Bachelor's Thesis

IMPROVING ANALYZABILITY OF LIBRARY INTERACTIONS IN VARA-TOOL-SUITE

SIMON LICHTNECKER

March 12, 2021

Examiner: Prof. Dr.-Ing. Sven Apel
(Chair of Software Engineering I)

Advisor: Florian Sattler, M.Sc.
(Chair of Software Engineering I)

Chair of Software Engineering
Faculty of Computer Science and Mathematics
University of Passau

It's been true in my life
that when I've needed a mentor,
the right person shows up.

— Ken Blanchard

Dedicated to my family, my girlfriend,
and all the incredible people,
who have supported me and my work.

DECLARATION

Hiermit bestätige ich, dass ich diese Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind. Ich habe diese Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Passau, March 12, 2021

Simon Lichtnecker

ABSTRACT

Software projects grow larger and become more complex as they evolve. This development makes it increasingly difficult to estimate the impact of code changes on other parts of the program. This problem becomes more apparent when the projects include library code. The LLVM-based software analysis framework VaRA tackles this problem by analyzing data flow interactions between commits and reporting them to the user. The computed interaction data can be visualized to understand the impact of commits on other parts of the program and to illustrate their development over time.
One application that is able to visualize this interaction data is called VaRA-Tool-Suite. However, it currently lacks the ability to utilize and visualize interaction data that stems from commits of different libraries. This constraint limits its use cases, since many projects include other libraries, which in turn introduce more commits.
To solve this problem, we extend the scope of VaRA-Tool-Suite to commits that originate from multiple libraries. Furthermore, we introduce a set of new visualizations that are able to depict interactions between library commits on different levels of detail and illustrate their development over time. We demonstrate the usefulness of our visualizations on three different projects, where the first one is a small hand-crafted example project, to showcase interesting scenarios, and the other two are real-world projects.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

VaRA Variability-aware Region Analyzer

VaRA-TS VaRA-Tool-Suite

1 INTRODUCTION

During the development of a software project, new lines of source code are constantly added, increasing its size and complexity. This makes it increasingly difficult for developers to predict how a particular change in the code will affect other parts of the program. Unexpected interactions between code parts can lead to bugs that are particularly hard to find in a large code base. This problem becomes worse when source code is contributed by more than one person. That is especially the case when projects contain code from other projects that are included as libraries. With VaRA (Variability-aware Region Analyzer), one is able to analyze the data flows between source code instructions to gain information about the interactions between different program parts. Additionally, if a project and its libraries are Git repositories, VaRA can relate the interactions between them to the commits that introduce their interacting source code instructions.
Since it is difficult to interpret this interaction data, we want to visualize the interactions between commits. The auxiliary application of VaRA, called VaRA-TS, serves this purpose well. However, its visualizations are currently limited to interactions between commits that originate from the same Git repository. We believe that it is critical to also include the interactions of library commits into the visualizations of VaRA-TS to get a better understanding of the entire software project. These visualizations help researchers and developers, e.g., to find commits that impact the project in an unexpected way or to illustrate the development of commit interactions over time. Additionally, one is able to visualize dependencies between commits of the main program and its libraries and commits of the libraries themselves.

1.1 GOALS

Our goal is to improve the analyzability of library interactions in VaRA-TS by visualizing interactions between commits, regardless of whether those commits stem from the main repository or its including libraries. These visualizations can be used by researchers and developers to get a visual impression of data flows between interacting commits, regardless of whether the commits stem from the same or different libraries. We propose a concept of accessing and persisting the interaction data that results from the data-flow analysis of VaRA, in combination with its VaRA-TS. Furthermore, we use this data to create a set of four library aware interaction plots that allow users of VaRA-TS to inspect the interactions between multiple libraries at different levels of detail.

1.2 OVERVIEW

In Chapter 2, we introduce the essential background of this thesis. We describe the distributed version-control system Git and the plotting libraries we use to implement our visualizations. We also explain the features of VaRA and its auxiliary application VaRA-TS that are most important to us. In Chapter 3, we present our improvements to the library

interaction analysis of VaRA-TS that include an extension of the commit lookup to libraries and the concepts of persisting the data of `BlameReports` in appropriate database layouts. Furthermore, we introduce the reader to the implementations and concepts of our blame library interaction plots. We evaluate our work in Chapter 4, by creating our visualizations on a set of case studies, summarize the results, discuss them, and illustrate the factors that could endanger the validity of our approaches. In Chapter 5, we mention work that is related to ours. Finally, we conclude our work in Chapter 6 and preview possible extensions to our work.

2 BACKGROUND

In this chapter, we introduce the distributed version-control system Git. Next, we give a deeper insight in our used Plotting Libraries. Last, we explain the Variability-aware Region Analyzer framework and the VaRA-Tool-Suite.

2.1 GIT

In this section, we introduce Git, a distributed version-control system. The development of a software project includes different tasks, where many can be reduced to a basic set of actions like adding, deleting, or modifying files. Applying any of these actions results in a new version of the project's state, which cannot be easily undone or kept track of without a version-control system. Git, as such a system, provides a memory-efficient way of tracking all project versions and their contents. This is achieved by saving the complete history of modifications done to the project. A project with all its files that is under version-control, is called a *repository*. Each version of this repository can then be searched, accessed, and rolled back to.
As a developer, one is very likely to collaborate with other people on the same project. This collaboration can lead to problems if they are not using a version-control system and start editing the same files simultaneously. Git tackles this issue with the concept of a repository *clone*. By cloning a repository one gets a complete copy of it, which can be modified and treated in the same way as the original one. By doing so, the clone keeps a reference to its *origin*. File changes are then made to each developer's own local clone and do not interfere with changes made to local clones of other developers.
After modifying the local clones they usually need to be synchronized at some point in the development process to gain a common project state again, which could be further distributed. By integrating local file changes into the origin repository instance, Git tries to merge the current repository state with the new one. Changes that do not interfere with each other can be handled by Git automatically. In cases where conflicting files have to be merged, e.g., the same line of code in a file got edited, Git asks the developer to resolve the conflict.
After all files have been merged the origin repository is again in a consistent state and other developers can now integrate the new state into their local clones.

2.1.1 *Commits*

One of Git's core concepts is the *commit* that resembles a snapshot of the project's current state. The creation of a commit incorporates changes of the project into the repository and stores a snapshot of the new state of the project at this time. Further changes of the project can be incorporated into its repository as new commit, with the currently referenced commit as parent commit. A series of commits represents the history of the project, where each

commit by itself is enough to get the full state of the project at the time a commit was made. A commit is uniquely identifiable within one repository by its SHA1 or SHA256-checksum, called *commit hash*. One can choose a commit to view the changes that it incorporated into the repository and use it to revert the project to the previous state at the time the commit was created. A series of new commits can then be added to the repository with this commit (the commit one reverted to) as parent commit. A commit in its textual representation contains a line about its tree object, its parent commit, its author and committer.
An example commit is shown in Listing 2.1. Line 1 contains a reference to its tree object, which points to the project's snapshot. It consists of one or more hashes that refer to either a file or subtree. Line 2 contains the commit hash of the commit's predecessor. Line 3 and 4 contain information about the author and committer with their name, e-mail address and time (in the UNIX format). The author of a commit is the person who originally made the file change. The committer is the person who applied the commit to the repository. Line 6 contains the commit message, which is a short descriptive text of the changes that this commit introduces.

```
1  tree 8af77d117b923eec332c3b43781be1ae24f72811
2  parent 6ba631b2f38cbf0175fb70adbe5a358d7940bdb5
3  author Simon Lichtnecker <simon@example-mail.com> 1610895177 +0100
4  committer Florian Sattler <florian@example-mail.com> 1610898241 +0100
5
6  Example commit message
```

Listing 2.1: Example of a Git commit object.

2.1.2 *Blame*

The `git blame` command facilitates a developer's insight on how project files progress over time. By passing a file of interest `git blame` annotates each line with the commit hash and author name that modified the line last. This feature is especially useful if questions about a specific code part arise and its author or introducing commit has to be determined. Listing 2.2 shows a short section of an example blame annotation. For example, the commit `f1bc62b9` by Sebastian Böhm in year 2020 was the last change that edited line 4.

```
2  704f3e73 (Simon Lichtnecker 2019-11-09 23:41:22 +0100   2) import os
3  2a6bfbbe (Florian Sattler   2019-01-15 23:45:26 +0100   3) import re
4  f1bc62b9 (Sebastian Böhm    2020-05-07 17:28:29 +0200   4) import typing as tp
```

Listing 2.2: Example of a `git blame` annotated code part with the commit hashes (left column) followed by author names, times in UNIX format, and the actual code with corresponding line numbers (right column).

2.1.3 *Submodules*

Incorporating code from external sources into a software project is a common practice in the software development process. The access to other code sources from within a project alleviates its development, saves time, and extends its functionality. To integrate a project into another one, the foreign project could simply be copied and stored as subdirectory. This approach has the drawback that it is cumbersome to integrate each update of the foreign project into the other one. To combine the above mentioned benefits of a Git repository with the inclusion of foreign projects and the option to easily update them, Git has the concept of *submodules*.
Including another repository as submodule adds it to the repository as a subdirectory. Submodules are like every Git repository in a defined state, which is the current commit they are referencing. Submodules have a reference to their original source location. This allows developers to get the latest commits of the foreign source. As the full commit history of the submodule can be accessed, each version of it can be used in the main repository. For example, this can be beneficial if the main repository wants to skip newer versions of the submodule for compatibility reasons.
By adding the first submodule Git creates a *.gitmodules* file in the repository. Each submodule that is added creates a mapping entry in this file that contains a `path` and an `url`.
Listing 2.3 shows, as an example, the .gitmodules file of the project GNU Bison, which contains two submodules (`gnulib` and `autoconf`). The entry of the submodule `gnulib` (line 1) includes its file `path` (line 2) relative to the main repository. The `url` (line 3) specifies the remote location from which it was originally cloned.

```
1   [submodule "gnulib"]
2     path = gnulib
3     url = git://git.savannah.gnu.org/gnulib.git
4   [submodule "submodules/autoconf"]
5     path = submodules/autoconf
6     url = git://git.sv.gnu.org/autoconf.git
```

Listing 2.3: The .gitmodules file of the GNU Bison parser generator containing two submodule entries (gnulib and autoconf) with the their path and url.

2.2 PLOTTING LIBRARIES

This section introduces the plotting libraries that we use to implement out different kinds of plots in Section 3.3. We begin with the Matplotlib library, followed by Plotly, and Graphviz.

2.2.1 *Matplotlib*

Matplotlib [5] is a 2D graphics environment used to generate high-quality images in Python. We use its *MATLAB*, state-based interface in all of our Matplotlib based plots, which is encapsulated in the *pyplot* module. Hence, this subsection covers only the pyplot interface.

The general idea behind the MATLAB-oriented interface is that its functions change a figure's state while preserving a reference to its new state across function calls. The access to the figure's state throughout the whole program simplifies plot adjustments without the need of passing the figure to every function. Figures contain one or more *axes*, which are the point of reference for most of the plot elements, like the coordinate system, plot type, and legend. A sectional overview of some of the most important figure parts is shown in Figure 2.1[1].
Matplotlib refers to the word 'axes' as the data space or generally speaking the complete plot with all its including plots, e.g., the red and blue line plots in Figure 2.1. Therefore, axes must not be confused with the term '*axis*', which refers to, e.g., the x-, y-, or z-axis of the Cartesian coordinate system.
Besides multiple plots on the same axes, one can add multiple axes to the same figure. The positions of axes are determined by the *grid* of the figure. The grid groups the figure's plotting area in rows and columns, which are modifiable in their number and used as reference point of an axes' position.
After arranging the axes on the grid, one is able to generate plots of different types and treat them independently. After the plot generation, the user is able to view and export the whole figure with all its axes.

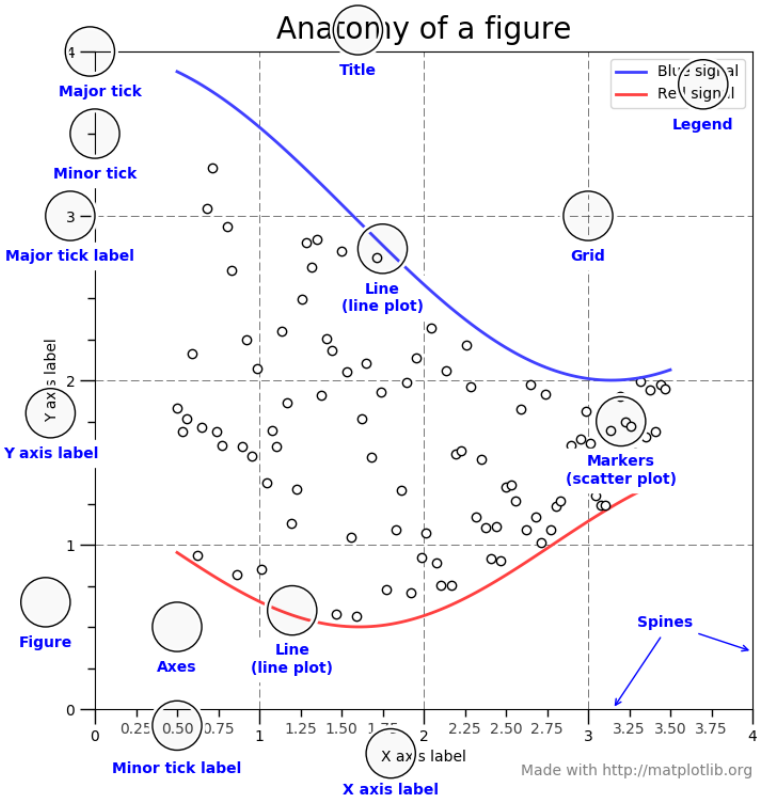

Figure 2.1: A sectional overview of Matplotlib figure components.

1 Available online at https://matplotlib.org/3.3.3/tutorials/introductory/usage.html#figure-parts; visited on January 21st, 2021.

2.2.2 *Plotly*

The Python graphing library Plotly[2] follows the concept of interactive plots. The interaction with plots after their generation is beneficial to represent extra information, e.g., via tooltips, rearrange elements to improve their readability, or even display animations. Furthermore, besides many basic plot types, Plotly also offers more advanced ones, like various 3-dimensional charts and supports more than 40 different chart types in total. The plots are rendered per default in the web browser where the user is able to download them. By using, e.g., the library *Kaleido*[3], one is able to generate static images locally in the first place.
To create a `Figure` that is later rendered as chart in the browser, one must define the plotting `data` with its plot type. The layout of the resulting plot can be specified by, e.g., using the `autosize` (determines the plot size automatically) option, to fit the plot in the web browser. If the user wants more control about the plot's layout and appearance, they can set the corresponding options explicitly.

2.2.3 *Graphviz*

The representation of structural information in the form of (directed) graphs is a common way to visualize data, like networks, data-flows, or dependencies in general. A popular graph visualization tool that serves this purpose well is *Graphviz* [2], which offers a variety of tools to build, manipulate, and render graphs. Graphviz provides implementations of common graph layouts and their access through various interfaces. We use its correspondent Python library to build our Graphviz Plots, which is a wrapper for creating graphs in the *DOT*[4] language.
A `graph` or directed graph (`digraph`) in DOT consists of the main components `node`, `edge`, and optionally `subgraph`. Subgraphs build a semantic group of nodes and their corresponding edges within a graph. This allows one to reference the elements of the subgraph altogether for further modification. Furthermore, attributes can be added to the subgraph that then apply to all contained elements. This is especially useful when the contents of one subgraph have to be distinguishable from those of others.
Interpreting, arranging, and drawing graphical and textual elements is done by so-called *layout engines*. While this term can vary in the field of application, we use it in the context of displaying graphs. There exist many layout engines to represent different kinds of graphs, like directed or undirected graphs that can be drawn with, e.g., the engines *dot* [4] and *fdp* [3].
Subgraphs beginning with the name '*cluster*' are treated specially by most of the Graphviz layout engines, like `dot` and `fdp`. Cluster subgraphs are no native part of the DOT language but induce engines that are sensible to this keyword to draw a rectangle around the corresponding subgraph. This groups subgraph elements visually together and tells the layout engine that they must be treated as unit.
Figure 2.2 shows an example of a Graphviz `digraph` with its source code, written in the DOT language, and utilizing the dot engine. The subgraphs `foo` and `bar` are defined as

2 Available online at `https://plotly.com/python/`; visited on January 26th, 2021.
3 Available online at `https://github.com/plotly/Kaleido`; visited on January 26th, 2021.
4 Available online at `https://graphviz.org/doc/info/lang.html`; visited on January 27th, 2021.

`cluster` subgraphs (line 3 and 11). The dot engine interprets this notation and draws a rectangle in their defined `style` around them. Node `a` in `foo` interacts with node `a` in `bar` and vice versa. The nodes are distinguished by their unique names in line 8 and line 15, but can have the same `label` as seen in line 18 and line 19. Labels are used to explicitly set the displayed name of graph elements, like subgraphs or nodes. If not used, the displayed name of, e.g., the nodes `a1` and `a2` would be the same as their definition name, which is often undesirable when the only difference between these nodes is their membership in a subgraph. To give another example, the `color` attribute in line 5 and 13 is used to apply a color to the whole subgraph area, excluding the node area, which is defined as different color.
In the latter analysis, the Graphviz library allows us to write Python code that gets translated into DOT and is finally drawn by our layout engine `fdp` to produce our Graphviz Plot.

```
1   digraph G {
2
3     subgraph cluster_foo {
4       label = "foo";
5       color = lightgrey;
6       style = filled;
7       node [style=filled, color=white];
8       a1 -> b;
9     }
10
11    subgraph cluster_bar {
12      label = "bar";
13      color = blue;
14      node [style=filled];
15      a2 -> c;
16    }
17
18    a1 [label=a]
19    a2 [label=a]
20    a1 -> a2
21    a2 -> a1
22    x -> c
23  }
```

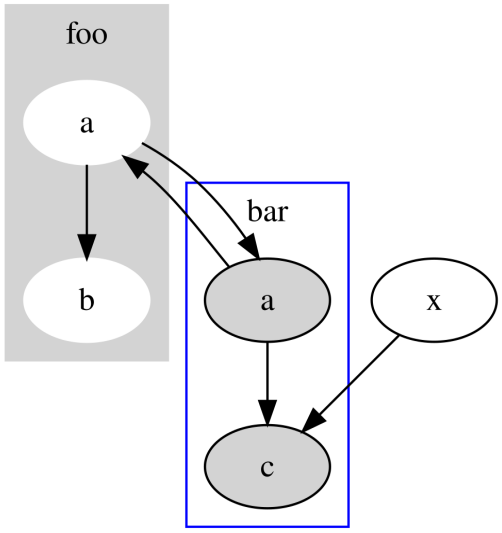

Figure 2.2: Example of DOT source code (on the left) that generates the directed graph (on the right) containing two cluster subgraphs.

2.3 VARIABILITY-AWARE REGION ANALYZER

In this section, we introduce the Variability-aware Region Analyzer (VaRA), which is a software analysis framework based on the LLVM Compiler Infrastructure (LLVM)[5]. VaRA can perform different types of analyses on a given software project. One of these analyses is the `BlameAnalysis`, which is especially relevant to us, as it generates the information we need for our implementations in the latter sections. To understand the concept of the `BlameAnalysis`, it is important to know what the basics of control-flow and data-flow analyses are.

Control-flow analysis aims to determine the order in which program instructions are evaluated. A set of program statements that is definitely executed directly after each other can be grouped together as a *basic block*. To visualize how control flows through a program, it can be represented as directed graph where a node is such a basic block, and an edge, starting from a basic block, connects to a basic block that may be evaluated next. This representation is called *control-flow graph*. *Data-flow analysis* collects information about a program by analyzing how variables change across edges in a control-flow graph and aggregates the collected data along the control flow. To give an example of a data flow that is detected by the data-flow analysis, one may write a function that takes a number, increases its value by one and returns it. Then this function gets called and the returned value assigned to a variable. First, this results in a data flow from the instruction of the assignment to the called function, as we pass our number as parameter to it. Second, this number is increased by the function and gets returned to the instruction of the assignment, which results in a data flow from the function, back to the instruction of the variable assignment.

The control-flow and data-flow analysis are types of static program analyses, which means that they can be performed without executing the program. Now, the `BlameAnalysis`, which is implemented on top of a data-flow analysis, aims to find interactions between commits for all source code files of the analyzed project. These interactions result from data flows between instructions. Each instruction in the source code can be mapped, similar to `git blame`, to the commit that introduced the code for which the instruction is created. The `BlameAnalysis` is then able to relate the data-flow information between the instructions to its corresponding commits. This relation is used by the `BlameAnalysis` to report interacting commits that contain such data flows from one commit to another. We call an interaction from one commit to one or more commits, a *blame interaction*.

The `BlameAnalysis` of VaRA detects the blame interactions of a software project by combining its data-flow analysis with `git blame` annotations. The next section explains how the `BlameAnalysis` is used with the VaRA-Tool-Suite, and introduces a way of persisting the analysis data in a file. [6, 8]

5 Available online at `https://llvm.org/`; visited on January 28th, 2021.

2.4 VARA-TOOL-SUITE

Researchers often have to invest a lot of time in the process of setting up the projects and the analyses they want to perform on them. Analyzing a new project or changing the analysis method restarts this time-consuming process, as the requirements of the analysis or projects can change. While some work a researcher has to perform cannot be reduced or optimized, essential steps in the analysis pipeline, like defining projects and experiments, analyzing multiple project states, persisting analysis results and evaluating them, can.
We introduce VaRA-TS[6], a software application that aids users of VaRA in setting up their own experiments and to make experimenting on projects easier. VaRA-TS offers a consistent interface for many steps of an analysis pipeline by abstracting redundant operations. Analyses with VaRA require a working installation on the user's system. VaRA-TS accelerates this installation process by downloading, building, and finally installing VaRA automatically. Analyzing software projects requires researchers to download, configure, and compile them often numerous times. This process is error-prone and often tedious. The project interface of VaRA-TS tackles this issue by allowing the user to define each step of this process separately with the aid of helper methods. Once defined, all steps can be run automatically by VaRA-TS. Adding new projects is easy, as the structure of a project definition never changes. Analog to project definitions in VaRA-TS, the implementation of experiments also has a fixed structure. Similar to project definitions, the experiment interface offers methods that cover necessary steps in an analysis and hides redundant ones from the user. Furthermore, the projects and experiments of VaRA-TS are defined independently, which allows a user to arbitrarily combine them.
After adding a project to VaRA-TS, a user is not limited to run experiments on only one version of the project. Some analyses require to be run on many versions of a project, e.g., to gain information about the project's development over time. For this reason, VaRA-TS provides a way to sample an arbitrary number of project revisions, if the project is a Git repository. When sampling the revisions of a repository's commit history, VaRA-TS stores the sampled commit hashes in a text file, called *CaseStudy*. If specified, the generated `CaseStudy` file can be passed to an experiment run of VaRA-TS. This allows VaRA-TS to perform the experiment on each of the project states that are listed in the `CaseStudy` file. After executing an experiment, the results are collected in a text file, called *report*.
Reports are usually specific to the type of experiment data they are storing. Therefore, VaRA-TS allows a user to define a report's structure that contains their analysis results in a later easily accessible and condensed way. Helper functions for common tasks, like deriving report names from their corresponding experiment and accessing reports are already provided by the report interface of VaRA-TS. After the experiment data is persisted in a report file it can be evaluated in the form of a plot. Implementing a plot can be divided into the parts of retrieving the plotting data, building the actual plot, showing the plot, and saving it for later inspections. To facilitate the retrieval of plotting data, VaRA-TS provides a database layout that is used to aggregate the data of multiple reports. If the underlying report data does not change, the database can be reused without the need of aggregating the data again. VaRA-TS alleviates the creation of new plots by providing a plot interface that includes functions to, e.g., retrieve the prior stored data, build multiple plots consistently,

6 Available online at `https://github.com/se-sic/VaRA-Tool-Suite`

save and display the resulting plot. A schematic overview of the analysis pipeline of VaRA-TS is shown in Figure 2.3 [7]. To give an example for a possible analysis pipeline, a user of VaRA-TS could build a `CaseStudy` by sampling 20 revision from the already incorporated project GNU `grep`. Then they start an experiment, like the `GenerateBlameReport` experiment on `grep` with the passed `CaseStudy` file. This results in the generation of 20 reports, which for the `GenerateBlameReport` experiment, corresponds to 20 BlameReports. By starting a plot, like the `BlameInteractionDegree` plot, the `BlameReport` data is automatically aggregated and saved in a corresponding database. To generate the final plot image from the data persisted in a database, one uses the *`vara-plot`* tool of VaRA-TS. This tool takes several arguments, like the name of the analyzed project, the path to the used `CaseStudy` file, the type of the plot, which is going to be generated, and an option to either view or store the resulting image.
VaRA-TS facilitates the work of VaRA users by reducing redundant and error-prone tasks in the analysis pipeline through various abstractions. Two important interfaces that abstract a software project and an experiment rely on an underlying application, named Benchbuild that is introduced in the following subsection.

Figure 2.3: Overview of the VaRA-Tool-Suite pipeline.

2.4.1 *Benchbuild*

The large-scale empirical-research toolkit *Benchbuild* [9] aids researchers by facilitating the preparation and execution of compile-time and run-time experiments. Benchbuild offers researches a broad spectrum of supported software projects and is able to automatically download, set up, and compile them. VaRA-TS and Benchbuild are strongly linked, since the prior one relies on Benchbuild's definition of a project and runs its experiments with Benchbuild.
The concept of a *project* in Benchbuild allows a user to define an abstract representation of a software project with the aid of helper functions through a consistent interface. After

7 Available online at `https://vara.readthedocs.io/en/vara-dev/vara-ts/pipeline-overview.html#tool-suite-pipeline-overview`; visited on March 10th, 2021.

defining the project's download source and its compilation steps, a user can reuse the project and Benchbuild performs the defined actions automatically. Running analyses on a Benchbuild project can be automated by using its experiment interface. An *experiment* is a list of actions that are executed on one or more projects that are part of the experiment. Benchbuild already provides many predefined actions that facilitate the implementation of experiments. Running analyses on projects is now easy, as the necessary steps to correctly set up projects and executing analysis actions are performed automatically and can be adjusted with minimal effort.
VaRA-TS relies on Benchbuild's abstractions of projects and experiments, which minimize the necessary work for researchers to perform their analyses. As an example, to run the experiment `GenerateBlameReport` of VaRA-TS on the project `grep`, one only has to run the command `benchbuild run -E GenerateBlameReport grep`. The experiment results are stored in a result file, called BlameReport, which we explain in detail in the following subsection.

2.4.2 *BlameReport*

As briefly described in Section 2.4, reports are files that store analysis results that are specific to the performed experiment type. The *BlameReport* is such a report and stores condensed textual information about the results of the `GenerateBlameReport` experiment.
The `GenerateBlameReport` experiment of VaRA-TS runs the `BlameAnalysis` of VaRA that was introduced in Section 2.3 on the specified project (revisions). The contents of an example `BlameReport` are represented in Listing 2.4, which results from the `BlameAnalysis` on the project `gzip`. The `demangled-name` in line 581 represents the name of the function (`create_outfile`) that was analyzed by the `BlameAnalysis`. The number of analyzed program instructions is represented in line 582. The following line starts the list of blame interactions that the `BlameAnalysis` found within this function. These blame interactions are broken down into the interactions between one base commit, which corresponds to a `base-hash`, e.g., line 584, and one or more interacting commits, which correspond to the `interacting-hashes`, e.g., line 586. The commits are represented as `CommitRepoPairs`, which are combinations of their commit hash, a dash, and the library name they are originating from. The `amount`, e.g., in line 587, represents that this exact blame interaction was found 3 times in the analyzed source code. This is important to prevent redundant entries in the `BlameReport` and to keep the file as small as possible without losing information.
Since we are interested in relating the blame interactions between commits to their corresponding libraries, we can build subgroups of interactions within a blame interaction that group the `interacting-hashes` by their library names. To give an example, the blame interaction starting in line 588 and ending in line 592 can be divided in an interaction between the base commit in line 588 and the interacting commit in line 590, which originates from the library `gnulib`, and an interaction between the base commit in line 588 and the interacting commit in line 591, which stems from the library `gzip`. Both of these subgroups have the same `amount` as their containing blame interaction, which is 2. Now we are able to derive important values from the blame interaction data that refer to each of these blame interaction subgroups.
The first value is the *`degree`* of a blame interaction subgroup. It is defined as the number

of `interacting-hashes` that originate from the same library, grouped by the library name of their `base-hash` within one blame interaction. It describes the interaction `degree` of a blame interaction, e.g., the number of interactions to other commits, in relation to its base and interacting libraries. For example, if we consider each of the previously mentioned subgroups on their own, we find that both contain one `interacting-hash`, hence they both have a interaction `degree` of 1. The second derived value is the *`fraction`* of a blame interaction subgroup. It is calculated by dividing the `amount` of a blame interaction through the sum of all `amounts` that result from building blame interaction subgroups in every blame interaction entry of the `BlameReport`. It indicates how large the proportion of one interaction is in relation to all interactions of a `BlameReport`.
The presence of an interaction in a `BlameReport` means that the `BlameAnalysis` of VaRA detected a data flow between instructions, whose source code they are originating from, got added by different commits. Therefore, the listed blame interactions show the presence of data flows between different commits.

```
[...]
demangled-name:  create_outfile
num-instructions: 142
insts:
- base-hash:       5ef892a9248e02dac13840f0acefe0fe72605dfa-gzip
interacting-hashes:
- a979d9c4db0adbf341eb329abaf3560aa12f10fd-gzip
amount:          3
- base-hash:       5ef892a9248e02dac13840f0acefe0fe72605dfa-gzip
interacting-hashes:
- 48e0ba6d23bdc0fcad1b620cb410bb0a57684edc-gnulib
- 95ace546d3f6c5909a636017f141784105f9dab2-gzip
amount:          2
[...]
```

Listing 2.4: Example section of a BlameReport of project GNU Gzip with its base and interacting hashes and their corresponding amounts.

3 IMPROVING VARA-TOOL-SUITE'S LIBRARY INTERACTION ANALYSIS

In this chapter, we explain how we improve the ability of VaRA-TS to visualize blame interactions by using an extended commit lookup strategy to create library-aware blame interaction plots. First, we describe the concept of gathering extended commit lookup data with the `BlameAnalysis` of VaRA to determine interactions between libraries. Second, we introduce our database layouts, in which we store the aggregated library blame interaction data of BlameReports. Third, we explain how we implement a set of new plots to visualize the blame interaction data between multiple libraries.

3.1 EXTENDING VARA-TOOL-SUITE'S COMMIT LOOKUP TO LIBRARIES

Commit-dependent evaluations with VaRA-TS are currently limited to commits that stem from the same Git repository within a project. As described in Section 2.1.3, many software projects depend on library code from external projects that are incorporated as, e.g., `GitSubmodules`. To allow users of VaRA-TS to create analyses that depend on interactions between a main program and one or more libraries, we explain the concept of using library related blame interaction data, and describe how we implement this concept.
To better understand how a program and its libraries can interact, it is important to know the basics about using library code in a project. Projects make use of libraries to reuse resources from other projects, like already written source code. After including the libraries into a project, one can access the library functionality by, e.g., calling library functions in the source code of the project. To create an executable from the main program and its used library code, one has to *compile* and *link* them. Compiling the source code results in individual object files for the main program and each library. These object files contain the output of the compiler in the form of machine code. To create a single executable, one has to link the object files together. By doing so, function references that were not found in the main object file are searched in the object files of the libraries. When all referenced functions were found and the object files are successfully linked together, an executable is created. This file can be executed and contains the combined functionality of the main program and its used library functions. If the project and libraries are Git repositories, a common way to integrate the libraries into the project is to add them as Submodules. This allows the incorporation of multiple libraries as repositories into a project that is managed by a main repository instance. As introduced in Section 2.3, VaRA is able to detect blame interactions between commits and report them to the user. This analysis is not limited to the commits of one repository, but can also detect blame interactions between commits that stem from different repositories, e.g., the project repository and its library submodules. A user of VaRA can then analyze the instructions of the source code with the `BlameAnalysis` and retrieve all blame interactions between the main repository and its libraries, or between the libraries themselves, including their corresponding library names. While VaRA-TS currently does not use

blame interaction data from different libraries, it already implements an experiment that executes the `BlameAnalysis` of VaRA on a given project, which delivers blame interactions even between commits of multiple repositories. This experiment is called `GenerateBlameReport`, which generates a BlameReport containing the results of the `BlameAnalysis`. For example, Listing 2.4 represents the contents of a `BlameReport` file after analyzing the project `gzip`. As represented in, e.g., line 590 and 591, the `CommitRepoPairs` (commit hash with library name) already provide information about the origin of a commit. If a user of VaRA-TS wants to run this experiment on a project with submodules to, e.g., retrieve the blame interactions with their corresponding library names, they must define each repository in the `SOURCE` list of a project. The `SOURCE` list of a project in VaRA-TS contains a named main repository, which is defined through the `Git` interface of Benchbuild and all its submodules that are defined through Benchbuild's `GitSubmodule` interface including a specified name. These interfaces are used to download, initialize and set up the repositories correctly in relation to each other. The `SOURCE` list is used to reference each of the repositories for later identification. Users of VaRA-TS can now run the `GenerateBlameReport` experiment on the prior defined project and receive the resulting `BlameReport` file. With this file, a user can retrieve the commit hash of a blame interaction. Additionally, the user is provided with the library name from which the commit originates. We can use this information to search the library by its specified name in the `SOURCE` list and look up the commit that corresponds to its commit hash within this library. To facilitate the lookup of commits, VaRA-TS already implements a commit lookup function that takes a commit hash and returns the corresponding commit. This lookup fails when the passed commit hash does not stem from the repository that was defined first in the `SOURCE` list. To solve this problem, we implemented an extension of this commit lookup function that is able to find commits from any repository in the `SOURCE` list. This extension takes a commit hash and also the repository name chosen in the `SOURCE` list to identify the repository from which the commit originates. When the repository name is found in the `SOURCE` list, the commit that corresponds to the passed commit hash is searched within this repository and is returned.
We proposed a concept of using the `CommitRepoPairs` of blame interactions from a `BlameReport` to find commits that stem from different repositories. We use this extended commit lookup strategy to relate blame interactions between commits to their corresponding libraries.

3.2 UTILIZING BLAMEREPORT DATA

Now that we know the concepts of finding blame interactions with VaRA, storing them with VaRA-TS as BlameReport, and relating the commits of blame interactions to their repositories, we are able to aggregate the received data in, e.g., a database, and build analyses that depend on interactions between multiple libraries. Therefore, we introduce the concepts and benefits of caching aggregated analysis data in databases of VaRA-TS.
It is a common practice to persist analysis data in a database for storage efficiency and improved accessibility. VaRA-TS already offers an abstraction of a database schema that allows a user to define their own layout with the help of functions implementing tasks that most database implementations have in common. When creating a database layout, it is important to consider the requirements of the applications that access the database. Knowing in what use cases a database is involved helps to create database designs that avoid

redundant entries, save only necessary data, and are versatile in their application. Therefore, we have to clarify our use cases first to create database layouts that satisfy our requirements. Our use cases are the visualizations of blame interactions between commits that possibly stem from different repositories, where each visualization focuses on different parts of the blame interaction data. Since the interaction data is provided by `BlameReports`, we need to store them in database layouts that provide the data each specific visualization needs. By breaking down a blame interaction into its components we can already see what information could be saved in a database. Looking at the `BlameReport` of Listing 2.4, we can divide a blame interaction into 4 parts: the type of commit hash (`base-hash or interacting-hash`), the commit hash itself, the library name a commit originates from, and the `amount`, which is the number of times this blame interaction was found within a function. In this example `BlameReport`, we choose the blame interaction that starts in line 588 and ends in line 592. We can see that the commit hash `5ef892a` in line 588 is of type `base-hash` and that the commit hashes `48e0ba6` and `95ace54` in line 590 and 591 are categorized as `interacting-hashes`. Furthermore, we notice that the commit hashes in line 590 and 591 stem from the libraries `gnulib` and `gzip`, as denoted at the end of each commit hash. The blame interaction ends with its `amount` in line 592. This is the data we need to extract from a `BlameReport` file and store it in a database.

A user of VaRA-TS that uses a `CaseStudy` file in an experiment, like the `GenerateBlameReport` experiment, to analyze multiple revisions of a project, will most likely want to persist all the report data in a single database. Storing and accessing the aggregated data of all reports in a database class is facilitated by inheriting from the `EvaluationDatabase` class of VaRA-TS. By doing so, one must implement a function called `_load_dataframe()`, which is used to load and cache a dataframe. Creating a dataframe and caching all report data is easy with the `build_cached_report_table()` function, which builds up an automatically cached dataframe. While the accumulation of report data over multiple reports and their automatic caching is mostly abstracted by VaRA-TS, the work that remains to implement our new database is the extraction of analysis data from a report file. VaRA-TS already provides many functions to extract the report data for various use cases, but we need a new implementation to receive the `CommitRepoPairs` (including their library names) from a `BlameReport`. For this purpose we created a function that takes a `BlameReport` and collects all blame interactions from its analyzed functions. This function returns the extracted data as a mapping that is similar to the structure of a blame interaction within a `BlameReport`. We use this mapping within the database definition to build dataframe rows that mirror the relation of the `base-hash CommitReporPair` to each `CommitReporPair` of the individual `interacting-hashes` within a blame interaction. To prevent redundant entries, we additionally add the `amount` of the blame interaction to every corresponding blame interaction row. Using the example of the prior described blame interaction (line 588-592) of Listing 2.4 results in 2 dataframe rows that are shown in Figure 3.1. As illustrated in the example database, the `base-hash` (`5ef892a`) and its corresponding library name (`gzip`) is used for both rows and the `interacting-hash` of the first row is `48e0ba6` with library name `gnulib` and `amount 2`, and the `interacting-hash` of the second row is `95ace54` with library name `gzip` and `amount 2`. Since both entries originate from the same `BlameReport` file, they are denoted with the same `revision` hash, for which we have chosen `8d506e2` for example.

	revision	base_hash	base_lib	inter_hash	inter_lib	amount
0	8d506e2	5ef892a	gzip	48e0ba6	gnulib	2
1	8d506e2	5ef892a	gzip	95ace54	gzip	2

Figure 3.1: Example database layout that stores blame library interactions with their corresponding revision, base hash, base library name, interacting hash, interacting library name, and amount.

Finding blame interactions within a report file that have exactly the same `base-hash` and `interacting-hashes` would lead to duplicated entries. Therefore, the blame interaction gets only added once in the database, with the `amounts` of the identical blame interactions being added together. The proposed layout is able to store very specific information about the interacting commits within every blame interaction of a `BlameReport`. Furthermore, we are able to reuse the cached database if the underlying `BlameReport` data does not change. To efficiently access all `BlameReport` data that we need for our later visualizations, we have to implement another database layout that slightly varies from the one we described before. This database layout introduces the columns `degree` and `fraction`, which calculate and store the `degree` and `fraction` value of a blame interaction as explained in Section 2.4.2. Database entries that would result in rows with the same base library name, interacting library name, and `degree` within a report, are saved only once and their `amounts` are added together. This database layout does not save the commit hashes of the extracted `CommitRepoPairs`, but relates their library suffix to their corresponding `degree`, `amount` and `fraction`. This layout allows us to additionally access the `degree` and `fraction` of blame library interactions and prevents redundant computations in the visualization step by saving the already computed `degrees` and `fractions`. An example of this database layout is depicted in Figure 3.2.

	revision	base_lib	inter_lib	degree	amount	fraction
0	38ae6a4	gzip	gzip	13	17	0.133008356
1	38ae6a4	gnulib	gzip	10	6	0.001973073
[...]	[...]	[...]	[...]	[...]	[...]	[...]
874	598fbc2	gnulib	gnulib	4	303	0.035248953
[...]	[...]	[...]	[...]	[...]	[...]	[...]

Figure 3.2: Example database layout that stores blame library interactions with their corresponding revision, base library name, interacting library name, degree, amount, and fraction.

We conclude with our last database layout that is similar to the layout represented in Figure 3.1, but saves the difference between `BlameReports`. By comparing two `BlameReports`, we notice that blame interactions can change in, e.g., their `amount`. The information about changing `amounts` of blame interactions is important to us, as it facilitates our understanding of the blame interaction's development. We define blame interactions that change from a

`BlameReport` to the next successive `BlameReport` of a unique and total order as *`blame diff interactions`* and save the differences of their `amounts` in our `blame diff database`.

3.3 IMPLEMENTATION OF VARA-TOOL-SUITE'S LIBRARY INTERACTION PLOTS

Visualizing analysis data makes its interpretation easier and allows one to represent information in an intelligible and condensed way. This section introduces the implementations of our plots that are able to represent blame interactions between multiple libraries. First, we describe our Fraction Plot, which represents an overview of the distribution of ingoing and outgoing library interactions over a range of specified project revisions. Second, we explain the details of our interactive Sankey Plot that we use to get a deeper insight in the library interactions of specific project revisions. Third, we illustrate a new implementation of the Degree Plot that visualizes the degree of blame library interactions between 2 libraries over multiple project revisions. Last, the concept of a detailed blame interaction plot is implemented in the Graphviz Plot that represents interactions of specific project revisions on the level of their interacting commits.

3.3.1 *Fraction Plot*

To help researches analyze the proportions of blame interactions between libraries, we implemented the *Fraction Plot* in VaRA-TS.
This plot is used in combination with a `CaseStudy` file to illustrate the distribution of blame interactions between multiple libraries over time. Furthermore, the plot is split into 2 subplots, where the upper part represents the `outgoing interactions`, and the lower part shows the `ingoing (incoming) interactions` of the interacting libraries for each of the project revisions. We define an *`outgoing interaction`* of a commit as a blame interaction that originates from this (base) commit and targets another (interacting) commit. Correspondingly, we define an *`ingoing (incoming) interaction`* of a commit as a blame interaction that originates from another (base) commit and targets this (interacting) commit. The plot is used to facilitate the identification of all interacting libraries of the analyzed project, while providing information about their `fraction ratio`.
We implemented the `Fraction Plot` with the Matplotlib library as represented in Figure 3.3. The x-axis consists of the truncated commit hashes originating from the selected `CaseStudy`. These commit hashes (revisions) are in a total and unique order that is similar to the output order of the `git log` command. The y-axis is split into two subplots (`axes`) with individual y-axes that relate to the `fraction ratio` of `outgoing` and `ingoing` interactions. Each subplot contains a legend that maps the library names to the color that is used to mark the area of the resulting *`stackplot`*. This plot allows us to see the distribution of all blame interactions among the libraries. As the example shows, the values that are denoted as stacked areas always add up to 1. That means that the visualized libraries together take part in all found blame library interactions. We implemented this plot by creating a class that inherits from the `Plot` interface of VaRA-TS, which allows us to easily access the saved BlameReport data and facilitates the implementation of its `view` and `save` functionality. To generate a plot, e.g., the `Fraction Plot`, a user of VaRA-TS uses the `vara-plot` tool, which takes several arguments, like the path to the `CaseStudy` file and the name of the concrete plot.

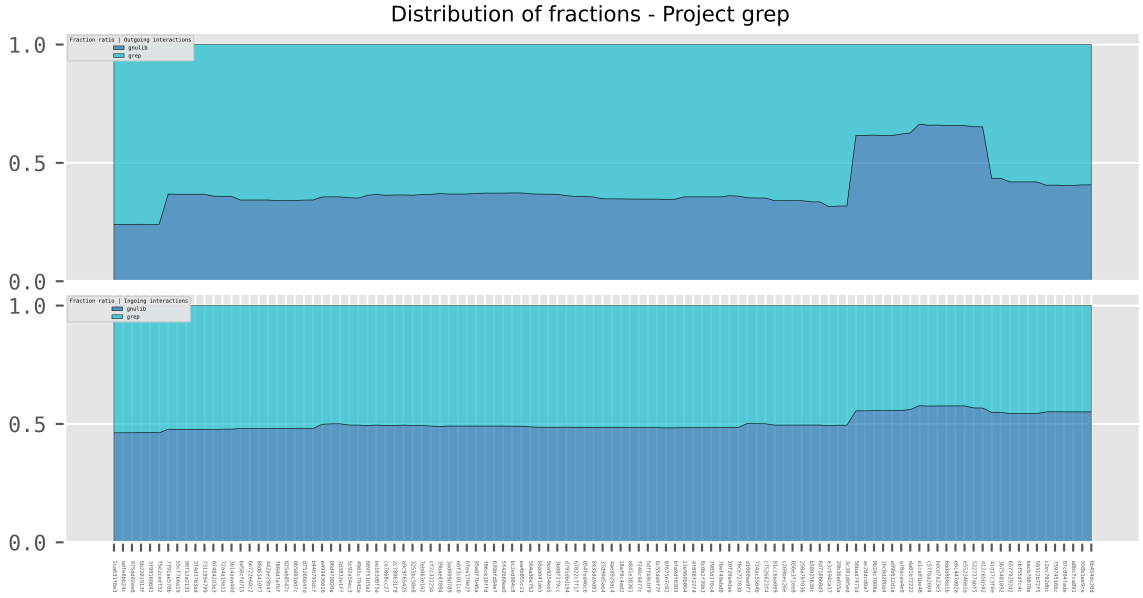

Figure 3.3: Example Fraction Plot of the project GNU grep over 109 revisions showing the distribution of fractions that stem from blame interactions between the libraries grep and gnulib.

3.3.2 *Sankey Plot*

We continue with a plot that allows researchers to select a specific project revision and view the blame interactions between libraries in more detail. The idea behind the *Sankey Plot* is to visualize blame interactions between multiple libraries of one specific project revision in an easily interpretable way. Furthermore, the `degree` of a blame interaction is encoded into the plot to help a researcher understand with how many commits a base commit interacts.
An example `Sankey Plot` is shown in Figure 3.4, which illustrates the blame interactions between 2 libraries (grep and gnulib). The revision for which we created the plot is depicted in the top left corner of the plot. The left side of the plot depicts the library names of the `base-hashes` an interaction originates from. The right side shows the library names of their corresponding `interacting-hashes`. Each color is assigned to a library to identify their blame interactions more easily. The color saturation of a blame interaction encodes the `degree` that we mentioned earlier, where the saturation increases with the `degree` level. Blame interactions can occur between commits of the same library, which is depicted in the example `Sankey Plot` by visualizing interactions between libraries that have the same name on the left and right side. The width of a blame interaction line represents its `fraction` value, which is the proportion of the interaction's `amount` relative to the total `amount` of all interactions. We implemented this plot within VaRA-TS using the Python library of Plotly, which allows us to view this plot also interactively in the browser. Users of VaRA-TS can generate this plot in two different ways. By passing the `-view` argument to the `vara-plot` tool, they can select a specific project revision to render the plot interactively in the browser. This allows a user to move the library bars vertically to rearrange the blame interactions. Furthermore, additional information is displayed for blame interactions over which the mouse curser hovers. This extra information contains the concrete values of

a blame interaction's `fraction` ratio in percent and its `degree`. Users that are interested in multiple `Sankey Plots` over a range of revisions can pass a corresponding `CaseStudy` file and omit the `-view` flag. This results in the generation of individual plots for every revision of the `CaseStudy` file. Plotting multiple revisions is especially useful to analyze the development of blame library interactions over time.

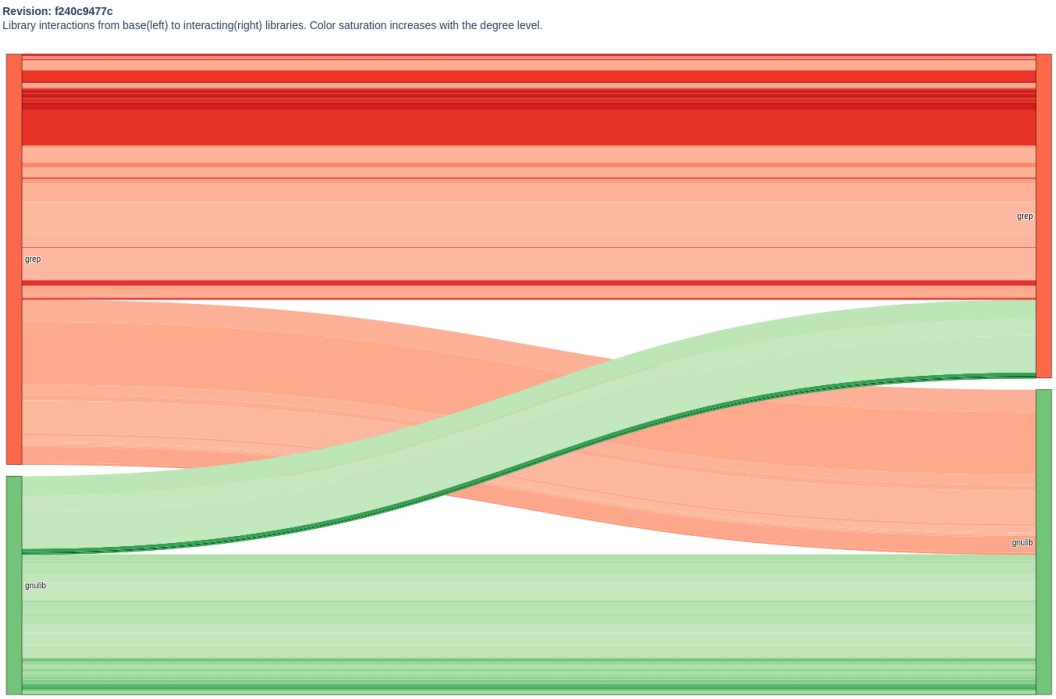

Figure 3.4: Example Sankey Plot depicting one revision of the project GNU grep by showing the blame interactions between the libraries grep and gnulib with different color saturations indicating different interaction degrees.

3.3.3 *Degree Plot*

We introduce our *Degree Plot* of VaRA-TS that allows researchers to get a more detailed insight in the proportions of different `degree` levels of blame interactions that stem from 2 specified libraries within a project. Using one of the proposed plots from the prior sections facilitates the identification of all interacting libaries. However, the blame interactions of some libraries might be of special interest to a researcher and appear more important than others. To filter these blame interactions and visualize the ratio of their different `interaction degrees` in more detail, we implemented the `Degree Plot`.
With this plot we can depict the development of blame interaction `degrees` and their ratio over a range of revisions. In comparison to the prior `Degree Plot`, we are not limited to interactions between commits of the same repository, but can choose the libraries we are interested in. Furthermore, a user of VaRA-TS can select one of the chosen libraries as a base library (`base_lib`) and the other one as interacting library (`inter_lib`). This results in a

plot that only considers blame interactions, where base commits stem from the selected base library and the corresponding interacting commits stem from the selected interacting library. As illustrated in Figure 3.5, we analyze the project `grep`, where `grep` is selected as base library and `gnulib` is chosen as interacting library. The x-axis labels of the `Degree Plot` mark the analyzed revisions of the passed `CaseStudy` file. The y-axis represents the `fraction` values of the different `degrees` within a revision. Similar to the Fraction Plot, we implemented this `stackplot` with the Matplotlib library that consists of stacked areas, where each area with a certain color denotes the `fraction` value of blame interactions that share the same interaction `degree`. The legend shows the found `interaction degrees`, which are mapped to colors that relate a certain `degree` level to the area that denotes its `fraction` ratio. The `fraction` value in this plot refers to the `amount` of blame interactions that share the same `degree`, which is divided by the total `amount` of all blame interactions that were found between base commits of the selected base library and interacting commits of the selected interacting library. To visualize the `degrees` and their `fraction` ratio development over time, users of VaRA-TS have to pass a `CaseStudy` file to the `vara-plot` tool. Furthermore, one has to specify the `base_lib` and `interacting_lib` after the plot name to filter the blame interactions of interest.

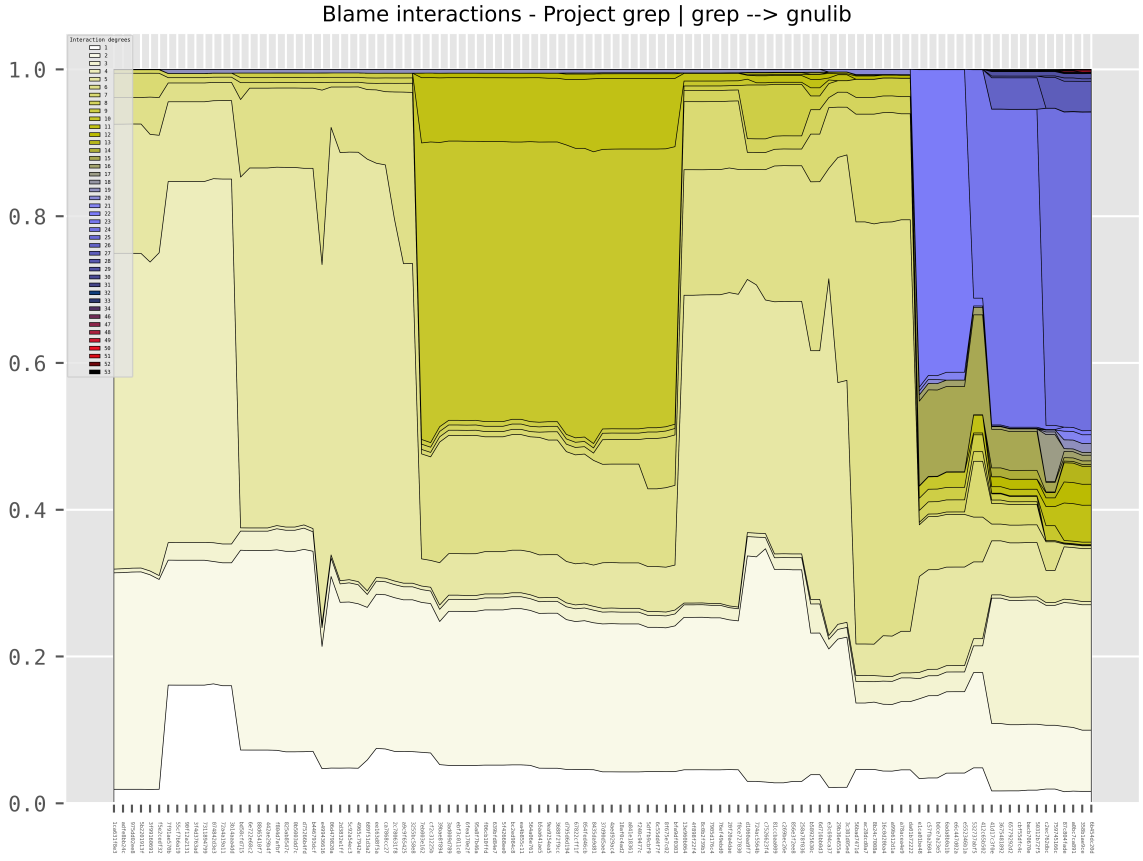

Figure 3.5: Example Degree Plot of the project GNU grep over 109 revisions showing the partition of degrees that stem from blame interactions, where the base hash originates from the library grep and the interacting hashes from library gnulib.

3.3.4 *Graphviz Plot*

To take an in depth look into the blame interactions of one specific project revision, we introduce our *Graphviz Plot* of VaRA-TS. This plot visualizes every blame interaction including its interacting commits as directed graph. The interacting commits are grouped by their libraries from which they originate. Building these clusters of commits based on their library names facilitates the identification of blame interactions within libraries or between them. To visualize blame interactions in detail, we represent the interacting commits as nodes and the interactions between them as directed edges. The direction of an edge mirrors the relation of base and interacting commits, where the edge starts from the blame interaction's `base-hash` and points to its `interacting-hash`. We implemented this plot using the Graphviz library and generated an example `Graphviz Plot` that is shown in Figure 3.6. The top middle of the plot states the chosen project revision (`bd693d7bc2`). Commits are grouped by the library they are originating from, which is illustrated as red rectangle around each `cluster`. The number near an edge indicates the `amount` of times this exact blame interaction was found. Users of VaRA-TS can enable or disable the `amounts` represented as edge weights, or select a threshold that removes edges from the graph that have a weight less than the threshold. Depending on the size and desired layout of the `Graphviz Plot`, it is useful to choose different layout engines. Therefore, users of VaRA-TS can choose between several layout engines, such as `dot` and `fdp`. While `dot` can arrange smaller numbers of nodes and edges very clearly, it takes much more computation time, unlike `fdp`. To counter the disadvantage of `fdp`'s somewhat cluttered rendering of large graphs, we pass additional graph attributes to `fdp`, namely `splines=True`, `overlap=False`, and `nodesep=1`. These attributes prevent edges from crossing nodes, inhibit node overlaps, and specify the distance between edges that interact with the same node. This improves the plot's readability, while preserving most of `fdp`'s performance benefits. Furthermore, one is able to either visualize the blame interactions of one revision, as already shown in the example plot, or generate a plot of the `blame diff interactions` that result from comparing the current revision with its previous revision. These `blame diff interactions` can either be plotted on top of the current blame interactions or being visualized as standalone plot. To reduce the size of a `Graphviz Plot` to the blame interactions of a specific commit, a user is able to pass a commit by its hash to the plot. This leads to the generation of a plot where only the blame interactions are shown that contain the chosen commit as base or interacting hash. Additionally, one can either visualize the blame interactions of one revision by passing the `-view` flag to the `vara-plot` tool and specifying a concrete project revision, or omit the flag and pass a `CaseStudy` file, which creates individual plots for every of its contained revisions.

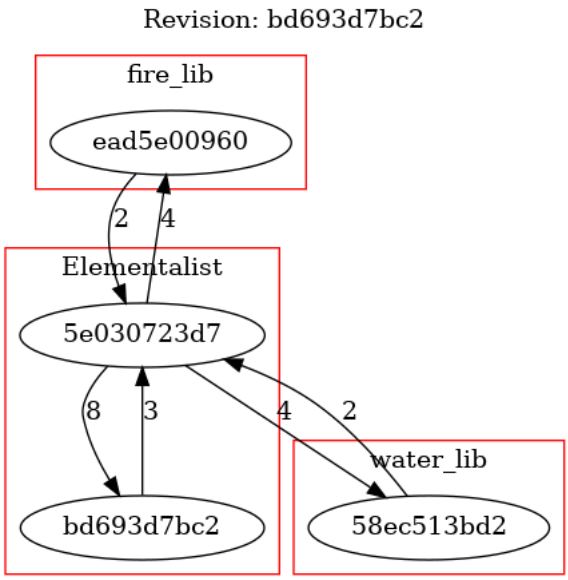

Figure 3.6: Graphviz Plot depicting one revision of the demo project Elementalist, showing the blame interactions between commits that are grouped into cluster subgraphs of Elementalist and its libraries fire_lib and water_lib.

4 EVALUATION

In the previous chapter, we described how we use blame interaction data between multiple libraries and how we build new visualizations in VaRA-TS to illustrate these blame library interactions. In this chapter, we evaluate how our visualizations improve the analyzability of library interactions in VaRA-TS by running them on a set of Case Studies. Therefore, we evaluate the following research questions.

- **RQ1**: Are our proposed illustrations useful to visualize blame interactions between multiple libraries?
- **RQ2**: Are our proposed illustrations useful to visualize the development of blame library interactions over time?

An example order in which our visualizations can be created to move from a high level overview to the details of blame interactions is shown in Figure 4.1, although this order can be changed as desired by the user.

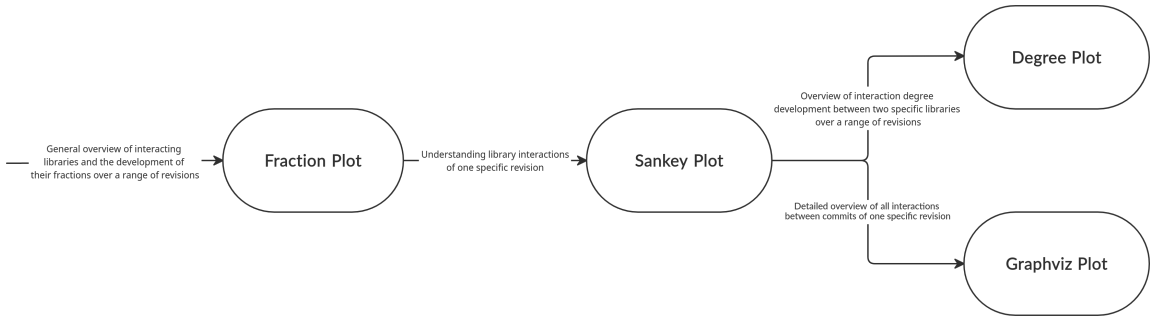

Figure 4.1: Possible order of visualizations.

4.1 CASE STUDIES

This section introduces the case studies that we use to evaluate our different blame library interaction plots. The first `CaseStudy` is our small demo project Elementalist that represents a project with multiple interacting libraries. Our second `CaseStudy` is the real-world software project GNU Grep, which allows us to sample many revisions of its commit history from a uniform distribution. Last, we use the real-world software project GNU Gzip and focus on newer revisions of its commit history by sampling from a half-normal distribution.

4.1.1 *Elementalist*

The implementations we described in the previous chapter require that we continually test our source code and its outcomes for validity. To build a controlled environment where we

are able to evaluate the results of our implementations on a small sample set, we created the demo project `Elementalist`.
`Elementalist` is desgined to include 3 external libraries, called `fire_lib`, `water_lib` and `earth_lib` to simulate blame library interactions between `Elementalist` and the other libraries, and additionally between `earth_lib` and `water_lib`. The source code of the main program and the libraries is written in C++. To ensure that the creation of the necessary project files is reproducible, we use the automation tool *Ansible*[1]. Ansible facilitates the repeated setup of our project files and their modification with an executable configuration file. With Ansible, we build our main program `Elementalist` and its external libraries `fire_lib`, `water_lib`, and `earth_lib` as repositories and create commits that result in blame interactions between them. Then, we incorporate `Elementalist` and its libraries in VaRA-TS as project, where we define `Elementalist` as main repository and the 3 libraries as `GitSubmodules`. After this setup, we are able to continually run experiments, like the `GenerateBlameReport` experiment, on our demo project to create a few example BlameReports that we use to test our databases and visualizations. For our evaluation, we select two project revisions, where the first revision (`e64923e69e`) results only in blame interactions between the main repository `Elementalist` and the libraries `fire_lib` and `water_lib`. The second revision (`5e8fe1616d`) adds a commit, which introduces the `earth_lib` library that interacts with `Elementalist` and additionally with the library `water_lib`. We choose these revisions as they allow us to evaluate two important cases. The first revision corresponds to the case where blame interactions occur only between the main program and its libraries. The second revision illustrates the case where blame interactions also arise between the libraries themselves.

4.1.2 *GNU Grep*

GNU `grep`[2] is a command line tool written in C that is used to find text patterns in a file and print the lines that match that pattern. We choose `grep` as `CaseStudy` since it includes an additional library, called `gnulib`. Furthermore, the large number of commits in `grep`'s commit history allows us to sample many revisions for our analysis. In total, we sample 109 revisions from a uniform distribution.

4.1.3 *GNU Gzip*

GNU `gzip`[3] is a popular program for file compression and decompression written in C. We choose `gzip` as `CaseStudy` since it includes the same library as our prior `CaseStudy` GNU Grep, which is `gnulib`. Since `gzip` and `grep` both include `gnulib` as library, we can compare the two projects more easily. In total, we sample 24 revisions from `gzip`'s commit history from a half-normal distribution. The choice of this distribution allows us to sample primarily more recent revisions from the commit history with comparatively smaller gaps between the commits, as opposed to sampling from the uniform distribution of the `CaseStudy` `grep`.

1 Available online at https://www.ansible.com/; visited on February 19th, 2021.
2 Available online at https://www.gnu.org/software/grep/; visited on February 27th, 2021.
3 Available online at https://www.gnu.org/software/gzip/; visited on March 2nd, 2021.

4.2 OPERATIONALIZATION

This section describes the necessary steps one has to take to generate our blame interaction plots that we use to answer our research questions RQ1 and RQ2. To visualize a project over multiple revisions that is already incorporated in VaRA-TS, we first have to create a `CaseStudy`, like the Case Studies of the prior section. To sample multiple revisions and store them in a `CaseStudy` file, we use the `vara-cs gen` command of VaRA-TS and define the location of our resulting `CaseStudy`, choose our sampling method, which is either the `UniformSamplingMethod` or the `HalfNormalSamplingMethod`, pass the name of the project we are interested in, and specify the number of revisions we want to sample. Next, we run Benchbuild to generate a BlameReport for each of the revisions we sampled in our `CaseStudy`. For that, we specify our `CaseStudy` in VaRA-TS and run the command `benchbuild run -E GenerateBlameReport our-project`, where '`our-project`' is the name of the project we choose to analyze. Finally, we choose one of our blame interaction plots and pass it to the `vara-plot` tool of VaRA-TS with the additional arguments we want to set for this plot. Depending on the plot, this results in the creation of either one plot file that considers all revisions of the specified `CaseStudy` or in individual plot files for every revision.

4.3 RESULTS

In this section, we evaluate our visualizations on the prior defined Case Studies and gather the results to answer our research questions RQ1 and RQ2 in Section 4.4. We start by analyzing the project Elementalist, then we proceed with GNU Grep and finish with GNU Gzip.

4.3.1 *Elementalist*

We begin our evaluation by generating two `BlameReports` based on our Elementalist `CaseStudy` with the `GenerateBlameReport` experiment of VaRA-TS. We select the revisions `e64923e69e` (1) and `5e8fe1616d` (2) from our `CaseStudy` for our evaluation. Revisions that are of special interest to us are assigned a number to reference them more easily.
By looking at the `Fraction Plot` in Figure 4.2, we immediately notice the differently sized and colored areas that illustrate the `fraction ratios` of the main program and its interacting libraries. The `fractions` of `Elementalist` (color dark blue) take the biggest part of the `outgoing interactions`, as well as the `ingoing interactions` on both revisions. This predominant area results from our design of the main program of `Elementalist` that contains most of the function calls, which result in bidirectional blame interactions between commits of the main program itself and the other libraries. We further notice that the `fraction` value of the library `earth_lib` (color cyan) is 0 at revision `e64923e69e` (1) for both subplots and around 0.1 at revision `5e8fe1616d` (2). This means that blame interactions of `earth_lib` are first found in revision `5e8fe1616d` (2) and since both subplots show a `fraction` value higher than 0 at this revision, we know that this library contains `outgoing` as well as `ingoing interactions`. As the `earth_lib` library is added, we notice a decrease in the `fraction` values of all other libraries, since the total `amount` of blame interactions increases with the

addition of the `earth_lib` interactions.
While we now know what libraries interact in general and what proportions of blame interactions these libraries have from a high level perspective, we cannot identify which libraries are specifically interacting. This leads us to the Sankey Plot, which we use to depict the interactions between libraries more precisely.

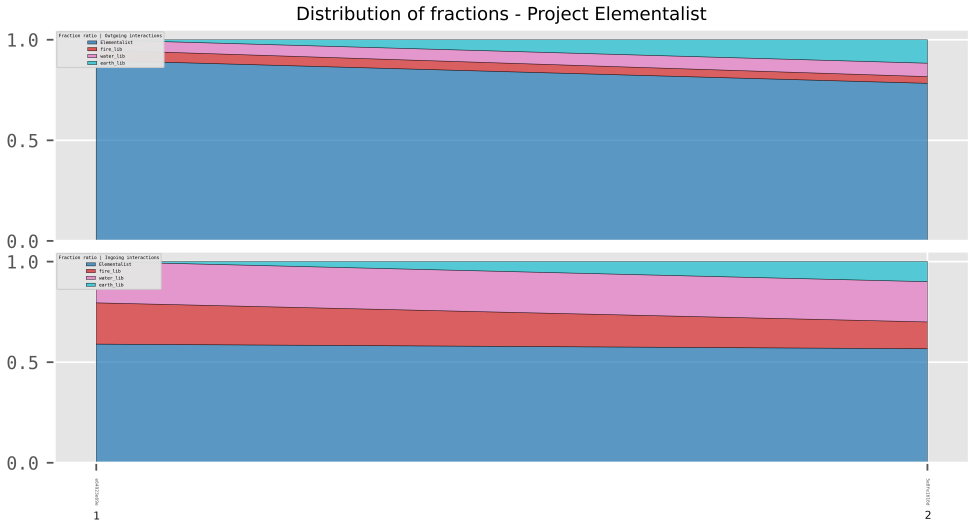

Figure 4.2: Fraction plot of project Elementalist over 2 revisions.

We continue with our `Sankey Plots` for the revisions `e64923e69e` (1) and `5e8fe1616d` (2), which are shown in Figure 4.3 and Figure 4.4. Starting with the plot of revision `e64923e69e` (1), we first notice that all interacting libraries, which we found with the prior plot at this revision, are also depicted in this plot as vertically aligned bars of different colors. The left side shows the base libraries `Elementalist`, `fire_lib`, and `water_lib`, meaning that all base commits of the detected blame interactions stem from these libraries, the right side denotes the libraries that contain the corresponding interacting commits. Blame interactions are visualized as lines between the library bars and are drawn in the color of their base library. We can approximate the `fraction` value by looking at the width of a blame interaction, which increases with its corresponding `fraction` value. Furthermore, we notice the different color saturations in, e.g., the interaction from `Elementalist` to `Elementalist`, which denotes its interaction `degree`. Moreover, we can inspect the precise `fraction` and `degree` value, by rendering the plot interactively in the browser. Looking at the middle of the `Elementalist` interaction with the highest degree, we can trigger an information box by hovering over it with our mouse, which is depicted in our example as small white frame. This box shows the blame interaction's exact `fraction` ratio of 7.69% and its degree of 2. In contrast to our `Fraction Plot`, we are now able to determine the specific libraries that are interacting and get a clearer image of their interaction proportions. Now we can see that for the revision `e64923e69e` (1) only `Elementalist` interacts with `fire_lib` and `water_lib`. Furthermore, we immediately notice that blame interactions between `Elementalist` and the other libraries are bidirectional, which means that some commits of `Elementalist` interact with commits

of `fire_lib` and `water_lib` as base library and vice versa.

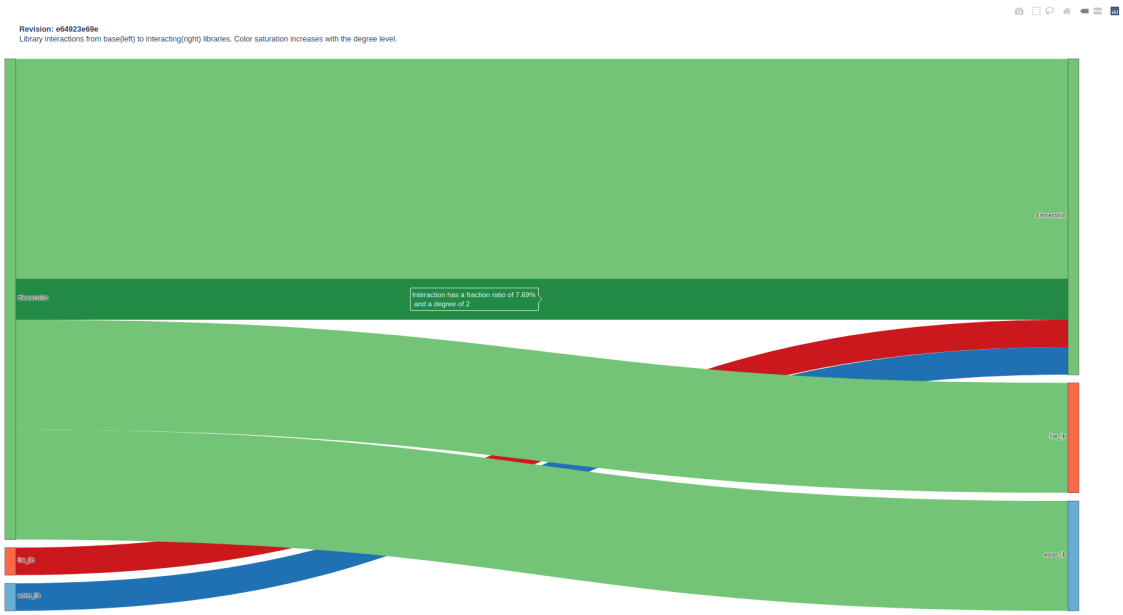

Figure 4.3: Sankey plot of project Elementalist of revision e64923e69e.

Let us now consider the second plot of revision `5e8fe1616d` (2), which is illustrated in Figure 4.4. As mentioned in a prior section, this revision adds another commit to `Elementalist`, which adds function calls of `earth_lib` into the main program that in turn call a function of the library `water_lib`. We can now inspect the blame interactions that are introduced by this commit by comparing the two plots. The newly added library `earth_lib` (color gray) can be easily identified as we notice that there are now 3 library bars and one main program (`Elementalist`) bar on the left and right hand side. As the blame interactions between `Elementalist` (color green) and all other libraries are detected, they get illustrated as green lines from the `Elementalist` bar to every of the library bars, including the new library `earth_lib`. We further notice that `earth_lib` also interacts with the library `water_lib` (color blue), which means that interactions between the libraries themselves are also detected and correctly drawn. Comparing the two plots, we also notice a change in the color of the `Elementalist` to `Elementalist` interaction. The darkened green indicates an increase of the interaction `degree` compared to the prior revision `e64923e69e` (1). This is additionally depicted in the hover info, which allows us to determine an increase from the interactions `degree` 2 (`e64923e69e` (1)) to `degree` 3 (`5e8fe1616d` (2)).
With this plot, we are able to determine blame interactions and their corresponding libraries more precisely than with the `Fraction Plot`. The comparison of the interaction `degree` that we did by hand proves to be inefficient when we already know the libraries of the blame interactions we are interested in. Therefore, to visualize the `degree` development of blame interactions between two known libraries and over multiple revisions, we use our Degree Plot. This allows us to illustrate the changes of interaction `degrees` over time, similar to the

development of `fractions` in the `Fraction Plot`.

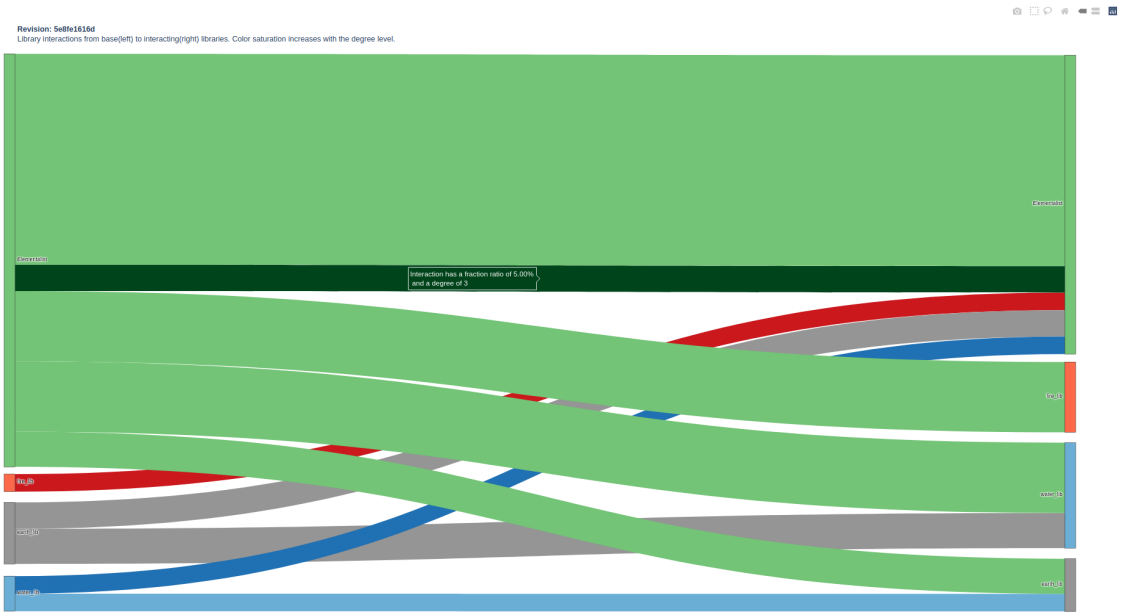

Figure 4.4: Sankey plot of project Elementalist of revision 5e8fe1616d.

We proceed with the `Degree Plot` that is depicted in Figure 4.5. We choose `Elementalist` as base library and also as interacting library, since we saw in the prior plot that the `degree` of a blame interaction between commits of `Elementalist` changed. At first glance, we notice that blame interactions with the `degree` 1 make up the majority in both revisions, which is illustrated as white area. Furthermore, we are able to see that revision `e64923e69e` (1) already contains blame interactions of `degree` 2. We can spot an interesting development of the represented degrees. When we compare the two revisions, we notice that as the revision `5e8fe1616d` (2) is introduced, the `degree` increases from 2 to 3, while the proportion of blame interactions with `degree` 1 increases from around 0.85 to about 0.9.
Now that we know what libraries are interacting and how the `fractions` and `degrees` of their blame interactions develop, we can take an in depth look into the interactions of one specific revision using our Graphviz Plot.

Figure 4.5: Degree plot of project Elementalist over 2 revisions.

We can inspect the interactions of revision `5e8fe1616d` (2) in more detail by generating our `Graphviz Plot`, which is illustrated in Figure 4.6. This graph is drawn with the layout engine `fdp`, to which we pass the additional graph attributes mentioned in Section 3.3.4, which improve the general readability. We clearly see the interacting commits that are represented as hashes and the libraries they originate from as red rectangles. Blame interactions between `Elementalist` and its libraries, as well as the interactions between `earth_lib` and `water_lib`, can be determined by following the ingoing and outgoing edges. Since this graph is rather small, we choose to also display the `amount` of a blame interaction, as denoted by the edge labels. The detailed depiction allows us to determine that each library consists of only one interacting commit. Furthermore, we see that `Elementalist` contains 5 commits that either interact with each other or with the other libraries. Looking at the amounts of `Elementalist`'s interactions, explains its predominant `fraction` value within the `Fraction Plot`.

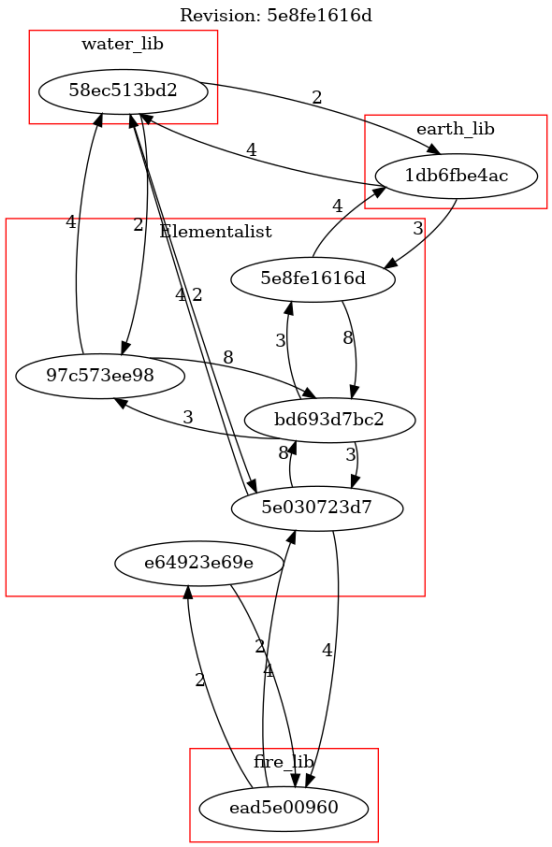

Figure 4.6: Graphviz plot of project Elementalist of revision 5e8fe1616d.

4.3.2 *GNU Grep*

We continue our evaluation by generating 109 `BlameReports` with our GNU Grep `CaseStudy`. We use the `CaseStudy` of the project `grep` as our first real-world example.
The large number of project revisions in our `CaseStudy` allows us to generate a Fraction Plot, which is depicted in Figure 4.7 that illustrates the development of `grep`'s `fraction ratios` over a larger period of time compared to the `Fraction Plot` of our previous evaluation. The commit date of the revisions ranges from December 2011 (`1ca631f8e3`) to January 2021 (`6b454dc20d`). We immediately notice a new dimension in the plot, which is shown at the bottom as subplot with its own y-axis. This subplot represents the *`code churn`* of the main repository at each revision. It shows the difference of code changes between two successive revisions, where the insertions are illustrated as green color and the deletions as red color. These code changes (`diff`) between revisions are calculated by using the *git diff* command, which we use to show the number of source code lines that changed between two commits. The y-axis represents the total number of line insertions for values bigger than 0 and the total number of line deletions for values less than 0. VaRA-TS already implements the `code churn` subplot and allows us to easily integrate it in every Matplotlib based plot, like our `Fraction Plot`. Furthermore, this subplot adds a prefix to each revision on the x-axis, where each prefix number stems from a total and unique order of all revisions. We are now able to measure the impact of a revision based on its code insertions and deletions, which helps us

to interpret the development of `grep` more precisely. Looking at the `code churn`, we notice that, e.g., revision `7f91ae570b` (1) introduces more than 1000 code insertions and even more code deletions. We interpret this nearly balanced change as a commit that moves more than 1000 lines of code to another location, which results in deletions for the lines where the code used to be and insertions for its new location. Besides smaller changes for nearly every of the shown revisions, we notice that revision `e499436616` (2) has a similar impact on `grep` as the prior mentioned one. As illustrated in the upper two subplots, we immediately see the two interacting libraries `grep` and `gnulib`. We benefit from the large number of project revisions, as we get a bigger picture of the libraries' `fraction` development. While we notice that the distribution of `fractions` of `ingoing interactions` varies rather little for both libraries, the ratio of their `outgoing interactions` changes noticeably. By looking at the revision `56bad7471d` (4), we see that `gnulib` now contains most of the `outgoing interactions`, where in all previous revisions `grep` makes up the largest part. This major change in the `outgoing interactions` could result from larger changes in the code that this revision introduces. Therefore, we look at the `code churn` at this specific revision and notice that this commit deletes many lines of source code. We can relate these two observations to each other and infer that the deletions occur in files of `grep` and result in a comparatively larger distribution of `outgoing interactions` of the library `gnulib`.

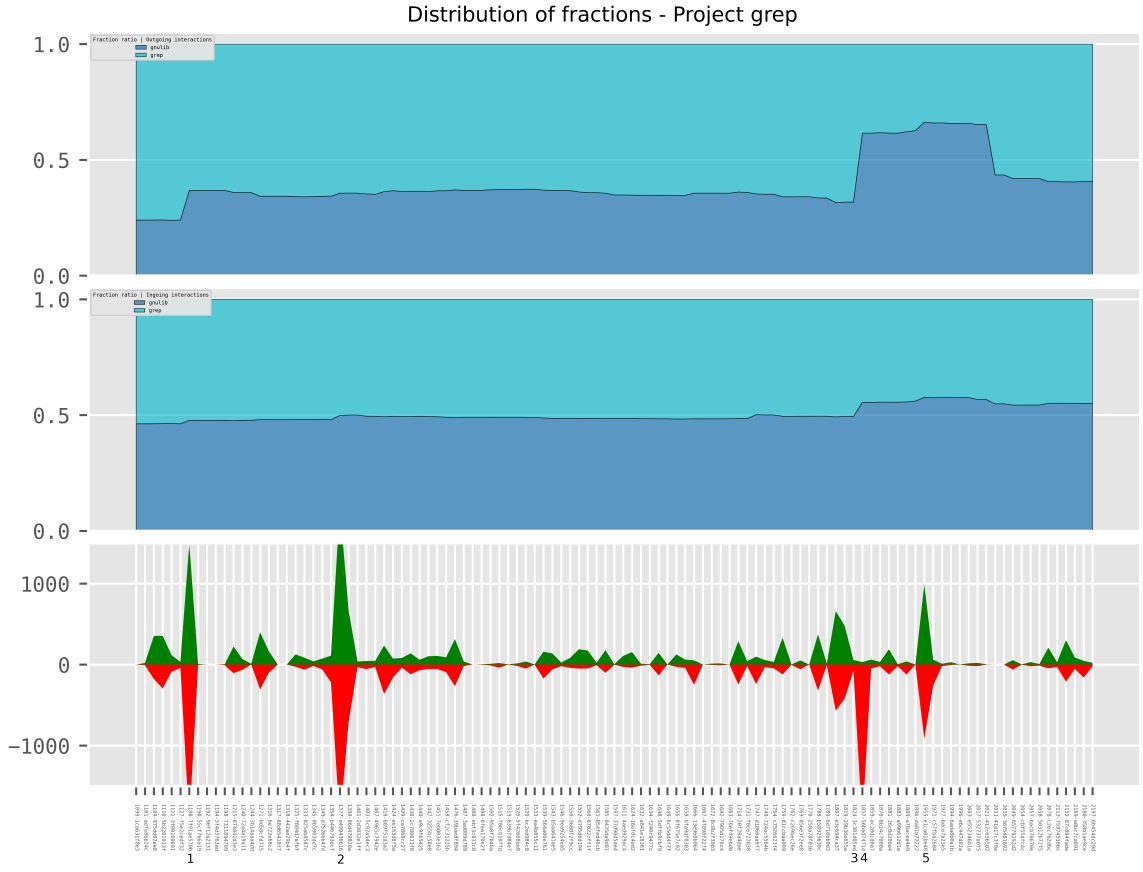

Figure 4.7: Fraction plot of project GNU grep over 109 revisions with its code churn.

We proceed our evaluation with two Sankey Plots, which are illustrated in Figure 4.8 and Figure 4.9. The plots depict the blame interactions between `grep` and `gnulib` in more detail, where the first plot visualizes the revision `3c381d05ed` (3) committed on September 2, 2016, which adds a new option for anchored searches in `grep`. The second plot of revision `56bad7471d` (4) committed on October 2, 2016, makes some small changes after a new release of `grep`. We choose these two revisions as we already noticed an interesting change of the `outgoing interactions` between these two in the prior described `Fraction Plot`. To view the changes between two or more successive `Sankey Plots`, we benefit from the internal numeration of each revision that we already see as prefix of the `Fraction Plot`'s revisions. The ascending numbers before each revision allow us to view and search generated plots, like the `Sankey Plots` according to their total order. Therefore, it is easy to view the predecessor of the revision `56bad7471d` (4), in which we are also interested in. Looking at both plots, we instantly see that both `grep` and `gnulib` interact as base and interacting library with each other. The first plot of revision `3c381d05ed` (3) shows that most of the blame interactions originate from library `grep`, which is denoted by the size of the red lines' width. Now we compare this distribution with the following revision's plot. We can clearly see that the bigger part of all blame interactions now originates from the library `gnulib` and not `grep` anymore. While we already noticed this development in our `Fraction Plot`, we are now able to inspect each blame interaction precisely and with a focus only on the revisions that are of interest to us. This development could be due to the new release of `grep` that got introduced somewhere between revision `3c381d05ed` (3) and `56bad7471d` (4).

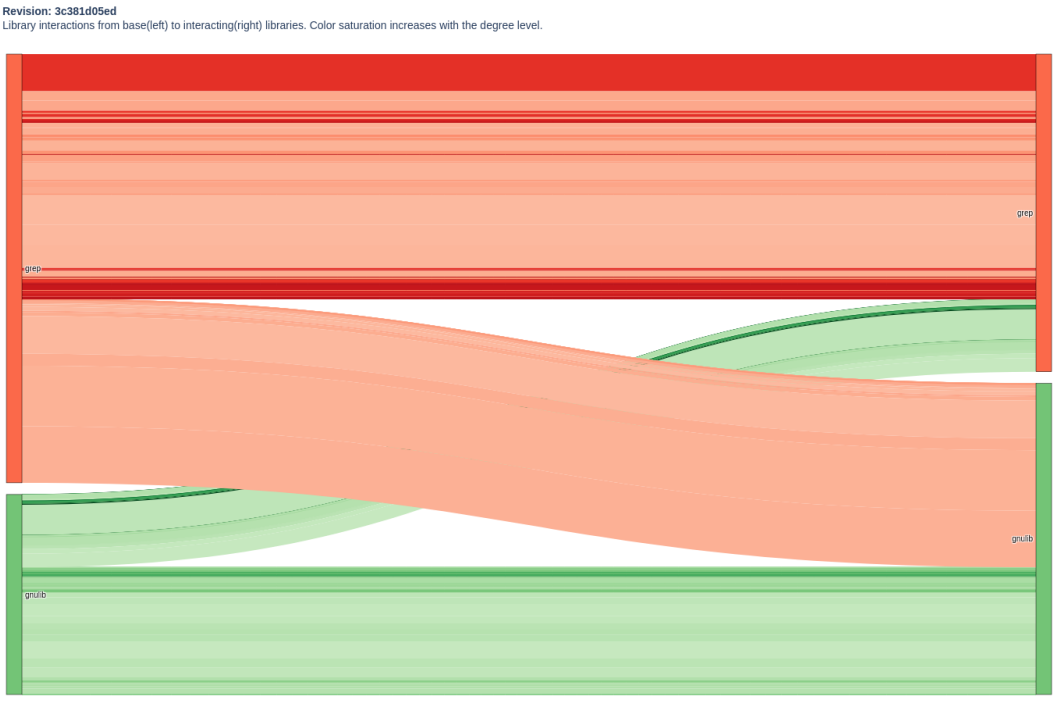

Figure 4.8: Sankey plot of project GNU grep of revision 3c381d05ed.

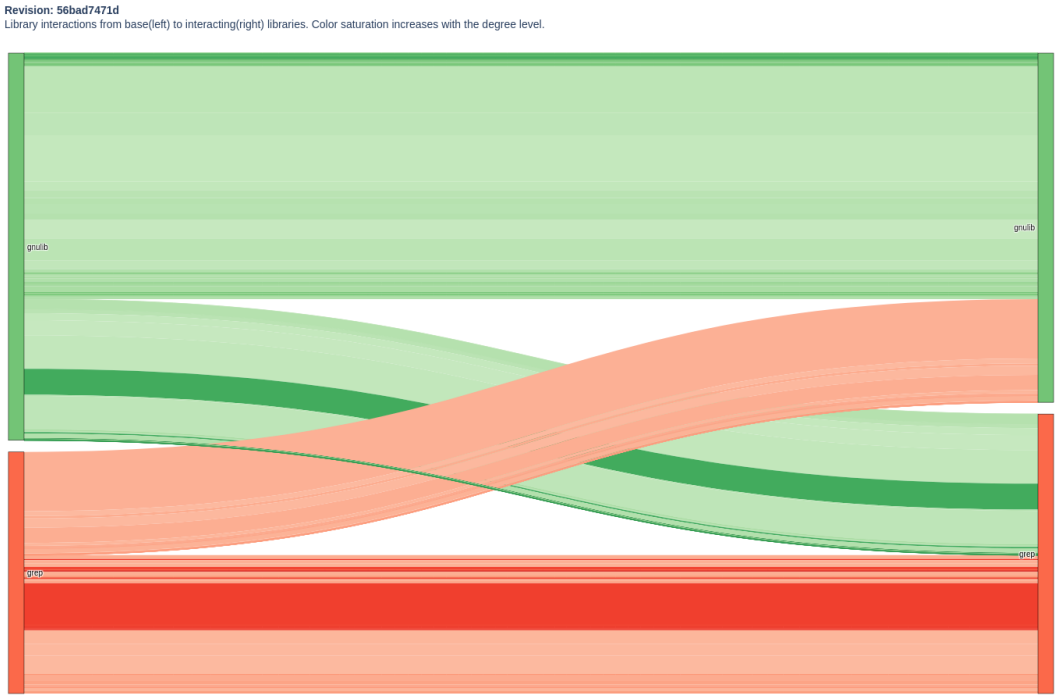

Figure 4.9: Sankey plot of project GNU grep of revision 56bad7471d.

We depict the development of `interaction degrees` between the chosen libraries `grep` (base library) and `gnulib` (interacting library) as Degree Plot in Figure 4.10. In contrast to our evaluation of Elementalist, we immediately notice the large number of `interaction degrees` in the plot's legend that reaches its peak at `degree` 53. Furthermore, we use our `code churn` subplot again to relate the impact of code insertions and deletions to the `degrees` of the upper subplot. We notice a general tendency of increasing `degrees` over time. Comparing the highest `degree` of the first revision with the highest `degree` of the last revision shows an increase in its `degree` of around 40. Moreover, the small changes of the `degrees` around revision `7f91ae570b` (1) and revision `e499436616` (2) affirm our assumption that the comparatively balanced changes of code insertions and deletions at these revisions arise from moving big code parts to another location. In contrast, revision `e1ca01be48` (5) with a similar `code churn` pattern seems to introduce lines of code that result in blame interactions with a relatively high `degree` of around 22.

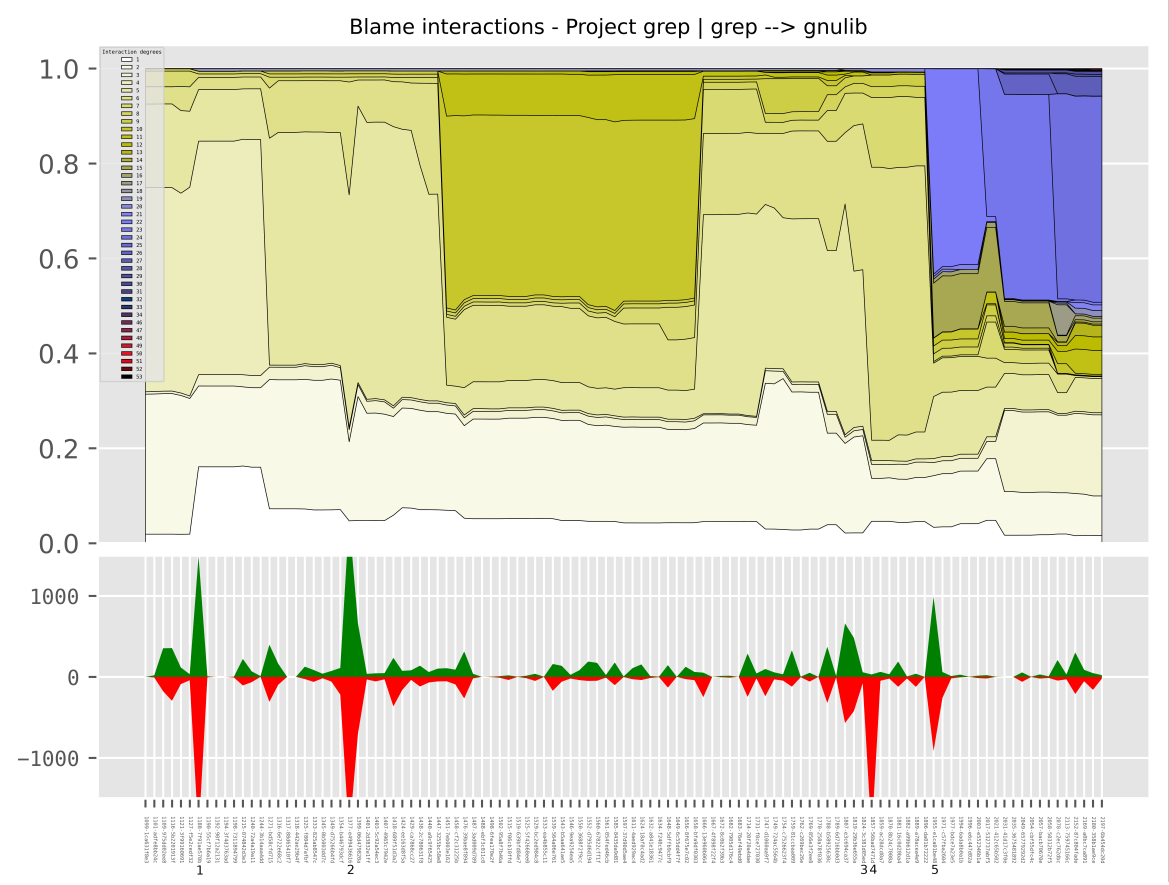

Figure 4.10: Degree plot of project GNU grep over 109 revisions with its code churn.

We continue with our last visualization for `grep`, which is illustrated in Figure 4.11 as Graphviz Plot, drawn with the layout engine `fdp`. The depicted plot shows all blame interactions between commits of the libraries `grep` and `gnulib` of the revision `adfe8bb24c` (second revision in `Fraction` and `Degree Plot`). We instantly notice that this visualization of only one `grep` revision is more crowded and less clearly than, e.g., the evaluated revision of our `Elementalist` project. `Grep` contains as one of our real-world projects many blame interactions that result in this rather complex view. Additionally, we see that a subset of the shown blame interactions are colored orange, which denotes that these are blame interactions that changed in their `amount` from the previous to the current revision, which we refer to as `blame diff interactions` (orange colored edges). While it is hard to interpret all blame interactions of this plot as a whole, we can use the `blame diff interactions` as indicator of how much impact the current revision has on the previous revision. For example, the large number of orange `blame diff interactions` tells us that the current revision has a rather big impact on the previous revision. Furthermore, this plot illustrates that most of the blame interaction changes this revision introduces, concern the library `gnulib` and not blame interactions between commits of `grep`. By zooming into the graph, we notice that the `blame diff interactions` contain two numbers near their edge. The first number is the `amount` of the current revision as already denoted by the labels of unchanged blame

interactions. The second number in brackets states the difference of the blame interaction's `amount` to the previous revision.

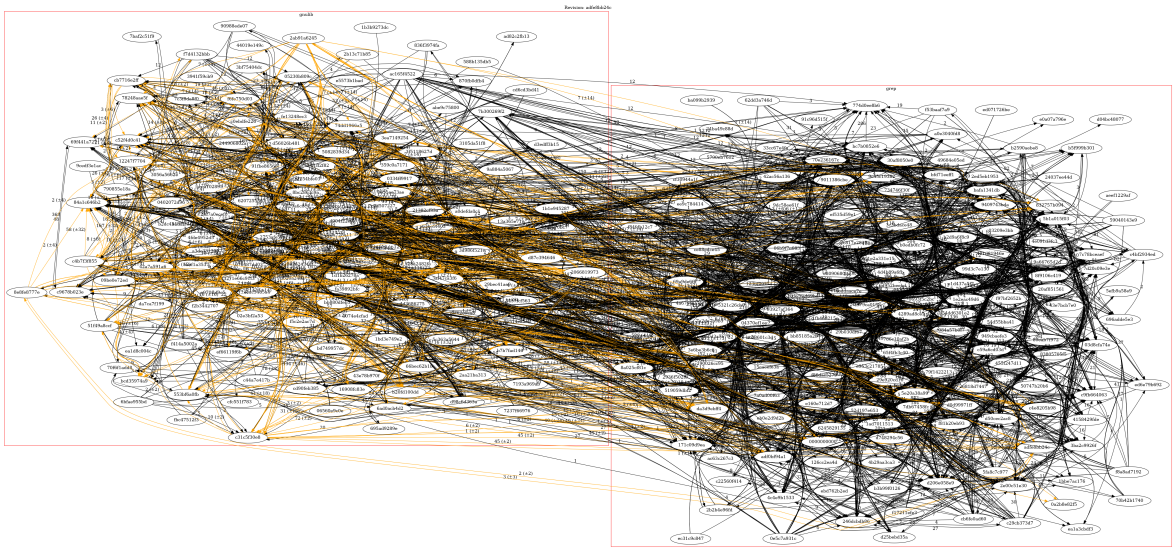

Figure 4.11: Graphviz plot of project GNU grep of revision adfe8bb24c.

4.3.3 *GNU Gzip*

We proceed with our GNU Gzip `CaseStudy` by generating 24 `BlameReports` for our evaluation.
The resulting Fraction Plot that is illustrated in Figure 4.12 shows that the interacting libraries `gzip` and `gnulib` have `outgoing` and `ingoing interactions`. We immediately notice the similar subplots of `outgoing` and `ingoing interactions` that represent a `fraction` distribution of around 75% for `gzip` and around 25% for `gnulib`. Furthermore, the represented plot shows no significant changes in the proportions of `fractions` in both subplots over time, which means that our revisions do not introduce source code commits that lead to significantly less or more blame interactions. As the `fraction` areas in both subplots for each of the libraries seem to be nearly identical, we infer that the largest part of the `outgoing interactions` of `gzip` are also its own `ingoing interactions`. This also applies for the library `gnulib`. Based on these observations we can interpret that either most of the blame interactions occur between commits of the same library and not between commits of different libraries, or that the distribution of blame interactions that occur between these libraries are nearly equal. Looking at the `code churn` at the bottom affirms our observation that the `fraction ratio`'s change over time is rather small, since no major source code changes are detected between the revisions. Even revision `7a6f9c932` (1) and `be0c5581e3` (2) that represent the biggest change of inserted and deleted source code lines do not change the `fraction ratio` noticeably. Due to the samples that origin from a half-normal distribution as mentioned in Section 4.1.3, we get more recent revisions that are relatively close in their commit time. Therefore, the changes in code are rather small compared to revisions that stem from a uniform distribution. Our `Fraction Plot` correctly shows only minor changes

in the `fraction` values because only small changes in code are made between the newer revisions.

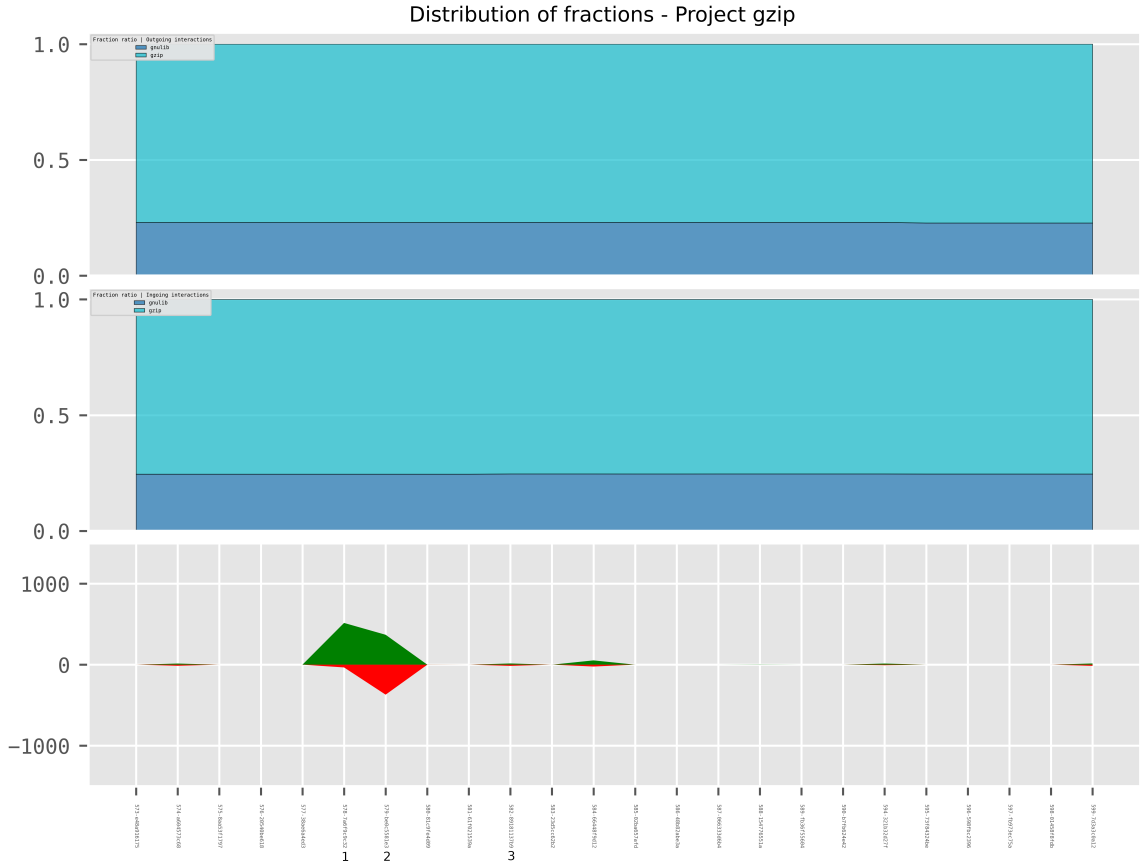

Figure 4.12: Fraction plot of project GNU gzip over 24 revisions with its code churn.

We proceed with our Sankey Plot by visualizing the revision `89181137b9` (3) that is shown in Figure 4.13. As illustrated in the plot, we see the blame interactions between `gzip` and `gnulib` as well as their different `degrees` denoted by their color saturation. Our first observation is that `gzip` and `gnulib` interact not only with each other, but also with themselves, where the blame interactions between commits of `gzip` build the biggest part of all interactions. Furthermore, with the detailed view of our `Sankey Plot`, we can now verify if one of our assumptions that we had in the prior `Fraction Plot`, regarding the similar `fraction` distribution between `outgoing` and `ingoing interactions`, is correct. Our first guess that there is no significant number of blame interactions between the libraries proves to be wrong, as we see that `gzip` interacts in noticeable `amounts` with `gnulib` and vice versa. In contrast, our second assumption turns out to be true, as we notice that the `amount` of blame interactions that stem from `gzip` and interact with `gnulib` is nearly identical to the `amount` of blame interactions that originate from `gnulib` and interact with `gzip`. The balanced `amount` of blame interactions between these libraries results in the `fraction ratios` of similar size in both of our `Fraction Plot` subplots.

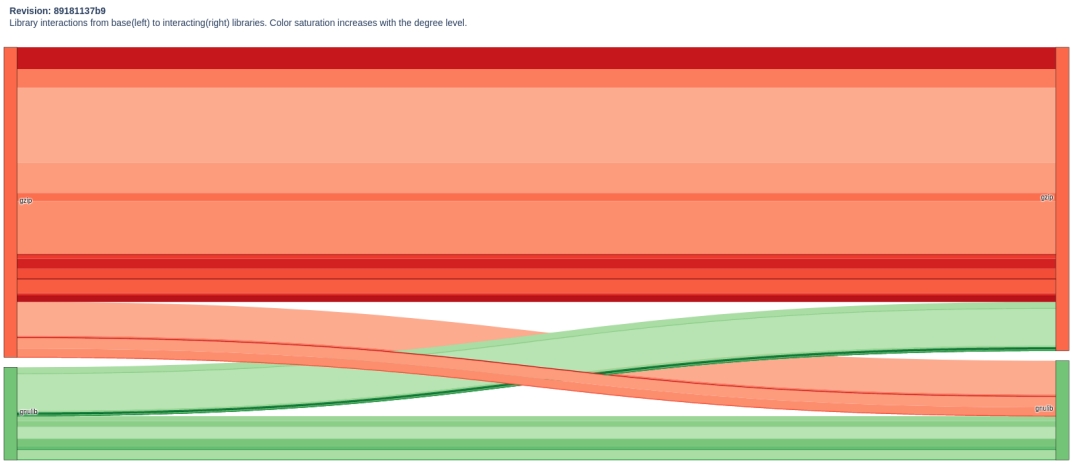

Figure 4.13: Sankey plot of project GNU gzip of revision 89181137b9.

To visualize the development of interaction `degrees`, we generate our Degree Plot, which is represented in Figure 4.14. We choose `gnulib` as base library and `gzip` as interacting library to illustrate blame interactions that origin from library commits and target interacting commits of the main program `gzip`. By looking at the plot, we instantly notice that similar to our `Fraction Plot`, the `degrees` and their `fraction ratio` do not change significantly over time. Furthermore, we see that the majority of blame interactions over all revisions has the `degree` 1, which is depicted as white area. The highest `degree` of all revisions is 14. By zooming into the plot at revision `89181137b9` (3), we notice a small change in the `fraction` distribution of the `degrees`. We have already investigated this revision in our prior `Sankey Plot` and use it in our next plot to get a more in depth look.

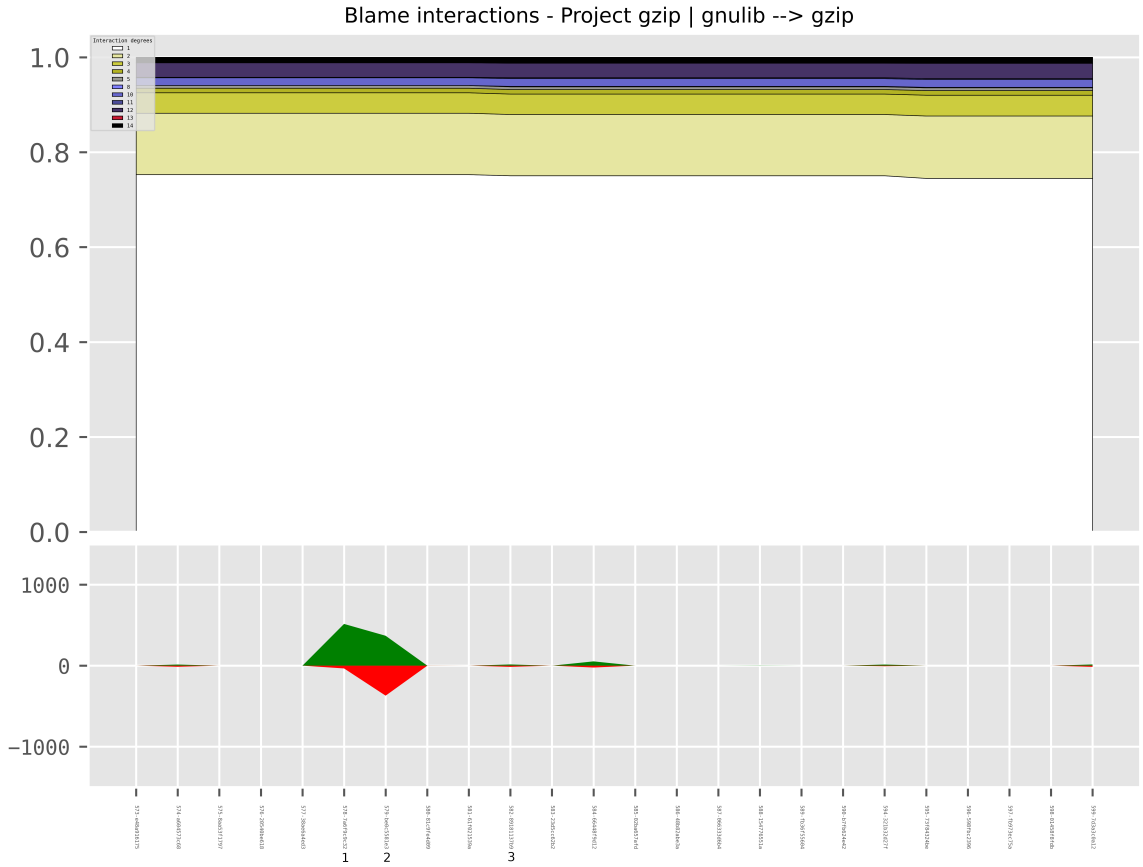

Figure 4.14: Degree plot of project GNU gzip over 24 revisions with its code churn.

We conclude our evaluation of `gzip` with a Graphviz Plot drawn with the layout engine `fdp` that is illustrated in Figure 4.15 and visualizes the revision `89181137b9` (3). We notice that this plot has a comparable complexity to the `Graphviz Plot` of the project `grep`, even though there are less blame interactions in total. All blame interactions between the commits of `gzip` and `gnulib` are represented with their `amount` near their edge. We again use the `blame diff interactions` illustrated as orange edges to find the blame interactions that changed in their `amount` from the previous revision to the current. We observe that the impact of the current revision on previous revision measured by their `blame diff interactions` is rather small, compared to the one we saw in the `Graphviz Plot` of the `CaseStudy grep`. This minor impact is expected, as we already know from the previous plots that the `fractions`, `degrees`, and code changes are hardly noticeable.

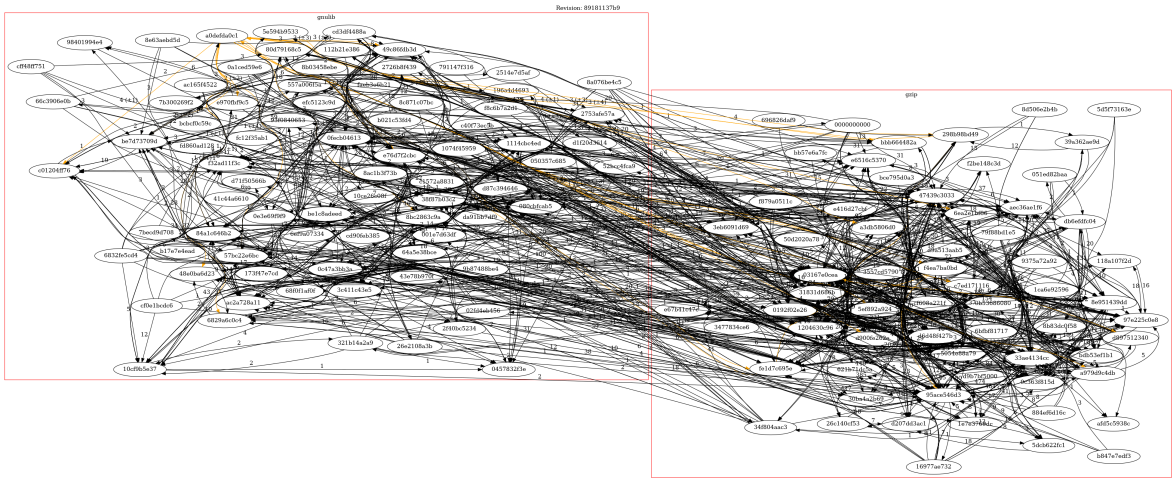

Figure 4.15: Graphviz plot of project GNU gzip of revision 89181137b9.

We notice that the commit `a0defda0c1` at the top of the `gnulib` cluster is involved in many of the `blame diff interactions`. It is hard to follow the blame interaction edges of one specific commit to another, therefore we choose the commit `a0defda0c1` and generate a `Graphviz Plot` that filters all interactions that involve this commit. The resulting plot represented in Figure 4.16 is much clearer and easier to interpret. Now we can see that this commit has a rather big impact on the previous revision's blame interactions compared to the other commits of the complete `Graphviz Plot`.

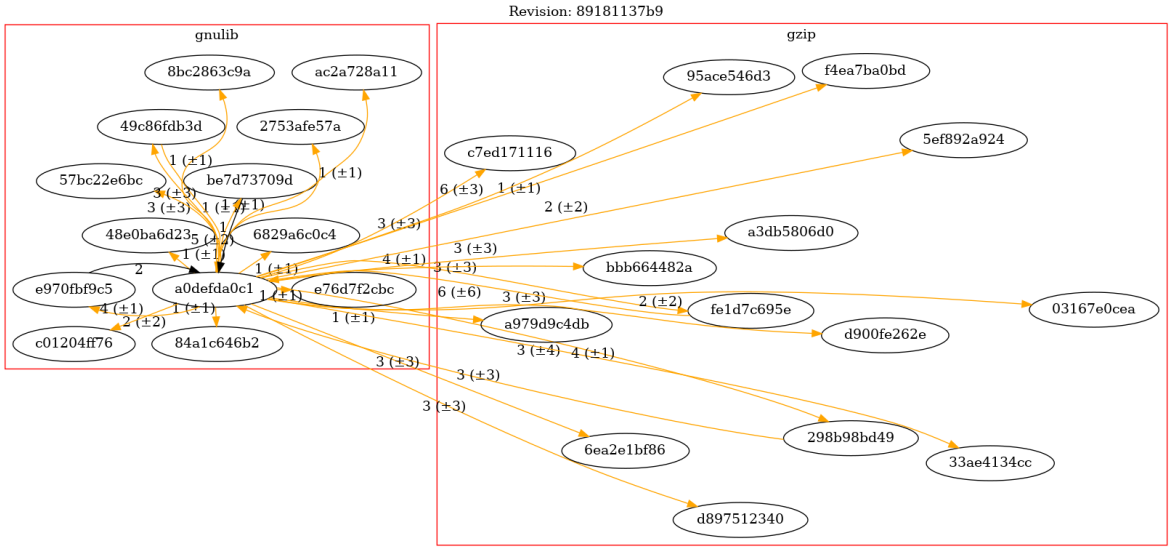

Figure 4.16: Graphviz plot of project GNU gzip of revision 89181137b9 only with blame interactions that contain commit a0defda0c1.

4.4 DISCUSSION

In this section, we answer our research questions and discuss the results of our evaluation. To answer RQ1 and RQ2, we look at the different visualization types that we evaluated before. We can choose between four different plots, where each plot covers a specific level of detail and is used to represent certain attributes of blame interactions, like their `degree` or `fraction` value. The `Fraction Plot` illustrates blame library interactions with a focus on their `fraction` development over time and provides the names of all interacting libraries through its legend. Our `Sankey Plot` depicts the concrete interacting libraries with their blame interactions of one specific revision and creates individual plot files for all revisions of a `CaseStudy`. Blame interactions are visualized by drawing lines of different colors and widths according to the library colors and the distribution of their `fractions`. The plot is able to illustrate interactions between a main program and its libraries and also interactions between libraries themselves. The `Degree Plot` groups blame interactions between two specified libraries by their `degree` and represents the development of their `fraction` ratios within these groups over time. With our `Graphviz Plot`, we are able to display libraries, their commits, and their interactions of one project revision as directed graph. Interactions are denoted as directed edges between nodes that represent commits. These blame interactions can either occur between commits within one library, which is denoted as edges that do not cross their library rectangle, or between commits of different libraries. No prior plots of VaRA-TS are able to represent blame interactions between multiple libraries.

> Our proposed illustrations prove useful for visualizing the interactions between multiple libraries, with each illustration having a different focus on the interactions and their libraries. This leads us to accept RQ1.

Furthermore, the `Fraction Plot` and `Degree Plot` are able to illustrate multiple project revisions and their development over time by passing a `CaseStudy`. Even the plots that generate one plot file for a specific revision, like the `Sankey Plot` or `Graphviz Plot` can be used to generate multiple numbered plot files that one can render in a file viewer to pass through them sequentially and get a better understanding of the project's progression over time.

> Our proposed illustrations prove useful for visualizing the development of blame interactions over time, with two illustrations depicting them over a range of revisions and the other two illustrating each revision specifically that can be viewed sequentially to achieve a similar effect. This leads us to accept RQ2.

By comparing the `Graphviz Plot` of our `Elementalist` project with the `Graphviz Plot` of `grep` and `gzip`, we notice that the clarity of these plots is strongly dependent on the total number of blame interactions the represented revision has.
One way to tackle the issue of their decreased readability is to additionally draw the `blame diff interactions` in the plot like we did in our evaluation of `grep` and `gzip`. This gives one a general idea of how much impact the represented revision has on the prior one. Furthermore, we are able to generate a `Graphviz Plot` that puts its focus on blame

interactions that involve only one specific commit and makes the blame interactions of the specified commit more readable, by omitting all other blame interactions.
Overall, our evaluation shows that our plots are able to visualize blame interactions in projects with multiple interacting libraries, like the `Elementalist` project and also projects with many specified revisions, like `grep` and `gzip`. Users of VaRA-TS can generate the different visualizations to first get an overview of a project's development over time and then proceed with plots that illustrate a specific revision or commit of interest more precisely.

4.5 THREATS TO VALIDITY

This section introduces the factors that could endanger the validity of our evaluation results. We start by explaining a possible threat regarding our blame interactions and proceed with a possible limitation of the validity of our `Graphviz Plot` results regarding their layout.

4.5.1 *Blame Interaction Detection*

The data we base our visualizations on depends on the findings of data flows between source code instructions and their corresponding commits. The data-flow analysis we use to gather these blame interactions is conducted by VaRA. While our concepts and visualizations are independent of the correctness of this analysis, it is worth mentioning that our visualization results may vary, if the data-flow analysis of VaRA changes.

4.5.2 *Layout Engine*

The arrangement of nodes and edges in our Graphviz Plots strongly depends on the used layout engine. While a different layout is often desirable when choosing another layout engine, it is also possible that a different layout results from the same layout engine when we rerun the plot generation on the same data. This factor of randomness can lead to different arrangements of the nodes and edges of our `Graphviz Plots`, which limits the reproducibility of our visualizations.

5 RELATED WORK

To the best of our knowledge, there is no previous work that has visualized the data flows between source code instructions that stem from commits of multiple libraries, represented on the level of their interacting commits.
However, Mysore et. al [7] presented visualizations of data flows through multiple layers of abstractions on the example of a network server program. Their representation resembles our `Graphviz Plot`, as they grouped the interacting processes, their modules, and their inner functions in clusters that are enclosed by rectangles and illustrated the data flow interactions between them as edges. These groups look similar to our clusters of commits that are drawn inside the rectangle that corresponds to the library they are originating from. Furthermore, they illustrated the ingoing and outgoing data flows as edges that connect to dots and circles, similar to the directed edges that denote the blame interactions in our `Graphviz Plot`.
Additionally, we found that Wongsuphasawat et. al [10] visualized the underlying data flows between components of a machine learning model from a high level view. The grouping of these components resembles the clustering of commits in our `Graphviz Plot`. Furthermore, their visualized data flows are illustrated in a flow layout, which allows them to encode additional information in an edge's stroke width. This layout is similar to our `Sankey Plot`, which is also a type of flow diagram that encodes the `amount` of our blame interactions in its stroke width.
Behnamghader et. al [1] proposed their *commit-impact analysis*, which distinguishes commits in *impactful* and *non-impactful* commits and analyzes them on certain metrics to gain information about the quality and development of software projects. Similar to the data-flow analysis of VaRA, they use various tools to perform large-scale static code analyses on commits that stem from software projects written in Java. A major difference to our work lies in their scope of analyzed revisions, which is determined by their definition of impactful commits in the context of their commit-impact analysis. Impactful commits are described as commits that have an impact on the main module (module that contains most of the project's source code) of the software project in terms of their defined metrics, like project size, complexity, and security vulnerabilities. Therefore, included modules with less source code than the main module are ignored in this analysis. In the context of our work, this would mean that we ignore interactions between library (smaller module) commits (as they are not considered impactful), and visualize only commits that interact with our main repository (main module). However, we do not follow this approach because we consider the interactions between libraries in our visualizations relevant to facilitate the understanding of how a software project evolves and to estimate the impact of commits in the main repository and its libraries.

6 CONCLUDING REMARKS

In this chapter, we conclude the thesis by summarizing our approaches to visualize the blame interactions between multiple libraries and providing possible approaches to further improve the presented visualizations.

6.1 CONCLUSION

In this thesis, we presented four different plots that allow users of VaRA-TS to visualize blame interactions between multiple libraries, since prior visualizations of VaRA-TS are limited to blame interactions between commits of the same library.
To gather the necessary data, one needs to relate the commits of blame interactions to their corresponding libraries, we proposed a concept of using the `BlameReport` data of VaRA to extend the commit lookup of VaRA-TS. Furthermore, we described our database layouts, in which we cache the extracted blame interaction data for easier access and reusability. We then explained the concepts and implementations of our blame library interaction plots that allow us to show the development of blame interactions between multiple libraries over time, with different emphasis on their attributes.
We evaluated our approaches on three different software projects, where the first one was a small demo project called `Elementalist`, and the other two were the real-world projects `GNU grep` and `GNU gzip`. We found that our plots were able to illustrate the blame interactions between multiple libraries in different ways that facilitate a researcher's understanding of the underlying data flows, and that they could be used to represent the development of different attributes of blame interactions over time.

6.2 FUTURE WORK

6.2.1 *Graphviz Viewer*

The current representation of our `Graphviz Plots` is rather hard to read when we visualize revisions with many blame interactions. Therefore, we would like to use programs that allow us to import the underlying `DOT` file and visualize them more clearly. Many of these programs do currently not support `DOT` graphs that contain large amounts of nodes and edges, or they result in a similar representation like ours. To increase the readability of these graphs, we would like to illustrate them as three-dimensional graph, where one can inspect the blame interactions more clearly, or find a new layout that, e.g., removes the space-consuming edges without loosing information about the commits that are interacting. Additionally, we want to use graph viewers that support XML based file formats, like GraphML. This means that we have to adapt our current implementation of the `Graphviz Plot` to additionally generate graph files in the GraphML format.

6.2.2 *Coloring Nodes and Edges*

The nodes and edges of our current `Graphivz Plot` are all the same color, disregarding the color of `blame diff interactions`. This makes it hard to follow specific edges to their corresponding nodes to find interacting commits. Therefore, we would like to color blame interactions and their corresponding nodes differently to make them easier to distinguish from others in the plot.

6.2.3 *Blame Diff Interactions*

The labels of our `blame diff interactions` are currently limited to the `amount` of the current revision and the difference of the `amount` to the previous revision. At the moment, we do not discern if this difference to the previous revision is an increase or decrease of the blame interaction's `amount`. Therefore, we want to implement this differentiation of the blame diff `amount` in our edge labels to better understand the impact of a revision on its predecessor.

BIBLIOGRAPHY

[1] P. Behnamghader, R. Alfayez, K. Srisopha, and B. Boehm. "Towards Better Understanding of Software Quality Evolution through Commit-Impact Analysis." In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2017, pp. 251–262. DOI: 10.1109/QRS.2017.36.

[2] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. "Graphviz and dynagraph – static and dynamic graph drawing tools." In: *GRAPH DRAWING SOFTWARE*. Springer-Verlag, 2003, pp. 127–148.

[3] Emden R. Gansner. "Drawing graphs with Graphviz." In: (2009). Available online at http://www.ammd.ch/1.pdf.

[4] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. "Drawing graphs with dot." In: (2015). Available online at https://www.graphviz.org/pdf/dotguide.pdf.

[5] J. D. Hunter. "Matplotlib: A 2D graphics environment." In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.5281/zenodo.4268928.

[6] Julian Breiteneicher. "Enhancing Program Analysis with Git Metadata in VaRA." Masterarbeit. Germany: University of Passau, 2019.

[7] Shashidhar Mysore, Bita Mazloom, Banit Agrawal, and Timothy Sherwood. "Understanding and Visualizing Full Systems with Data Flow Tomography." In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 211–221. ISBN: 9781595939586. DOI: 10.1145/1346281.1346308. URL: https://doi.org/10.1145/1346281.1346308.

[8] Florian Sattler. "A Variability-aware Region Analyzer in LLVM." Masterarbeit. Germany: University of Passau, 2017.

[9] Andreas Simbürger, Florian Sattler, Armin Größlinger, and Christian Lengauer. *BenchBuild: A large-scale empirical-research toolkit*. Tech. rep. Available online at https://www.fim.uni-passau.de/fileadmin/dokumente/fakultaeten/fim/forschung/mip-berichte/MIP-1602.pdf. Technical Report MIP-1602, Faculty of Computer Science and Mathematics, 2016.

[10] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mané, D. Fritz, D. Krishnan, F. B. Viégas, and M. Wattenberg. "Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow." In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 1–12. DOI: 10.1109/TVCG.2017.2744878.