Development of the GitCouplingTool for Change Coupling Analysis and its Evaluation

Bachelor's Thesis

Sven Hofmann

September 29, 2021

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Priv.-Doz. Dr. Henning Schnoor
M.Sc. Sören Henning

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 29. September 2021

Abstract

Software systems are constantly increasing in size and complexity. However, increasing complexity leads to more likely software failures. Coupling Analysis is one way of measuring the complexity of a system. A particular type of Coupling Analysis - *Change Coupling Analysis* - is examined in more detail. This thesis aims to create a valuable and high-performance tool for measuring the coupling of software systems tracked in Git repositories. It turns out that such a tool can analyze large repositories with more than a million commits in a very manageable amount of time. In addition, the metric of the developed *GitCouplingTool* is compared with the metric of a tool for measuring Static Coupling in order to identify possible advantages or disadvantages.

Contents

List of Figures

List of Acronyms

DRC	Directed Raw Counting
GML	Graph Modelling Language
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
PWD	Present Working Directory
URC	Undirected Raw Counting
VCS	Version Control System
WSL	Windows Subsystem for Linux

Chapter 1

Introduction

This thesis aims at creating a tool for Change Coupling analysis in Git repositories, resulting in the GitCouplingTool. The GitCouplingTool is a console application written in Java to generate graphs representing the coupling of files within a Git repository. Thanks to its utilization of multi-core CPUs, it can analyze repositories with hundreds of thousand commits within minutes or even seconds. The source code for this tool can be found on GitHub `https://github.com/svnlib/GitCouplingTool`.

As this thesis is part of a bachelor's project that is done in partnership with Voigt [2021], the results generated by each tool are compared.

1.1 Motivation

The motivation for this tool is to provide a robust and performant solution for analyzing Change Coupling in Git repositories. A similar tool was already created by Issa [2021], but it is mainly suitable for small repositories with a few hundred to a maximum of a few thousand commits, depending on the amount of available memory. Large amounts of commits cause the tool to take up large amounts of memory space, making it impossible to use for large projects. The GitCouplingTool, developed in this thesis, focuses on large repositories with hundreds of thousands of commits. The advantage in performance and decrease in memory consumption is prooven later in Chapter 4.

Another motivation for this thesis is to compare the results of the GitCouplingTool with the results of the StaticCouplingTool by Voigt [2021]. The StaticCouplingTool is part of the same bachelor's project as the GitCouplingTool.

1.2 Goals

This thesis focuses on two primary goals. (1) The first goal is to create a tool for Change Coupling Analysis that can analyze huge repositories with hundreds of thousand commits in a reasonable time and memory consumption. (2) The second goal is to compare the GitCouplingTool's metric with the StaticCouplingTool's one by Voigt [2021]. This comparison should give information about the comparability and advantages of each tool.

1.3 Approach

Especially the memory consumption is a problem in the tool of Issa [2021] when analyzing large repositories with thousands of commits. This problem is solved by storing commits as succinctly in the memory as possible. The commits must be processed directly without collecting them in a list structure to keep the memory usage low. Creating a pipeline that takes in the commits, processes them, and delivers a list of diffs for each commit would perfectly fit our needs. With such a pipeline, it is possible to first collect a commit by reading the first object file from Git, second to filter the commit for user-defined criteria, and lastly processing the diff algorithm. The last step of this pipeline is quite CPU expensive so parallelize it would be a good idea. All other information can be discarded along this pipeline. As a result, the memory consumption is lowered, and the performance is increased by parallelizing the diffing algorithm. The CPU utilization with this approach is almost 100%, leaving quite enough room for other tasks such as displaying the progress.

1.4 Bachelor's Projekt

As mentioned before, this thesis is part of a bachelor's project to compare Change Coupling with StaticCouplingTool Coupling. Change Coupling is analyzed in this thesis while Static Coupling is in Voigt [2021]. Each thesis aims at developing a tool for either Change or Static Change Analysis. The resulting tools are the GitCouplingTool and the StaticCouplingTool [Voigt 2021]. Another goal the theses have in common is the comparison of the results generated by the tools. Therefore both theses have the same Section 2.1 and a similar Section 2.2 and Section 2.3. The comparison Chapter 6 can also be found in both theses. Lastly, the conclusion about the comparison (in Section 7.1) is the same. The other chapters and sections are only part of this thesis.

Chapter 2

Foundation and Related Work

This chapter gives the foundation to *Coupling Analysis* as well as *Change Coupling* and other essential topics. Afterward, this foundation is used to develop an approach for creating the GitCouplingTool.

2.1 Coupling Analysis

Nagappan et al. [2006] have shown that complexity metrics can predict software defects. Coupling Analysis is one possible metric that is suitable for some projects. Coupling describes the inner connection of software components. Measuring coupling can be done in many ways. "In the literature, there exists a wide range of different approaches to defining and measuring coupling." [Schnoor and Hasselbring 2020] In the case of this thesis, the measuring of dependence between files is considered. How files can be dependent on each other is defined in Section 2.3 and Section 2.2. Each of these sections describes a particular type of measuring the coupling between source code files.

2.2 Change Coupling

Oliva and Gerosa [2015] define *Change Coupling* as a connection between artifacts "from an evolutionary point of view". In this case, an artifact describes a tracked file of a repository that might change its name over time. Change Coupling provides two major benefits over Static Coupling. First, Change Coupling can identify hidden relations that are not visible in code. Second, it is based entirely on the changelog of a repository, making additional steps related to the project's code unnecessary [Oliva and Gerosa 2015]. For instance, the StaticCouplingTool by Voigt [2021] has to build an Abstract Syntax Tree to be able to measure the Static Coupling. Only depending on the changelog makes Change Coupling independent from specific languages. The GitCouplingTool developed in this thesis can analyze every Git repository, making it applicable to every software system tracked in Git.

Nevertheless, Change Coupling is not the one solution for Coupling Analysis suitable for every project. Some repositories are more suitable for Change Coupling than others. Like Oliva and Gerosa [2015] mention, repositories that link commits to tasks are more suitable because such links make commits more contextual. Furthermore, repositories that often move or migrate data are less suitable. Renamed or moved files can sometimes be

detected but in other cases not, which results in duplicate artifacts and less accurate results. Mirror repositories are another counterexample as they tend to merge commits since the last synchronization. That makes it impossible to measure the Coupling of individual artifacts.

2.3 Static Coupling

The GitCouplingTool tool is compared to the StaticCouplingTool by Voigt [2021] in a Chapter 6. Therefore this section (adapted from Voigt [2021]) gives a short introduction to Static Coupling.

Static Coupling is the coupling that can be measured by analyzing the source code or the compiled code of a program [Schnoor and Hasselbring 2020]. That means that analyzing Static Coupling is language-specific. Thus, for different languages, there may be different approaches to measure the coupling. Possible approaches for C++ are discussed in [Briand et al. 1997]. These approaches are:

- Class-Attribute interaction (class A has an attribute of class type B)
- Class-Method interaction (class A has a method which signature contains class type B)
- Method-Method interaction (method of class A calls a method of class B)

There can also be a relationship between classes caused by Inheritance or friendship [Briand et al. 1997]. However, these approaches will not be used in this thesis. Instead, a coupling metric is used, which considers coupling as a dependency between source code files. In [Schnoor and Hasselbring 2020] coupling is considered as a dependency graph.
File coupling for static analysis in C and C++ can also be represented as such a dependency graph. The nodes of this graph are the names of the source code files which are analyzed. Since C and C++ supports separated header and source files, header and source files that only differ in their file extension can be seen as a single node in this graph to prevent a high coupling caused by implementing header signatures in a separate source file. In Static Coupling, the edges of this graph are references in source code from the current analyzed file to another file. An example is a method call. If in file `A` a method which is defined in file `B` is called, there is an directed edge from node `A` to `B` in the corresponding file coupling graph.

2.4 Introduction to Git's Datastructure

To better understand the functionality of the GitCouplingTool, this section will give a brief overview of the internal data storage of Git. Internally Git uses a tree-based data structure whose nodes are stored in files called objects. These objects are located under `.git/objects` within the repository's folder. Objects are divided into three types: commit, tree,

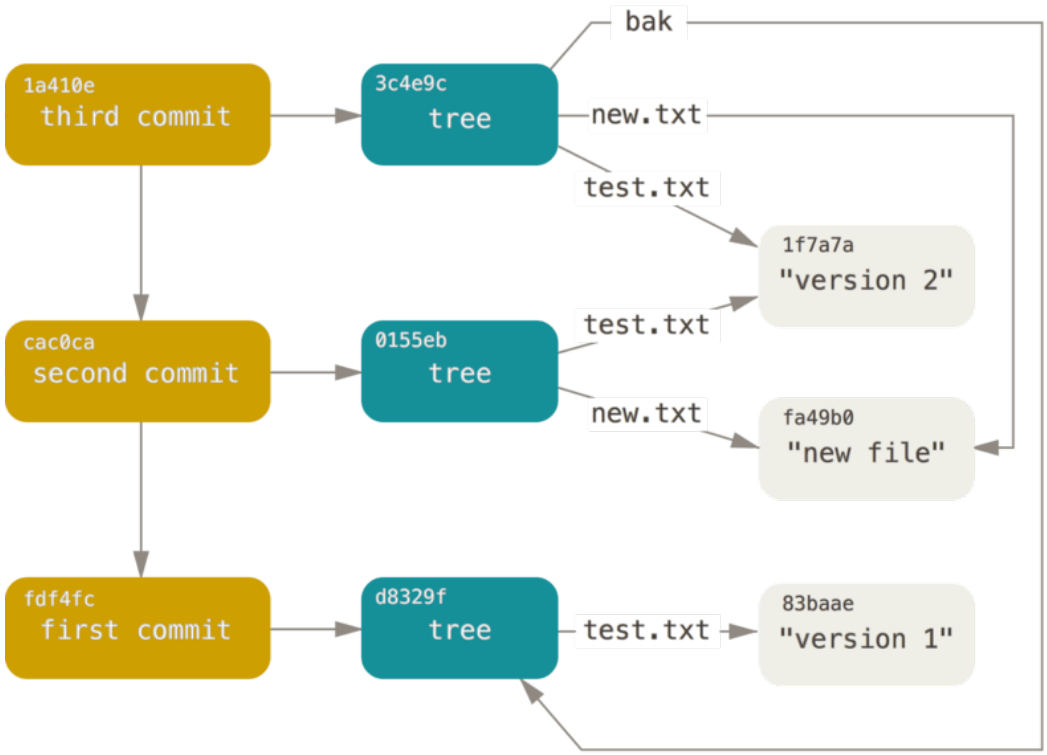


Figure 2.1. Git's internal data structure [git-internals]

and blob. As the name suggests, commit-objects store the information about a commit, like a commit's author, the date, and the message. Further, they link to the previous commit and a tree object representing the folder structure and files at the commit. A tree object represents a folder and links to other trees (subfolders) or blobs containing the file's data [git-internals] (see Figure 2.1). How the tool utilizes this data structure to achieve maximum performance is discussed in the following section.

Chapter 3

Development

This chapter is about the actual implementation of the GitCouplingTool. It introduces the libraries, data structures, algorithms, and architecture used in the GitCouplingTool.

3.1 Project Structure

As the GitCouplingTool is written in Java, it is structured as a Gradle project. Gradle is utilized for dependency resolution as well as the building process. For the latter, the Gradle plugin shadowJar is used [shadowjar]. This plugin gives the ability to produce jar files, including all necessary libraries to execute it. A jar file can be created using this plugin by `./gradlew shadowJar`. The execution of the created jar file is described in Section 5.2.

3.2 Libraries

In the following, the used libraries are introduced. These libraries are mainly the pipeline framework TeeTime and the Java Git implementation JGit. Some other utility libraries are also mentioned in the following.

3.2.1 TeeTime

The first library is *TeeTime* [teetime] which is used to create the already mentioned pipelines. It provides an easy way to define stages like filters and transformations and delivers multithreading ability out of the box. Custom stages that run in separate threads in parallel can be created using the build-in distributor and merge stages. Parallelization is essential to achieve satisfactory performance.

3.2.2 JGit

The second library is *JGit* [jgit]. JGit is a Java implementation of Git and allows to perform all kinds of operations with Git repositories. The GitCouplingTool uses the ability to walk the object tree described in Section 2.4 to collect the commits, which then get fed in the TeeTime pipeline. Another used ability of JGit is the diff-algorithm to get a list of files that changed from one commit to the next.

3.2.3 Utility Libraries

- *picocli* - for parsing program arguments and creating helpful information for the user. Command options are described by annotated class fields, which makes this library easy and clear to use [picocli].

- *Progressbar* - for creating progress bars on the standard output. It shows not only the progress but also the speed of units per second. Displaying the speed can be useful to compare different systems and repository sizes. The output of the library runs on a separate thread which makes this library more responsive [progressbar].

- *JColor* - for coloring strings in the console output. The library is used for formating the configuration overview at the beginning of the program execution [jcolor].

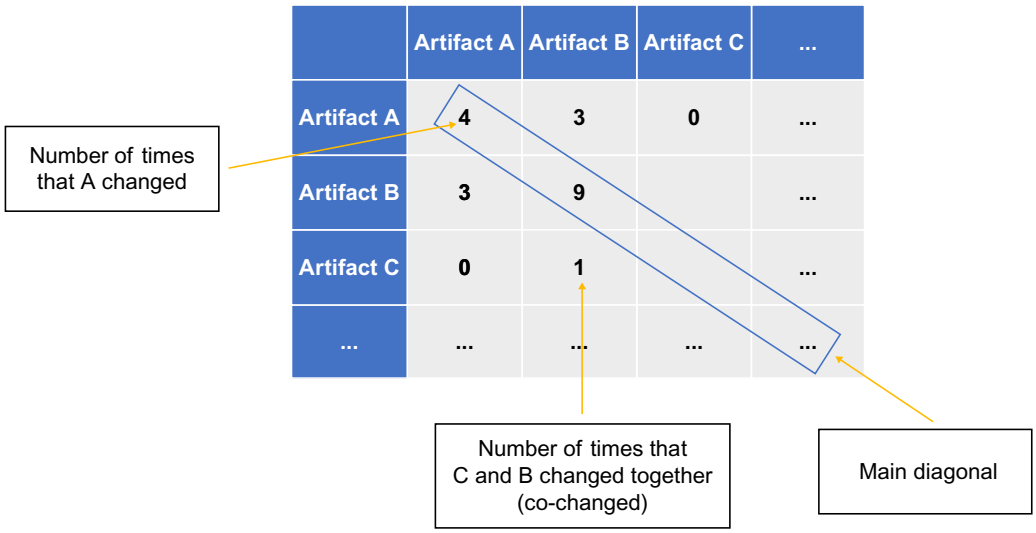
3.3 Architecture

The central part of the architecture is based on two pipelines created with the framework TeeTime. One of them is responsible for collecting the commits, filtering them for user-defined options, and parsing the diffs for each commit. The resulting diffs are collected in a list at the end of the pipeline. When the first pipeline finishes, the collected diffs are fed into the second pipeline, responsible for performing the coupling algorithm. The implemented coupling algorithms are described in Section 3.4. A more in-depth description of the pipeline structure is given in Section 3.6. After the second pipeline finishes, the collected data gets formed into a coupling graph exported into a file in a format chosen by the user. The available export formats are mentioned in Section 5.3.

3.4 Algorithms

The GitCouplingTool uses the *Raw Counting* approach for measuring the Change Coupling described by Oliva and Gerosa [2015]. The same approach is used by Issa [2021]. Each commit of a branch gets converted into a set of changed artifacts. (See Section 3.6) In a later step these change-sets are transferred into a *co-change matrix*. (See Figure 3.1) The co-change matrix is a symmetrical $N \times N$ matrix with N being the number of artifacts in total. Each cell $[i, j]$ represents how often the artifacts i and j have been changed together. Cells where $i = j$ represent the number of times the artifact i changed in total.

After every change-set has been transferred into the co-change matrix, two strategies can identify the coupling. These strategies are Undirected Raw Counting (URC) and Directed Raw Counting (DRC). Both are implemented and create a coupling graph, representing the coupling (edges) between the artifacts (nodes).



.

Figure 3.1. Hypothetical co-change matrix [Oliva and Gerosa 2015]

3.4.1 Undirected Raw Counting (URC)

URC, as the name suggests, results in an undirected relation between artifacts. Each cell below the main diagonal with $[i, j] \neq 0$ gets translated into an undirected edge between artifacts i and j with the weight $[i, j]$. The higher the weight, the stronger the evolutionary connection [Oliva and Gerosa 2015].

In Figure 3.1 the artifacts A and B changed three times together, which results in an edge between them with the weight 3.

3.4.2 Directed Raw Counting (DRC)

DRC on the other hand, creates a directed relation between artifacts. The idea is to set the number of co-changes between two artifacts in relation to each artifacts' changes. Each cell below the main diagonal with $[i, j] \neq 0$ gets translated into two directed edges between artifacts i and j. The first edge starts at artifact i and points to artifact j. The edge's weight is calculated by $\frac{[i,j]}{[i,i]}$. The second edge starts at artifact j and points in the opposite direction. The weight is calculated analog to the first edge by $\frac{[i,j]}{[j,j]}$.

When looking at Figure 3.1 the directed edge from artifact A to artifact B would have a weight of $\frac{3}{4} = 0.75$, where as an edge in the opposite direction would have a weight of $\frac{3}{9} = 0.333$. These two edges show that artifact A is more coupled with artifact B than vise

versa.

3.5 Datastructures

During the development of the tool, one unique data structure was implemented. It is called *PushList* which represents a list of comparable elements with a fixed length. Every added element is sorted to the position after the first element that is bigger than the added one. The last element is removed as soon as the list has one element more than the maximum size. This data structure collects the edges with the highest weight without listing them all and sorting them. Creating a list of all edges first would result in much larger memory consumption.

3.6 Pipeline Structure

The heart of the GitCouplingTool is a pipeline structure consisting of two consecutively executed pipelines. They are introduced in this additional section. Two pipelines are mainly used to create two separate steps with their individual memory consumption. If both of them were merged, the garbage collector of the Java Virtual Machine (JVM) is not able to free the memory as efficiently as with two pipelines. Each pipline has one section in which a stage runs multiple time parallel on separate threads. The number of threads is based on the number of CPU threads.

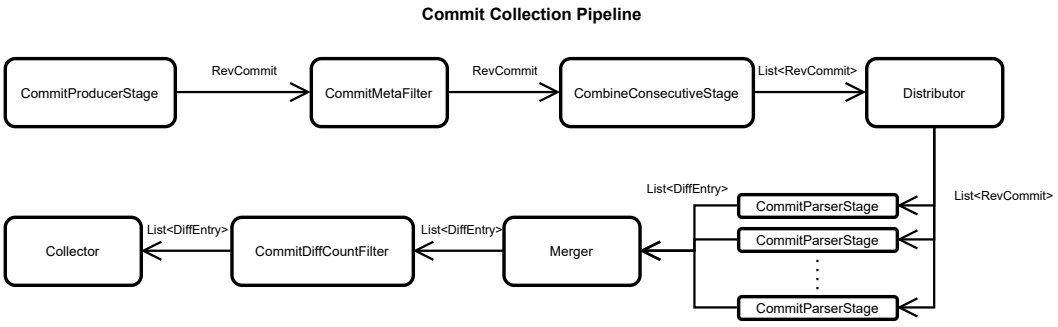


Figure 3.2. Commit Collection Pipeline

3.6.1 Commit Collection Pipeline

The *Commit Collection Pipeline* (Figure 3.2) is the first pipeline and is used to collect a list of commits. A commit is defined as a list of `DiffEntry`, which represents a modificaton of a file. A `DiffEntry` can be of the type *ADD*, *MODIFY*, *DELETE*, *RENAME* or *COPY*. To collect the described list, the following steps have to be accomplished.

- *CommitProducerStage*: responsible for walking the commit tree and pushing every collected `RevCommit` to the next stage. `RevCommit` is a class defined by JGit [jgit] and contains all information of a commit.
- *CommitMetaFilter*: filters the commits for criteria provided by the use. For example this can be the date or the author.
- *CombineConsecutiveStage*: if the user provides a time interval in witch consecutive commits should be combined, this stage combines the commits in a list of commits. Consecutive commits share the same author and are authored within the time interval are in such a list. However, most commits do not match these criteria resulting in mostly one-element lists.
- *Distributor*: distributes the incoming lists of commits on the following stages.
- *CommitParserStage*: responsible for parsing the binary commit body and performing the Git diff algorithm to retrieve a list of `DiffEntry`. This stage runs multiple times on different threads in parallel. The diff algorithm is the most CPU-intensive task in this pipeline, and it is crucial to use multithreading.
- *Merger*: collects the results from the previous stages and forwards them to the following stage.
- *CommitDiffCountFilter*: removes commits with a higher number of changed files than the user-defined threshold for large commits.

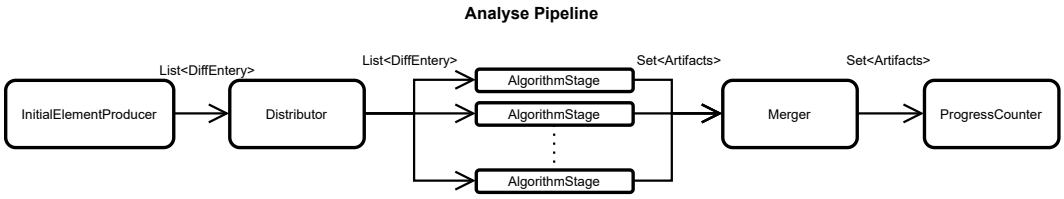


Figure 3.3. Analyse Pipeline

▻ *Collector*: collects all parsed commits in a list. It also displays the progress of this pipeline.

3.6.2 Analyse Pipeline

The *Analyse Pipeline* (Figure 3.3) is the second pipeline and is used to perform the coupling algorithm on the list of collected commits. The files changed in a commit are translated into artifacts and counted afterward. For this process, the pipeline uses the following stages.

▻ *InitialElementProducer*: takes in the list of commits from the first pipeline and pushes them one after another into the pipeline.

▻ *Distributor*: distributes the incoming lists of commits on the following stages.

▻ *AlgorithmStage*: converts the changed files to changed artifacts and counts them in a co-change matrix. This stage is parallelized to maximize performance.

▻ *Merger*: collects the results from the previous stages and forwards them to the following stage.

▻ *ProgressCounter*: counts the finished commits and displays the progress bar.

Chapter 4

Performance Analysis

As the performance is the focus of the GitCouplingTool, this section compares the performance and memory consumption with the tool of Issa [2021] (repository-mining).

4.1 Testing environment

The analysis is done on the following system.

- *CPU:* AMD Ryzen 9 5900X 12 Core / 24 Threads @ 3.7 GHz (watercooled)
- *RAM:* 32 GB DDR4 @ 3600 MHz
- *SSD:* M.2 NVMe PCIe x4 Gen4 @ 7.000 MB/s read and 5.500 MB/s write
- *OS:* Windows 10 with Windows Subsystem for Linux (WSL) 2 running Ubuntu

WSL is chosen over Windows 10 itself because Windows' default filesystem is case insensitive and causes problems when checking out repositories that rely on case sensitive filesystems. The Linux kernel is an excellent example of a case like that.

The performance of the GitCouplingTool depends mainly on the number of commits the tool should analyze and the number of files that Git tracks in the commit first to analyze. The files tracked in a commit are defined as files available in the working directory when checking out the commit. If all commits have almost the same amount of changed files, the overall execution time is in $O(c \cdot f^2)$ with $c = \textit{number of commits}$ and $f = \textit{number of tracked files}$. The exponential increase results from the $N \times N$ co-change matrix introduced in Section 3.4.

For the comparison, two repositories analyzed in Issa [2021] are used. In addition, the repositories of Git itself and the Linux kernel are analyzed. Both of them have too many commits to analyze them with repository-mining.

To get a comparable result, repository-mining is called by

```
$ python3 mining analysis -al urc -o result.json -nm -r <path>
```

and the GitCouplingTool is called by

```
$ java -Xmx16g -jar <jar> <path> -a URC -c 1 -o result.json -f JSON
```

Important is the JVM option `-Xmx16g` which increases the maximum heap size to 16 GB. Without this option, an Exception is thrown as soon as the heap increases more than the default 8 GB. The statistics about the execution are tracked with the GNU `time` command, by calling

```
$ /bin/time -v <cmd>
```

4.2 Results

The Results of the analysis are listed in the following Table 4.1. For each example project, the number of commits and tracked files are provided. Additionally, the latest commit at the time of testing is given along with the URL of the project. The `--from-commit <hash>` flag of the GitCouplingTool can be set to the listed hash for recreating the same test environment.

It is worth mentioning that the memory consumption of the JVM is a little bit higher than the memory consumption of the application itself. This extra amount of used memory space is allocated by the JVM in advance for more required space at a later point in time.

Table 4.1. Results of the comparison between GitCouplingTool and repository-mining

	GitCouplingTool	repository-mining
ExplorViz 1,264 commits ∣ 315 tracked files Commit: `7cd4b189797a5f0117bab0d33834aea8237cd321` `https://github.com/ExplorViz/explorviz-backend`		
Elapsed time	0.68 seconds	20.54 seconds
Percent of CPU the process got	865%	119%
Max peak RAM usage	245 MB	344 MB
Kieker 7,692 commits ∣ 3,377 tracked files Commit: `9e77f732e3f2a352a977c2467e8b2fa0d320f9ac` `https://github.com/kieker-monitoring/kieker`		
Elapsed time	8.99 seconds	2 hours 26 minutes
Percent of CPU the process got	518%	99%
Max peak RAM usage	1.93 GB	13.12 GB
Git 63,931 commits ∣ 3,975 tracked files Commit: `e0a2f5cbc585657e757385ad918f167f519cfb96` `https://github.com/git/git`		
Elapsed time	4.80 seconds	
Percent of CPU the process got	1153%	-
Max peak RAM usage	5.61 GB	
Linux 1,032,541 commits ∣ 72,886 tracked files Commit: `b91db6a0b52e019b6bdabea3f1dbe36d85c7e52c` `https://github.com/torvalds/linux`		
Elapsed time	6 minutes 27 seconds	
Percent of CPU the process got	1879%	-
Max peak RAM usage	15,20 GB	

4.3 Visualization

A visualization of the resulting graph is not part of the GitCouplingTool, but a suitable option is a tool called Gephi. Gephi can import the Graph Modelling Language (GML) file and provides numerous options for the visualization of the graph. Force Atlas, a force-directed algorithm, is used for the layout [Bastian et al. 2009].

The results of the previously analyzed Git repository of *Kieker* are visualized in Figure 4.1. The visualization is for illustration purposes only and created manually. Interesting about the result is the one large group of files in the lower-left corner. These files changed numerous times in the same commits. That probably means that these files have something in common. It turns out that all files in this group are located in `kieker-common/src-gen/kieker/common/record/`.

The results of the analysis of *Git* are visualized in Figure 4.2. Based on the visualization, there are two main groups of related files. On the left are files located in `Documentation/` that are often changed in the same commit. In contrast to that is the group on the right that consists of `.c` and `.h` files, with `cache.h` and `diff.c` beeing the files most commonly changed with numerus other files.

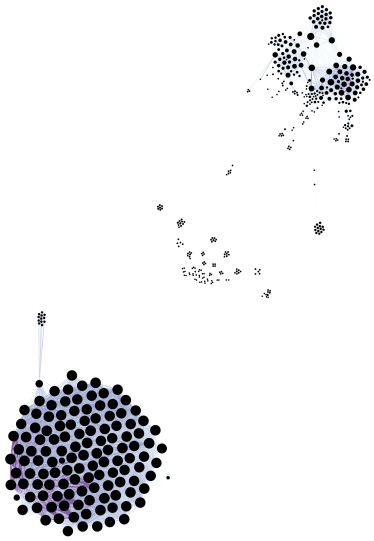


Figure 4.1. Visualization of the example Kieker

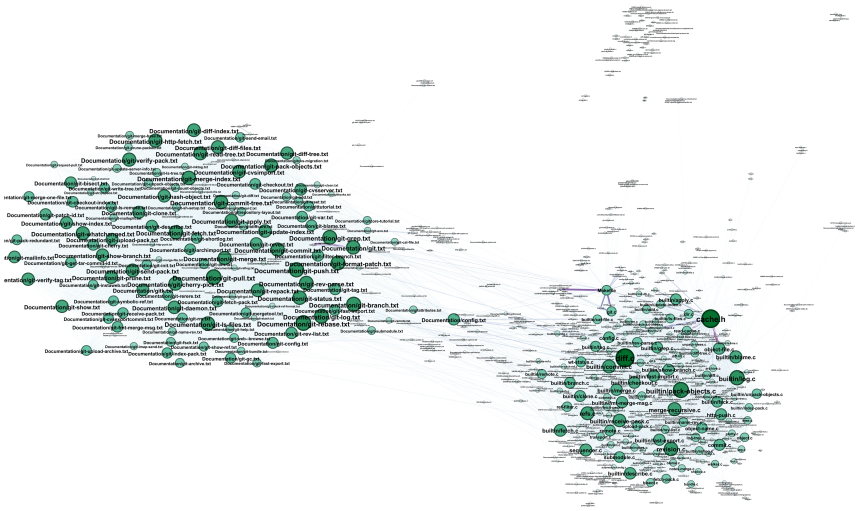


Figure 4.2. Visualization of the example Git

Chapter 5

Documentation

The following sections are meant as a guide on how to use the tool and its options.

5.1 Quick Start

The first step to use the tool is to build the jar file. That can be done by calling

```
$ ./gradlew shadowJar
```

which builds the project and creates a bundled jar in `build/libs/GitCouplingTool*.jar`. The GitCouplingTool has a default for every option. That makes it very easy to use. The most simple command to get a coupling graph is

```
$ java -jar GitCouplingTool*.jar <path>
```

with the path beeing the root directory of a Git repository or the `.git` directory itself. This command will use URC and export the graph in `result.gml` in the Present Working Directory (PWD).

5.2 User Guide

5.2.1 Options

The GitCouplingTool provides some options for filtering commits and files. These options are the following.

```
$ java -jar GitCouplingTool.jar <path> [options]
```

Table 5.1. A list of available options.

Name	Type	Default	Description
Arguments			
path	string	-	The path to the repository to analyze. This can be either the repository's root directory or the `.git` directory.
Miscellaneous Options			
`-a, --algorithm`	string	URC	The couplin algorithm to use. Can be URC or DRC.
`-r, --follow-renames`	flag	-	If a file is renamed in one commit, it is technically deleted and recreated with a new name. With this flag set, renames are tracked, and the following couplings are collected. Therefore the counting process has to be synchronus, which results in longer execution times but more precise results.
`-cc, --combine-consecutive`	int	-	Combine commits that are authored with in the given span of seconds.
`--large-threshold`	int	50	Number of changed files per commit to considder the commit as large. (see `--no-large`)
Export Options			
`-o, --output`	string	`./result.gml`	Path of a file to export the graph to.
`-f, --format`	string	GML	Export the data in the given file format. Can be either GML or JSON.
`-c, --min-couplings`	int	2	The minimal number of common commits to considder two files coupled.
`-e, --edges`	int	0 (all)	Export the given number of edges with the highest weights as well as the connected nodes. Or all if set to 0.
Commit and File Filter Options			
`--author`	string	-	Only include commits if the author's name or email are equal to the given string.
`--from-date`	string	-	Only include commits authored after the given date. E.g. 2021-09-01.

Table 5.1. A list of available options.

Name	Type	Default	Description
`--to-date`	string	-	Only include commits authored before the given date. E.g. 2021-09-31.
`--file-type`	string	-	Include only files ending with the given extension e.g. ".java". Can be defined multiple times to include more types.
`--merges`	flag	no	Include merge commits, which are commits with more than one parrent commit.
`--no-large`	flag	no	Exclude large commits, which are commits with more than a certain number of changed files. (see `--large-threshold`)
Commit Traversl Options			
`-b, --branch`	string	HEAD	Start the commit traversal at the given branch.
`--from-commit`	hash	-	Start the commit traversal at the given commit's hash.
`--to-commit`	hash	-	End the traversal at a given commit's hash.
`--from-tag`	string	-	Start the commit traversal at the given tag.
`--to-tag`	string	-	End the traversal at a given tag.

5.2.2 Examples

```
$ java -jar GitCouplingTool.jar . -r -e 100000 -cc 60
```

Analyse the PWD, follow renames, combine commits that are commit within 1 minute to the previous commit and export the 100,000 edges (couplings) with the highest weights along with their nodes in GML format in `result.gml`.

```
$ java -jar GitCouplingTool.jar /home/user/git/repo -e 100 -format JSON -o file.json
```

Analyse `/home/user/git/repo` and export the 100 edges (couplings) with the highest weights along with their nodes in JSON format in `result.json`.

```
$ java -jar GitCouplingTool.jar ../repo -c 5 -a DRC -from-commit 8f8e32df -to-commit 231de9fa
-no-large
```

Analyse `../repo` with DRC and export all edges (couplings) with a common change count of at least 5 along with their nodes in GML format in `result.gml`. Start with the commit `8f8e32df`, stop when reaching `231de9fa` and skip commits that have more than 50 changed files.

5.3 Exported File Format

The GitCouplingTool is capable of exporting the final graph in two formats. The first one and also the default is Graph Modelling Language (GML). GML is a simple format (Listing 5.1) dedicated to representing graphs. It can be used to export the graph to other tools like Gephi [Bastian et al. 2009]. The second format is JavaScript Object Notation (JSON) which is a more universal format than GML. (Listing 5.2) The JSON format is additionally used for the comparison with Voigt [2021], whose tool exports into a very similar format.

Listing 5.1. Example GML file.

```
graph [
  node [
    label "fileA.txt"
    id "fileA.txt"
  ]
  node [
    label "fileB.txt"
    id "fileB.txt"
  ]
  edge [
    source "fileA.txt"
    target "fileB.txt"
    value 4.0
    directed 0
  ]
]
```

Listing 5.2. Example JSON file.

```
{
  "nodes":[
    {
      "id":"fileA.txt"
    },
    {
      "id":"fileB.txt"
    }
  ],
  "edges":[
    {
      "id":"fileA.txt::fileB.txt",
      "start":"fileA.txt",
      "end":"fileB.txt",
      "weight":4.0,
      "directed":false
    }
  ]
}
```

Chapter 6

Comparison with Static Coupling of Example Projects

In the following chapter, the results delivered by the GitCouplingTool will be compared with the results of Static Coupling. The StaticCouplingTool by Voigt [2021] delivers the data for the Static Coupling.

Three open-source projects available on GitHub are selected for comparison. Two of them are Git [git] and BitKeeper [bitkeeper]. Both of them are Version Control Systems (VCS) with the difference that Git is a community-driven project while BitKeeper was used to be a closed-source and company-driven project [bitkeeper].

6.1 Git

The first project to be analyzed is git. Git is a distributed version control tool [git]. It is mainly written in C.

6.1.1 Static Coupling

To be able to analyze the tool, a compilation database must first be created. This can be done as follows:

```
$ cd <path to git project> && bear make .
```

The Git project is to be analyzed with the following configuration:

```
{
 "language": "cpp",
 "merge": true,
 "output": <path to output folder>,
 "project-path": "<path to git project>",
 "whitelist": "<path to git project>"
}
```

Table 6.1. Nodes of Static and Change Coupling of Git with highest coupling degree

Static Coupling		Change Coupling	
Node Name	**Coupling Degree**	**Node Name**	**Coupling Degree**
strbuf	5654	cache	6141
cache	4164	diff	4344
gettext	3052	pack-objects	3794
git-compat-util	2691	commit	3389
config	1367	revision	3355
commit	1305	log	3326
sequencer	1168	refs	3300
repository	1167	object-file	3287
hash	1082	receive-pack	3234
refs	1008	checkout	3066

6.1.2 Change Coupling

The generation of the coupling graph can be done with:

```
$ java -jar GitCouplingTool.jar <path> -r -c 1 -cc 60 --file-type .c --file-type .cpp
--file-type .h -o result.json -f JSON
```

This command includes every coupling of files with the ending ".c", ".cpp", and ".h".

6.1.3 Comparsion

In order to compare the results, a ranking of the results according to the coupling degree of the individual nodes is to be considered. The Table 6.1 lists the ten nodes with the highest coupling degree. The 'cache', 'commit', and 'refs' nodes are among the ten with the highest coupling degree for Static and Change Coupling. So since three of ten nodes are present in both lists, there is an overlap of 30%.
In the graph of Figure 6.1, the overlap was performed for all list lengths. The list lengths are on the x-axis, the corresponding overlap is on the y-axis. The number of overlaps increases with the number of considered nodes. This increase was to be expected since the probability of a match increases with a higher number of considered nodes.
Next, it is investigated how the coupling number depends on the file size. The list of ten nodes with the highest coupling degree is appended with the linecount and the rank in the list of files with highest line count of the files for each corresponding node in Table 6.2. It is noticeable that the files for the nodes with the highest coupling degree of the Change Coupling graph have a higher total number of lines than those of the Static Coupling graph.

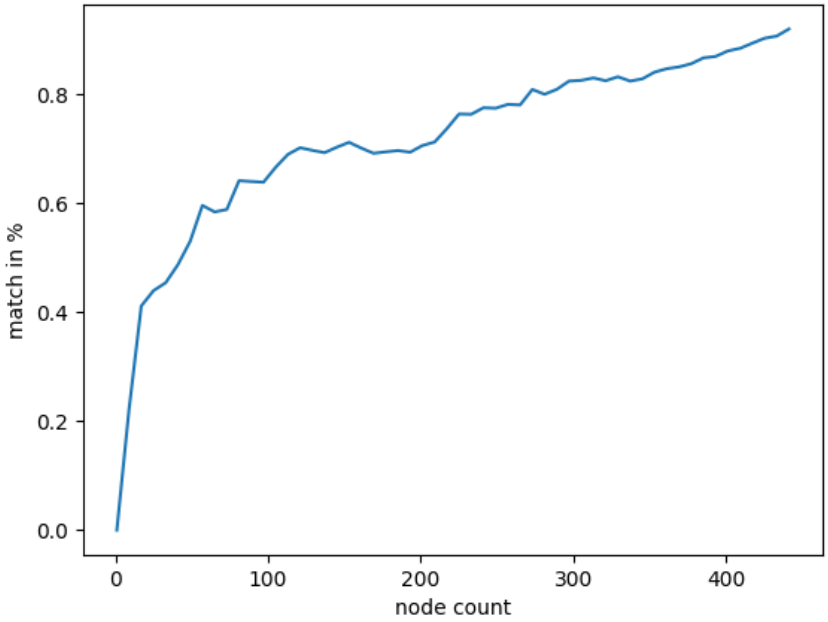


Figure 6.1. Overlap of nodes with highest coupling degree of Git for different count of considered nodes

Table 6.2. Coupling degree of Git in relation to line count of nodes files

Static Coupling			Change Coupling		
Node Name	**Line Count**	**Rank**	**Node Name**	**Line Count**	**Rank**
strbuf	1924	53	cache	1917	54
cache	1917	54	diff	8338	1
gettext	218	271	pack-objects	4700	9
git-compat-util	1380	77	commit	3965	14
config	5190	5	revision	4706	7
commit	3965	14	log	2363	41
sequencer	6166	2	refs	3346	21
repository	519	173	object-file	2651	30
hash	334	223	receive-pack	2600	31
refs	3346	21	checkout	1989	48

6.2 BitKeeper

The next project to be analyzed is BitKeeper [bitkeeper]. Similar to git, Bitkeeper is a tool for distributed version control. Until 2016 Bitkeeper was proprietary software, then it became an open-source project with the Apache 2.0 license.

6.2.1 Static Coupling

To be able to analyze the BitKeeper project, a compilation database must first be created. This can be done as follows:

```
$ cd <path to git project> && bear make
```

The Static Coupling analysis of BitKeeper can be performed with the following configuration:

```
  {
    "language": "cpp",
    "merge": true,
    "output": <path to output folder>,
    "project-path": "<path to bitkeeper project>",
    "whitelist": "<path to bitkeeper project>"
}
```

6.2.2 Change Coupling

The generation of the coupling graph can be done with:

```
$ java -jar GitCouplingTool.jar <path> -r -c 1 -cc 60 --file-type .c --file-type .cpp
--file-type .h -o result.json -f JSON
```

This command includes every coupling of files with the ending ".c", ".cpp", and ".h".

6.2.3 Comparsion

The coupling degree of the nodes of the result graphs of the Static and the Change Coupling are compared in the following. Table 6.3 compares the 10 nodes with the highest coupling degree. Like in the Git project, there is an overlap of some nodes of these lists. The nodes 'sccs' and 'slib' are among the top 10 nodes with the highest coupling degree for both lists. So there is an overlap of 20%.

Table 6.3. Nodes of Static and Change Coupling of BitKeeper with highest coupling degree

Static Coupling		Change Coupling	
Node Name	**Coupling Degree**	**Node Name**	**Coupling Degree**
sccs	4493	sccs	7187
string	4126	slib	6682
stdlib	2908	takepatch	4273
system	1616	check	4007
Last	1568	resolve	3856
slib	1490	cset	3609
lines	1439	clone	3415
stdio	1175	bk	3248
proj	1134	utils	3243
Lcompile	1110	commit	2696

Remarkably, several reimplemented parts of the C standard library can be found in the result list of the Change Coupling. These are, for example, 'string', 'stdlib', and 'stdio'. These nodes have hardly any outgoing edges and are represented in the top 10 coupling degree nodes because they have many incoming edges. These classes have a shallow coupling degree in the resulting graph of the Change Coupling because only infrequent changes were made to the corresponding files after adding them. In the graph of Figure 6.2, the overlap was performed for all list lengths. The list lengths are on the x-axis, the corresponding overlap is on the y-axis. Here it is noticeable that the graph only reaches an overlap of just under 60 percent at the end. This low value is caused by the StaticCouplingTool finding fewer nodes that contain coupling than the GitCouplingTool. The compilation database used for this analysis and generated with Bear [bear] does not contain compile commands for all files. The whitelist option is set to the root project directory to compensate for this, but the StaticCouplingTool sometimes has problems finding coupling in all of these files because their compile commands are missing in the compilation database.

Next, it is investigated how the coupling number depends on the file size. The list of ten nodes with the highest coupling degree is appended with the linecount and the rank in the list of files with highest line count of the files for each corresponding node in Table 6.4 as it was already done in the previous section for git. Also, at the results of BitKeeper, it is noticeable that the files for the nodes with the highest coupling degree of the Change Coupling graph have a higher total number of lines than those of the Static Coupling graph.

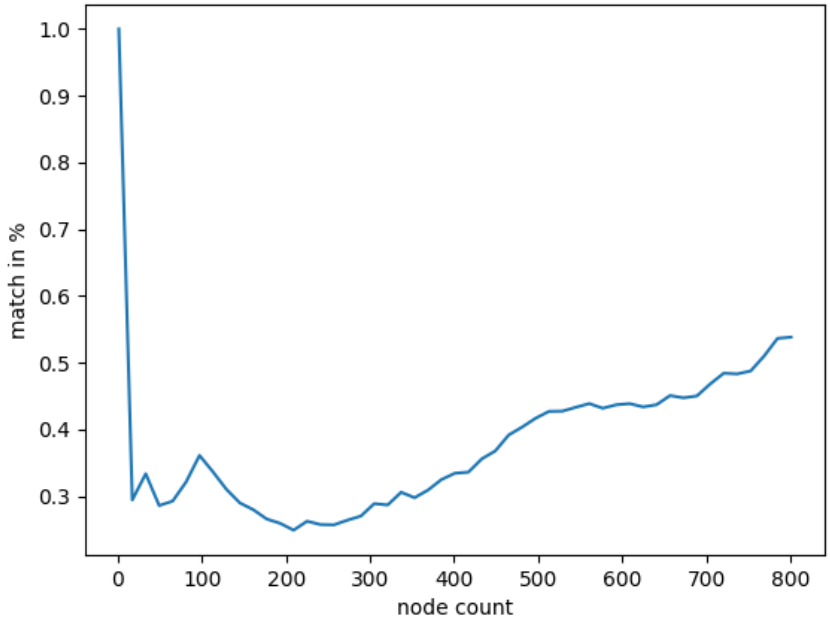


Figure 6.2. Overlap of nodes with highest coupling degree of BitKeeper for different count of considered nodes

Table 6.4. Coupling degree of BitKeeper in relation to line count of nodes files

Static Coupling			Change Coupling		
Node Name	**Line Count**	**Rank**	**Node Name**	**Line Count**	**Rank**
sccs	1827	152	sccs	1827	152
string	57	1252	slib	18608	1
stdlib	76	1119	takepatch	2322	104
system	1394	203	check	2856	83
Last	876	321	resolve	3613	52
slib	18608	1	cset	1018	291
lines	2124	117	clone	2210	109
stdio	597	418	bk	1955	138
proj	1875	148	utils	2353	102
Lcompile	8792	8	commit	1296	220

6.3 linux

6.3.1 Static Coupling

Creating the compilation database is complex for the Linux kernel than for other projects. At first, a configuration for the Linux kernel has to be set. This can be done as follows:

```
$ make CC=clang-12 defconfig
```

After that, the compilation database can be created with Bear:

```
$ bear make CC=Clang-12
```

The Git project is to be analyzed with the following configuration:

```
{
 "language": "cpp",
 "merge": true,
 "output": <path to output folder>,
 "project-path": "<path to Linux kernel build folder>",
}
```

6.3.2 Change Coupling

The generation of the coupling graph can be done with:

```
$ java -jar GitCouplingTool.jar <path> -r -c 10 -cc 60 --file-type .c --file-type .cpp
--file-type .h -o result.json -f JSON
```

This command includes every coupling with a co-change count of at least ten and only files with the ending ".c", ".cpp", and ".h".

6.3.3 Comparsion

Comparing the results of the Linux kernel causes some problems. The compilation database used by the StaticCouplingTool only contains the files which are relevant for the configuration selected in Section 6.3.1. The Linux kernel consists of 51903 C files (sum of header

Table 6.5. Nodes of Static and Change Coupling of Linux kernel with highest coupling degree

Static Coupling		Change Coupling	
Node Name	**Coupling Degree**	**Node Name**	**Coupling Degree}**
spinlock	7663	dev	11535
err	6394	fs	11423
mutex	5917	i915_drv	10317
netlink	5527	intel_display	9539
nl80211	4055	inode	8442
string	4002	route	7721
skbuff	3962	netdevice	7081
pci	3236	ipmr	7029
core	2858	shmem	6988
intel_display	2236	udp	6964

and source files), but the configuration of Section 6.3.1 only contains 2671 (source files only). The results are therefore difficult to compare. Nevertheless, the coupling degree of the nodes of the result graphs of the Static and the Change Coupling are compared in the following. Table 6.5 compares the ten nodes with the highest coupling degree. Despite the high difference of analyzed files, there is again an overlap. The 'intel_display' node is part of both graphs. This is an overlap of 10%. In the graph of Figure 6.3, the overlap was performed for all list lengths. The list lengths are on the x-axis, the corresponding overlap is on the y-axis. Remarkable here is that the max overlapping is less than 30% and the graph decreases for a node count of approximately 250. This is a difference to the corresponding graphs of Git Figure 6.1) and BitKeeper (Figure 6.2). This difference is due to the problems mentioned at the beginning of this section.

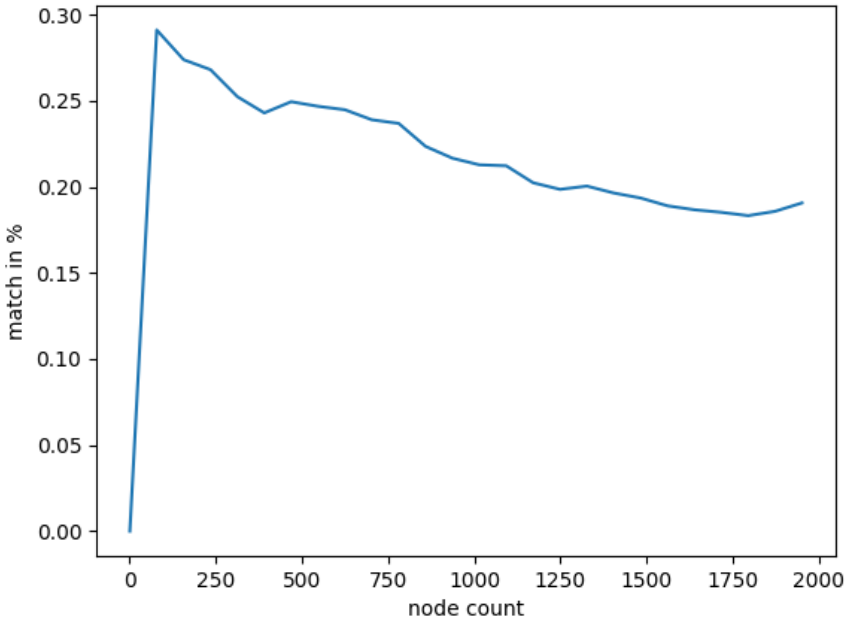


Figure 6.3. Overlap of nodes with highest coupling degree of Linux kernel for different count of considered nodes

Chapter 7

Conclusions and Future Work

7.1 Conclusion

It was shown how to develop a tool for Change Coupling Analysis in the context of Git repositories. The resulting GitCouplingTool has been proven to have a much lower execution time and memory consumption than the reference tool repository-mining by Issa [2021]. The performance analysis in Section 4.2 showed 970 times faster execution time compared to repository-mining for the example project Kieker. Furthermore, repositories that could not be analyzed with repository-mining due to the limitation in memory consumption can now be analyzed. The Linux kernel with a little more than 1 million commits can be analyzed in under 10 minutes with a memory consumption below 16 GB. These results accomplish the first of the two goals for this thesis.

In addition to the GitCouplingTool, the second goal is achieved by comparing it with the StaticCouplingTool by Voigt [2021]. The comparison has shown that both used metrics for Change and Static Coupling share similarities but also have differences. The file size often plays a more significant role in Change Coupling than in Static Coupling in the analyzed sample projects. In Static Coupling, on the other hand, smaller files, whose nodes have a high number of incoming edges in the coupling graph, often have a high coupling degree overall. The analysis with Change Coupling is easier because it is not necessary to create a compilation database first, and also files can be analyzed reliably, which are not present in this compilation database. In conclusion, both analysis methods are helpful, and neither can replace the other.

7.2 Future Work

The performance of the GitCouplingTool is still limited to one thread when using the option to follow renames. Some more investigation has to be done to count the changing files of the commits in parallel threads. Another option for future work would be to implement a more sophisticated algorithm for measuring Change Coupling.

Bibliography

[Bastian et al. 2009] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An Open Source Software for Exploring and Manipulating Networks. In: International AAAI Conference on Weblogs and Social Media. 2009. URL: http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154. (Cited on pages 16, 22)

[bear]. *Bear*. László Nagy. URL: https://github.com/rizsotto/Bear (visited on 09/28/2021). (Cited on page 29)

[bitkeeper]. *BitKeeper*. BitMover Inc. URL: http://www.bitkeeper.org (visited on 09/25/2021). (Cited on pages 25, 28)

[Briand et al. 1997] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for c++. In: *Proceedings of the 19th international conference on Software engineering*. 1997, pages 412–421. (Cited on page 4)

[git]. *Git –everything-is-local*. URL: https://git-scm.com (visited on 09/23/2021). (Cited on page 25)

[git-internals]. *Git Internals - Git Objects*. URL: https://git-scm.com/book/en/v2/Git-Internals-Git-Objects (visited on 09/12/2021). (Cited on page 5)

[Issa 2021] K. Issa. Developing a git Repository Mining Tool for Change Coupling Analysis. Bachelor thesis. Kiel University, Mar. 2021. URL: http://oceanrep.geomar.de/52240/. (Cited on pages 1, 2, 8, 13, 14, 35)

[jcolor]. *JColor - easy syntax to format your strings with colored fonts and backgrounds*. Diogo Nunes. URL: https://github.com/dialex/JColor (visited on 09/12/2021). (Cited on page 8)

[jgit]. *JGit - Java library implementing the Git version control system*. Eclipse Foundation. URL: https://www.eclipse.org/jgit/ (visited on 09/12/2021). (Cited on pages 7, 11)

[Nagappan et al. 2006] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In: *Proceedings of the 28th international conference on Software engineering*. ACM, May 2006. (Cited on page 3)

[Oliva and Gerosa 2015] G. A. Oliva and M. A. Gerosa. "Change Coupling Between Software Artifacts". In: *The Art and Science of Analyzing Software Data*. Elsevier, 2015, pages 285–323. (Cited on pages 3, 8, 9)

[picocli]. *picocli - a mighty tiny command line interface*. Remko Popma. URL: https://picocli.info (visited on 09/12/2021). (Cited on page 8)

[progressbar]. *Progressbar - a console-based progress bar for Java*. Tongfei Chen. URL: http://tongfei.me/progressbar/ (visited on 09/12/2021). (Cited on page 8)

[Schnoor and Hasselbring 2020] H. Schnoor and W. Hasselbring. Comparing Static and Dynamic Weighted Software Coupling Metrics. *Computers* 9.2 (Mar. 2020), page 24. (Cited on pages 3, 4)

[shadowjar]. *shadowJar - Gradle plugin to create fat/uber JARs*. John Engelman. URL: https://imperceptiblethoughts.com/shadow/ (visited on 09/12/2021). (Cited on page 7)

[teetime]. *TeeTime - The Next-Generation Pipe-and-Filter Framework*. Kiel University. URL: https://teetime-framework.github.io (visited on 09/12/2021). (Cited on page 7)

[Voigt 2021] L. Voigt. Development of the StaticCouplingTool and its Evalualtion. Bachelor thesis. Kiel University, Sept. 2021. URL: https://github.com/lukas0820/StaticCouplingTool. (Cited on pages 1–4, 22, 25, 35)