Development of StaticCouplingTool for Static Coupling Analysis and its Evaluation

Bachelor's Thesis

Lukas Voigt

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

Coupling has an impact on the quality of software and can be measured in different ways. One of these is the static coupling analysis. In this thesis the StaticCouplingTool is developed, which is a tool for the static analysis of coupling for C and C++ projects. It is also extendable for other languages, and it measures the coupling between source files of a project. In addition, the analysis of three example projects is evaluated and compared with the results of another coupling tool analysing Change Coupling.

Contents

List of Figures

List of Acronyms

AST	Abstract Syntax Tree
GML	Graph Modelling Language
IR	Intermediate Representation
JSON	JavaScript Object Notation

Chapter 1

Introduction

1.1 Motivation

A low coupling in software has many advantages. Software is better maintainable if it has a low coupling, since the change at a component does not lead to a necessary change in other components. A low coupling facilitates the testing with unit tests, since when testing a component, less code of other components is tested implicitly, so bugs can be better located. In addition, components with low coupling can be reused in other software more easily.
There are different ways to measure coupling. This thesis focuses on static coupling and the comparison to Change Coupling. In static coupling the coupling is measured by static analysis, in Change Coupling [Hofmann 2021] the coupling is measured by metadata of the version control system git.
Although there are already tools for static analysis, there are hardly any tools that support coupling analysis in the languages C and C++. The motivation for this thesis is to provide a way to make coupling in C and C++ measurable with with the StaticCouplingTool developed in this thesis and compare its result to the result of Change Coupling in C and C++.

1.2 Joint Thesis Project

This thesis is part of a joint thesis project together with [Hofmann 2021]. In this thesis the StaticCouplingTool is developed, in [Hofmann 2021] the GitCouplingTool is developed. Parts of Chapter 2, Chapter 5, and Chapter 6 are part of both thesis.

1.3 Goals

1.3.1 Developing a static coupling analysis tool for C and C++ projects

The main goal in this thesis is to develop the StaticCouplingTool, a static analysis tool that can analyze coupling in the source code of C and C++ projects.

1.3.2 Evaluation and comparison with Change Coupling

In [Hofmann 2021], the GitCouplingTool was developed for analyzing change coupling. This tool is language-independent, so it can be used for the analysis of C and C++ projects. A goal of our joint thesis project is to analyze selected projects with the StaticCouplingTool and the GitCouplingTool and evaluate and compare their results.

Chapter 2

Foundation and Related Work

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 2.2 and Section 2.3. Each of these sections describes a particular type of measuring the coupling between source code files.

2.2 Static Coupling

Static coupling is the coupling that can be measured analysing the source code or the compiled code of a program [Schnoor and Hasselbring 2020]. That means that analysing 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 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 analysed. 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 signatures of a header in a separate source file.

In static coupling the edges of this graph are references in source code from the current analysed 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 a directed edge from node `A` to `B` in the corresponding file coupling graph. Other possible references between files are discussed in Section 3.3.3.
The file coupling metric is used to make the result of the StaticCouplingTool comparable to the results of Change Coupling, which is discussed in the next Section 2.3.

2.3 Change Coupling

In this section (adapted from Hofmann [2021]) the Change Coupling is introduced. 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 has to build an Abstract Syntax Tree to be able to measure the static coupling. Depending only on the changelog makes Change Coupling independent from specific languages. The GitCouplingTool developed in [Hofmann 2021] 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. Renames 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.

2.4 Abstract Syntax Tree (AST)

Compilers parse source code into an Intermediate Representation (IR) in order to be able to use the source code as a data structure internally. One of the most common IR is the AST [Torres et al. 2018].
An AST is the representation of the source code as a tree. Clang [clang] as a compiler frontend supports the parsing of C and C++ source code into an AST. In the case of clang, there are three distinct node types from which all other node types are derived. These are types, declarations, and statements [Torres et al. 2018; clang]. With clang an AST can be created for each translation unit. The AST is important for the implementation of StaticCouplingTool. A depth-first search is performed on the AST to find AST nodes that are

relevant for coupling. This is discussed in more detail in Section 3.3.3.

Chapter 3

Tool Development

The StaticCouplingTool is a tool written in C++ for static analyse of coupling in C and C++ projects. It can be found on Github `https://github.com/lukas0820/StaticCouplingTool`.

3.1 Requirements

In order to run, build and develop the tool, some programs are required. These are named in the following and are also listed in the `README.md` file.
The whole project is written in C++ and can be built using CMake [cmake]. For this, of course, a C++ compiler and an installation of CMake is required. Furthermore, libraries are used, which also have to be installed. These are Qt5 [qt] and clang [clang] (version 12 is recommended). To use clang as a library, it is important that the headers are available for inclusion. For this, clang must either be built from source or libclang-dev must be installed in a version that matches the clang installation.
The tool works on different platforms. Tested were Ubuntu 20.04, macOS Big Sur (with an Intel-based Mac), and Windows 10. Libclang-dev is not available on Windows, so it is important to build clang and not install a precompiled version on Windows.

3.2 External Libraries

As mentioned in the previous section, two libraries are used in the tool. These are clang and Qt5.

3.2.1 clang

Clang is a compiler frontend for LLVM. It supports the programming languages C, C++ and Objective C. Clang is modular and individual modules can be integrated into own projects [clang]. This is not possible with other known compiler frontends such as gcc. Clang offers a library to create tools for static code analysis. For this, only the appropriate modules must be integrated into the own project. In the tool, clang is used to generate an AST from the files to be analyzed. In this AST, a depth-first search is then performed for certain nodes that are relevant for coupling. This will be discussed in more detail in Section 3.3.3.

3.2.2 Qt5

Qt is a framework that can be used to write applications and graphical user interfaces. It has a modular structure, and individual modules can be integrated individually [qt]. Most of these modules provide functions for graphical user interfaces, but these were not used here. For this tool, only the qt-core library was used, which contains valuable implementations and which is also used by other Qt modules.
In the tool, the qt-core library is used for parsing and writing JavaScript Object Notation (JSON) objects and for accessing the filesystem. These will be discussed in more detail in the following two subsections.

File Access

Qt can be used to access, read and write files. The advantage is that Qt is available for different platforms, so the file access is platform-independent for the supported platforms of Qt. Qt offers different classes for file access. With `QFileInfo` the metadata of a file can be accessed. With `QFile` a file can be written and read. This will be demonstrated in the following example:

```
1 QFile file(path);
2 if (file.open(QIODevice::ReadWrite))
3 {
4   QByteArray contentBytes = file.readAll();
5   std::string content(contentBytes.toStdString());
6   content += "foo";
7   file.write(QByteArray::fromStdString(content));
8   file.close();
9 }
```

First, a `QFile` object is created, which is passed the path of the file to be opened. The file is then opened in ReadWrite mode in line 2. This allows both read and write access to the file. After that, the file content is read, appended and written back to the file.

Parsing JSON Objects

For parsing and writing JSON objects mainly three Qt classes are used. The `QJsonDocument` class handles the actual writing and parsing of JSON objects. `QJsonObject` and `QJsonArray` represent JSON objects and JSON arrays respectively. The following is an example of how to create and export a JSON object using Qt:

```
1 QJsonObject rootObject;
2 rootObject.insert("string", "foo");
3 QJsonArray array;
```

```
4  array.push_back("item 1");
5  array.push_back("item 2");
6  rootObject.insert("array", array);
7
8  QJsonDocument doc;
9  doc.setObject(rootObject);
10
11 std::cout << doc.toJson().toStdString() << std::endl;
```

A JSON object is created as a `QJsonObject`, which is then passed to a `QJsonDocument`. The `QJsonDocument` converts this into a human readable format and outputs it. The corresponding JSON file looks like this:

```
1 {
2   "array": [
3     "item 1",
4     "item 2"
5   ],
6 "string": "foo"
7 }
```

3.3 Structure

The StaticCouplingTool has a modular structure as shown in the component diagram in Figure 3.1. It consists of four modules in total. Each of these modules has its own directory in the root directory of the tool. First, there is a `language` module, which contains the language-specific implementation of the static analysis. Second, there is a `configuration` module, which creates configurations from command line arguments, user input, and configuration files and exports the used configuration. A further module is the `coupling` module, which administers internally a Coupling graph as data structure and can also export this. The fourth module is then the `application` module, which combines the three modules mentioned before to an application. The `application` module also contains the `main` function. Furthermore, there is a `shared` directory, which contains utils classes and classes used by all modules. Finally, a `scripts` directory contains Python scripts to modify and visualize the tool's results.

3.3.1 CMake

The `CMakeLists.txt` which contains the CMake configuration is located in the root directory. Each individual directory also contains a `filelist.cmake` file, in which source files are added to lists, which in turn are linked to the executable. This is done through lists in

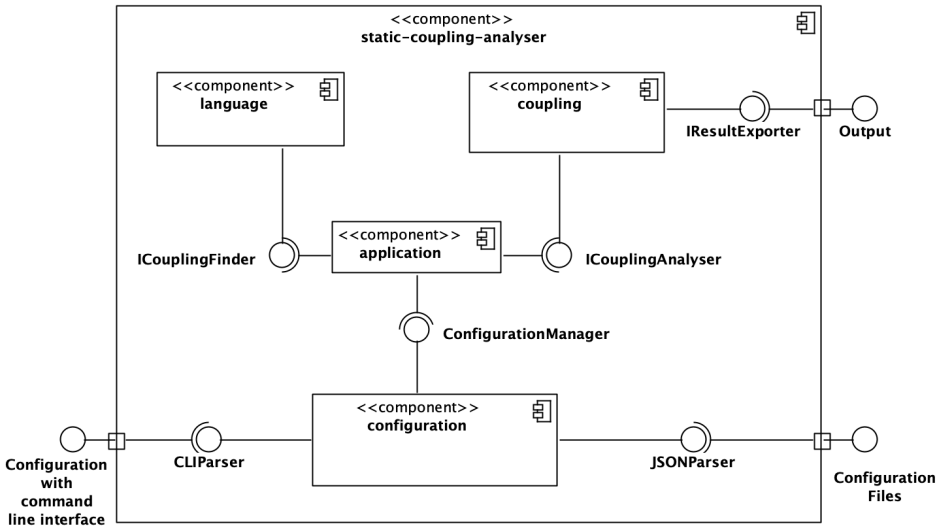

Figure 3.1. Structure of StaticCouplingTool

CMake, where each of the modules and the shared directory has its own list. These lists are named after the corresponding directory name, e.g. `APPLICATION_SOURCE_FILES` for the list from the `application` directory. Because of this the `language`, the `configuration`, and the `coupling` module can be built individually if a corresponding `CMakeLists.txt` is added, because these modules do not depend on other modules than the classes in the shared directory. The `application` module can not be built individually, because it depends on all other modules. Building a module individually can be practical if only individual modules are to be used in other projects or if unit tests are to be written for individual modules. Each of the directories has its own namespace, which is named after the corresponding directory.

3.3.2 shared

The `shared` directory contains all classes that are used from all modules. If modules are to be built individually, this directory must be included.
On the one hand, there are the two utils classes `shared::FileUtils` and `shared::ContainerUtils` for the access to files and container classes. On the other hand, there is the `shared::AbstractCoupling` class and other implementations of it. This represents the coupling between two different objects. The definition of a coupling object is not defined more precisely in the `shared::AbstractCoupling` class and is left to the classes implementing it. There is a caller object and a callee object in `shared::AbstractCoupling`. Because the tool is to be used to compare the static coupling with the Change Coupling,

only the file coupling metric is implemented as `shared::FileCoupling`.

3.3.3 Language Module

The `language` module is the module that contains the actual static analysis of the source code. It can provide implementations for different languages. Currently, there is one implementation that handles both C and C++. The module consists of the interface `language::ICouplingFinder` and one directory for each language-specific implementation. This language-specific implementation must provide an implementation of the interface `ICouplingFinder`.
The task of this module is to create `shared::AbsractCoupling` objects from the source code of the files to be analyzed, which then can be processed by other modules. The connection of this module with other modules works with the help of publish-subscribe mechanisms. The `language::ICouplingFinder` interface provides function signatures for registering callback functions. Functions, that process the AbstractCoupling object found by the `language` module in the source code, can be registered in the implementations of `language::ICouplingFinder`.
The implementation for C and C++ is located in the `cpp` directory. It provides `language::cpp::ClangCouplingFinder` as an implementation of the `language::ICouplingFinder` interface. In addition to the functions of the interface specific options for the static analysis can be set here. It is the only class in this module that can be accessed from outside the module.
In the C and C++ specific implementation clang was used as a library for static analysis as already mentioned in Section 3.2.1. Clang provides an implementation of the AST for its supported languages. The nodes of the branch consist of classes whose base types are `clang::Stmt`, `clang::Type` or `clang::Decl`, which represents statements, types and declarations.
A central role in the module is played by the `clang::RecursiveASTVisitor` class. This class provides a function `TraverseAST`, which performs a depth-first search starting from a node in the AST using the visitor pattern. For each possible AST node type, a function is provided, which can be overridden and which is called for the corresponding AST node during the depth-first search. The following types are relevant for file coupling:

- `clang::DeclRefExpr` is a reference to a declaration. For example, such a declaration can be a function, a variable or an enumeration. With this AST node all relevant coupling for C code can be found. For C++ there are other node types in addition that are relevant for coupling.
- `clang::CXXConstructExpr` represents a call of a constructor and is only relevant for C++.
- `clang::CXXRecordDecl` represents a class declaration. It contains references to possible super classes. Inheritance coupling can be found with this AST node and it is only relevant for C++ code.

▷ `clang::CXXMemberCallExpr` represents a class member call.

Using the example of `clang::CXXConstructExpr`, the following example shows how depth-first search works using the Visitor pattern in the tool. The corresponding Visit function in `clang::RecursiveASTVisitor` is named `VisitCXXConstructExpr`. The implementation of `clang::RecursiveASTVisitor` in the `language` module is `language::cpp::CouplingVisitor`. Its `TraverseAST` function is called for each translation unit by `CouplingASTConsumer`. In the following the implementation of the Visit function from this class will be used as an example to explain how this works:

```
1  bool VisitCXXConstructExpr(clang::CXXConstructExpr* call) override
2  {
3    if (call && call->getConstructor())
4    {
5      clang::SourceLocation caller = call->getBeginLoc();
6      clang::SourceLocation callee = call->getConstructor()->getBeginLoc();
7      if (isCoupling(caller, callee))
8      {
9        std::string callerName = getSourceLocationFileName(caller);
10       std::string calleeName = getSourceLocationFileName(callee);
11       shared::FileCoupling coupling(callerName, calleeName);
12       couplingCallback(&coupling);
13     }
14   }
15   return true;
16 }
```

First of all, in lines five and six, the source files are determined in which the object is created (caller) and in which the corresponding class is defined (callee). The `isCoupling` function now checks whether it is a coupling that is to be analyzed. The following criteria must be fulfilled for that:

▷ caller is in the current file to be analyzed

▷ caller and callee are not in the same file

▷ caller and callee are both in the files to be analyzed

If all of these criteria are fulfilled, a `shared::FileCoupling` object is created in line 11, which base class is `shared::AbstractCoupling`, and a pointer to this object is passed to the callback function, which has to be registered in `language::cpp::ClangCouplingFinder` beforehand.

3.3.4 Coupling Module

In the `coupling` module `shared::AbstractCoupling` objects are processed. This happens using a graph as datastructure, which can also be exported. Two interfaces are pro-

vided for this purpose. On the one hand, there is the `coupling::ICouplingAnalyser` interface, which provides a function signature for the `shared::AbstractCoupling` objects, which can be registered as callback in other modules. On the other hand, there is the `coupling::IResultExporter` interface, which provides a function signature for exporting a `coupling::CouplingGraph`. Each CouplingAnalyser needs an object of type `coupling::IResultExporter` to export the results.
In this module, the implementations of the previously mentioned interfaces can be called from outside the module.
Currently, there are two different implementations of the `coupling::IResultExporter` interface. On the one hand, there is the `coupling::CommandLineExporter`, which exports all nodes and edges of the graph on the command line on `std::cout`. On the other hand, there is the `coupling::JSONExporter`, which exports the graph as a JSON file.

3.3.5 Configuration Module

The `configuration` module offers the `configuration::ConfigurationManager`. This is the only class that is accessed from outside the `configuration` module. Three different possibilities are offered to provide configurations:

- Set command line arguments
- Use a configuration file in JSON format
- Use interactive input while the program is running

For command line arguments the class `configuration::CLIParser` is used. For the configuration file there is the class `configuration::JSONParser`. This parses the configuration file, which is in JSON format. Interactive input works directly with the `configuration::ConfigurationManager` class.
In the configuration folder there is a `configuration_args.json` file. All configuration options contained in this file can be used by the `configuration::ConfigurationManager`. An example entry in this JSON file looks like this:

```
1 {
2   "optionName": "language",
3   "shortOptionName": "l",
4   "mandatory": true,
5   "description": "language of project to be analysed",
6   "possibleValues": [
7     "cpp"
8   ]
9 }
```

If the mandatory flag is set for an option, then this option has to be provided. This can be done either by command-line arguments or by a configuration file. If the mandatory option is not provided by these, the interactive mode of the `configuration::ConfigurationManager` starts and the corresponding option has to be set at runtime. Command-line arguments overwrite entries in a configuration file and interactive mode is only used if options are missing at runtime for which the mandatory flag is set. The `configuration_args.json` only has to be changed if the tool is appended. The mandatory flag must not be removed for options that have this flag currently set. Adding it to other options, which currently are not mandatory, to enable the configuration of this option at runtime is possible.

3.3.6 Application Module

The `application` module contains the main function and is the only module, which depends on all other modules, so it cannot be built alone. In this module the other modules are linked. The `configuration::ConfigurationManager` loads the configuration. Then the callback function of `coupling::ICouplingAnalyser` is registered in the `language::ICouplingFinder`. In `coupling::ICouplingAnalyser` a `coupling::ResultExporter` is set, which outputs the results at the end. All of these functionalities are provided within `application::AbstractCoupling-Application`.
Configuration options for the language-specific implementation `language::cpp::ClangCoup-lingFinder` are set in `application::AbstractCouplingApplications` subclass `application::ClangCouplingApplication`.

3.4 Extandibility

There are different possibilities to extend the tool. In this section, two of these possible extensions will be discussed.

3.4.1 Append Configuration Options

For the extension of the tool it may be necessary to provide further configuration options. This works by extending the `configurarion_args.json` file in the `configuration` folder.
The current JSON file contains two lists of different configuration options, the base list and the `cpp` list. The `cpp` list is the list for the language specific C and C++ options. The options in this list are only loaded and useable if `cpp` is selected as language. To add a new language create a new list for this and add the language to the possibleValues list of the language option. It is important that the added name in the language option matches the name of the new list. For other extensions that do not involve adding a new language, the base list can be extended. The options can then be queried with the ConfigurationManager as follows:

```
configuration::ConfigurationManager* configurationManager =
 configuration::ConfigurationManager::getInstance();
std::string optionValue = configurationManager->getOptionValue("option-name");
```

3.4.2 Add another Language

To add support for a new language, different places in the in the `application` module, the `configurarion` module and the `language` module have to be extended. First an implementation of the `language::ICouplingFinder` interface for the corresponding language is needed. In the `application` module an implementation of `application::AbstractCouplingApplication` for the appropriate language must be added to the `application` module. In this implementation the new implementation of `language::ICoupliungFinder` can be configured. The main function in the `main-coupling.cpp` file has to be extended as documented there. Configuration options for the language can be appended in the `configurarion_args.json` as described in Section 3.4.1. At least the name of the new language must be added to the possibleValues list of the language option.

3.5 Performance Analysis

Performance did not play a major role in the development of this tool. The runtime of the tool varies depending on the size and file count of the project to be analyzed and the configuration of the tool. The static analyses presented in the following sections were performed with the following hardware:

- *CPU:* Intel Core i7 10700k 8 Core / 16 Threads @ 3.8 GHz
- *RAM:* 64 GB DDR4 @ 2667 MHz
- *SSD:* M.2 NVMe @ 3.400 MB/s read and 3.400 MB/s write
- *OS:* macOS 11.5.2 with VMWare Fusion running Ubuntu 20.04.

Chapter 4

Tool Documentation

4.1 Build the tool

The tool can be built with CMake. For this the requirements mentioned in Section 3.1 are needed. In the following, it is described how to build the project. First a `build` folder is created, e.g. with:

```
1 $ mkdir build && cd build
```

Afterward, CMake can be executed. The generator must be selected for the build system used. Building with make as build system works as follows:

```
1 $ cmake -G "Unix Makefiles" .. && make
```

Alternatively, other build systems can be used. The build command must then be adapted accordingly. The executable is named `StaticCouplingTool` located in the `build` directory.

4.2 Prepare project to be analysed

In order to be able to analyze c and c++ projects, a compilation database for the project to be analyzed is needed. A compilation database is a JSON file that specifies how a source file from the project can be parsed. There are several ways to create this compilation database. Projects that are built with CMake can create it directly during the build process. For this, the `CMAKE_EXPORT_COMPILE_COMMANDS` flag must be set in the main `CMakeLists.txt`. For the StaticCouplingTool, this flag is already set. The compilation database can be found in the build directory.
For other projects, the tool `Bear` [bear] can create the compilation database. This works with:

```
1 $ bear <build command>
```

In both cases the file `compile_commands.json` is created in the `build` folder.

4.3 Configuration options

There are different configuration options, which are shown in the following table:

option name	short name	type	required	count	description
general options					
project-path	p	directory	yes	1	path to project folder to be analysed. In case of C or C++ project this is the path to the directory which contains the compilation database
language	l	string	yes	1	Language of project to be analyzed. Currently only cpp is available. Both C and C++ projects can be analysed with it
config-path	c	file	no	0...1	path to configuration file (only useful as command line argument)
whitelist	w	directory file	no	0...n	Path to directories or files which should be analysed. All other files are ignored. Can be used with multiple files and directories
blacklist	b	directory file	no	0...n	path to a folder or a file which should be ignored
output	o	directory	no	0...1	Path to a folder, where results and configuration should be exported after analysis
help	h	-	no	0	show help (only useful as command line argument)
C/C++ specific options					
merge	m	-	no	0	header and source files are combined into one coupling artifact during analysis

There are three different ways to set the configuration arguments for a run of the tool. These are:

1. Set options as command-line arguments
2. Use a configuration file in JSON format

3. Use interactive input while the program is running

The interactive runtime input is only possible if an option is marked as required in the table from the previous Section 4.3, but the option is neither set as a command-line argument nor in a configuration file. A combination of these options is also possible. It should be noted that values passed as command-line arguments will overwrite the values from the configuration files.

4.3.1 Command Line Arguments

The command-line arguments can be used with the option name or the short option name from the Section 4.3. The name of the option can be used beginning with a double dash and the short name can be used with only a single dash. The following example uses both:

```
$ ./StaticCouplingTool --language cpp --merge -o <path> -p <path>
```

In this example, the language, merge, output, and project-path options are set.

4.3.2 Configuration File

A configuration file must be a JSON file. It can contain all options with their full option name. The short option name cannot be used in a configuration file. The help option and the config-path option cannot be used from a configuration file. To use a configuration file, the config-path option has to be set on the command line. An example for a configuration file is the following JSON file:

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

The tool can use this configuration file with:

```
$ ./StaticCouplingTool --config-path <path to config file>
```

4.4 Exporting Results

The results can either be output to the command line or written to the JSON file specified in the output-path option. With the help of the python scripts in the scripts directory, they can be visualized and modified.

4.4.1 Modify Results

Using `reduce.py`, the number of nodes in the result file can be reduced to $n \in \mathbb{N}_{>0}$ nodes. First, the sum of the incoming and outgoing edges is determined for each node in the graph, then the graph is restricted to the n nodes with the highest value, and subsequently all edges having at least one node, that is no longer in the new graph, are removed.
The Python script takes three command-line arguments in the following order:

1. number of nodes after reducing
2. Source result JSON file
3. Destination JSON file for reduced graph

The script can be executed as follows:

```
1 $ python3 scripts/reduce.py <number> <source> <destination>
```

4.4.2 Visualize Results

With the `render.py` script, the graph from a JSON file can be visualized. It is displayed as a weighted, directed graph. The path to the JSON file must be passed as a command-line argument. This works as follows:

```
1 $ python3 scripts/render.py <source>
```

4.4.3 Convert to Graph Modelling Language (GML)

The JSON file can be converted to an GML file. GML is a file format that describe graphs. This graphs can be visualized with several graph visualization software. One of these is gephi [Bastian et al. 2009].
The result JSON file can be converted using the `to_gml.py` of the `scripts` directory with:

```
1  $ python3 scripts/to_gml.py <source JSON file> <destination gml file>
```

4.4.4 Coupling Degree

In order to be able to analyze the results, the term coupling degree should be defined for a node of the resulting graph of the static or the Change Coupling analysis. The coupling degree of a node of the resulting graph represents the sum of the weights of all incoming and outgoing edges. With the help of the coupling degree, it is possible to get a hierarchical order of the nodes. This hierarchical order can be exported of an `result.json` file with the help of `highest_coupling_degree.py` from the `script` directory. To export a hierarchical list, the list length and the path to the `result.json` file is needed:

```
$ python3 highest_coupling_degree.py <list length> <source JSON file>
```

4.5 Example

In the following, the tool shall analyze itself. The following configuration file is used for this:

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

The `render.py` script visualizes the graph of the `result.json` file as shown in Figure 4.1. In

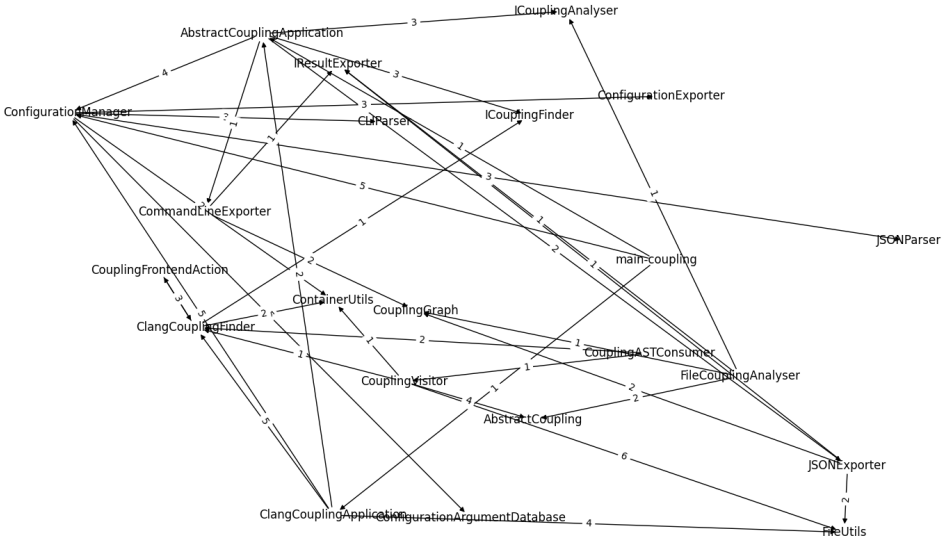

Figure 4.1. Visualized result graph of StaticCouplingTool analyzing itself

Figure 4.2 the modules mentioned in Section 3.3 are highlited. The files of `shared` directory are not considered in Figure 4.2, because they do not belong to a module.

A comparison with the previously presented component diagram in Figure 3.1 shows that the result of the static analysis is very close to the component diagram. Also, the

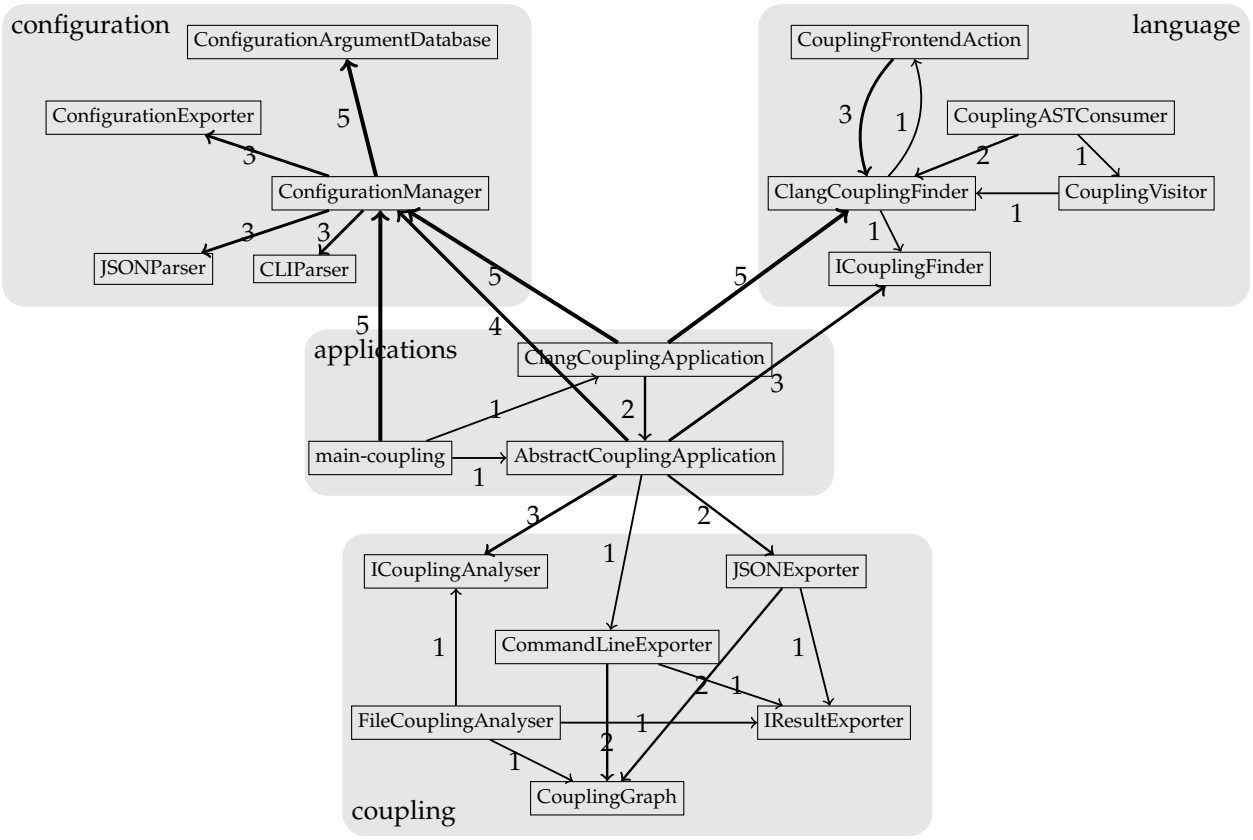

Figure 4.2. Result graph with modules

modularity mentioned in Section 3.3 becomes visible. There are only a few edges between nodes of different modules and more between nodes of the same module.
Next, the `result.json` file is converted to a GML file as described in Section 4.4.3 and can then be visualized with the help of gephi. The result can be seen in Figure 4.3. The node size indicates the sum of the weight of the incoming and outgoing edges. The biggest nodes have the highest value. These values can also be calculated as described in Section 4.4.4. The five nodes with the highest nodes are:

1. ConfigurationManager (30)
2. ClangCouplingApplication (17)
3. AbstractCouplingApplication (16)
4. ClangCouplingFinder (15)

5. CouplingVisitor (13)

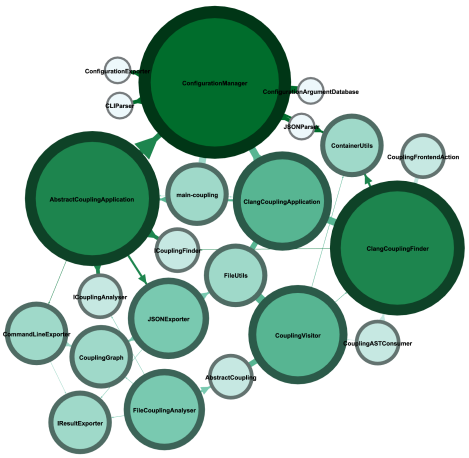

Figure 4.3. Gephi graph of result of tool analyzing itself

Chapter 5

Comparison with Change Coupling of Example Projects

In this chapter, the results of the StaticCouplingTool are compared with the results of GitCouplingTool developed in [Hofmann 2021]. For analyzing the results, header and source files are combined. C and C++ specific files, which differ only in the file extension, are considered as a common node in the resulting graph.

5.1 Git

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

5.1.1 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 5.1. 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

5.1.2 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".

5.1.3 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 5.1 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 5.1, 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 5.2. 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.

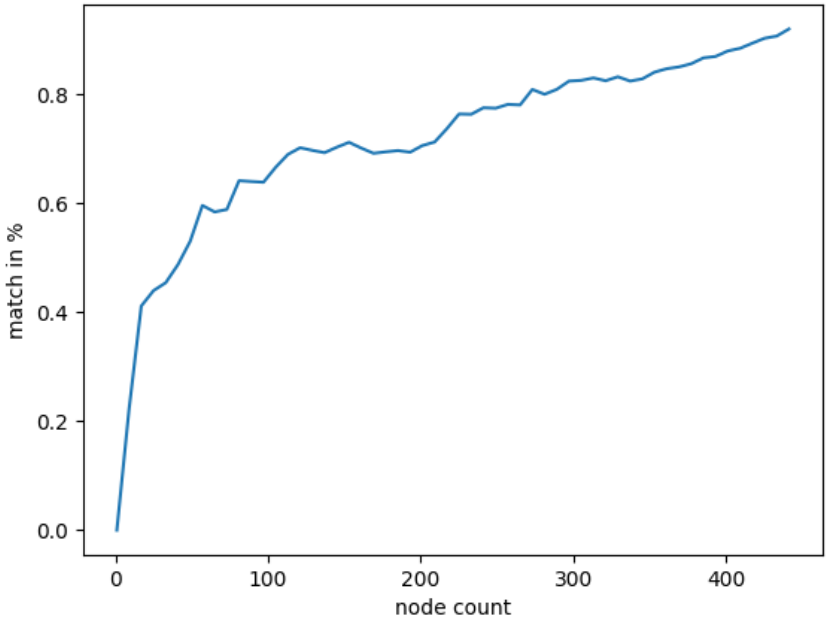

Figure 5.1. Overlap of nodes with highest coupling degree of Git for different count of considered nodes

Table 5.2. 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

5.2 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.

5.2.1 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>"
}
```

5.2.2 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".

5.2.3 Comparsion

The coupling degree of the nodes of the result graphs of the Static and the Change Coupling are compared in the following. Table 5.3 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 5.3. 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 5.2, 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 5.4 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.

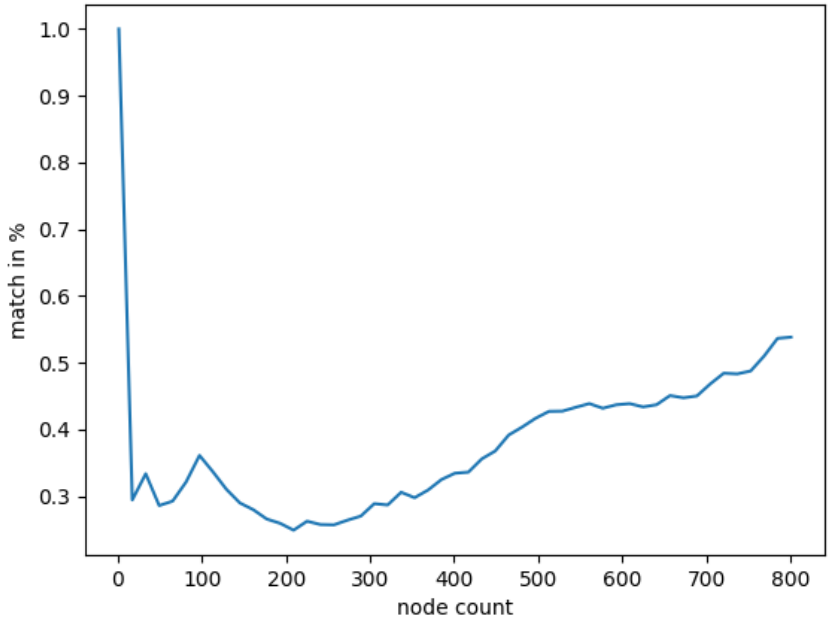

Figure 5.2. Overlap of nodes with highest coupling degree of BitKeeper for different count of considered nodes

Table 5.4. 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

5.3 linux

5.3.1 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>",
}
```

5.3.2 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".

5.3.3 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 5.3.1. The Linux kernel consists of 51903 C files (sum of header

Table 5.5. 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 5.3.1 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 5.5 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 5.3, 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 5.1) and BitKeeper (Figure 5.2). This difference is due to the problems mentioned at the beginning of this section.

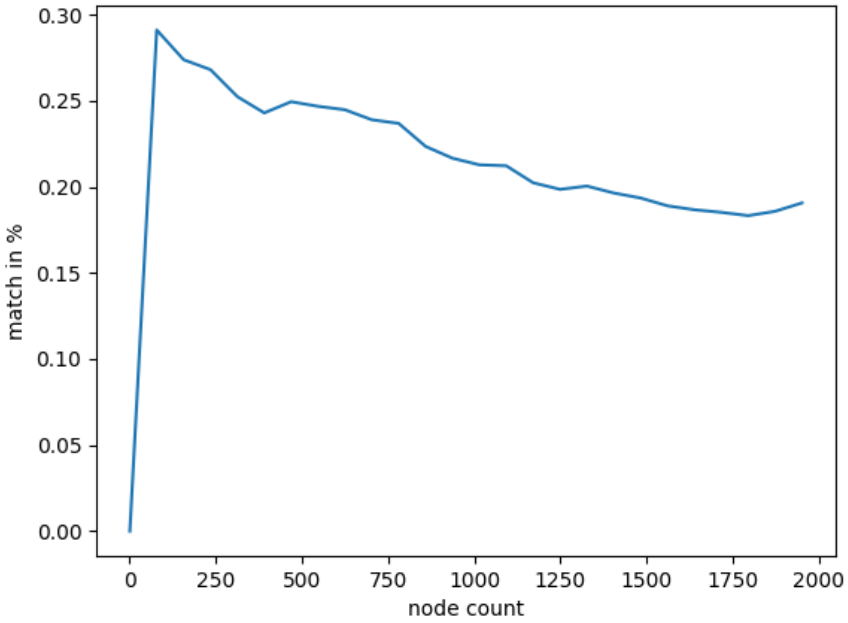

Figure 5.3. Overlap of nodes with highest coupling degree of Linux kernel for different count of considered nodes

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The main goal of this thesis was to develop a tool for static Coupling analysis for C and C++. The implementation of StaticCouplingTool fulfills this goal. The tool is configurable, extendable and was able to analyse the selected example projects with reasonable results. However, the performance could still be improved. The analysis of a configuration of the Linux kernel still took a few hours.
The second goal, the evaluation and comparison to Change Coupling of example projects, were also achieved. The results show some similarities, but there were also big differences between the results of Change Coupling and static coupling. In the analyzed sample projects, the file size often plays a more significant role in Change Coupling than in static coupling. 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.

6.2 Future Work

Some points can be improved or added to the StaticCouplingTool. One point is the performance. Since there is no parallelization, the analysis of big projects can take several hours or days. For example, it would be possible to parallelize by files. Multiple files could be analyzed at the same time.
Furthermore, the tool could be extended for other languages.
When analyzing and comparing the results, it might also be interesting to compare it not only with the Change Coupling but also with results of the dynamic analysis.

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 page 20)

[bear]. *Bear*. László Nagy. URL: https://github.com/rizsotto/Bear (visited on 09/28/2021). (Cited on pages 17, 29)

[bitkeeper]. *BitKeeper*. BitMover Inc. URL: http://www.bitkeeper.org (visited on 09/25/2021). (Cited on page 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 3)

[clang]. *Clang: a C language family frontend for LLVM*. The Clang Team. URL: https://clang.llvm.org (visited on 09/21/2021). (Cited on pages 4, 7)

[cmake]. *Build with CMake*. Kitware. URL: https://cmake.org (visited on 09/21/2021). (Cited on page 7)

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

[Hofmann 2021] S. Hofmann. Development of the GitCouplingTool and its Evalualtion. Bachelor thesis. Kiel University, Sept. 2021. URL: https://github.com/svnlib/GitCouplingTool. (Cited on pages 1, 2, 4, 25)

[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 page 4)

[qt]. *Qt 5.15*. The Qt Company Ltd. URL: https://doc.qt.io/qt-5/ (visited on 09/21/2021). (Cited on pages 7, 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 page 3)

[Torres et al. 2018] R. Torres, T. Ludwig, J. M. Kunkel, and M. F. Dolz. Comparison of Clang Abstract Syntax Trees using String Kernels. In: *2018 International Conference on High Performance Computing Simulation (HPCS)*. 2018, pages 106–113. (Cited on page 4)