
**Master Thesis**
**in Computer Science**

# Feature-Aware Collaboration Tracking with Codeface

Matthias Dittrich

Supervisor:
Prof. Dr. Sven Apel

June 24, 2015

**Abstract**

Current research has provided increasingly accurate collaboration networks. At the same time, the growing number of applications for these networks, for example, to predict software failures, leads to the direct need of more accurate networks.

This thesis enlightens a new way of gathering developer collaboration by using the notion of features established by the research of software product lines (SPL). Within this work we show how feature-aware collaboration tracking can be implemented within the software analysis tool CODEFACE, which already supports the creation of tagging, file, and function based collaboration networks. Additionally we benchmark our implementation and evaluate our networks by comparing them with the established function based networks on four open source projects using three 3-month commit windows.

# Contents

# List of Figures

# List of Tables

# 1. Introduction

Software product lines (SPLs) are an effective way to implement software products for a specific domain [28]. SPLs consist of a shared code base and are configured by specifying a set of features, each provided by feature-specific code. Every configuration yields a different software product. Due to the characteristics of the implementation mechanism features may also have inter-dependencies and often require extra code (glue code) depending on the configuration. In the end the idea is to maximize the reuse of software artifacts between different products [28].

For readability, we assume that the concrete implementation of SPLs is done in C/C++ with the help of the C-Preprocessor (CPP) to annotate feature-specific code. This is reasonable as most of the examples discussed within this work can be implemented using other concrete methods of feature separation as well. Furthermore, the CPP is widely used for SPLs, even when research shows that alternatives exist [20, 14].

Collaboration tracking or collaboration analysis as suggested within this work aims at answering the question "Who is working with whom?" by only using the version control system. These collaboration networks provide crucial information for project managers or new project members [29, 9, 11, 12, 25, 26]. A project manager could compare these networks with the networks of the actual communication, extracted from, for example, mailing lists. This way, a lack of communication or an overhead of communication could be detected. On the other hand, new members are able to see with whom they should connect with for their changes. Research strongly indicates that developer organization and coordination is critical for software quality [9, 8, 11, 12, 26, 25], some are even able to predict software failures [24, 26].

It has already been shown that modern version control systems (VCS) provide sufficient information to construct such collaboration networks with various methods [18, 22, 21, 16, 17, 23]. Research by Joblin et al. has shown that function-based collaboration networks are more accurate than file- and tagging-based networks [18]. However, the authors suggest that there are still missing links, so it is likely that a more specialized analysis can uncover those. We implement and investigate two more approaches specialized to SPLs, which incorporate the feature view into the analysis. Additionally we evaluate our new approaches by comparing the feature-based networks with the function-based ones.

# 2. Related Work

To get a general overview on the current state of research, we give a short historical overview about the related work leading to this thesis.

Initially extracting collaboration information from the VCS was done by Lopez-Fernandez et al., by linking developers based on the collaboration on the same module [21, 22]. This work was later improved by Huang et al. by automating module classification by analyzing the directory structure [16]. However, the developer networks generated by this approach are still to dense to apply any collaboration analysis. One important step was done by Jermakovics et al. to use file information to capture collaboration [17]. With this method collaboration is identified by working on the same file instead of a complete directory. This helped to find more fine-grained collaboration but the generated networks where still too dense for collaboration analysis methods. But the authors still managed to run a community-detection and visualize their networks by filtering them beforehand. A more fine-grained approach was used by Bird et al. by using the Linux tagging convention to construct a developer network without analyzing it [7].

Joblin et al. found a general and fully automated approach to produce collaboration networks just with the right amount of density to be able to use them in collaboration analysis algorithms [18]. The general idea was to take the code structure into account and detect collaboration if developers collaborated in the same function of a file. They implemented this approach within the freely available Codeface tool [1], they verified the generated function-based networks in an empirical study and compared the Linux graph against the tagging approach. They also extended the work of Meneely et al. who addressed the question of real-world significance of file-based developer networks by showing missing links and false positives by extending the survey and showing that function-based networks only miss edges [23]. Our work starts at their conclusion that the function-based approach reflects collaboration in reality, but is missing some edges. By using a feature instead of a function induced code structure we aim to uncover those missing links.

Research has used other sources to create collaboration networks. Mailing lists have been used by using a sent e-mail as an edge between sender and recipient. Social-network analysis has been applied to investigate different roles in the Linux mailing list [27] Bird at al. have used mailing-list networks to compare the relationship between email and VCS participation and to study the small-world property as well as social-network analysis [5]. They also investigated the organization and structure of four open source projects and have shown statistically significant communities [6]. However, they miss to show real-world significance.

Bug-tracking systems have been used to create and analyze collaboration graphs by

finding community structure as well [10, 15]. Their work focused on social networks in contrast to developer networks as it seems obvious that the bug tracker is a place for social interaction, but not necessarily suited to find developer collaboration, as most of the time a single developer is responsible for fixing a bug and the bug is often reported by people not directly committing to the repository.

# 3. Codeface

Within this chapter we explain the CODEFACE architecture and important implementation details. This provides the necessary background to explain our changes and the new implementation, which will be shown in the following chapter.

## 3.1. Architecture

CODEFACE is a software-project analysis tool, which can analyze multiple data sources within a common framework [18, 1]. The tool can extract and analyze information from version control systems and mailing lists. An overview over the CODEFACE architecture can be found in Figure 3.1.

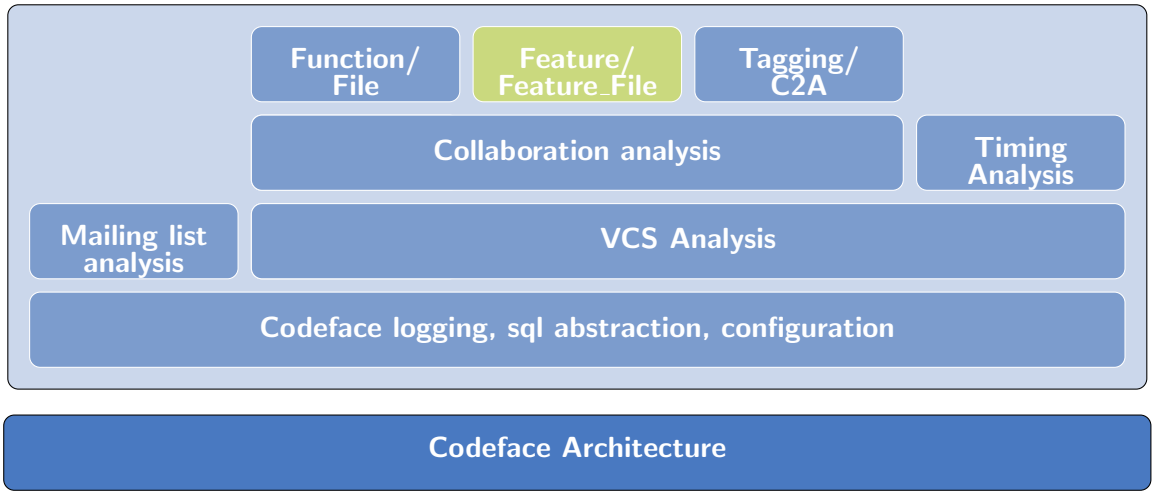


Figure 3.1.: Architectural overview of CODEFACE. CODEFACE provides analysis methods for two sources, namely the mailing list and the VCS. The VCS Analysis can be split up in two categories: First the timing analysis which calculates evolutionary project metrics like the number of files, and the collaboration analysis which creates the developer network and runs a cluster analysis to extract clusters of developers. CODEFACE can find collaborations between developers based on files, functions and tagging (where C2A stands for *Commiter2Author*) based approaches. And we present two completely new approaches, namely *Feature* and *Feature_File* as part of this thesis.

CODEFACE is an open-source project, initially developed internally by Siemens to analyze software projects. The tool outputs PDF and LaTeX reports of the collaborations

and communications in the system. Additionally, a web-service is included to visualize the results within a website. A screenshot can be found in Figure 3.2.

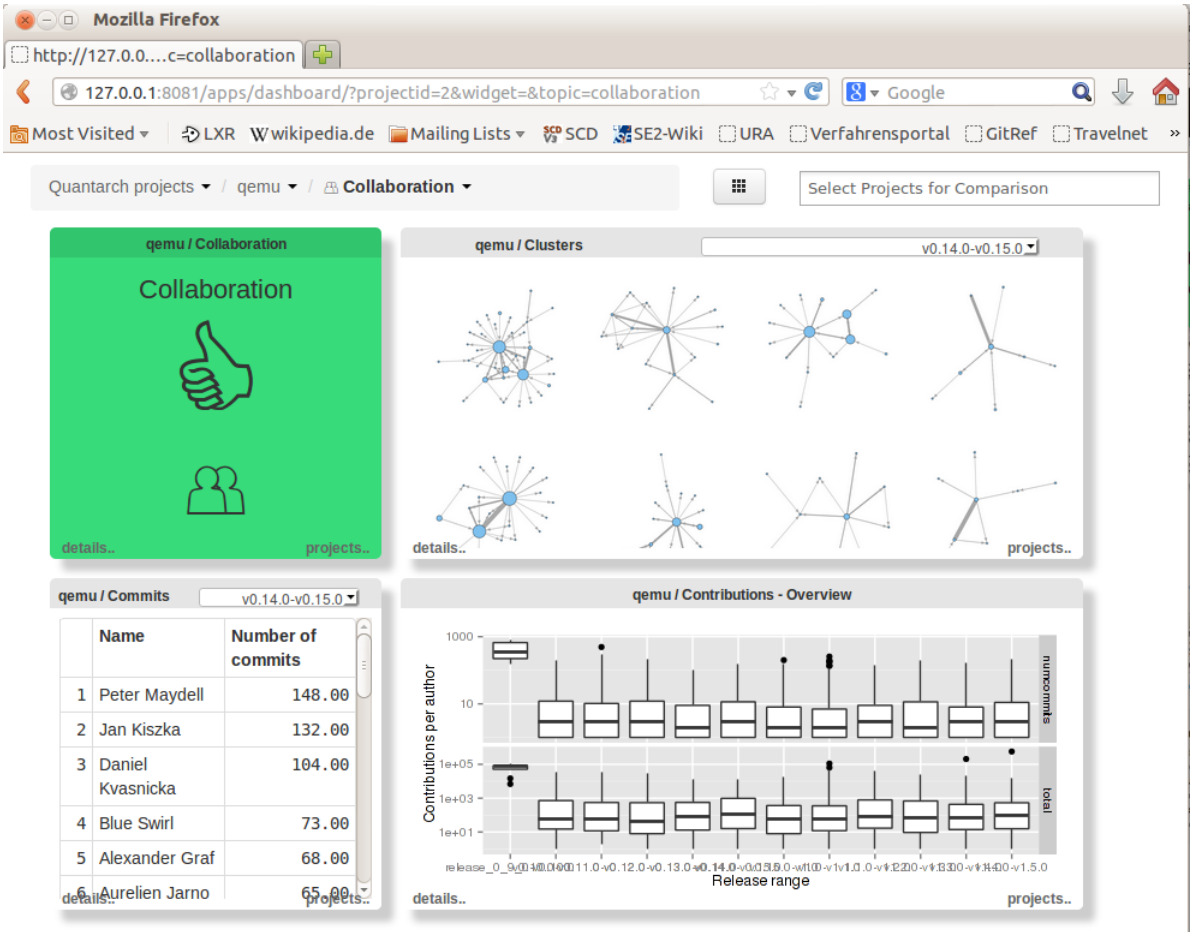


Figure 3.2.: A screenshot of the SHINY web service of CODEFACE. It shows the collaboration section which shows the analysis of a collaboration network and the detected clusters, i.e., groups of developers working together on the selected time frame.

Within the scope of this work, only the collaboration analysis techniques of CODEFACE are discussed.

## 3.2. Collaboration Analysis

CODEFACE can generate a developer or collaboration network within its collaboration analysis by using various ways for collaboration tracking. This process is defined for a single time-frame of the VCS, however, CODEFACE can show the evolution of the data by repeating its collaboration analysis process for multiple commit ranges. Hereinafter,

we explain a single iteration and use the range $(r_1, r_2)$, where $r_1$ is the first commit and $r_2$ is the last. It is possible to configure multiple commit ranges for a project in the CODEFACE configuration files and CODEFACE is able to process them in parallel. If no commit range is specified, CODEFACE calculates and uses three 3-month windows automatically, which have historically yielded good results [18].

The CODEFACE collaboration analysis powered by the VCS data is described in Figure 3.3 and is given by a multiple-step process. In the first step, an internal CODEFACE structure is generated from the version control system[1]. This first step gets the commit and blame[2] data out of the repository and saves it into an easy-and-fast-to-process, in-memory, python data structure. Blame analysis is quite costly for most VCS systems, so this step currently takes the most amount of the time. Additionally CODEFACE collects line information depending on the selected analysis method. The first step is only done for implementation and performance reasons and the internal data structure does only contain data fetchable with a simple GIT query or additional tools for the line information. Currently CODEFACE is calling sub-processes and parsing their output to fetch the commit, blame and additional information.

The second step is to analyze the raw data and the code changes to determine the collaboration strength between different developers. The collaboration graph and strength between the developers is then given by an adjacency matrix. The concrete meaning of strength depends on the used analysis method.

This matrix is then used in the next step, the cluster analysis, to find clusters or networks of developers. Codeface uses the *order statistics local optimization method* (OSLOM) for community detection or cluster analysis. The main reason is that OSLOM can handle weighted and directed networks [18, 19]. These clusters are then written to various LaTeX and PDF files in the output directory. Additionally CODEFACE saves the developer network and the captured clusters in its database, the data-model is shown in Figure 3.4.

The following subsections give further insight into the "Collaboration Analysis" box of Figure 3.3.

### 3.2.1. Signed-Off/Committer2Author Analysis

The *Signed-Off* and *Committer2Author* analysis methods are enumerating all commits from the current commit range and compute the collaboration directly from the commit meta-data. The process is shown in Figure 3.5.

For the *Committer2Author* analysis, CODEFACE is reading the `Author` and `Committer` meta-data fields from the CODEFACE internal data structure and adds a relation from the committer to the author.

For the *Signed-Off* analysis, CODEFACE is only using the `Commit-Message` meta-data field from the internal data structure. Projects use various tags within the commit

[1] Currently CODEFACE only supports the VCS system GIT

[2] A common feature of modern VCS to extract additional line information, like who contributed a particular line on which particular commit. See `git blame` for details.

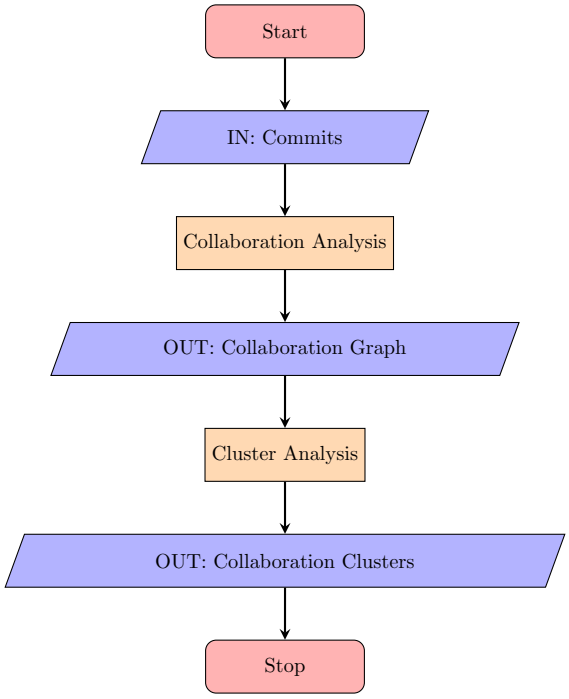


Figure 3.3.: Flowchart of the collaboration-analysis process in CODEFACE. Initially, CODEFACE collects data about the configured repository. Then, CODEFACE runs the configured collaboration analysis. The analysis produces an adjacency matrix, which represents the collaboration graph[3]. Now the graph is processed by the cluster analysis to find the clusters of the network. Finally, the output files are generated and results are saved to the database.

[3] This graph is saved into the CODEFACE database as edgelist as well.

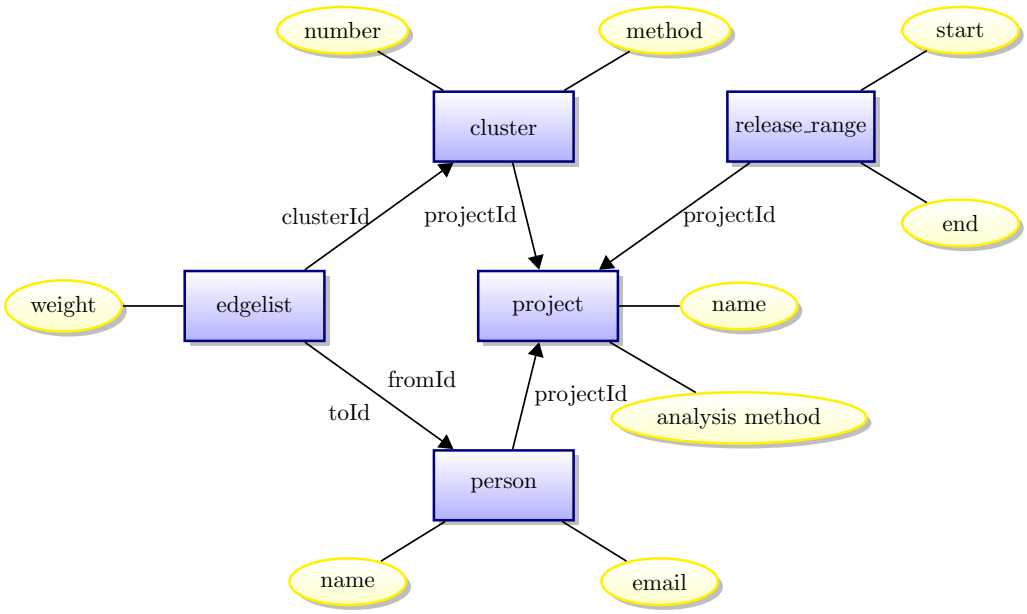


Figure 3.4.: A (simplified) ER-Model of the CODEFACE database which only shows tables relevant to saving collaboration analysis results. Clusters are saved by creating a new entry in the *cluster* table and adding the edges to the *edgelist* table. The complete developer network is saved in CODEFACE as a special *cluster* with *number* 0.

message to add some additional information to a commit. For example, the `Signed-Off` tag is used by many projects to establish a developer certificate of origin[4] and is directly supported by GIT itself[5]. Some projects use similar tags like `Acked-by`, `Reviewed-by`. CODEFACE supports these tags and establishes a relationship between the mentioned developers or authors.

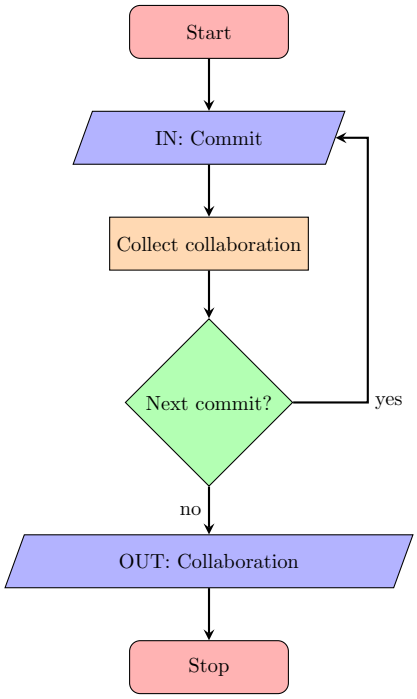


Figure 3.5.: Flowchart of the *Signed-Off / Committer2Author* collaboration analysis in CODEFACE. It shows the most simple analysis method where just commits are enumerated and the collaboration is collected by analyzing commit meta-data. The *Signed-Off* analysis uses the `Author` and `Committer` meta-data fields, while the *Committer2Author* parses the commit message to extract common tags like `Reviewed-by` and `Acked-by`.

### 3.2.2. File/Function Analysis

While the previous analysis are pretty straightforward to understand from their implementation details, we want to give a much deeper insight into the *File-* and *Function-*based analysis before discussing the technical details. This makes it easier to point out the analogies and differences between the function-/file- and the feature-based methods, that we implement.

[4] `http://elinux.org/Developer_Certificate_Of_Origin`
[5] `http://git-scm.com/docs/git-commit`

### High-Level Example

One way to explain the *Function* and *File* analysis is to look at an in-depth example and show the processing step by step. The example should only provide the general idea and doesn't show the full process. To simplify the example, special cases like removing older commits are not considered. Those details are explained later when the concrete implementation is discussed.

For example, consider the code in Figure 3.6 to be a file within the currently analyzed project at the time $r_2$ (the end of our analysis window).

```
1  // Initial comment to foo
2  int foo (int arg) {
3      doFunc();
4      otherFunc();
5      return 0;
6  }
7  // Some comment to bar
8  int bar (int b) {
9      a ();
10     return 4;
11 }
```

Listing 3.6.: A simple example source code to explain the *Function* analysis.

CODEFACE will now run CTAGS and/or DOXYGEN to gather function information and GIT BLAME for the blame data [3, 4]. This process will yield information about the lines of the code. In particular CODEFACE can now assign lines to specific functions and to specific commits. The commits provide additional information about the user and the time when the commits were added. CODEFACE uses the time of every commit, to define an order on all the commits. Therefore we further simplify the example by using abstract time points instead of concrete times. For our example code from Listing 3.6, the result will look similar to Table 3.1.

Once function information is available the collaboration is collected by enumerating the time steps and considering all older lines, working on the same function. The weight is given by the sum of the number of lines. Performing these steps for the example returns a set of collaborations for each time frame. The results from our example can be found in Table 3.2.

Finally, CODEFACE creates a collaboration network from the aggregated collaboration data. The graph for the current example, a project with a single code file, can be found in Figure 3.7.

| Line | Function | User | Time |
|---|---|---|---|
| 1 | FILE | UserC | 2 |
| 2 | foo | UserA | 0 |
| 3 | foo | UserB | 1 |
| 4 | foo | UserA | 4 |
| 5 | foo | UserB | 1 |
| 6 | foo | UserA | 0 |
| 7 | FILE | UserD | 5 |
| 8 | bar | UserC | 2 |
| 9 | bar | UserC | 2 |
| 10 | bar | UserC | 2 |
| 11 | bar | UserA | 3 |

Table 3.1.: Data available to CODEFACE after running the VCS blame analysis and CTAGS/DOXYGEN. Note that the abstract time used in this table is given by an implicit commit order used within CODEFACE based on the commit time.

| Time | User | Function | Collaborating User | Weight |
|---|---|---|---|---|
| 1 | UserB | foo | UserA | $2+2=4$ |
| 3 | UserA | bar | UserC | $1+3=4$ |
| 4 | UserA | foo | UserB | $1+2=3$ |
| 5 | UserD | FILE | UserC | $1+1=2$ |

Table 3.2.: Collaboration collection process by enumerating time steps, and finding collaboration with earlier times.

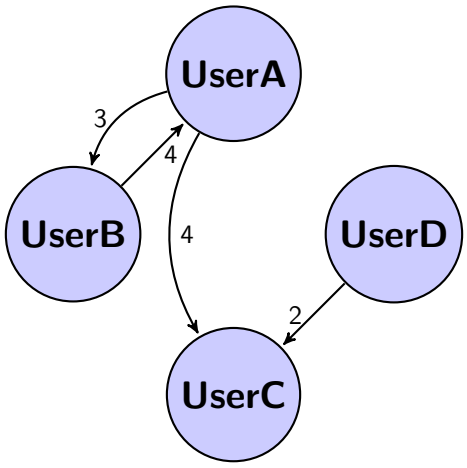


Figure 3.7.: A simple example function-based network generated from the code in Listing 3.6 and the collected metadata from Table 3.1.

### Implementation Details

In the *File* or *Function*[6] analysis CODEFACE enumerates *FileCommit* objects instead of raw commits. The process is shown in Figure 3.8. *FileCommit* objects are built by splitting up every commit into the files it modifies and then collecting all parts, which belong to the same file. Because the *File* analysis is a special case of the *Function* analysis, where every file is seen as being one function, we only explain the *Function* analysis.

While the analysis is complex, the basic idea is this: The collaboration of two authors is given by the sum of the number of line changes on a function. For fetching the function locations of a file, CODEFACES uses DOCYGEN and falls back to CTAGS which covers a wide range of programming languages. To include the time-information, the analysis enumerates all commits of a file (*FileCommit*) and calculates the collaboration in the following way:

1. Get the blame analysis of the file at commit $r_2$.
2. Remove all lines *after* the current commit ($c$).
3. Remove all lines *before* the current commit-range start commit $r_1$.
4. Group code-blocks (a group of lines with the same committer) by *function* name.
5. Calculate the collaboration for each group:
   a) Separate the group into the blocks which belong to the current commit and the *rest*.
   b) Enumerate all commits ($d$) of the *rest* (commit $d$ is always *before* commit $c$)
      i. The collaboration strength is the sum of the blocks belonging to either commit $c$ or $d$.
      ii. The direction of the collaboration is from committer of $c$ to the committer of $d$.

This procedure ensures that the ordering of the commits is taken into account. The grouping of the commits is done for performance reasons and to simplify the implementation.

Naturally, one would use the file at the time of commit $c$ at step 1, but for performance reasons CODEFACE uses the blame analysis of the file at the time $r_2$ (the last commit of the current range). While this is a potential threat to validity it has already been analyzed and discussed by Joblin et al. [18] without any indication of practical relevance. Although, we might miss collaborations.

[6] Within CODEFACE, the *Function* analysis is called the *Proximity* analysis as well.

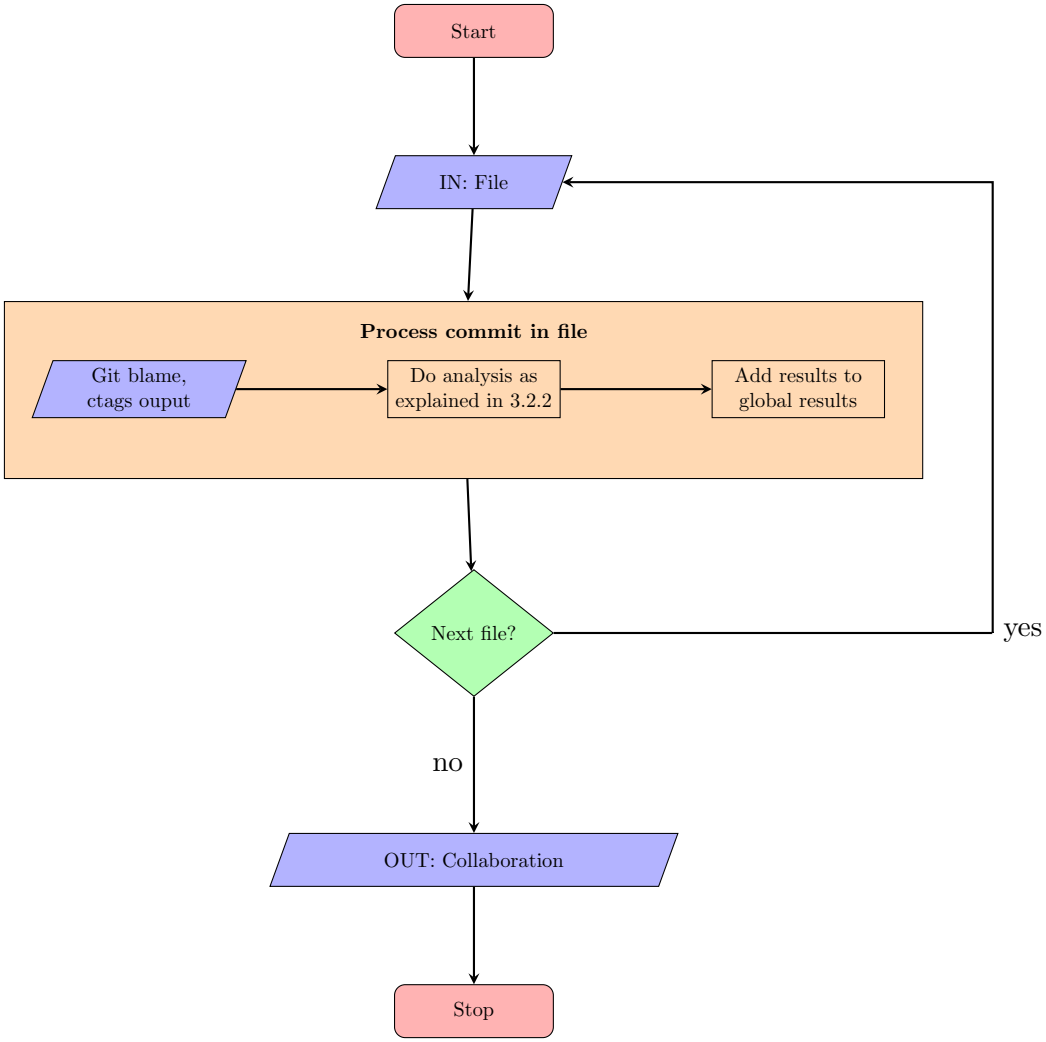


Figure 3.8.: Flowchart of the *File/Function* collaboration analysis in CODEFACE. All files are iterated and for each file all commits (*FileCommit* objects) are iterated. For each file, the `git blame` and `Doxygen` or `ctags` output is used to calculate the collaboration as explained in Section 3.2.2.

# 4. Feature-Aware Collaboration Analysis with Codeface

With the knowledge of the CODEFACE internals, we can now discuss how the new feature-based analysis methods are implemented on top of the CODEFACE framework.

Basically two new analysis methods, namely the per-file *Feature_File* and the project-wide *Feature* analysis, have been added. While the *Feature_File* analysis is heavily based on the *Function* analysis, the *Feature* analysis implements a new approach[7].

## 4.1. Feature detection

Before we can create any community networks based on features we first need to be able to detect them. We therefore need to replace the CTAGS/DOXYGEN-based detection with CPPSTATS, a tool for preprocessor-based SPLs, which allows detection of feature-annotated code blocks [2]. We show this detection process with a simple code example from Listing 4.1.

```
1 #if A||B
2  doFunc();
3 #if C
4  otherFunc();
5 #endif
6 #endif
```

Listing 4.1.: A simple example source code to explain the feature detection algorithm.

First we run CPPSTATS to detect feature lines and get the results written as a CSV file as shown in Figure 4.2. The file contains a line for each feature, marks its start- and end-line as well as the feature condition.

While we could use this format, create an entry for each source code line and insert the feature information, there is a more efficient way to save this information by only saving the lines which change the feature set. This is because we expect only a very limited number of feature annotation lines in comparison to other lines. The idea is to order this list of changed lines by the actual line number and use binary search when the information is accessed. To construct such a list, we first need to parse the CPPSTATS

[7]The list of contributions can be found here: `https://github.com/siemens/codeface/commits?author=matthid`

```
#file ,  #start ,#end ,  #if , #ifcond , #feature_list
test.c ,    1 ,     6 ,    #if , A||B,        A;B
test.c ,    3 ,     5 ,    #if , C,           C
```

Listing 4.2.: The result CSV file written by CPPSTATS by using our test-code from Figure 4.1. CODEFACE doesn't evaluate the **#file**, **#if**, **#ifcond** columns.

data and convert it to a simpler structure which splits the start- and end-lines into separate items. The result looks similar to Table 4.1.

| **Line** | **Type** | **Features** |
|---|---|---|
| 1 | Start | A,B |
| 3 | Start | C |
| 5 | End | C |
| 6 | End | A,B |

Table 4.1.: The Table shows the simplified data extracted and converted from the generated CPPSTATS CSV file from Figure 4.2.

Now we simply need to iterate over this list and construct the list of changed lines. CODEFACE combines this list with the blame data, as already seen in Section 3.2.2. The result is semantically equivalent with a lookup table which yields feature and commit information for each line. An example can be found in Table 4.2.

| **Line** | **User** | **Features** | **Commit** |
|---|---|---|---|
| 1 | X | A,B | 1 |
| 2 | Y | A,B | 2 |
| 3 | X | A,B,C | 1 |
| 4 | X | A,B,C | 1 |
| 5 | Y | A,B,C | 2 |
| 6 | X | A,B | 1 |

Table 4.2.: The Table shows the final data after running the blame analysis, returning the commit and user information, and the feature detection with CPPSTATS, returning the feature information, for the code in Figure 4.1.

## 4.2. Feature_File Analysis

Now that feature detection is available we can implement a new *Feature_File* analysis based on the *Function* analysis.

### High-Level Example

To explain the *Feature_File* analysis we provide an example and show the processing step by step. As in Subsection 3.2.2 the example should only provide the general idea and doesn't show the full process. To simplify the example some special cases, like removing older commits, are not considered. Those details are explained later when the concrete implementation is discussed.

For example, consider the source code files in Listing 4.3 as the currently analyzed project at the time $r_2$ (the end of our analysis window).

```
1 int main (int arg) {
2 #if A||B
3   doFunc();
4 #if C
5   otherFunc();
6 #endif
7   return 0;
8 #endif
9 }
```

```
1 #if C
2 int bar (int b) {
3   a ();
4 #else
5   return 4;
6 }
7 #endif
```

Listing 4.3.: Two simple example source code files to explain the *Feature_File* analysis. The left is called **main.c**, while the right one is called **bar.c**.

CODEFACE will now run feature detection as explained in Section 4.1 and the VCS blame analysis to gather line-specific information for the source code. Similar to the result of the previous function-based example, shown in Table 3.1, we now get data similar to Table 4.3. CODEFACE can now assign lines to specific features and to specific commits and, for our example, we abstract from concrete times the same way we did in Subsection 3.2.2.

Once feature and blame information is available, the collaboration is collected by enumerating the time steps and considering all previous (in time) lines, working on the same feature. Performing these steps for the example returns a set of collaborations for each time frame. The results from our example can be found in Table 4.4.

Finally CODEFACE creates a collaboration network from the aggregated collaboration data. The graph for the current example, a project with two code files, can be found in Figure 4.4.

### Implementation Details

As already mentioned in the introduction of Section 4, the *Feature_File* analysis is quite similar to the *Function* analysis. The main difference is that CODEFACE now groups code lines by features instead of functions. We basically replace the DOXYGEN/CTAGS-based process with the process explained in Section 4.1. On this occasion we needed to generalize some concepts about line information within CODEFACE, as a single line can

| Line | Features | User | Time |
|---|---|---|---|
| 1 | - | UserA | 0 |
| 2 | A,B | UserA | 1 |
| 3 | A,B | UserA | 1 |
| 4 | A,B,C | UserB | 2 |
| 5 | A,B,C | UserB | 2 |
| 6 | A,B,C | UserB | 2 |
| 7 | A,B | UserA | 1 |
| 8 | A,B | UserA | 1 |
| 9 | - | UserA | 1 |

| Line | Features | User | Time |
|---|---|---|---|
| 1 | C | UserC | 3 |
| 2 | C | UserC | 3 |
| 3 | C | UserC | 3 |
| 4 | C | UserD | 4 |
| 5 | C | UserD | 4 |
| 6 | C | UserC | 5 |
| 7 | C | UserC | 5 |

Table 4.3.: Data available to CODEFACE after running the VCS blame analysis, parsing CPPSTATS output file and analysis the results. Note that the abstract time used in this table is given by an implicit commit order used within CODEFACE based on the commit time.

| Time | User | Feature | Collaborating User | Weight |
|---|---|---|---|---|
| 2 | UserB | A | UserA | $3+4=7$ |
| 2 | UserB | B | UserA | $3+4=7$ |
| 4 | UserD | C | UserC | $2+3=5$ |
| 5 | UserC | C | UserD | $2+2=4$ |

Table 4.4.: The Table shows the *Feature_File* collaboration collection process by enumerating time steps, and finding collaboration with earlier times. Weights are given by the sum of the number of lines working on the same feature.

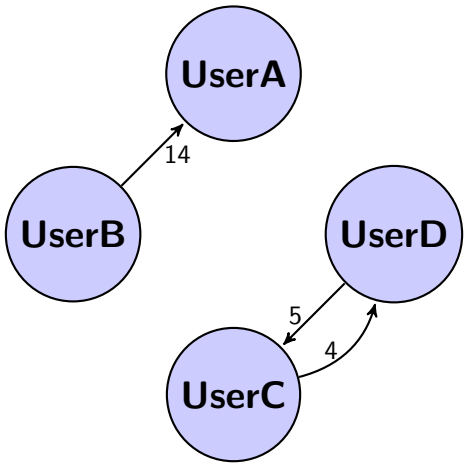


Figure 4.4.: A simple example *Feature_File*-based network generated from the code in Listing 4.3.

belong to multiple features but not to multiple functions. For example grouping lines by function is just a matter of enumerating all lines and tracking the current function name. For features, we need to track a list of current features and properly handle cases like a feature nested in itself. We can describe the complete process with the following steps:

1. Get the blame analysis of the file.
2. Remove all lines *after* the current commit ($c$).
3. Remove all lines *before* the current commit range start commit $r_1$.
4. Group code-blocks (a group of lines with the same committer) by *feature name*.
5. Calculate the collaboration for each group:
   a) Separate the group into the blocks which belong to the current commit and the *rest*.
   b) Enumerate all commits ($d$) of the *rest* (commit $d$ is always *before* commit $c$)
      i. The collaboration strength is the sum of the blocks belonging to either commit $c$ or $d$.
      ii. The direction of the collaboration is from committer of $c$ to committer of $d$.

Note that these are almost the same steps as in Section 3.2.2, but here code lines are grouped by features in step 4. As mentioned above, the implementation could not be re-used and had to be more generalized. An interesting side effect is that now a single line-change can lead to multiple lines of collaboration if the line belongs to multiple features, this is shown in the example as well.

## 4.3. Feature Analysis

### High-Level Example

To explain the *Feature* analysis, we use the same example code as the *Feature_File* example and show the processing step by step. Again we start with the source code files in Figure 4.3 to be the currently analyzed project at the time $r_2$ (the end of our analysis window). CODEFACE will now run feature detection as explained in Section 4.1 and the VCS blame analysis to gather line specific information for the source code. Exactly like in the *Feature_File* case we now get data similar to Table 4.3. Therefore CODEFACE can now again assign lines to specific features and to specific commits.

Once feature and blame information is available the collaboration can be collected by enumerating the time steps and considering all previous (in time) lines, working on the same feature. In contrast to the *Feature_File* analysis, we now take cross-file changes on the same feature into account. Performing these steps for the example returns a set

of collaborations for each time frame. The results from our example can be found in Table 4.5.

| Time | User | Feature | Collaborating User | Weight |
|---|---|---|---|---|
| 2 | UserB | A | UserA | $3+4=7$ |
| 2 | UserB | B | UserA | $3+4=7$ |
| 3 | UserC | C | UserB | $3+3=6$ |
| 4 | UserD | C | UserC | $2+3=5$ |
| 4 | UserD | C | UserB | $2+3=5$ |
| 5 | UserC | C | UserD | $2+2=4$ |
| 5 | UserC | C | UserB | $2+3=5$ |

Table 4.5.: *Feature* collaboration collection process by enumerating time steps, and finding collaboration with earlier times.

The collaboration network generated from this data can be found in Figure 4.5.

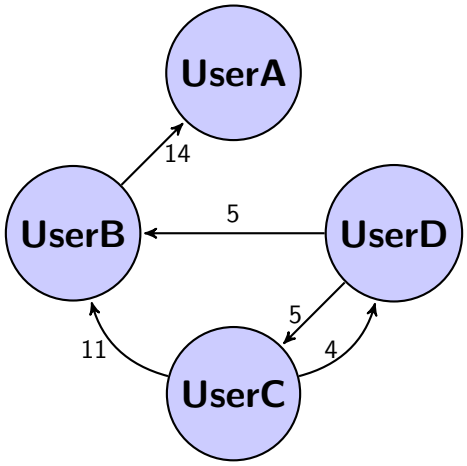


Figure 4.5.: A simple example *Feature*-based network generated from the code in Figure 4.3.

To limit the scope of this thesis, to ease the implementation and to improve performance, we decided to approximate the shown solution by removing the timing aspect of the analysis. Instead of collecting the time-based analysis we simply collect the number of changes for each user, for our example the result can be extracted from Table 4.3 and is shown in Table 4.6.

Now we define the collaboration on a single feature as the minimum number of lines two people worked on the same feature. With this definition we can collect the collaboration from Table 4.6 and get Table 4.7.

CODEFACE now can create a collaboration network from the collaboration data. The resulting network is shown in Figure 4.6. As visualized in the network, we basically merge all edges of the "exact" method; this makes sense, as we basically just lifted the

| User | Feature | Lines |
|---|---|---|
| UserA | A | 4 |
| UserA | B | 4 |
| UserB | A | 3 |
| UserB | B | 3 |
| UserB | C | 3 |
| UserC | C | 5 |
| UserD | C | 2 |

Table 4.6.: Users and the features they are working on, collected from Table 4.3.

| Feature | User | Collaborating User | Weight |
|---|---|---|---|
| A | UserA | UserB | $min(3,4) = 3$ |
| A | UserB | UserA | $min(4,3) = 3$ |
| B | UserA | UserB | $min(3,4) = 3$ |
| B | UserB | UserA | $min(4,3) = 3$ |
| C | UserB | UserC | $min(3,5) = 3$ |
| C | UserC | UserB | $min(5,3) = 3$ |
| C | UserB | UserD | $min(3,2) = 2$ |
| C | UserD | UserB | $min(2,3) = 2$ |
| C | UserC | UserD | $min(5,2) = 2$ |
| C | UserD | UserC | $min(2,5) = 2$ |

Table 4.7.: The Table shows the *Feature* collaboration collection process by processing all users, and finding collaboration with other users. The data is collected from Table 4.6.

time restriction, which means the edge-weights are different and no longer directed.

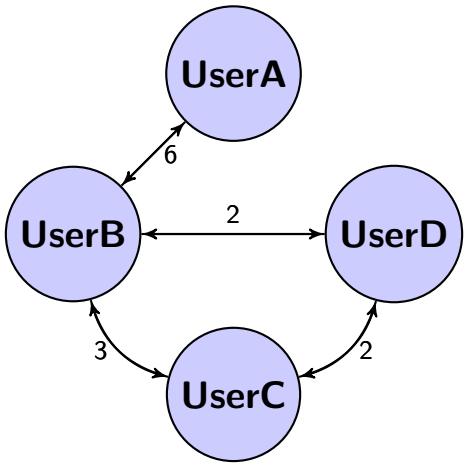


Figure 4.6.: A simple example approximated *Feature*-based network generated from the code in Figure 4.3.

### Implementation Details

With the project-wide *Feature* analysis, we introduce another new analysis method to CODEFACE. The example above shows how the *Feature* and *Feature_File* methods are related. The main goal is to track collaborations across files and only limited by feature boundaries. To calculate a number representative for a collaboration on a feature, we first need to define how we measure collaboration within this context. A natural choice would be the line numbers two developers worked on a feature for a given time period. Assume now $F_P$ to be the set of features and $A_P$ the set of authors of a given project $P$. Next we let $lines_{(t_1,t_2)}(a, f)$ be number of changed lines of author $a \in A_P$ on the feature $f \in F_P$ in the time between $t_1$ and $t_2$. So, given our analysis time period $(r_1, r_2)$ and setting $lines = lines_{(r_1,r_2)}$ we can calculate the collaboration between two authors $a_1, a_2 \in A_P$ with:

$$C_{a_1,a_2} = \sum_{x \in F_P} \min\left(lines(a_1, x), lines(a_2, x)\right)$$

The adjacency matrix $C$ is the result of this calculation and represents the collaboration within the project for a given time window. To easily calculate the sum, we first need to calculate a mapping $M$ : feature $\rightarrow$ (author $\rightarrow$ line number). From this mapping, we can directly calculate $C$ by enumerating all features and all authors, because $lines(a, x) = M(x)(a)$.

To get the mapping $M$, we enumerate all files in the project, group feature lines (as shown in Section 4.2) and collect all groups based on the author and the feature of the group in a lookup table. This lookup table represents $M$.

After the calculation of $C$, CODEFACE will write the resulting matrix into a file and continue with the cluster analysis as shown in Figure 3.3. At the same time $C$ already

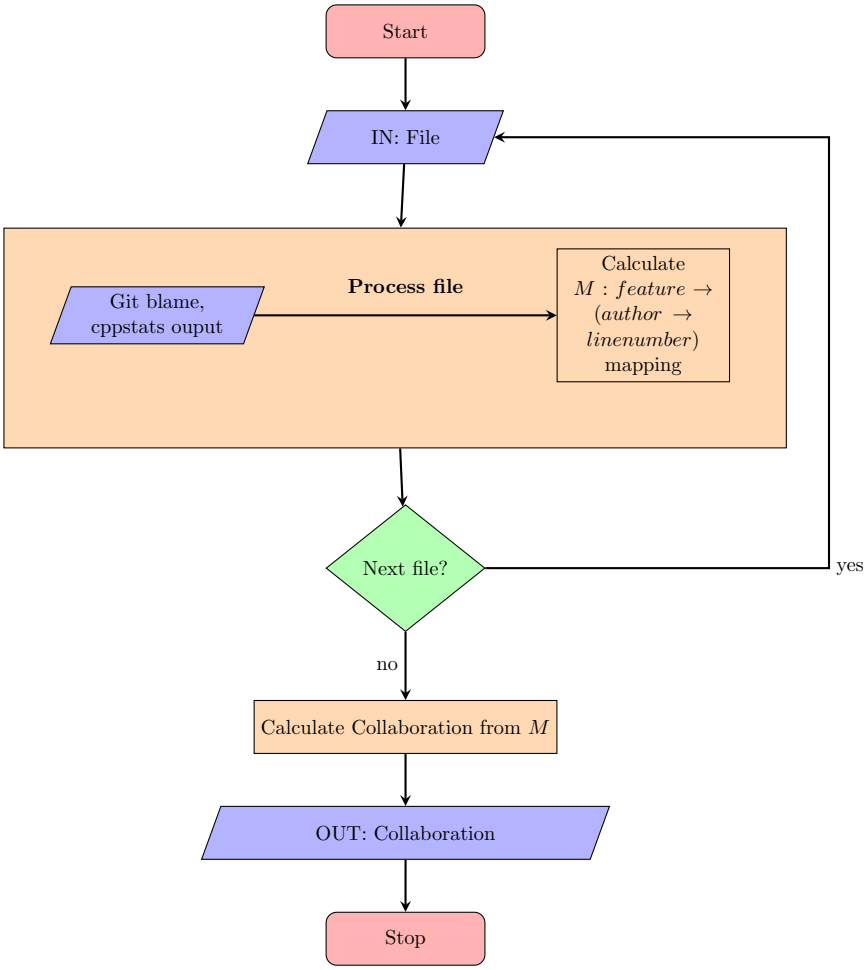


Figure 4.7.: Flowchart of the *Feature* collaboration analysis in CODEFACE. All files are iterated to calculate the number of line changes of an author on any feature. After calculation of this table $M$, we can calculate and output the collaboration matrix as shown in Section 4.3.

describes the developer network or the collaboration graph and is written into the CODEFACE database as an edgelist by using the model of Figure 3.4. The complete process is shown in Figure 4.7.

# 5. Evaluation

In Sections 3 and 4, we discussed the implementation details of the *Function* collaboration analysis within CODEFACE (Section 3.2.2) and the new feature-based analysis methods, namely *Feature* and *Feature_File* (Sections 4.2 and 4.3). In this chapter, we compare the three collaboration analyses, as all of them extract collaboration networks directly from the source code of the VCS. Our focus is to get an initial idea of the validity of the newly generated data and how it compares to previous methods. First, we look very briefly into the performance. Then, we take a closer look on the generated collaboration graphs. Because Joblin et al. have already shown the accuracy of their *Function* analysis [18], we compare our feature-based networks with the function-based ones to get an overview on the real-world relevance of our networks.

## 5.1. Performance

To measure the impact of our changes we run a simple collaboration analysis on an *Intel Core i5-4200 CPU @ 1.6GHZ* with 8 GB RAM and an SSD hard-drive. As operating system an up-to-date *Gentoo* system running within *VirtualBox* and a *Windows* 8.1 host was used. The virtual machine itself had access to 4 GB RAM and 2 cores with a limit of 100% (no limit). As the focus was on graph comparison (Section 5.2) we did not measure the performance in detail, instead we did only a very minimal benchmark.

We used the command `./codeface/runCli.py run --recreate --profile-r -p conf/benchmark.conf -c codeface.conf res/benchmark ../git-repos/` to execute a simple OPENSSL analysis and used the same configuration for each analysis method. All the benchmark configuration files can be found in Appendix C. We ran the OPENSSL analysis command 5 times and took the average run-time of those 5 runs. This process was then repeated 3 times and we took the lowest average value. The results are presented in Table 5.1.

The performance between the different methods is pretty equal with a deviation of about 10% and every run took about 10 to 12 minutes. This clearly shows that our new analysis doesn't introduce any new performance impact.

This matches our experience with long-running CODEFACE analyses, which indicated that there is no significant difference between the used methods. Nevertheless, we want to note here that CODEFACE lacks an internal profiling system, which means it is difficult to measure the run-time of a specific subsystem. Thus we can not exactly know where the run-time is spent within CODEFACE.

| Analysis Method | Time (best of 3)[8] |
|---|---|
| *Function* | 665 sec |
| *Feature* | 709 sec (+6.6%) |
| *Feature_File* | 611 sec (-8.1%) |

measured with `python -m timeit -n 5 -v "__import__('os').system('codeface ...')"`

Table 5.1.: Benchmark results

## 5.2. Graph Comparison

To evaluate the generated feature-based networks we compared the *Feature*- and *Feature_File*-based networks[9] with the networks generated by using the *Function* analysis. We use the same Jaccard-distance-based method Joblin et al. used to compare the function-based (*Function*) against the tagging-based (*Signed-Off / Committer2Author*) approach [13]. Additionally we extend this method to get an idea of how this distance behaves when non-matching nodes and edges are considered as well. The result is a number indicating the dissimilarity between the graphs.

In the following we describe the matching algorithms in detail: Consider two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, which represent two collaboration networks. Now every node represents a developer and every edge a collaboration between two developers. Edges can have a weight, which is modeled by $weight_E : E \mapsto \mathbb{N}$. To compare a matching node $v \in V_1$ and $v \in V_2$, we set the following:

1. $V = V_1 \cap V_2$ (matching nodes)
2. $\tilde{E_1} = E_1 \cap (V \times V)$ (edges of first graph between matching nodes)
3. $\tilde{E_2} = E_2 \cap (V \times V)$ (edges of second graph between matching nodes)
4. $in(E, v) = \{(a, v) \in E\}$ (incoming edges)
5. $out(E, v) = \{(v, a) \in E\}$ (outgoing edges)
6. $matchingEdges(E_1, E_2, v) = |in(E_2, v) \cap in(E_1, v)| + |out(E_2, v) \cap out(E_1, v)|$
7. $totalEdges(E_1, E_2, v) = |in(E_2, v) \cup in(E_1, v)| + |out(E_2, v) \cup out(E_1, v)|$

Further, we define

$$Diff_{node}(G_1, G_2, v) = \begin{cases} \frac{matchingEdges(\tilde{E_1}, \tilde{E_2}, v)}{totalEdges(\tilde{E_1}, \tilde{E_2}, v)} & \text{if } totalEdges(\tilde{E_1}, \tilde{E_2}, v) > 0 \\ 0 & \text{otherwise} \end{cases}$$

$Diff_{node}(G_1, G_2, v)$ describes the difference of the given vertex $v$ between $G_1$ and $G_2$, where 0 means they have the same edges and 1 means they are completely different (have no matching edges).

[9] written as matrix files and saved in the CODEFACE database as list of edges

### Unmerged graph comparison

Now we can define the comparison method by numbering the matching vectors $v_i \in V$ and setting

$$\mathit{Diff}(G_1, G_2) = (\mathit{Diff}_{node}(G_1, G_2, v_1), ..., \mathit{Diff}_{node}(G_1, G_2, v_n)) \tag{5.1}$$

$\mathit{Diff}(G_1, G_2)$ calculates a vector of comparison results of matching nodes between the graphs $G_1$ and $G_2$. This is called the *unmerged* comparison method and it does not consider edges which have an endpoint at a node appearing in only one of the graphs.

### Merged graph comparison

To see what happens if all nodes are considered, we define a similar, *merged*, comparison method:

$$\overline{\mathit{Diff}}(G_1, G_2) = \mathit{Diff}(add(G_1, V_2), add(G_2, V_1)) \tag{5.2}$$

where $add(G, \widehat{V}) = (V \cup \widehat{V}, E)$.

This way, we add the missing nodes to the graph before performing the comparison. $\overline{\mathit{Diff}}(G_1, G_2)$ will therefore always return a vector of comparisons for all nodes, instead of only matching nodes (and without limiting the edges beforehand because $V = V_1 = V_2$).

### Aggregation

To get a single aggregated number for the *Diff* and $\overline{\mathit{Diff}}$ comparison vectors, we explored various possible solutions and terminology as defined in Table 5.2. Basically, we use the simple *average* function and define a modified *averageWeighted* function that takes the weight of the incoming and outgoing edges into account. Then, these functions are used on the vectors generated by the *merged* and the *unmerged* comparison methods.

| **Definition** | ***Name*, Explanation** |
|---|---|
| $average(d_1, ..., d_n) = \frac{\sum_{i=1}^{n} d_i}{n}$ | The *average* calculates the average of the given tuple as returned by *Diff*. |
| $averageWeighted(d_1, ..., d_n) = \frac{\sum_{i=1}^{n} d_i * weight(v_i)}{\sum_{i=1}^{n} weight(v_i)}$<br>where<br>$d_i = Diff_{node}(G_1, G_2, v_i)$ as in equation 5.1 and<br>$Edges_1(v_i) = in(E_1, v_i) \cup out(E_1, v_i)$,<br>$Edges_2(v_i) = in(E_2, v_i) \cup out(E_2, v_i)$ and<br>$weight(v_i) = \sum_{e \in Edges_1} weight_{E_1}(e) + \sum_{e \in Edges_2} weight_{E_2}(e)$ | The *weighted average* calculates an average value which takes the weight of edges into account. |
| $average(Diff(G_1, G_2))$ | The *unmerged total vertex diff* calculates the average difference between two graphs. |
| $average(\overline{Diff}(G_1, G_2))$ | The *(merged) total vertex diff* calculates the average difference between two graphs, by first merging the graphs. |
| $averageWeighted(Diff(G_1, G_2))$ | The *unmerged weighted vertex diff* calculates the average difference between two graphs, by taking the weight of the edges into account. |

Continued on next page

| Definition | Explanation |
|---|---|
| $averageWeighted(\overline{Diff}(G_1, G_2))$ | The *(merged) weighted vertex diff* calculates the average difference between two graphs, by taking the weight of the edges into account and merging the graphs. |

Table 5.2.: Table of used network graph comparison methods.

The previously defined comparison methods, as well as the aggregation methods defined in Table 5.2, are completely implemented in R and are part of the CODEFACE repository[10]. However the comparison has to be started manually and the code used to produce the results of this thesis can be found in Appendix A.

With these comparison methods and aggregations, we empirically check the following hypotheses in a short empirical study:

**H.1** Networks generated based on feature information show the system from another perspective, but are still useful for the creation of collaboration networks. In detail, we argue that if both methods agree on two nodes, they likely agree on the edges between the nodes.

**H.2** *Feature_File* and *Function* analyses agree more than *Feature* and *Function* analyses in comparison.

**H.3** *Feature_File*- and *Function*-based networks agree more if features (*ifdefs*) are used more thoroughly. If features are not used widely enough *Feature_File* should be unable to detect any collaborations.

In the empirical study, we analyzed three 3-month windows of commits for the projects LINUX, LLVM, CLANG, and BUSYBOX. An overview of the projects can be found in Table 5.3 and the analyzed commit windows in Table 5.4.

We selected the projects, because they are representative for a specific set of projects. LINUX, for example, is known for its thorough use of features throughout the source code. BUSYBOX on the other hand is representative for small projects with a limited number of developers and activity. LLVM and CLANG are representative for a minimal use of features, but on the other hand have a very large code-base with lots of developers. For each 3-month window, we calculated the three collaboration networks *Function*, *Feature_file*, and *Feature*. Then, we compared the *Function* collaboration network against the *Feature_file* and *Feature* networks.

[10] At the time of writing, the code was not fully merged, but it can be found as part of this pull request: `https://github.com/siemens/codeface/pull/17`.

| Name | Language | Size (LOC)[1] | Feature Code[2] | Description |
|---|---|---|---|---|
| BusyBox | C | 189 170 | 22.6% | stripped down unix tool box |
| Linux | C | 12 209 732 | 8.9% | free and open source operating system. |
| Clang | C++ | 730 198 | 6.4% | C/C++/Objective-C compiler and library. |
| LLVM | C++ | 793 847 | 6.5% | source and target independent optimizer, generation support for many CPUs. LLVM IR (intermediate representation). |

[1]measured with `sloccount`, Version 2.26
[2]measured with cppstats: normalized Feature LOC / normalized LOC

Table 5.3.: Analyzed projects

Additionally, we randomized both graphs to get an idea on how the comparison values for the original networks compare to those random graphs. If we get the same results with randomized graphs, we can directly follow that the graphs are already different enough such that randomization doesn't make them more different. To randomize the graphs, we rewire edges, i.e., take two edges $(v_1, v_2) \in E$ and $(w_1, w_2) \in E$ and replace them with the edges $(v_1, w_2)$ and $(w_1, v_2)$. We did this $|E|$ times on both graphs and calculated the same comparison metrics as for the non-rewired graphs. The results are presented in the Figures 5.1, 5.2, 5.4, and 5.3.

## 5.3. Results

First, we noticed that one of the release ranges of BusyBox was too small for Codeface to produce a network with any analysis method. The remaining two ranges are very small as well, with only several nodes and edges. This means the networks are too small for sophisticated further analysis like statistical approaches.

In general, we found that the feature-based networks are smaller than the function-based ones. The exception is the *Feature* network in Linux which still has fewer nodes, but is a lot more dense. The details can be found in Table 5.5.

There seems to be no information regarding the weight of the edges as we cannot recognize any tendency regarding the weighted comparison methods. The weighted average comparison seems to be on par with the simple average.

When analyzing Linux we found that the *Feature_file* analysis yields developer networks that are more similar to *File*-based networks than the *Feature*-based networks. Figure 5.2 shows this for example when comparing the bars (e) with (h) by the higher similarity on (h) when using the *unmerged* comparison method. On the other hand

| IDs | Project | **Start SHA1** End SHA1 is below |
|---|---|---|
| (a),(b) | BusyBox | `50e4cc29f56b5401579cddaf42d0404320b9efa7` |
| (c),(d) | BusyBox | `539e2802ebd2680602de0a2c76069b7f555392d9` |
| none | end of above | `feac9b607dc68ea63992a46b3b8361f00f663cdc` |
| (e),(f) | Linux | `a8b33654b1e3b0c74d4a1fed041c9aae50b3c427` |
| (g),(h) | Linux | `e560623050693d2550d0bfb3b092e6398249176e` |
| (i),(j) | Linux | `9797eb83c85d553dbc062b2953597fb1086649e9` |
| none | end of above | `64aa90f26c06e1cb2aacfb98a7d0eccfbd6c1a91` |
| (k),(l) | Clang | `3390ed3341a0848f37333d55bb34bcff4b9364c2` |
| (m),(n) | Clang | `04f9ed5f8cf30195a85743d43e56d06f62b87ce2` |
| (o),(p) | Clang | `5b0b279f796ecf91b10ba8b0ca89f9dbf802bae4` |
| none | end of above | `33947ed22c57e11e7aa88e803bfdd664fe50412f` |
| (q),(r) | LLVM | `c413e016723673ca93d5700c72083194ac21b766` |
| (s),(t) | LLVM | `99cd10fe111560b9921e731a89109972a149dfab` |
| (u),(v) | LLVM | `e1e1392e13ef3f5e97fbdd15ca8d60d63f69a862` |
| none | end of above | `186332c0c98aab21acb91ae11055e44ec1acb95a` |

Table 5.4.: Analyzed commit-ranges and their exact **SHA1** hashes. All ranges are continuous, therefore the table only shows the beginning of a range. The end of the range is given by he start of the next range or a separate row. Because every range is used for two comparisons, *Feature* and *Feature_File* with *Function*, it is assigned two **IDs** which identify the comparison. The exact comparison can be found in Table 5.5.

when analyzing LLVM or Clang (Figure 5.4 and 5.3) the *Feature* analysis yield higher similarity with the *Function*-based approach.

The results show that the *merged* networks are more different to each other than their *unmerged* counterparts. This can be seen on all comparisons, for example, in Figure 5.3 (k) by comparing the first 4 bars (*merged*) with the last 4 (*unmerged*).

Additionally, we see that the corresponding randomized graph comparison always yield a higher result, besides three minor exceptions (in (d) and (r)). This means the randomized graphs are more different to each other.

| **Id** | **Compared with** | **Vertex** *Function* | **Vertex** | **Edges** *Function* | **Edges** |
|---|---|---|---|---|---|
| BusyBox | | | | | |
| (a) | *Feature* | 5 | 4 | 4 | 4 |
| (b) | *Feature_File* | 5 | 4 | 4 | 2 |
| (c) | *Feature* | 6 | 4 | 7 | 6 |
| (d) | *Feature_File* | 6 | 3 | 7 | 2 |
| Linux | | | | | |
| (e) | *Feature* | 610 | 242 | 1159 | 3178 |
| (f) | *Feature_File* | 610 | 110 | 1159 | 154 |
| (g) | *Feature* | 851 | 367 | 2000 | 10994 |
| (h) | *Feature_File* | 851 | 199 | 2000 | 245 |
| (i) | *Feature* | 760 | 309 | 1662 | 7152 |
| (j) | *Feature_File* | 760 | 161 | 1662 | 293 |
| Clang | | | | | |
| (k) | *Feature* | 103 | 28 | 1730 | 170 |
| (l) | *Feature_File* | 103 | 19 | 1730 | 19 |
| (m) | *Feature* | 104 | 19 | 1360 | 54 |
| (n) | *Feature_File* | 104 | 13 | 1360 | 11 |
| (o) | *Feature* | 98 | 21 | 1202 | 114 |
| (p) | *Feature_File* | 98 | 11 | 1202 | 7 |

| Id | Compared with | Vertex *Function* | Vertex | Edges *Function* | Edges |
|---|---|---|---|---|---|
| LLVM | | | | | |
| (q) | *Feature* | 128 | 27 | 1826 | 454 |
| (r) | *Feature_File* | 128 | 17 | 1826 | 16 |
| (s) | *Feature* | 139 | 16 | 1491 | 170 |
| (t) | *Feature_File* | 139 | 10 | 1491 | 13 |
| (u) | *Feature* | 146 | 27 | 1833 | 316 |
| (v) | *Feature_File* | 146 | 13 | 1833 | 8 |

Table 5.5.: The table shows the network sizes of the developer networks before performing the comparison. The **Id** specifies the project and the commit window from Table 5.4. Therefore, every line specifies two graphs that are compared later on. One graph is always given by the *Function*-based network and the other one is specified in the **Compared with** column.

## 5.4. Interpretation

The main focus of this thesis is on assessing the feature-aware data and the integration of the according changes into the CODEFACE infrastructure. Therefore, we provide only general interpretations, whereas in-depth analysis of the data will be done in future work.

The failure of one BUSYBOX range shows that we need a minimum number of regular commits and changes within the VCS system to successfully create a developer network. All three methods are similar in this regard, as all failed to provide a sophisticated network.

From the network sizes, we conclude that the feature-based approaches cannot capture all developers and relate them to each other. This is most likely due to the fact that our feature-based methods do not relate developers which are only working in the shared code base. Our feature-based methods only relate developers working on feature-specific code. The density exception in the LINUX analysis can most likely be explained with some generic feature symbols which relate a lot of developers to each other. For example, symbols like `DEBUG` or `CONFIG_PM` are used frequently within the LINUX code-base.

We are surprised by the LLVM analysis due the fact that the *Feature*-based networks have a higher similarity to the *Function*-based networks than the *Feature_File*-based networks. We generally expected *Feature_File*-based networks to be more similar as their generation process is almost identical to the *Function*-based networks and it is still

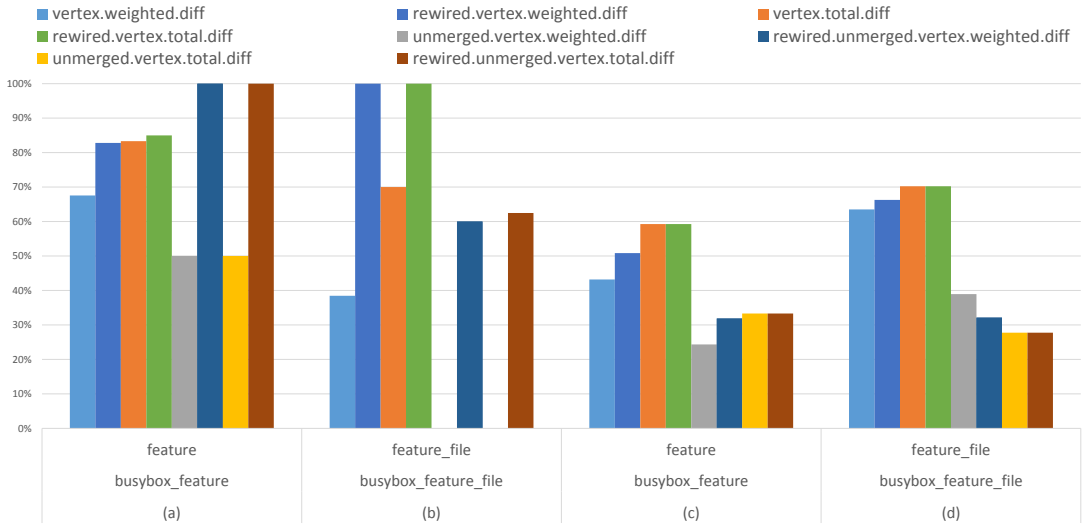


Figure 5.1.: Four comparisons of 3-month windows of commits for BusyBox. (a) and (c) are two comparisons between the *Function* and the *Feature* networks, while (b) and (c) compare the *Function* and the *Feature_File* networks. (a) and (b) as well as (c) and (d) are created from the same commit window. A height of 0% means totally equal, while 100% means totally different. One commit window of BusyBox does not contain enough commits to create a developer network for any analysis method.

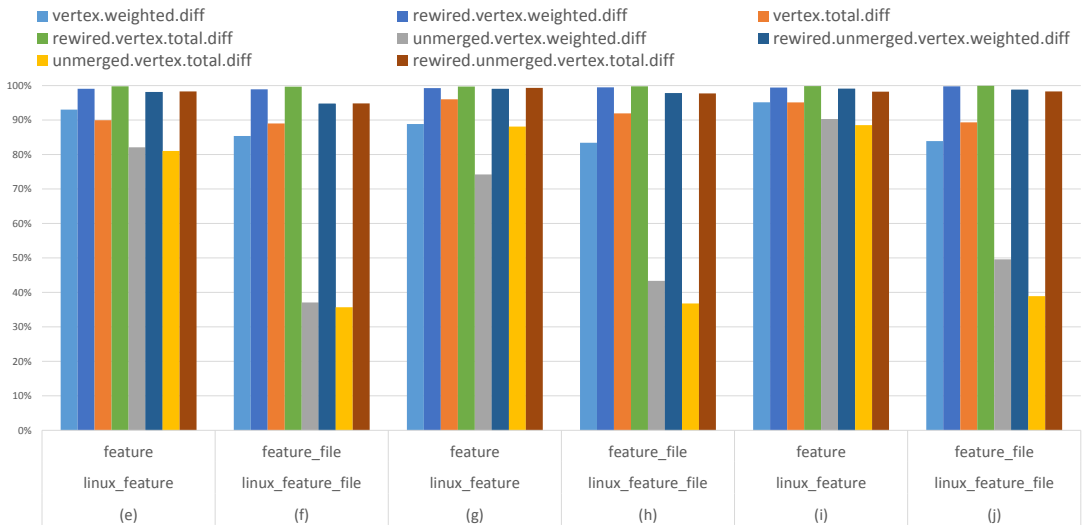


Figure 5.2.: Six comparisons of 3-month windows of commits for LINUX. The order of the bars is the following: Two bars form the same range, so (e) and (f), (g) and (h), and (i) and (j) are generated from the same commits respectively, as seen in Table 5.3. The bars (e), (g), (i) show the comparison results between the *Function* and *Feature* analysis, while (f), (h), (j) show the comparison between the *Function* and the *Feature_File* analysis. The names of the bars match the comparison methods defined in Table 5.2. A height of 0% means totally equals, while 100% means totally different.

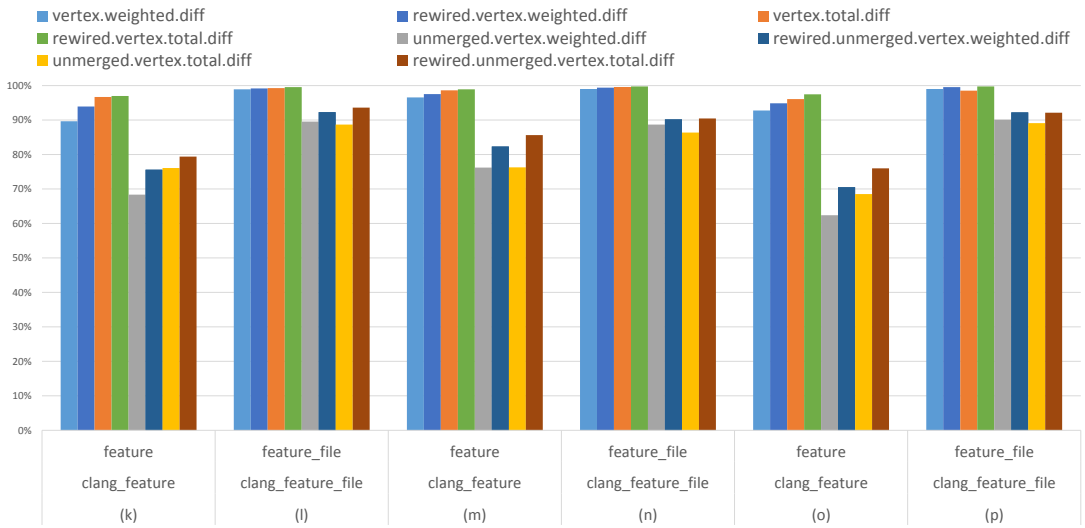


Figure 5.3.: Six comparisons of 3-month windows of commits for Clang. The order of the bars is the following: Two bars form the same range, so (k) and (l), (m) and (n), and (o) and (p) are generated from the same commits respectively, as seen in Table 5.3. The bars (k), (m), (o) show the comparison results between the *Function* and *Feature* analysis, while (l), (n), (p) show the comparison between the *Function* and the *Feature_File* analysis. The names of the bars match the comparison methods defined in Table 5.2. A height of 0% means totally equals, while 100% means totally different.

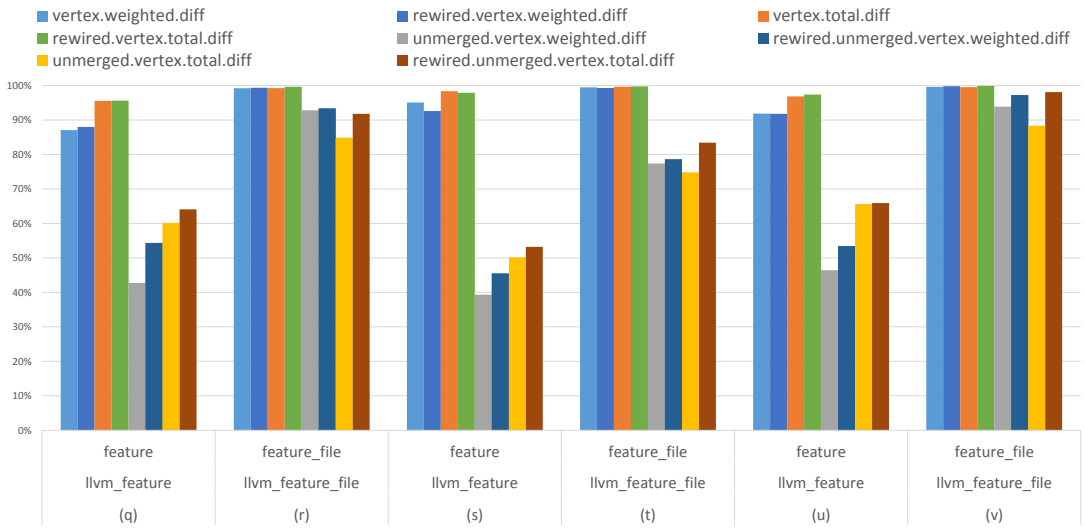


Figure 5.4.: Six comparisons of 3-month windows of commits for LLVM. The order of the bars is the following: Two bars form the same range, so (q) and (r), (s) and (t), and (u) and (v) are generated from the same commits respectively, as seen in Table 5.3. The bars (q), (s), (u) show the comparison results between the *Function* and *Feature* analysis, while (r), (t), (v) show the comparison between the *Function* and the *Feature_File* analysis. The names of the bars match the comparison methods defined in Table 5.2. A height of 0% means totally equals, while 100% means totally different.

file-based instead of project-wide. As the feature networks for LLVM are quite small in comparison to the function-based network, it seems to be more important to detect a larger number of connection (using the *Function* analysis) instead of using time and file-restrictions. However, we still accept our hypothesis that *Feature_File* is more similar to *Function*, but we need to restrict the hypothesis to projects using a minimum amount of features (*ifdefs*, in this case) (**H.2**).

We accept **H.3** as we can see that for Linux, which uses feature code more thouroughly with 8.9%, the *Feature_File* similarity is higher than for LLVM, which uses less feature code with 6.5%.

As we can generally see that *unmerged* comparisons yield higher similarity than *merged* comparison and because random graphs always yield more different results we accept our initial assumption, that feature-based networks show a different perspective of the system and are useful to capture feature-specific collaborations (**H.1**). We can see this exceptionally well in the Linux comparison (Figure 5.2) where all random graphs do not yield any similarity.

# 6. Conclusion

This thesis provided initial results from feature-based collaboration networks generated by two new methods, implemented as part of CODEFACE within the scope of this thesis. We have shown how CODEFACE works, how the new feature-based methods are implemented, provided initial results and an initial interpretation of the feature-based results. The main focus of this work was to provide a foundation for future research. If the new *feature*-based networks emerge as being to dense to analyze (our feature-based LINUX analysis generated dense networks), there are still methods to limit the number of edges by taking the ordering of commits and the direction of collaboration into account[11]. We noticed that the feature-based networks do not seem to relate nearly as many developers as the function-based networks do, so it should be considered to either relate developers based on non-feature code (the shared code base) or to fall back to function-based collaboration if the generated networks are too dense by directly relating the shared code base.

Another point that could be improved is the feature detection. Currently only *ifdefs* are supported, but not many projects use *ifdefs* throughout the project. Finally, the weight of a collaboration could be defined differently; currently, we use the minimum number of changed lines of two developers as their collaboration-strength (in the *feature*-based analysis), but we could use the sum or the maximum or a custom defined function as well. This would possibly change our weight-based comparisons. Taking this number in context to the size of the feature would also make sense.

After analyzing LLVM and CLANG, the question arises[12] how many features need to be used within a project for the data to become useful. We possibly need to introduce a *relative feature usage* scale which relates feature-specific to feature-shared code and gives an idea of the usefulness of our feature-based methods on a given project. Future work can possibly use this scale to evaluate which method is more accurate on a given project or file. Of course, additionally, we need to repeat the comparisons for other and more feature-using projects and relate the data or find general patterns.

We can also use this data for evolutionary analysis, for example an interesting question is: How does the comparison between *function*- and *feature*-based collaboration networks change over time or how do the networks themselves change?

[11] The *Feature_File* and *Function*-based methods already take direction and ordering into account.

[12] Their rewired(=random) comparison was not reasonably different to our data.

# A. Comparing networks

loadArgs.R

```
library(Rcpp)
library(igraph)
library(BiRewire)

# Setup the environment
codeface.dir <- "/mnt/codeface/projects/codeface"
R.wd <- "/codeface/R/cluster"
setwd(paste(codeface.dir, R.wd, sep="/"))

# In case you run out of connections
#library(RMySQL)
#for (con in (dbListConnections(MySQL()))) dbDisconnect(con)

# Load Codeface
source("../db.r", chdir=T)
source("../query.r", chdir=T)
source("../config.r", chdir=T)
source("../utils.r", chdir=T)
source("community_metrics.r")
source("graph_comparison.r")

# establish database connection
con <- connect.db("../../../codeface.conf")$con

## Configure graph comparison ids
## Extract the range ids from the codeface database
compare.ranges <- data.frame(
  original= c(172, 172, 173, 173,           # BUSYBOX
              208, 209, 210, 208, 209, 210, # LINUX
              202, 202, 203, 203, 204, 204, # CLANG
              214, 215, 216, 214, 215, 216),# LLVM
  compare = c(175, 178, 176, 179,           # BUSYBOX
              235, 236, 237, 238, 239, 240, # LINUX
              199, 205, 200, 206, 201, 207, # CLANG
              217, 218, 219, 220, 221, 222))# LLVM

# Run the comparsion
compare.result <- run.batch.comparison(con, compare.ranges)

# View the data
View(compare.result$overview)
print(compare.result$vertexdata)
```

```
# Get data of a specific comparison (here range-id 172 with range-id 175)
View(compare.result$vertexdata$'172/175')

# Use regular R to plot/save/extract data from the "compare" data-frame
```

---

# B. Codeface configuration files

conf/busybox_proximity.conf

```
---
project: busybox_proximity
repo: busybox
description: busybox_proximity
mailinglists:
    -   name: gmane.comp.emulators.qemu
        type: dev
        source: gmane
revisions: []
rcs: []
tagging: proximity
```

conf/busybox_feature.conf

```
---
project: busybox_feature
repo: busybox
description: busybox_feature
mailinglists:
    -   name: gmane.comp.emulators.qemu
        type: dev
        source: gmane
revisions: []
rcs: []
tagging: feature
```

conf/busybox_feature_file.conf

```
---
project: busybox_feature_file
repo: busybox
description: busybox_feature_file
mailinglists:
    -   name: gmane.comp.emulators.qemu
        type: dev
        source: gmane
revisions: []
```

```
rcs: []
tagging: feature_file
```

conf/linux_proximity.conf

```
---
project: linux_proximity
repo: linux
mailinglists:
    - {name: gmane.comp.encryption.proximity.devel, type: dev, source: gmane}
description: linux proximity
revisions: []
rcs: []
tagging: proximity
```

conf/linux_feature.conf

```
---
project: linux_feature
repo: linux
mailinglists:
    - {name: gmane.comp.encryption.feature.devel, type: dev, source: gmane}
description: linux feature
revisions: []
rcs: []
tagging: feature
```

conf/linux_feature_file.conf

```
---
project: linux_feature_file
repo: linux
mailinglists:
    - {name: gmane.comp.encryption.feature_file.devel, type: dev, source: gmane}
description: linux feature_file
revisions: []
rcs: []
tagging: feature_file
```

conf/clang_proximity.conf

```
---
project: clang_proximity
```

```
repo: clang
description: clang_proximity
mailinglists:
    -   name: gmane.comp.emulators.qemu
        type: dev
        source: gmane
revisions: []
rcs: []
tagging: proximity
```

conf/clang_feature.conf

```
---
project: clang_feature
repo: clang
description: clang_feature
mailinglists:
    -   name: gmane.comp.emulators.qemu
        type: dev
        source: gmane
revisions: []
rcs: []
tagging: feature
```

conf/clang_feature_file.conf

```
---
project: clang_feature_file
repo: clang
description: clang_feature_file
mailinglists:
    -   name: gmane.comp.emulators.qemu
        type: dev
        source: gmane
revisions: []
rcs: []
tagging: feature_file
```

conf/llvm_proximity.conf

```
---
project: llvm_proximity
repo: llvm
description: llvm_proximity
mailinglists:
```

```
    -   name: gmane.comp.emulators.qemu
        type: dev
        source: gmane
revisions: []
rcs: []
tagging: proximity
```

conf/llvm_feature.conf

```
---
project: llvm_feature
repo: llvm
description: llvm_feature
mailinglists:
    -   name: gmane.comp.emulators.qemu
        type: dev
        source: gmane
revisions: []
rcs: []
tagging: feature
```

conf/llvm_feature_file.conf

```
---
project: llvm_feature_file
repo: llvm
description: llvm_feature_file
mailinglists:
    -   name: gmane.comp.emulators.qemu
        type: dev
        source: gmane
revisions: []
rcs: []
tagging: feature_file
```

# C. Codeface benchmark files

conf/benchmark.conf

```
---
project: benchmark
repo: openssl
mailinglists:
    - {name: gmane.comp.encryption.openssl.devel, type: dev, source: gmane}
description: Benchmark test
revisions: [ ]
rcs: []
tagging: proximity
```

conf/benchmark_feature.conf

```
---
project: benchmark_feature
repo: openssl
mailinglists:
    - {name: gmane.comp.encryption.openssl.devel, type: dev, source: gmane}
description: Benchmark test
revisions: [ ]
rcs: []
tagging: feature
```

conf/benchmark_feature_file.conf

```
---
project: benchmark_file
repo: openssl
mailinglists:
    - {name: gmane.comp.encryption.openssl.devel, type: dev, source: gmane}
description: Benchmark test
revisions: [ ]
rcs: []
tagging: feature_file
```

# Bibliography


[1] Codeface home page. `http://siemens.github.io/codeface/#/home`. Accessed: 2015-03-26.

[2] cppstats home page. `http://www.infosun.fim.uni-passau.de/cl/staff/liebig/cppstats/`. Accessed: 2015-06-23.

[3] Ctags home page. `http://ctags.sourceforge.net/`. Accessed: 2015-06-23.

[4] Doxygen home page. `http://www.stack.nl/~dimitri/doxygen/`. Accessed: 2015-06-23.

[5] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 137–143. ACM, 2006.

[6] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 24–35. ACM, 2008.

[7] Christian Bird, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German, and Prem Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 1–10. IEEE, 2009.

[8] Marcelo Cataldo and James D Herbsleb. Architecting in software ecosystems: interface translucence as an enabler for scalable collaboration. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 65–72. ACM, 2010.

[9] Marcelo Cataldo and James D Herbsleb. Coordination breakdowns and their impact on development productivity and software failures. *Software Engineering, IEEE Transactions on*, 39(3):343–360, 2013.

[10] Subhajit Datta, Vikrant Kaulgud, Vibhu Saujanya Sharma, and Nishant Kumar. A social network based study of software team dynamics. In *Proceedings of the 3rd India software engineering conference*, pages 33–42. ACM, 2010.

[11] Cleidson RB De Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. How a good software practice thwarts collaboration: the multiple roles

of apis in software development. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 221–230. ACM, 2004.

[12] Steven D Eppinger, Daniel E Whitney, Robert P Smith, and David A Gebala. A model-based method for organizing tasks in product development. *Research in Engineering Design*, 6(1):1–13, 1994.

[13] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[14] Critina Gacek and Michalis Anastasopoules. Implementing product line variabilities. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 109–117. ACM, 2001.

[15] James Howison, Keisuke Inoue, and Kevin Crowston. Social dynamics of free and open source team communications. In *Open Source Systems*, pages 319–330. Springer, 2006.

[16] Shih-Kun Huang and Kang-min Liu. Mining version histories to verify the learning process of legitimate peripheral participants. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[17] Andrejs Jermakovics, Alberto Sillitti, and Giancarlo Succi. Mining and visualizing developer networks from version control systems. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 24–31. ACM, 2011.

[18] M. Joblin, W. Maurer, S. Apel, and D. Riehle. From developer networks to verified communities: A fine-grained approach. Submitted.

[19] Andrea Lancichinetti, Filippo Radicchi, José J Ramasco, and Santo Fortunato. Finding statistically significant communities in networks. *PloS one*, 6(4):e18961, 2011.

[20] Jörg Liebig, Sven Apel, Christian Lengauer, C Kastner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 105–114. IEEE, 2010.

[21] Luis Lopez-Fernandez, Gregorio Robles, Jesus M Gonzalez-Barahona, et al. Applying social network analysis to the information in cvs repositories. In *International Workshop on Mining Software Repositories*, pages 101–105. IET, 2004.

[22] Luis López-Fernández, Gregorio Robles, Jesus M Gonzalez-Barahona, and Israel Herraiz. Applying social network analysis techniques to community-driven libre software projects. *vol*, 1:28–50, 2008.

[23] Andrew Meneely and Laurie Williams. Socio-technical developer networks: should we trust our measurements? In *Proceedings of the 33rd International Conference on Software Engineering*, pages 281–290. ACM, 2011.

[24] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23. ACM, 2008.

[25] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering*, pages 521–530. ACM, 2008.

[26] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12. ACM, 2008.

[27] SL Toral, MR Martínez-Torres, and Federico Barrero. Analysis of virtual communities supporting oss projects using social network analysis. *Information and Software Technology*, 52(3):296–303, 2010.

[28] Frank J van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media, 2007.

[29] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540. ACM, 2008.

# Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Master-Arbeit 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, sowie dass ich die Master-Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, den 24. Juni 2015

_____

Matthias Dittrich