "silver",
55          "file" : "..\\silver.js"
56    },
57
58    "hidden": false,
59
60    "timeout": 600,
61
62    "max_file_size": 4194304,
63
64    "concurrency_limit": 15
65  }
```

Listing 3.2: tool hoster configuration file

3.3 User Interface

The user interface consists of the three web pages provided by the *GUIServlet* and the code input to edit and run the code. The main component is the code input section. The code input is implemented in Javascript and consists mainly of the file *viper_online.js*. The Javascript framework jQuery [10] is also used and simplifies many aspects of the implementation. Besides the stylesheets associated with the code input, all functionality is in *viper_online.js*. The portability of this single Javascript file allows to include Viper Online code inputs on every webpage. The following sections explain the implementation of the crucial parts of the code input.

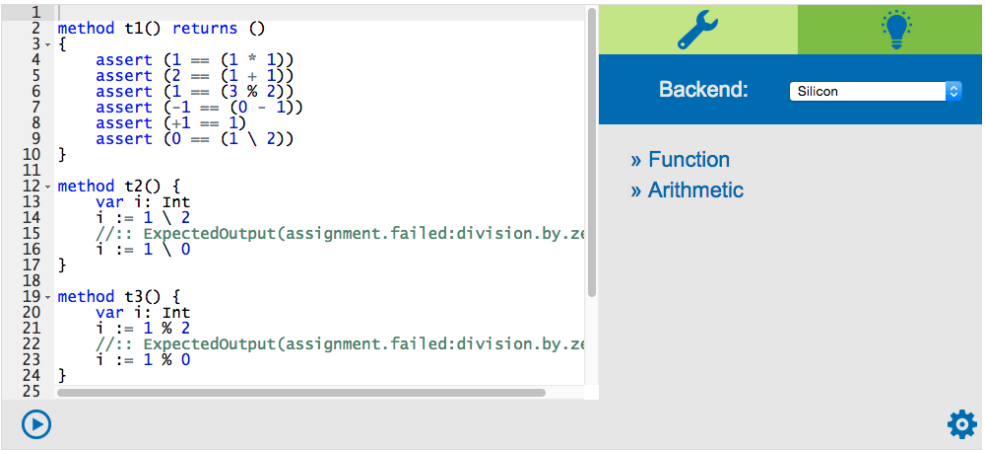

Figure 3.2: User interface code input

3.3.1 Configuration

In order to successfully integrate the Viper Online code input into different web-pages, with different purposes, the user interface offers different configuration parameters which can be set. All parameters are located in a specially marked section at the top of the file *viper_online.js*. Table 3.2 lists all parameters.

serverURL	String. The URL of the Viper Online server. (e.g. http://pm.inf.ethz.ch/viperonline)
tool	String. The tool identifier. Must be available under the specified serverURL.
language	String. The language identifier. This is used to load the language definition file.
enableMultipleEditors	Boolean. Used to specify whether more than one code input instance is to be displayed on the same web page (e.g. true for a tutorial).
resultBarFixedHeight	String. A HTML height measurement to indicate a fixed height of the result bar. Depending on the tool output, the result can be very long. This parameter is used to set a fixed height and scrollbar on overflow. (e.g. "300px"). The height gets set to "auto" if it is an empty string.

showOptions	Boolean. Allow the user to see and select the tool options offered by the backends.
showExamples	Boolean. Allow the user to see and select the tool examples offered by the backends.
loadRandomExample	Boolean. Set to true if a random example should be loaded into the code input on initialization.
initialSideBar	String. Defines the sidebar which is displayed to the user on initialization. • "settings" ≡ tool settings • "examples" ≡ tool examples • "" [empty string] ≡ sidebar closed
enablePermalinks	Boolean. Defines whether a button to display a permalink is displayed to the user.
enableLoggedSessions	Boolean. Defines whether a user is allowed to access a logged session. Logged sessions can be accessed by opening a valid permalink.
maxLines	Integer. The maximum of lines the editor will display. A scrollbar will be displayed on overflow.
minLines	Integer. The minimum of lines the editor will display. Set it to same number as *maxLines* and the height of the code input is fixed. It is not recommended to set this < 15, the design might break otherwise.
fontFamily	String. A specific font can be chosen for the code input. Only monospaced fonts allowed, otherwise the code input suffers unexpected behaviour.
fontSize	String. Font size of the code input. (e.g. "12pt")

Table 3.2: Viper Online user interface parameters

3.3.2 Multiple code input instances

As Viper Online is to be used in tutorials with multiple instances of the code input on one page, the user interface needed to be improved accordingly. In order to distinguish every code input from one another, it was necessary to give them unique identifiers. Those identifiers are needed on the one side to determine which results coming from the server belong to which code input. On the other hand, they are needed to identify user interactions with the code input, e.g. a user toggles the sidebar on one code input, then only this exact sidebar will be minimized. Multiple code input instances also offer the flexibility to try the different backends or run the same code with different parameters side-by-side.

The key to supporting multiple code input instances are 3-digit unique identifiers. On initialization of Viper Online every HTML div element ( ) with class *editor* gets assigned a unique id and the code input gets inserted in this div element. Every component of the code input (e.g. the *sidebar*, the *resultbar*, the *bottombar*, the *editor*) is accessible with the unique identifier, called the *editorNr*.

Also for the assignment of incoming result files to the correct code input the *editorNr* is important. As the user decides to run a code snippet, the method *run* makes an HTTP request with the input and tool options to the server. As response, the user interface gets a *sessionKey* which is used to identify the run-request. The user interface now links the *editorNr* to the *sessionKey*. As soon as results are coming in from the server, the correct code input can be identified with the *sessionKey*. This allows of course to have multiple open sessions with the server at once and the results are displayed next to the correct code input.

3.3.3 Editor and language specification

The editor is the main component of the code input. In Viper Online the Javascript code editor Ace [1] is used. Ace offers an extensive API, is well documented and widely adopted. It is very easy to embed Ace into any site and it offers many parameters to adjust it. Listing 3.3 shows how Ace is embedded into Viper Online.

```
<!-- Some html -->
  <div id="editor"></div>
<!-- Some html -->
<script type="text/javascript">
  var editor = ace.edit(editor);
  editor.setTheme("ace/theme/textmate");
  editor.getSession().setMode("ace/mode/language");
  editor.setOptions({
      maxLines: 25,
      minLines: 25
  });
</script>
```

Listing 3.3: Embedding Ace

For syntax highlighting, Ace comes with many predefined languages. The TextMate/Sublime *.tmlanguage* language specification file format is widely used and can be imported into Ace to enable syntax highlighting and code folding. The language specification is one Javascript file, which has to be provided by the tool hoster. On initialization of the code input, the correct language file gets loaded through the server and the mode is then available to Ace.

For more information how to create such a language specification file, see section 6.1.

3.4 Implementation Draft: Caching

Server-side caching of results from the tool hoster has been discussed in the course of this project. Due to time reasons, it has not been implemented. This section provides a draft for a possible implementation in the future.

It is important that the user knows whether the received result has been delivered from the cache or not. Especially for program verification it is possible that the user does not want to have cached results. Therefore, the user must be able to request an actual verification and not a result from the cache.

On the other side, in the environment of a tutorial, in which most users are very likely to run the exact same input, caching can speed up the response time significantly.

The following sections provide the technical details of how caching could be implemented.

Database

To store the hash of the input, an additional table in the database is required. The table *result_cache* could have the following layout:

id	Primary key. The input cache's unique id.
session_id	Foreign key. The session's unique id.
tool	The tool name of this session.
input_hash	The hash of the input.
timestamp	Time of hashing.

Table 3.3: layout of table *result_cache*

Process

On any run-request, the server would first check whether the user enforces the input to be verified or whether a result from the cache is acceptable. In case a result from the cache is acceptable, the server hashes the input and queries the database for an entry with the same tool and this exact hash string. Which hash function to use is not crucial, as the hash function doesn't need to be cryptographically secure and the possibility that two different, syntactically correct programs generate the same hash, is negligible. Therefore, *md5* or *sha* would both be fine.

In case the database does have an entry with the same input hash, the communication with the tool hoster can be skipped. The server can request the *session_key* from the database and then use the method *getLoggedSession* to retrieve the session with the results.

The *run-status* can be updated with the available results right away. As soon as the user interface requests the updated the *run-status*, it begins to fetch the result files.

The method *getResultFile* in the class *Session* has to be modified such that if the results are coming from another session, then it does load them from the correct *sessiondirectory*. As the directories are named with the the tool tag and the session id, it is no problem to find the correct directory, because both values have been queried from the database.

It is important to include the information that the results are coming from the cache, including the timestamp. As the result gets transferred as plain text, without any special format, it is not suitable to include caching infor-

mation. The better solution would be to include this information in the *run-status* which is JSON-encoded.

In case the input hash cannot be found in the database, the server proceeds normally and lets the tool hoster run the input. As soon as the tool hoster has finished and the *run-status* has status *finished*, meaning that the execution did terminate normally, the server creates a new entry of this session in the table *result_cache*.

Also in case a cached result would be available, but the user enforces verification, the database entry should get updated with the new result.

no_cache option

To offer the user the possibility to enforce a verification of the code input, it would be necessary to add a option *no_cache* to the list of options of every tool. This option can be handled separately on the server and also be removed before forwarding the run-request to the tool hoster. This implementation would have the advantage that no modification to the user interface is necessary, in order to give the user the control over caching.

Flush cache

There has to be a possibility to flush the cache. As the cache is only built around the database table *result_cache*, it is sufficient to delete the entries in this table.

In case a tool hoster updates the tool, it is necessary to flush all cache entries belonging to that tool. Therefore, the communication protocol between the server and the tool hoster has to be extended with an */tool/flush_cache* request.

This request can be implemented very similar to the */tool/reload* request, which enforces the server to reload the tool hoster information.

Similar to the reload-request, the secret key of the tool hoster has to be included, to prevent unauthorized flush-requests.

Chapter 4

Design

As Viper Online is intended to make tools of the Viper verification infrastructure available to a broader audience, it is important that the appearance is modern and appealing. Furthermore, to underline the affiliation to ETH Zürich, the design is loosely based on the corporate design of ETH Zürich [4].

The sections in this chapter give information about some of the design decisions taken in this project.

4.1 Viper Online

Basic Design

The corporate design of ETH Zürich has some characteristics that were used for the design of Viper Online. The site-wide, blue header in combination with the green rectangle at the bottom is an eye-catcher of the ETH web appearance. This iconic feature has been transferred to the Viper Online design.

Figure 4.1: Viper Online header

The overview of the available tools on the start page has to be simple and offer information about the tools. The initial implementation of Viper Online displayed large clickable blocks. Inside the blocks, the tool description

was displayed. Such blocks offer an intuitive way to navigate to the tools. Additionally they are suitable for touch-screen devices. Therefore, the block design has been taken over into the new design. But the blocks have been redesigned. As the model is very clean, with clear lines and no rounded borders, also the blocks need to be as simple as possible.

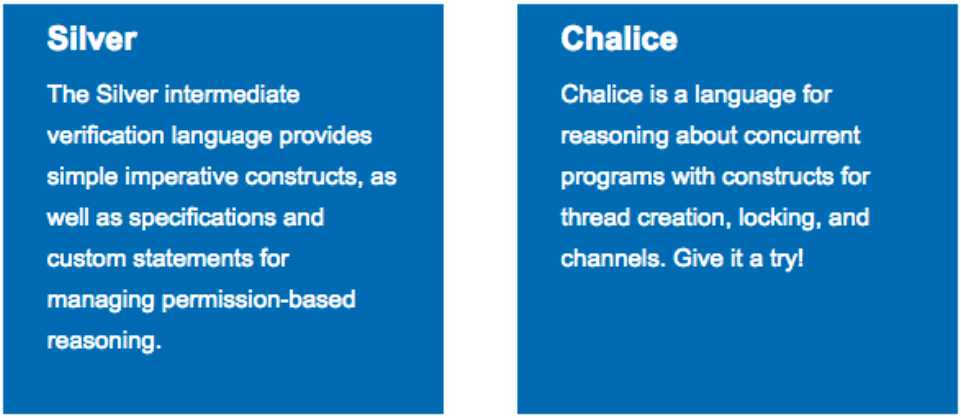

Figure 4.2: Viper Online tool overview

Code Input

The code input is the most important part of Viper Online. Furthermore, as it is going to be used on different web pages, the code input needs to be an independent unit. For the tutorial it has to be very space efficient and minimalistic, as it is displayed many times on the same page.

The code input is divided into 4 sections around the editor. Every section accommodates another kind of information. Figure 4.3 illustrates all sections.

The initial setup consists only of the *editor* and the *bottombar*. As the *editor* is the most important part of the code input, it has to be placed most prominently and with enough space. The *bottombar* cannot be hidden and should therefore not take up too much space and the colour should not be to intrusive. This setup gives a really clean look and focuses on the code in the editor.

The *sidebar* needs to display too much information to show it to the user permanently. It would take away too much of the screen and distract attention from the editor. Therefore, the *sidebar* is displayed by a click on the settings button on the *bottombar*.

The *sidebar* has the task to display two different kind of information: the tool options and the tool examples. Having both in the same *sidebar* would be too cluttered. Therefore, a menu is displayed on top of the *sidebar*, allowing the

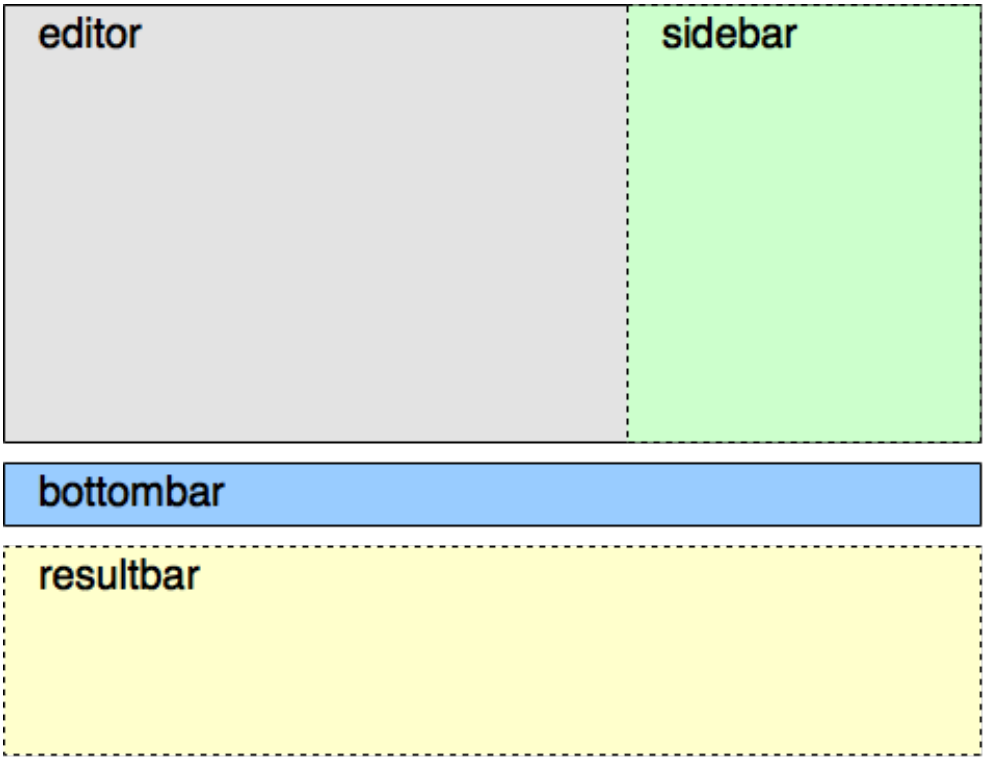

Figure 4.3: Modular code input

user to switch between the different *sidebars*. If the user has a big enough screen, the sidebar can be kept open permanently. As the tools can offer multiple backends and the tool settings and examples are backend-specific, right under the menu the user can choose the backend. If the user chooses another backend, the content of the *sidebar* gets reloaded.

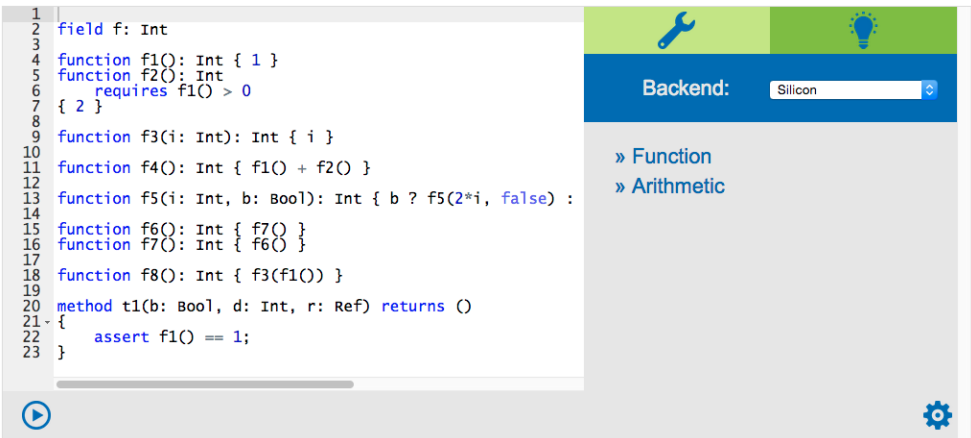

Figure 4.4: Editor with open sidebar

The *resultbar* is initially never displayed and can be shown only if a result is available. In case the tool hoster delivers the results in the format *text/table*, the user interface can automatically read possible verification errors and display them in the editor as annotations to the line numbers and visual feedback in the *bottombar*. This has the advantage that the user has all the important feedback without opening the *resultbar*.

Figure 4.5 shows the result of an execution returning verification errors. It is clearly visible that the execution was not successful and the lines containing an error are annotated with an icon. On hovering over this icon, the error message is shown.

If the execution is successful, the user gets a green visual feedback in the *bottombar*.

The user can view all details about the execution on selecting the "Details" button in the *bottombar*.

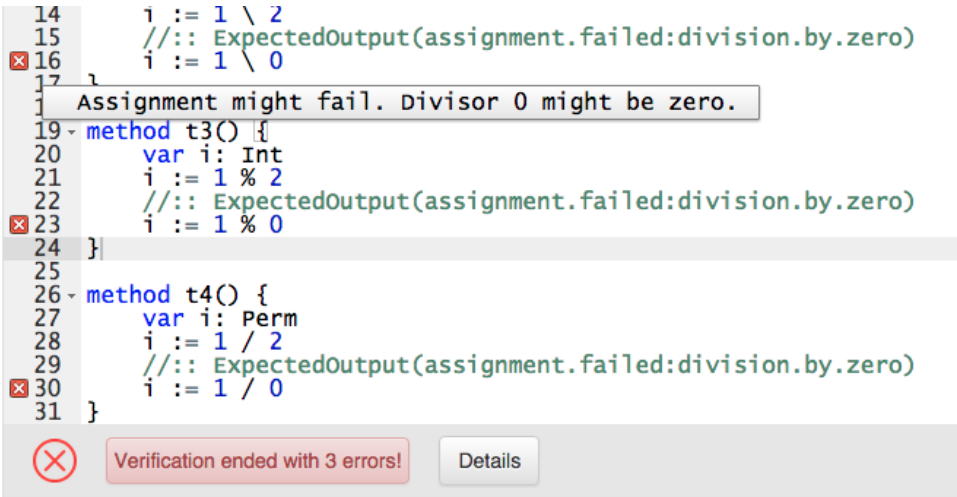

Figure 4.5: Unsuccessful execution

In case the user interface does not receive the result in the format *text/table*, the user interface does not know whether the verification was successful or not. Therefore, the *resultbar* is automatically expanded. Otherwise, only if the user selects the button "Details", the *resultbar* is displayed.

To accommodate all results, the different result files are shown as tabs. Depending on the tool and backend, there might be different numbers of tabs. The *resultbar* is designed very minimalistic, in order to blend in with the rest of the code input.

Figure 4.6: Open resultbar

Used libraries

Some parts of the design are achieved with the help of libraries. All the animations that are used (e.g. toggle, hide, slide) are part of jQuery [10]. jQuery offers a simple API to animate almost every aspect of a web page. The other library used is Twitter Bootstrap [2], which is mostly a CSS framework, powered with some Javascript. It offers easy to use, but very nice looking buttons, tabs, tables and more.

4.2 Mobile Devices

Viper Online is optimized for mobile devices. Although mobile devices are not very well suited to edit the code in the code input, it is possible to run the offered examples or open a logged session with a permalink.

If the device's screen width is smaller than a defined width, the header gets centred and the tool blocks are stacked above another on the start page. Also, the code input changes the width of the sidebar, depending on the device's screen width. Additionally, the *bottombar*'s height is increased in order to have enough space for the result notifications.

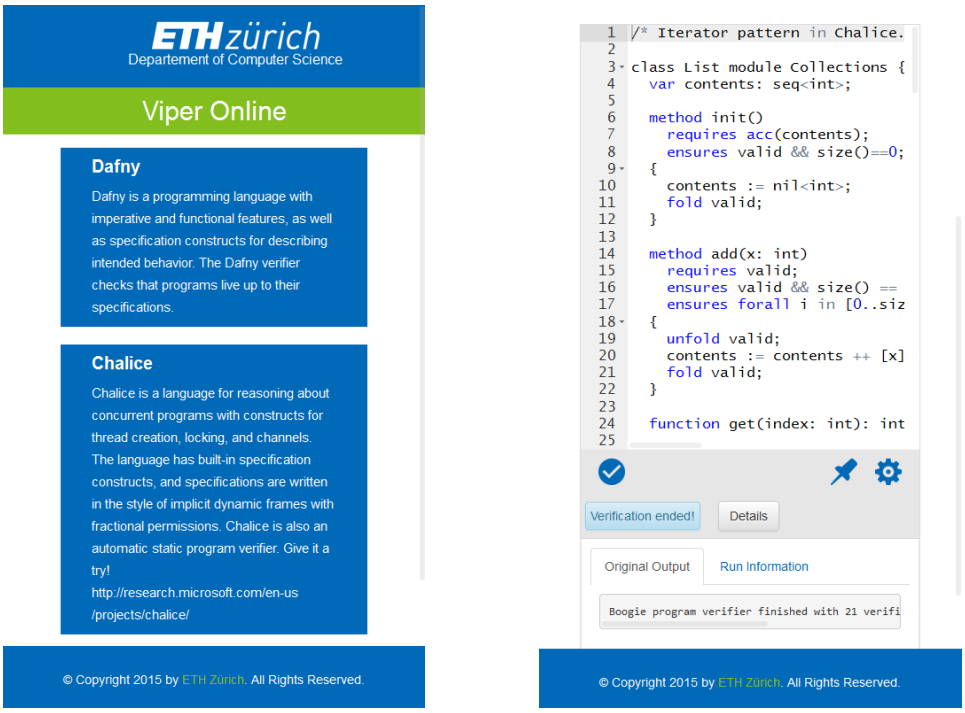

Figure 4.7: Viper Online on a mobile device

Chapter 5

Tutorial

This chapter covers the tutorial part of the project. First the implementation of the framework to easily create interactive tutorials based on Viper Online and then the created Silver tutorial.

5.1 Implementation

The tutorial uses Viper Online to realise the interactivity, meaning the possibility to run examples right on the page. The effort needed to create a tutorial should mainly be spent on creating the actual content and not on the design, the implementation or the setup. Therefore, the goal was to implement an automated workflow, which allows to easily create a new interactive tutorial and change existing ones. The following sections show the main parts of implementing this workflow.

5.1.1 Markdown

Writing the tutorial from scratch in HTML was not an option, as it would be to cumbersome and distracting. The solution is the markup language Markdown [12]. Markdown is a simple markup language. It features a very simple syntax, which allows produce web content in an intuitive and fast way. Markdown is a widely accepted markup language to write web content. For example, GitHub has adopted Markdown to let users write documentation. GitHub even extended the original Markdown syntax and created GitHub Flavored Markdown [12] , which offers additional functionality.

The GitHub Flavored Markdown is used to write the content of the interactive tutorial. There are many editors suitable for this task (for example, StackEdit [17]). Most important is the support for embedding code snippets. The syntax allows to insert whole code blocks with specifying the language or also just small inline code.

Additionally, some of the elements can be attributed with class and id tags that get translated into HTML. The tutorial makes use of that feature to annotate runnable code blocks with class *runnable* to later identify those code snippets that can be transformed into a Viper Online user interface. Listing 5.1 shows the Markdown syntax to embed code into the tutorial.

```
Markdown offers the possibility to
have small `<b>inline</b>` code blocks.

```java
class foo {
  //I cannot get executed in the tutorial
}
```

```silver {.runnable}
// Run me!
method t1() returns ()
{
    assert (1 == (1 * 1))
    assert (2 == (1 + 1))
    assert (1 == (3 % 2))
    assert (-1 == (0 - 1))
    assert (+1 == 1)
    assert (0 == (1 \ 2))
}
```
```

Listing 5.1: Markdown code blocks

On loading the HTML converted from Listing 5.1 into the tutorial, all code blocks will be styled as code and the syntax will be highlighted according to the language specified at the top of the code block. Only the last code block in Listing 5.1 will show a button to transform it to a Viper Online user interface and offer the possibility to run it in the browser. This is because it has the class attribute *runnable*.

Figure 5.1 shows the web page of listing 5.1. Only the second code snippet has a button, on the bottom right, to convert it to an Viper Online user interface.

5.1.2 Custom Elements

Many tutorials feature special content blocks to highlight very important content, more detailed information or hands-on exercises, that are visually

Markdown offers the possibility to have small `<b>inline</b>` code blocks.

```
class foo {
  //I cannot get executed in the tutorial
}
```

```
// Run me!
method t1() returns ()
{
    assert (1 == (1 * 1))
    assert (2 == (1 + 1))
    assert (1 == (3 % 2))
    assert (-1 == (0 - 1))
    assert (+1 == 1)
    assert (0 == (1 \ 2))
}
```

Figure 5.1: Markdown from listing 5.1 as converted code snippets

highlighted. Unfortunately, the possibilities to control the styling options in Markdown are limited. The class and id attributes that can be added to code blocks cannot be extended to normal text passages. Therefore, the automated workflow offers additional syntax to mark special content. Listing 5.2 shows the syntax to mark special passages of the tutorial.

```
...
//important//
This is important content that is crucial in
   understanding the concepts of the tutorial!
///
...
//further_reading//
In the library, you'll find more information about
   this topic.
///
...
```

Listing 5.2: Special elements

The passage between *//some_style//* and *///* will be wrapped in a HTML div element. This div element will be styled according to the stylesheet of the

tutorial. Those custom elements allow a fast and easy way to control the style of whole text passages and to give the reader a visual representation of different kinds of content. Supported styles are:

- further_study
- important
- warning
- exercise

Those styles are an initial set of possible styles. It is easy to register new styles or to alter the look of existing styles. More information about defining new styles is in section 6.2.

5.1.3 Automated Workflow

The goal was to implement an automated workflow which makes it easy to create and change an interactive tutorial. This workflow is implemented by providing a framework for the tutorial in which one only has to plug in the actual content, meaning the HTML converted from Markdown and optional pictures used in the tutorial.

The framework basically consists of a folder which can be placed on any web server. The folder contains all HTML files, all stylesheet files, the Viper Online javascript file and all needed javascript libraries. The content of the tutorial is placed as *tutorial.html* in the root of the folder.

In case the tutorial is to be divided over multiple web pages, the *index.html*, which is the entry point of the tutorial, has to be duplicated. To load the correct tutorial content into the pages, the configuration parameter *tutorialFile* has to be set differently on every page. Further information about multipage tutorials is in section 6.2.

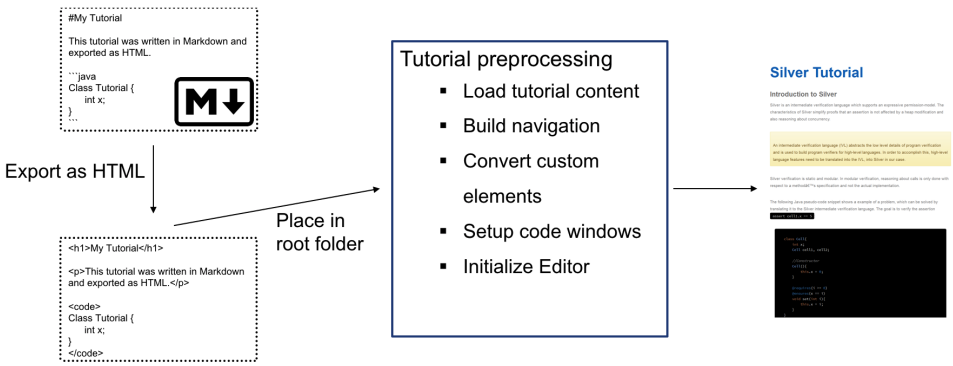

Figure 5.2: Automated workflow

The javascript file *tutorial.js* is the main component of the framework. On accessing the page *index.html* (or any copy of it on multipage tutorials) through a browser, *tutorial.js* is loaded and initializes the tutorial. The initialization process performs following steps:

- The content of *tutorialFile* gets loaded into the currently open document.
- All custom elements, described in section 5.1.2, get parsed and the correct styles are applied.
- All *<h1>* and *<h2>* elements in the tutorial are selected and being used to automatically build a simple navigation.
- All elements with class *runnable* are selected and registered for a possible transformation into a Viper Online interface. A button to start this transformation is added to the code snippets.

After the initialization, the user is displayed a fully functional, interactive tutorial.

Figure 5.2 illustrates the automated workflow described in this section. A complete guide to setup a tutorial can be found in section 6.2.

5.2 Design

The tutorial's design is based on a responsive HTML5 template [8] and has been adapted to fit the tutorial's needs.

Basic Design

The most important design goal in the tutorial was simplicity. The reader should be concentrated on the tutorial content. It is also important that the content has enough space for the code snippets and the Viper Online user interface. The site therefore has only two components: the content and a sidebar for navigation.

As the sidebar is coloured with the same blue as in the Viper Online user interface, the code input fits perfectly into the page.

Code snippets

The various code snippets are displayed on a black background to clearly separate them from the normal tutorial content. The code snippets are also syntax-highlighted, which is achieved by *highlight.js* [7], a simple Javascript library for syntax highlighting. The runnable code snippets have a small button at the bottom right. Clicking this button transforms them into Viper Online user interfaces and loads the code automatically into the editor.

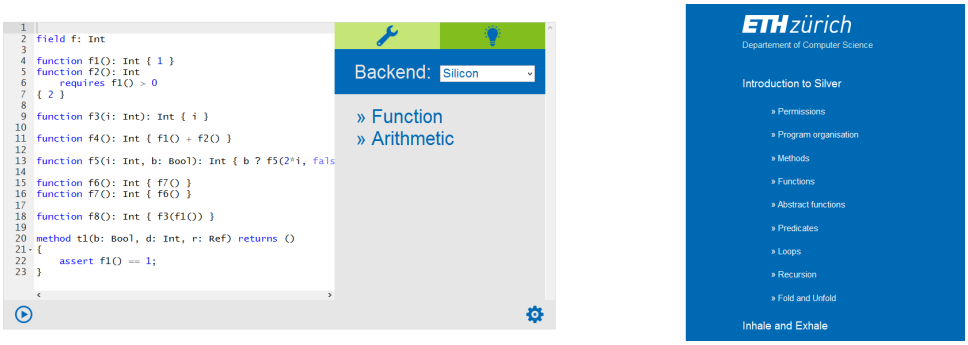

Figure 5.3: Tutorial with code input and sidebar

```
field x : Int

method set(a : Ref, i : Int)
    requires acc(a.x)
    ensures acc(a.x) && a.x == i
{
    a.x := i
}
method main()
{
    var cell1 : Ref
    var cell2 : Ref
    cell1 := new(x)
    cell2 := new(x)
    set(cell1, 5)
    set(cell2, 10)
    assert cell1.x == 5
}
```

Figure 5.4: Code snippet width edit button

Mobile Devices

The tutorial is also optimized for mobile devices. As the underlying template is responsive, it automatically adapts to screen sizes. If the device's screen width is too small, the sidebar gets hidden and the user needs to toggle the navigation.

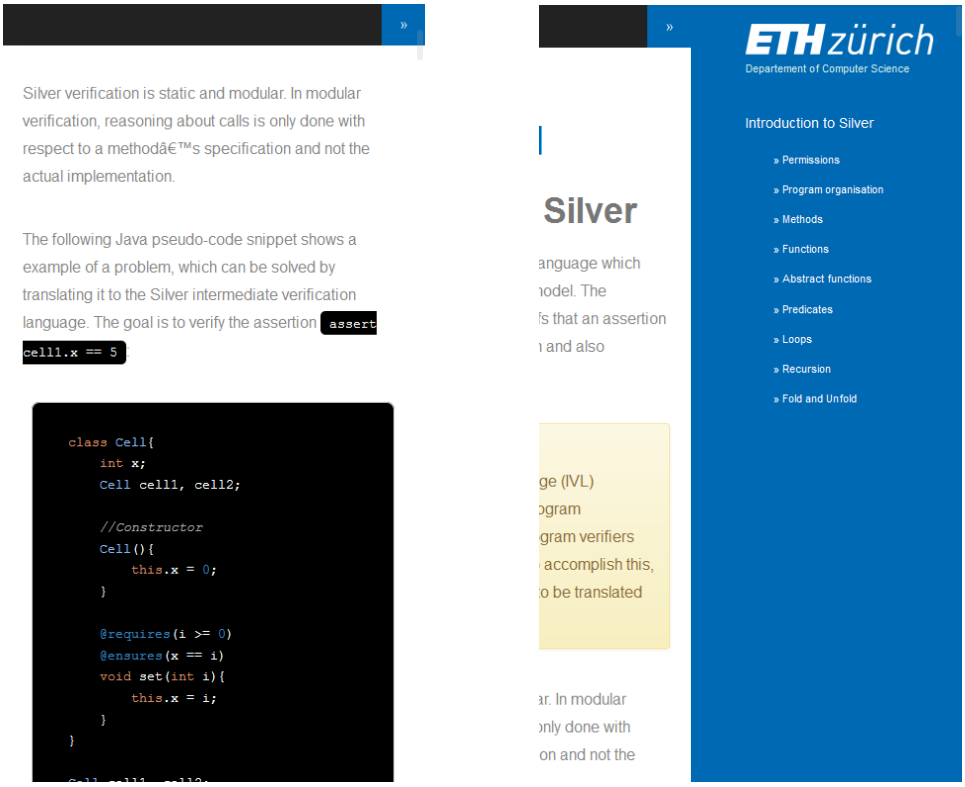

Figure 5.5: Tutorial on mobile device with closed and open navigation

5.3 Silver Tutorial

This section describes the details of the Silver tutorial, created for this project.

5.3.1 Idea

As Silver is currently only documented in the form of test cases and examples, and by a technical report, the effort to get started with Silver is very high. In order to improve visibility of the Viper verification infrastructure among researchers and interested students, it is useful to have a tutorial for Silver.

The tutorial is suitable for readers with knowledge about program verification, but not necessarily about permission-based reasoning. As the possibility to verify code snippets in the browser is given, the tutorial is intended to be example driven, e.g. it is evolving around an example that is gradually extended throughout the tutorial. Besides the example, the reader is invited to actively solve simple exercises. As solution to such exercises can be tested in the browser, immediate feedback is achieved. Overall, the tutorial is intended to offer a hands-on experience and allow the reader to learn about

Silver in a practical way.

5.3.2 Overview

The Silver tutorial consists of five chapters. Chapter 1 is an introduction to the Silver intermediate verification language. The characteristics of Silver are described and the use of Silver is motivated. The motivation is based on a Java pseudo-code example, which introduces the problems that occur when reasoning about concurrent programs. In order to overcome those problems, the pseudo-code example gets extended with accessibility predicates and finally the pseudo-code gets translated into equivalent Silver code. The composition of this first chapter allows the reader to start with something already known and simultaneously motivates the use of Silver.

The second chapter further clarifies the already introduced concept of permissions. The chapter introduces the concept of fractional permissions and permission transfer. This chapter is not specific to Silver, but defines the notion of permissions for the rest of the tutorial.

Chapter 3 is the main part of the tutorial and covers the basic Silver program constructs. It consists of several subchapters. The subchapters include *methods*, *functions*, *predicates*, *recursion* and *fold/unfold* statements. On introducing a new functionality, the subchapters give the general form and then apply it to an ongoing example. The example introduced in Chapter 1 is gradually expanded with the functionalities introduced in the subchapters of Chapter 3.

For example, functions are motivated with the concept of information hiding, extending the ongoing example with a function to hide the presence of a specific field. Similarly, predicates are motivated also with information hiding, but for encapsulating assertions instead of expressions. The ongoing example allows the reader to gradually develop understanding of Silver programs and the presented concepts.

Chapter 4 introduces the statements *inhale* and *exhale*. In order to motivate those statements, the tutorial first presents a Java pseudo-code example including *fork* and *join* statements. As the concept of *fork* and *join* is widely known, the equivalent translation into Silver code with statements *exhale* and *inhale* allows the reader to easily grasp the meaning and function of those statements. In the end of chapter 4, the reader is motivated to apply his new knowledge about *inhale* and *exhale* and model a monitor (for synchronization) with those statements.

The last chapter introduces *assume* and *assert*, which are similar to *inhale* and *exhale*.

5.3.3 Status of Implementation

At the time of writing, the tutorial covers the basic features of Silver. The overall scope of the project and the time restrictions did not allow to develop it further. However, this project introduces a framework, which enables an automated workflow to further extend the tutorial. Instead of focusing only on the Silver tutorial, effort has been put into designing the automated workflow as easy as possible and developing the tutorial framework. This allows to also create other tutorials in a convenient way and with minimal effort. The extension of the Silver tutorial is addressed in section "Future Work".

Chapter 6

Extending Viper Online

The following sections give an overview of how to add a new tool hoster to Viper Online and how to create a new tutorial with the introduced automated workflow. It is not a complete step-by-step guide, but intends to offer some guidelines and explains the most important steps to take.

6.1 Adding a new Tool

This section clarifies how to setup a new tool hoster. It is not mandatory to implement a tool hoster as described in this section; one could use an own implementation of the tool hoster, as long it follows the communication protocol with the server. It is also possible to use another web server.

The described setup is intended to work on a Windows Server 2008. Java version 8 and Scala version 2.11.4 are assumed to be installed on the machine.

Configure the tool hoster

There are a few settings that can be configured in the tool hoster's source code. Depending on the tool to be hosted, the following settings might be important:

- Input file name
- Maximum of concurrent sessions
- Maximum timeout

Those settings can be found in the object *utility.Settings*.

As the user interface needs the results in the format *text/table* to display verification errors to the user, the object *utility.TableBuilder* has to be configured

to parse the output correctly and produce such a result file. The following things need to be configured:

- Set the name of the original output which should be parsed. A tool may produce multiple output files, but only one of them will be parsed.
- Set the file name of the *text/table* file that is produced.
- Set the regular expressions to detect the various errors the tool can produce in the object *Errors*
- Add the defined regular expression(s) to the simple lexer in *processLine*

This step is not mandatory and Viper Online works also without having a result file in the format *text/table*. For an optimal user experience, it is recommended to provide it.

The current *TableBuilder* is a fairly simple and not suited to parse arbitrarily complex error reporting, e.g. multiple files or errors spanning multiple lines. Every developer who wants to add a tool to Viper Online is invited to extend *TableBuilder* to suit the tool's needs.

Build the tool hoster

The next step in adding a new tool hoster is to build a web archive (*.war*), which can be deployed on a jetty web server [9].

The tool hoster is built with sbt [13]. The deployable web archive can be built by starting sbt in the tool hoster project folder and then running command *package*, which packs all necessary file in one web archive.

Deploy

The web archive can be deployed on a Jetty web server. As of the date of writing, Jetty version 9.2 worked perfectly. A copy of the Jetty web server has to be placed in the root folder called *jetty*. The web archive has to be copied to *root/jetty/webapps*. In order to be recognized and started correctly, a XML configuration file has to be included in the *webapps* folder. Listing 6.1 shows an example of the XML file to be included.

```
<?XML version="1.0"  encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN"
    "http://www.eclipse.org/jetty/configure.dtd">

<Configure class="org.eclipse.jetty.webapp.
    WebAppContext">
  <Set name="contextPath">/tool_hoster/[tool]</Set>
  <Set name="war"><SystemProperty name="jetty.home"
    default="."/>
    /webapps/tool_hoster.war
  </Set>

  <Set name="extractWAR">true</Set>
  <Set name="copyWebDir">false</Set>


  <Call name="setInitParameter">
    <Arg>toolConfigFile</Arg>
    <Arg>../config.json</Arg>
  </Call>
</Configure>
```

Listing 6.1: tool_hoster.xml

The setting *contextPath* indicates the full path to the web application. This path can be chosen freely, but it is recommended to indicate the tool that is being hosted by replacing *[tool]* with the actual tool name.

If the tool hoster is not intended to be reachable under the default Jetty port, then the port has to be adjusted. The file *root/jetty/start.ini* provides several settings regarding the run-configuration of jetty. It is possible to change the port in this file.

For the setting *setInitParameter*, it is important to include the tool hoster's configuration file, which is discussed in section 3.2.1.

Configuration

The configuration file *config.js* has to be included in the root of the web server. All important details of the configuration file are discussed in section 3.2.1. Some comments to the configuration:

- The *command* setting is crucial the tool hoster. In case the tool needs to be called in a complex way, it could be useful to create a batch file, which calls the tool instead. This way, it is possible to keep the *command* simpler.

- The *wakeup_command* setting has to be a fully specified, executable command. Meaning that potential input and parameters have to be included. As the result is not relevant, the output can be discarded. It is not mandatory to use this setting.
- The list *results* can have entries that are not produced on every run of the tool. This allows to include result files, which may only be produced under special circumstances (e.g. flags, errors, tool-options)

Besides the configuration file, the *working_dir* folder also needs to be created and placed in the root folder of the tool hoster. All input and output files are going to be saved to this folder.

Language specification

The last important task is to provide the language specification, so that the user interface's editor (Ace) can highlight the syntax. The tool hoster works also without a language file, but the user interface doesn't support syntax highlighting in this case.

The language specification is provided in one Javascript file (e.g. *language.js*) in the root folder of the tool hoster, right along the *config.js* and Jetty folder. It has to be named as it is specified in *config.js* under the setting *language*.

An introduction to generating Ace language modes can be found at [1]. The site also offers a tool called Ace Mode Creator [2] which allows to test self-defined language modes in the browser. This is an ideal starting point to create further language modes, as it also allows to have a look on already defined modes and use them as a basis for new modes.

The Ace editor offers automatic conversion of *.tmlanguage* files to Ace language modes. This is a common format for language specifications from the TextMate/Sublime text editor, and it has been adapted by other editors. This file format can automatically translated into an Ace mode. Further information is online at [3].

Start

After completing all the steps above, the tool hoster is ready to be started. Inside the Jetty folder is *start.jar* which needs to be executed to start the tool hoster.

[1] http://ace.c9.io/#nav=higlighter
[2] http://ace.c9.io/tool/mode_creator.html
[3] https://github.com/ajaxorg/ace/tree/master/tool

Register the tool hoster at the server

As soon as the tool hoster is running, it can be registered with the Viper Online server. For this purpose, one has to contact the responsible person and provide the full web address to the tool hoster. In the admin interface (http://url_to_server.com/admin), the administrator can log in with the server password and add the new tool hoster. If the process is successful, the server produces a link which can be used to reload the tool hoster's configuration file. This link has to be forwarded to the tool hoster's responsible person. On every change to the configuration file, this link has to be opened in a browser to load the new configuration.

6.2 Adding a new Tutorial

The setup of a new interactive tutorial is very simple. It is assumed that the tool to be used for the user interface is available at the Viper Online server and working as intended. The only prerequisite to make it work is a web server. The tutorial only uses Javascript and has therefore no special requirements like a database or PHP.

Framework

First thing to do is copy the folder containing the framework to the root of the web server. The framework includes all required files. The only thing missing is the tutorial content.

Configuration

There are a few things that need to be configured. In the file *root/js/tutorial.js* the following settings need to be adjusted:

tutorialTitle	String. The title of the tutorial that gets displayed at the top of the tutorial.
serverUrl	String. The URL of the Viper Online server.
tool	String. The name of the tool to be used for this tutorial. The tool has to be available on the server (e.g **sometool** on http://viperonlineserver.com/tool/**sometool**).
language	String. The name of the language the tool uses. This is used to load the correct language specification for syntax highlighting.

tutorialFile	String. The file name of the tutorial content. In case of a multipage tutorial, this configuration parameter needs to be overwritten on the web page accessed by the user (e.g. *index.html*)

Table 6.1: Tutorial settings

The Viper Online user interface is already preconfigured to be used in a tutorial. Nevertheless, those settings can be adjusted in the file *root/js/viper_online.js*. More details about the settings available for the Viper Online user interface can be found in section 3.3.1.

Styling

In case the styling needs to be adjusted, all stylesheets are in the folder *root/css*. It is recommended to only change the file *style.css*.

Styling custom elements

In section 5.1.2 custom elements were introduced. It is possible to add new custom elements or to change the style of existing ones. In the file *root/js/tutorial.js* is the section custom elements right under the configuration.

To add a custom element, the following steps need to be performed:

- Add the element to the regular expression *regex_customElem*.
- Map the element in *mapCustomElem* to the CSS class(es) it should get styled with.
- Add the CSS class(es) to the stylesheet *css/style.css*.

The variable *additionalClasses* is used to specify CSS classes which get added to every custom element.

Since Viper Online makes use of the Bootstrap framework [2], it is possible to use classes from it. This allows to choose from a large set of predefined styles.

Creating content

The most time-consuming part of the tutorial is creating the actual content. As described in section 5.1.1, the content is written in Markdown [12]. This can be done in any Markdown editor that supports the GitHub Flavored Markdown extension with fenced code blocks and special attributes (for example StackEdit [17]).

Any details about the syntax of Markdown and how it is used to create the tutorial can be found in section 5.1.1, and in the Markdown documentation.

It is not mandatory to create the content in Markdown, as the framework accepts any HTML file as input. Nevertheless, as the framework is not arbitrarily flexible, the HTML file needs to follow some conventions in order to be correctly transformed into a tutorial. With the methods described in this chapter, it is guaranteed that those restrictions are met.

Adding the content

The last step is to convert the Markup code into HTML. The content has to be copied as *tutorial.html* in the root folder of the framework.

In case any images have been included into the tutorial, they also need to be placed in the framework. Best practice is to store them in a separate folder *tutorial_images* and link them accordingly in the Markdown code.

Multipage Tutorials

It is possible to distribute a tutorial over multiple pages. In order to achieve this, some additional steps need to be performed.

Initially, the framework only has *index.html* as web page that is accessible by the user. To successfully set up a multipage tutorial, *index.html* needs to be copied (and renamed) to match the number of pages the tutorial is going to have (e.g. *index.html*, *part2.html*, *part3.html*).

For all of the copies of index.html, the tutorial content file to be loaded on accessing the web page has to be specified separately. This can be done by overwriting the Javascript variable *tutorialFile*. It is important that this is done after the file *tutorial.js* is loaded, otherwise *tutorialFile* is set to default value *tutorial.html*. Listing 6.2 illustrates how to load specific tutorial content into a tutorial page.

```
<!-- Inside the HTML head element -->
<script src="js/tutorial.js"></script>

<!-- Define after tutorial.js is loaded -->
<script type="text/javascript">
  var tutorialFile = "tutorial_part2.html";
</script>
```

Listing 6.2: Load *tutorial_part2.html* into the tutorial

Setting the value of *tutorialFile*, allows to override the default value and loads the correct tutorial content into the web page.

A small example: *index.html* is copied and renamed to *part2.html*. The changes in listing 6.2 are applied to *part2.html*. The tutorial content for part

one is in *tutorial.html* and for part two in *tutorial_part2.html*. Accessing **/mytutorial/index.html** in a browser loads the tutorial content from *tutorial.html* and **/mytutorial/part2.html** loads the content from *tutorial_part2.html*.

Open tutorial

When all steps above have been completed, the tutorial is accessible through a web browser. On loading the page, the tutorial automatically loads the content, prepares the Viper Online user interface, builds the navigation and adds the styling to the custom elements.

Chapter 7

Quality Assurance

7.1 Documentation

Documentation for the Viper Online server and tool hoster is generated with the scaladoc tool [14]. For all the Javascript code in the project, documentation is generated with JSDoc3 [11]. All source code is commented, to allow good maintainability and extension, including all HTML and CSS files.

7.2 Compatibility

Compatibility of the web pages with different browsers has been tested. The Viper Online web pages, the code input and the tutorial is compatible with the latest versions of Chrome (44.0.24103), Firefox (39.0), Safari (8.0.7), Internet Explorer (11.0.9600) and Opera (31.0). The design of the web pages is also compatible with mobile devices. The Ace code editor used in the Viper Online user interface has limited compatibility with mobile devices. Depending on the device, the browser and the screen size, typing in the editor is not working correctly. Exact device compatibility has not been further tested. Therefore, it is not recommended to write code on a mobile device. However, running the predefined examples or a logged session from a permalink is no problem.

7.3 Testing

To assess the behaviour of Viper Online under heavy load, a series of stress tests were performed. The tests were performed with the tool Gatling [6] , which allows to simulate scenarios over HTTP requests. As the tests were performed, Viper Online was running on a machine with the following characteristics:

- Windows Server 2008 R2 Standard
- 4GB RAM, Intel Xeon CPU (2.40GHz, 2.39GHz), 64-bit

7.3.1 Tool Hoster

The first two stress tests were performed to test the tool hoster. The tool hoster had the following configuration:

- Tool: Silver (Silicon)
- Concurrency limit: 15
- Tool gets started with nailgun [1]

The scenario consisted of a run-request directly sent to the tool hoster. This scenario was used for two different tests.

Test A used a instant workload of 20 sessions at once. This test intended to asses how the tool hoster handles the concurrency limit.

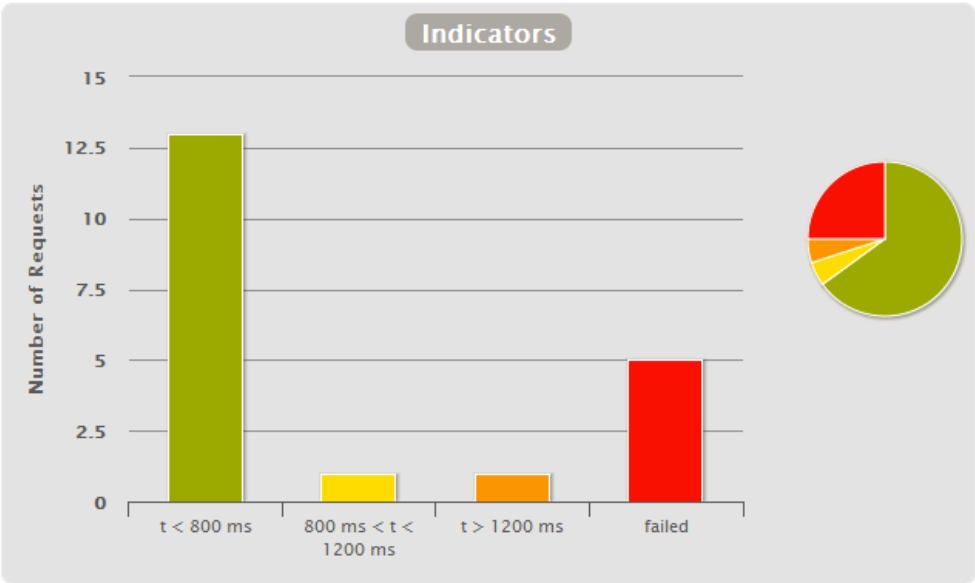

Figure 7.1: Results Test A

Figure 7.1 shows the results of test A. The results show that exactly 15 of 20 sessions were successful and the other 5 were rejected. Two of the 15 successful sessions had a relatively long response time. The fact that 15 request were accepted is exactly what is to be expected. As the tool hoster

[1] nailgun is an application which preloads user-defined classes into a persistently open Java Virtual Machine and therefore reduces the runtime of those classes significantly. http://martiansoftware.com/nailgun/quickstart.html

has a concurrency limit of 15, the first 15 session get accepted, the others don't. The longer response time of two requests is to be explained with the high workload, the tool hoster had to deal with.

Test B initiated 100 sessions over the time of 10 seconds. This test intended to asses the behaviour of the tool if it is under high workload for an extended time period.

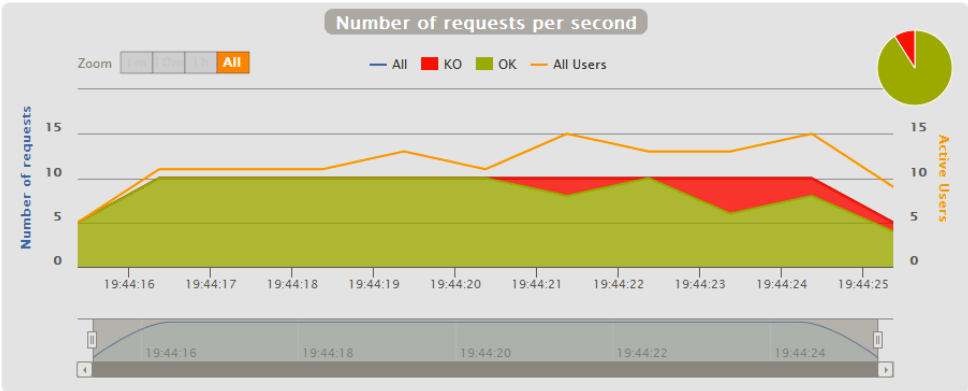

Figure 7.2: Results Test B

Figure 7.2 shows the result for test B. On the timeline we see the accepted sessions in green and the declined sessions in red. The first impression is that a huge part of those 100 sessions were actually accepted. Because the tool is started with nailgun, the verification time is only around one second. This allows the tool hoster to accept and execute this many sessions. In the end, the tool hoster is too busy and has to reject some of the sessions.

Test A and B have shown that the tool hoster is running stable. It handles the concurrency limit correctly and is able to execute a sufficient number of sessions in short time.

7.3.2 Tool Server

The next two stress tests were performed to test the server. The scenario of tests C and D is the following:

- Load the tool selection page
- Select a tool (Silver)
- Go back to tool selection page
- Select a tool (Dafny)
- Go back to tool selection page
- Select a tool (Chalice)

Test C is similarly designed like test A. The scenario was executed 50 times in parallel.

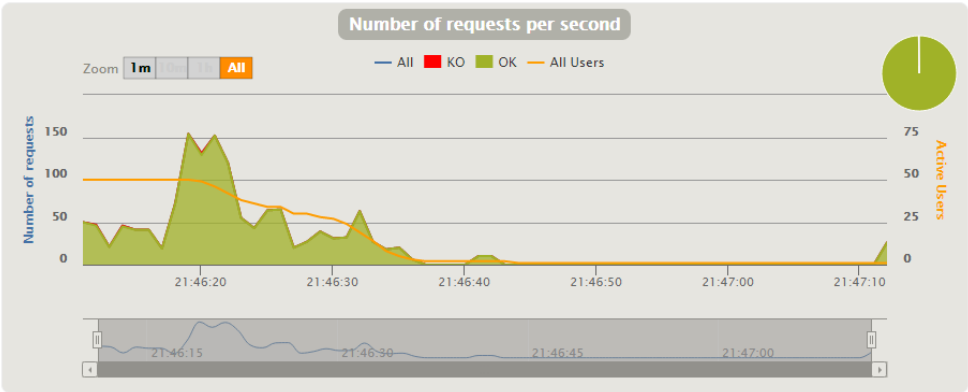

Figure 7.3: Results Test C

Figure 7.3 shows the results of test C. Unfortunately, not all requests have been answered. At least, the server did not crash and remained fully functional. The timeline indicates that at the highest peak, some few requests couldn't be answered.

Test D distributes the same load as in test C over 10 seconds.

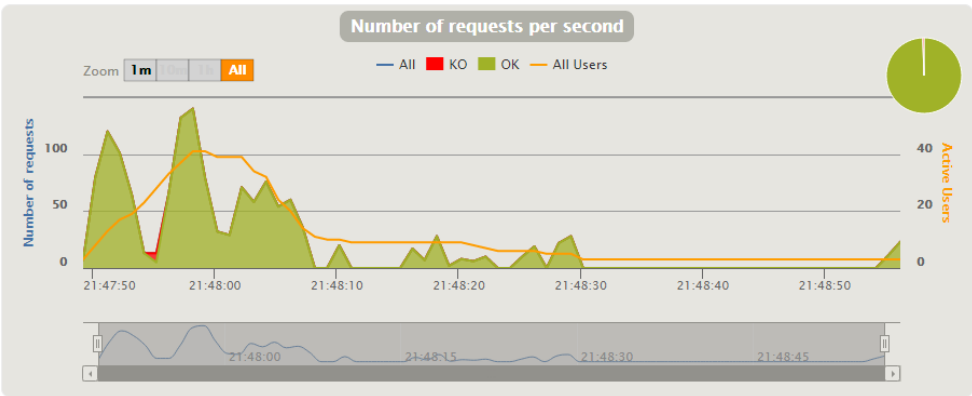

Figure 7.4: Results Test D

The results of test D in Figure 7.4 are a bit unexpected. As it is the same load as in test C, one could expect that all requests are accepted. Nevertheless, the server was not able to answer all requests.

The results of test C and D have shown, that the web server is not suitable for large numbers of synchronous sessions. As the server remains functional, user who experience timeouts of their session may reinitiate the session at a

later time.

The obvious difference in the results between tests A/B and C/D can be explained that tests C/D needed much more network traffic, because of the images, javascript files and stylesheets. But the implementation has proven itself stable, also under increased load, as it remained functional all the time.

In any case, the presented tests are scenarios to show the limits of the server. Under normal load, the server and the tool hoster work perfectly fine.

7.4 Response Time Measurements

One goal of this project was to reduce the response time significantly. To verify whether this goal has been achieved, multiple response time measurements have been performed with the new Viper Online.

Figure 7.5 summarizes the differences of response time between the old tool and the new Viper Online on five Chalice examples.

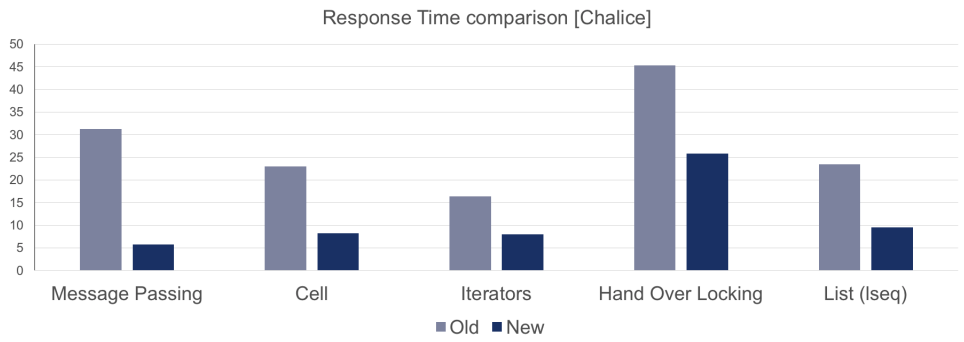

Figure 7.5: Response time comparison between the old tool and the new Viper Online

Figure 7.5 indicates clearly that project has indeed achieved a significant speedup over the old tool.

Also for the Silver tool available on Viper Online, response time measurements have been performed. As this tool is used for the Silver tutorial (section 5.3), a fast response time is important for the interactivity of the tutorial.

Figure 7.6 illustrates the response time of four different code examples on Viper Online. As Silver offers two different backends (verifiers), the response time has been measured with both.

The response time with verifier Silicon is for all examples around 2.5 seconds, which is sufficiently fast for the Silver tutorial.

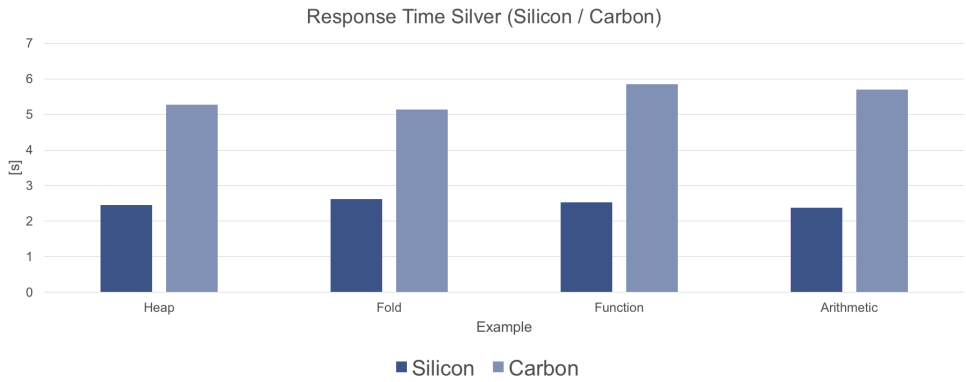

Figure 7.6: Response time of Silver examples verified by Silicon and Carbon

The same examples verified with Carbon need approximately twice the time until the user gets a result. This is because the verifier Carbon is indeed slower than Silicon. Nevertheless, a response time of 5 seconds is still acceptable.

All examples tested in this section are also available on Viper Online.

Chapter 8

Conclusion

This thesis introduces a simple way to create an interactive tutorial, based on Viper Online. The automated workflow to create the tutorial allows to concentrate on the content and not on implementation details or design of the tutorial. The included code snippets can be transformed automatically to an Viper Online user interface and be executed from there. This gives the reader of the tutorial a hands-on experience.

The underlying tool, Viper Online, has been improved in many ways. First of all, the response time has been reduced significantly. This is an essential point for an interactive tutorial. Furthermore, the Viper Online user interface has been made portable, meaning that it can now be embedded into any website. The user interface has been redesigned, giving it a modern look and adapting its design to the Viper verification infrastructure. To better represent the Viper verification infrastructure, the possibility to have multiple backends per tool hoster has been introduced. This also gives the user the possibility to change between different verification methods within the same user interface.

The developed tutorial covers the basics of Silver and is a good basis to develop it further.

Overall, the improved Viper Online, combined with the presented way to create a tutorial, offers a valuable tool to enhance the public perception of the Viper verification infrastructure and may also serve educational purposes.

8.1 Future Work

This project still has some things that can be improved or functionality that can be added.

8.1.1 Viper Online

One functionality that was discussed during the project, but not implemented, is the caching of results on the Viper Online server. How it can be done conceptually is described in section 3.4. This can be implemented in a future extension. There are also several improvements to the user interface that could be done: In case the server returns verification errors, they could be displayed as squiggly lines, directly in the editor. Another useful functionality would be real-time syntax checking for the code input. The Ace editor supports such a feature, but currently only for very few languages.

Support for mobile devices could also be improved, but this requires the availability of code editors that support mobile devices.

8.1.2 Tutorial

One issue with the tutorial are multiple pages. Currently, a multipage tutorial needs to be separated and linked manually. This process could also get automated with a special page break element, written in Markdown and applied to the tutorial later in the process.

The current implementation of custom elements is a work-around, because they are not natively supported by (Github Flavored) Markdown. As there are open-source Markdown tools, one of them could be extended in order to fully fit the arising needs during the creation of an interactive tutorial.

The current Silver tutorial covers only the basic functions of Silver, and it should eventually be extended to cover the whole functionality that Silver offers.

8.2 Acknowledgments

This project would not have been possible without the help of others. First of all, I want to thank Malte Schwerhoff for his patient assistance and valuable inputs throughout this project. Then I want to thank Prof. Dr. Peter Müller for giving me the chance to complete this project at his research group and contributing to the Viper verification infrastructure.

Many thanks also to Roland Meyer, the author of "Developing a common web interface to various verification tools" [19]. With his work he provided a good basis for what I achieved in this project.

Bibliography

[1] Ace. http://ace.c9.io/.
Accessed: 08.08.2015.

[2] Bootstrap. http://getbootstrap.com/.
Accessed: 08.08.2015.

[3] CodeMirror. https://codemirror.net/.
Accessed: 07.08.2015.

[4] Corporate Design ETH Zürich. https://www.ethz.ch/services/de/service/kommunikation/eth-identity.html.
Accessed: 08.08.2015.

[5] Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/.
Accessed: 07.08.2015.

[6] Gatling. http://gatling.io/#/.
Accessed: 08.08.2015.

[7] highlight.js. https://highlightjs.org/.
Accessed: 08.08.2015.

[8] HTML5 Up - Read Only. http://html5up.net/read-only.
Accessed: 08.08.2015.

[9] Jetty. http://www.eclipse.org/jetty/.
Accessed: 08.08.2015.

[10] jQuery. https://jquery.com/.
Accessed: 08.08.2015.

[11] JSDoc 3. https://github.com/jsdoc3/jsdoc.
Accessed: 08.08.2015.

[12] Markdown. http://daringfireball.net/projects/markdown/.
Accessed: 08.08.2015.

[13] SBT. http://www.scala-sbt.org/.
Accessed: 08.08.2015.

[14] Scaladoc. https://wiki.scala-lang.org/display/SW/Scaladoc.
Accessed: 08.08.2015.

[15] Scalatra. http://scalatra.org/.
Accessed: 07.08.2015.

[16] Silver. https://bitbucket.org/viperproject/silver.
Accessed: 06.08.2015.

[17] StackEdit. https://stackedit.io/.
Accessed: 08.08.2015.

[18] Viper (Verification Infrastructure for Permission-based Reasoning). http://www.pm.inf.ethz.ch/research/viper.html.
Accessed: 06.08.2015.

[19] Roland Meyer. Developing a common web interface to various verification tools. http://www.pm.inf.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Roland_Meyer_BA_report.pdf, 2012.
Accessed: 06.08.2015.

[20] Uri Juhasz, Ioannis T. Kassios, Peter Mueller, Milos Novacek, Malte Schwerhoff, and Alexander J. Summers.
Viper: A Verification Infrastructure for Permission-Based Reasoning.
http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=JKMNSS14.pdf,
2014.

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):

Developing an Interactive, Web-Based Tutorial for an Intermediate Verification Language

Authored by (in block letters):

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

Name(s):	**First name(s):**
Birrer	Mathias

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

Winterthur, 16.08.2015

Signature(s)

M. Birrer

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