HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Include-Analysis for C++ Source Code

Bachelor's Thesis

by

Aiko Bernehed, M. Sc.

submitted to

Software Engineering and Programming Languages
Prof. Dr. Michael Leuschel
Heinrich-Heine-University Düsseldorf

February 2022

Supervisor:
Dr. John Witulski

Abstract

The `C++` programming language is highly versatile and leaves many aspects of code organization to programmers, but provides functionality for code separation into *source* and *header* files. As projects have grown larger in size and `C++` has found its way into critical infrastructure, the community has started developing a host of best-practice guidelines for code organization. Since these are not programmatically enforced during development, static include analysis is an excellent tool for checking programmers adherence to these self-imposed guidelines. In this work, the Axivion Suite by the Axivion GmbH is used to develop a set of rules that enforce a number of guidelines as they may be encountered in real-world applications. The Axivion Suite uses a proprietary compiler to gather analysis data on existing `C++` source code of a given project. The rules presented here are enforced by running specially developed Python scripts on the analysis data generated by the Axivion compiler. Finally, the entire set of rules is used to analyze an existing application, "Notepad++", and the results are compared to Axivion's proprietary rules, gathered under the name of *Generic* stylechecks. "Notepad++" consists of a core program and the Scintilla and Boost libraries, all three of which exhibit a host of different issues. Overall, the rules developed here uncover 3,673 and Axivion's rules 3,890 violations. The minutiae of these results are discussed and final conclusions about this work and Axivion's implementation drawn.

Acknowledgments

I would like to acknowledge the extensive help my supervisor, Dr. John Witulski, provided me with this work. Furthermore, I would like to thank Axivion's Dr. Sebastian Krings for his insightful comments on an early version of the work carried out here, and Kristina Herbinger for her quick and industrious assistance with the Axivion licensing tools. Finally, I would like to thank my family for their continuous support of my work and education.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The C++ programming language was originally developed by Bjarne Stroustrup as an extension to C in 1979 [19, 21]. The intent was to augment the speed and low-level practicality of C by introducing object-oriented programming paradigms into the language. In fact, the precursor to C++ was called "C with classes". Over the subsequent years the language was developed further until the ISO C++ committee (officially named ISO/IEC JTC 1/SC 22/WG 21) promulgated the first C++ standard in 1998 under the title ISO/IEC 14882:1998 [16], informally known as C++98. The latest standard was published in December 2020 as ISO/ICE 14882:2020 [17], also known as C++20, and will likely be replaced in 2023. Since its inception C++ was, and still is, a very successful programming language rivaled in popularity only by Python, Java and its predecessor C [22].

1.1 The Necessity of Static Include Analysis

C++ supports the use of standalone compilation units, such that the user code only sees type and function declarations without any knowledge of their implementation [21]. The definitions of those types and functions can be located in separate *source* files, marked by endings such as ".cpp", ".cxx", or ".c++", which may be compiled independently from the user code. For the compiler to know which tokens are available, declarations of types and functions may be presented by *header* files, usually marked by a ".h" ending. The ISO C++ standard [17] does not require particular endings for *source* and *header* files, but a consistent naming convention is strongly recommended by Stroustrup [21].

The *header* files are best understood as an interface between the *source* files containing the actual code. They are made available to *source* files via the `#include` directive, so that *source* files subsequently have access to all the types, functions, variables, and macros that the targeted *header* file declares. For example, the `C++` Standard Library header `cmath` defines a function `sqrt`, which calculates the square root of a given numerical input. By writing `#include <cmath>` a user has access to `sqrt` without any knowledge how it is defined in the `cmath` source file.

`C++` compilation of a project consists of four major steps [14] as detailed in Fig. 1.1 and outlined as follows:

Preprocessor Directives for preprocessing are handled in each *source* file as they are encountered. `C++` has a plethora of preprocessor directives, e.g. `#define` statements for macro replacement or `#if`, `#ifdef`, and `#ifndef` statements to enable conditional compilation. `#include` statements lead to the statement being replaced with the entire target file, typically a *header*.

Compiler After the Preprocesser has finished the Compiler is presented with an augmented *source* file with macro and conditional text replacement and the *header* files copied into the text. The Compiler converts this pure `C++` code, without preprocessing directives, into an *assembly* file for the target platform.

Assembler The Assembler uses the generated *assembly* file to generate machine readable *object* code. The *object* files typically have a "`.o`" or "`.obj`" file ending. Often the Compiler invokes the Assembler directly, so that these intermediary steps are not transparent to the user.

Linker The Linker takes all the necessary *object* files and links them together into a library file or an executable program.

In the case of the `sqrt` example above, the *source* file has an `#include <cmath>` statement. The Preprocessor replaces that directive with the `cmath` *header* file, which, among other things, contains the forward declaration of the `sqrt` function. The compiler (and subsequent assembler) can now use this forward declaration to generate an *object* file using this reference to an undefined symbol. The Linker finally replaces that reference with the correct address from the `cmath` *object* file. Therefore, neither the user nor the compilation program ever needs to know about the `sqrt` source code and still have full access to it through the `cmath` *header* as an interface.

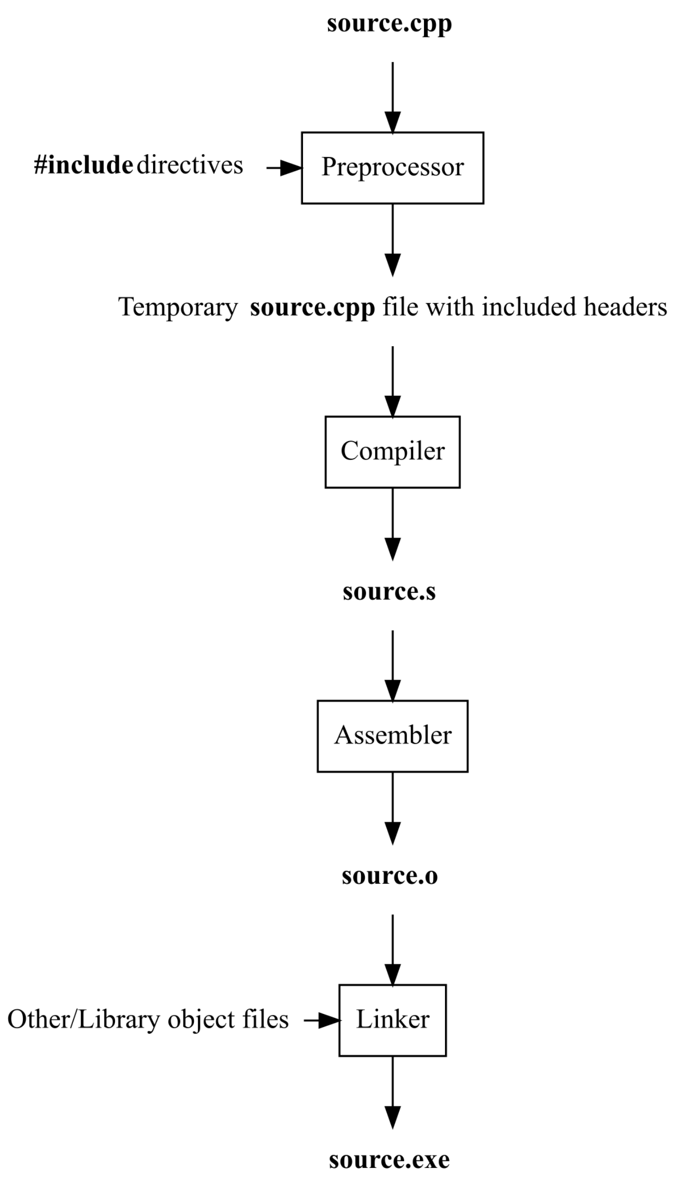


Figure 1.1: Compilation steps of a `C++` compilation program as presented by Mallia and Zoffoli [14, p. 3].

The ISO `C++` standard [17] states few requirements for code organization. Theoretically, one could provide a *source* and *header* file for every single variable and function used in a project or, in the other extreme, one could write the complete project into a single *source* file. Stroustrup himself writes: "Strictly speaking, using separate compilation isn't a language issue; it is an issue of how best to take advantage of a particular language implementation. However, it is of great practical importance. The best approach is to maximize modularity, represent that modularity logically through language features, and then exploit the modularity physically through files for effective separate compilation." [21, p. 53]. The ISO standard itself defines 124 Standard Library *headers*, also called system *headers*, which take up about three-quarters of the entire document in chapters 16-32 [17]. Next to that users can define their own *headers* and include them into *source* files.

Rules and regulations on how to structure a `C++` project and how to utilize *header* files is a matter of convention within the community. The exact guidelines followed vary between users, teams, and industries, may even be contradictory to one another, and change over time. A `C++` compiler adhering to the ISO standard will happily translate any correct program with little regard to code structure. Yet, adequate code structure is advantageous in providing a number of benefits:

- By increasing the organization of the project current and subsequent programmers have access to an understandable and serviceable code base.
- Code may be reused and shared between various projects and may even be published using *header* files without publishing the underlying source code. Within a project, often used functionality can be bundled along logical lines to have a single serviceable piece of code.
- Coding responsibility can be assigned to various programmers and teams, creating "experts" for a particular part of code.
- Standalone compilation units can be compiled separately from one another, thereby greatly increasing compilation time should changes be made to only one part of the code.

In static analysis the source code of an entire project can be automatically checked for adherence to a set of self-imposed rules the programming team wishes to follow. This is in contrast to unit tests and dynamic code analysis, which check a code's performance after compilation and during runtime on a set of predetermined inputs. Static include analysis, the main topic of this thesis, is a subset of static analysis tools and applies to *source* and *header* file organization and their contents. The rules imposed often follow best-practice advice by the `C++` community. Therefore, the concepts discussed here can be used for a wide range of `C++` projects.

1.2 State-of-the-Art

There are currently several options available for static include analysis, though only two open-source projects could be found. These will be touched upon briefly here.

cppclean

The open-source project **cppclean**, according to its own github page [4], "Finds problems in C++ source that slow development of large code bases". It does this by building an Abstract Syntax Tree of the source code to be analyzed and subsequently identifying various forms of unused code. Currently, **cppclean** can find the following

- Classes with virtual methods that may cause issues
- Global and static data that may cause errors in conjunction with threads
- Declared but undefined functions
- Unnecessary forward class and function declarations
- Undeclared function definitions
- Unnecessary `#include` directives in *header* files and inconsistent casing in `#include` directives and *header* file names.

Furthermore, there are a host of issues the development team has tagged as "planned", i.e. yet to be implemented. Unfortunately, the latest commit to the repository is from November 2019, so the project may be deemed discontinued.

Include What You Use

Include What You Use [6], or **IWYU** for short, is still under active development with **IWYU 0.17** released in December 2021. The first version was released in February 2011 and was a spin-off form work done at Google. The developers follow a philosophy that each file declares all the headers for all the symbols (types, functions, variables, and macros) used directly. When followed, this means that it is possible to edit any `#include` directive and statement without breaking any dependencies on that file.

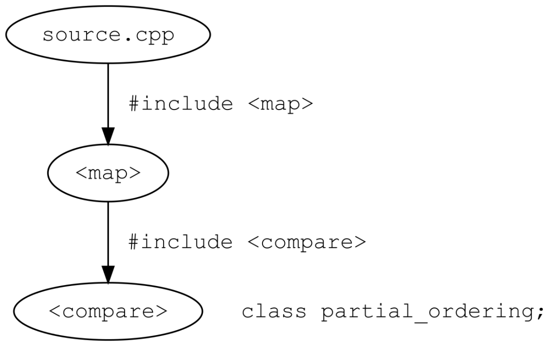


Figure 1.2: An illustration of how concatenating includes may lead to obscure and unexpected behavior.

For example, the Standard Library *header* `map` includes the *header* `compare`. If a source file includes `map` it has access to everything in `compare` as well, including the `class partial_ordering`. An overview of the dependencies is illustrated in Fig. 1.2. **Include What You Use** posits that `source.cpp` in this example should include `compare` directly. Otherwise, the include dependencies within the project may be opaque and deletion of `#include <map>` may lead to unexpected behavior.

The developers mention a host of benefits by using **IWYU** on the project's website [6]:

- Faster Compiles and fewer recompiles due to an elimination of unnecessary includes
- Easier refactoring due to each file being directly included, mitigating ambiguous include paths
- A certain level of self-documentation, since each *header* file clearly states its use
- Simpler cutting of dependencies, as the *header* files of irrelevant dependencies are easier to identify
- Reduction of code size and compile time by replacing `#include` directives with forward declarations. This aspect of **IWYU** is contentious though, since excessive forward declaration can lead to a vast quantity of new issues.

IWYU makes heavy use of the `clang` compiler-frontend. Integrating **IWYU** into a project can be cumbersome and it may not work on every programming environment. Furthermore, the development team itself is aware of major issues and writes: "This is alpha quality software – at best (as of July 2018). It was originally written to work specifically in the Google source tree, and may make assumptions, or have gaps, that are immediately and embarrassingly evident in other types of code." [6, "Instructions for users"]. Overall, **IWYU** is certainly a powerful tool, but it may be difficult to use and results must be operated on with care.

Commercial products

Apart from the two open-source tools **cppclean** and **Include What You Use** there is a host of commercial static analysis tools to chose from:

- Klocwork by Perforce (https://www.perforce.com/products/klocwork)
- CppDepend by CoderGears (https://www.cppdepend.com/)
- Parasoft C/C++ Test (https://www.parasoft.com/products/parasoft-c-ctest/)
- Coverity by Synopsys (https://www.synopsys.com/software-integrity/)
- Polyspace by Mathworks (https://www.mathworks.com/products/polyspace.html)

How many of these also provide some sort of static include analysis is unclear from the limited information available for each tool. The work carried out here was done using the Axivion Suite by the Axivion GmbH. The suite is a full-service package that provides static analysis, including static include analysis, and architecture analysis services. It supports a variety of build tools, is compatible with most modern version control systems, and provides plugins for Microsoft Visual Studio, Visual Studio Code and Eclipse.

1.3 The Axivion Suite

The Axivion Suite by the Axivion GmbH is a software analysis package for a plethora of static code analysis tools wrapped with professional reporting and presentation software. The origins of the suite date back to a project called *Bauhaus* originally developed at the Institute of Software Engineering at the University of Stuttgart in 1996 [18]. Together with the Working Group Software Engineering of the University of Bremen the project was further developed and spun out into the Axivion GmbH in 2006, which, as an independent company, was able to professionalize the software, address industry specific issues, and to handle and fulfill customer requests. The origins of the Axivion Suite can still be seen in some of the naming convention of software internals, where the name *Bauhaus* still appears.

The Axivion Suite uses two distinct tools to analyze project source code. Historically, the first is the proprietary *Intermediate Language* (IML), which contains semantic and syntactic information about the code in question [18, 20]. It has since been replaced by a construct called the *Intermediate Represenation* (IR), the internals of which are not published. However, it can be inferred from the documentation and usage of the suite how it is organized. The IR is built up as an *Abstract Syntax Tref* (AST), each node of which represents a general object within the programming language, e.g. a *source* or *header* file, a function, class, variable or namespace, and contains extensive information about that particular token. The IR is split into two parts, the *logical* and the *physical* part:

- The *logical* part, also called LIR, represents how a programmer might think about the code in the sense of functionality. As an example, it shows all entry points to a program, the logical execution calls, and the connection between different variables, functions, namespaces, classes, and other language tokens without any regard of where they are in the source code.
- In contrast, the *physical* part, abbreviated as PIR and somewhat misnamed for a software analysis tool, represents the actual structure of the code. It shows each *source* and *header* file, each declaration and definition of variables, macros, function, classes, etc. as they are encountered in each of these files and so on. The *physical* is a close representation of how the code is actually written.

The second tool the Axivion Suite uses is the *Resource Flow Graph* (RFG) [20]. The RFG is much more high-level than the IR and shows the project from a more global perspective. Nodes in the RFG represent elements that are architecturally relevant, whereas edges represent their relationship. A graphical analysis tool for the RFG, called *Gravis*, was developed and may pose the best way to analyze the RFG. The RFG has different views, which show different architectural parts of the project, for instance how function calls relate to one-another or how different *header* files are included. Furthermore, each node in the RFG contains all necessary information about the element it represents. The RFG is therefore best used in conjunction with *Gravis* to understand a project and its internal interdependencies as a whole.

To generate the IR and RFG for `C++` source code, the project has to be compiled with Axivion's proprietary build tool `CafeCC`. According to the suite's documentation [1], `CafeCC` is the frontend to `cafe`, `giraffe`, and `irlink`, which are the suite's compiler, assembler, and linker, respectively. `CafeCC` compiles code like an ordinary compiler, but gathers analysis data while doing so. It supports projects that would ordinarily be built with local build tools such as `make` or Visual Studio solutions. The Axivion Suite also allows tracking of issues via a large variety of version control systems like Git and Apache Subversion (SVN). Using the IR and RFG, users can access a large variety of analysis rules and also implement their own. The rules written and implemented in this thesis will be discussed in detail in Chapter 2. The results of a test of these rules on existing projects will be discussed in Chapter 3 and a conclusion and an outlook for future work provided in Chapter 4.

Chapter 2

Analysis Rules

The Axivion Suite provide access to a lot of different analysis and stylechecks out of the box, including major industry standardization guidelines like AUTOSAR [10] and MISRA [9]. Additionally, the suite allows the user to write their own analysis rules with an example of a custom rule provided in the documentation. The rules are written in Python and can be integrated into the analysis setup of an existing framework.

The configuration setup for the suite can be called with the command `axivion_config`. The Axivion documentation [1] provides instructions about how to implement and package custom rules, so that they can be found in and integrated into the configuration dialogue. By setting the environment variable `BAUHAUS_CONFIG` to a specific configuration, the shell command `axivion_ci` is used to build the IR (and RFG, if necessary), run the required analysis, and publish the results to the local Axivion dashboard, which can be accessed with a browser.

The setup of each custom rule follows the template provided in the documentation:

- The file needs to import `bauhaus.analysis, baushaus.ir` to access Axivion's functionality and the rule is written as a class descended from *analysis.AnalysisRule*
- The class has a couple instance variables which are required for the rule title and to output messages about the analyses results to the dashboard. Furthermore, the user can program any number of variables used by the rule, such as reporting thresholds. These can be changed when calling `axivion_config`.
- The method `get_rulehtml_description` is used to provide an HTML-coded

description of the rule's behavior to the configuration window and the dashboard.

- The method `execute` contains the intelligence of the rule. Depending on user programming, it usually receives an IR graph with which to carry out the required analysis. The class can also be augmented such that the method receives an RFG instead of or in addition to the IR graph.
- Messages are emitted by calling the inherited method `self.add_message(...)`, which allows to provide information about the node in question and a message to output to the programmer. For unit tests, this method is overwritten to return the messages to the testing suite, which uses `asserts` to ensure that the rule behaves as anticipated. Therefore, the tests are only minimally invasive to the script tested.

Most rules for static include analysis in the Axivion Suite fall under the category of "Generic Stylechecks". The rules are named *Generic-XXX*, but in order to aid legibility Axivion's rules will be called *Axivion Generic-XXX* in this work. Using these rules and the concepts presented in Chapter 1 as guidelines, several custom static include analysis rules were developed and tested. These rules will be called *AB - XXX* (for Aiko Bernehed) to differentiate them from Axivion's *Generic* rules. The Axivion Suite, version 7.2.5, provides the IR and RFG for subsequent analysis by the *AB* rules developed here. In this chapter, each rule's background and motivation will be discussed, the implementation shown on a pseudo-code basis and the functionality illustrated using include-graphs, where necessary. Each rule's behavior was tested using automated unit tests. An estimation of each rule's time complexity in relationship to the amount of `#include` directives *n* is also provided. The source code for this project is available upon reasonable request from the author's git repository at the Heinrich-Heine-University Düsseldorf[1].

2.1 Busy Headers

The rule about "Busy Headers" is directly inspired by *Axivion Generic-BusyHeaders* rule provided by the Axivion Suite. Each *header* file in the project is analyzed for the amount of files that include it and the amount of transitive includes it contains. Those two numbers are then multiplied and the *header* reported if a certain threshold is reached.

Whereas the number of including files is easily calculated, the number of transitive includes

[1]https://git.hhu.de/aiber100/ba_cs_code

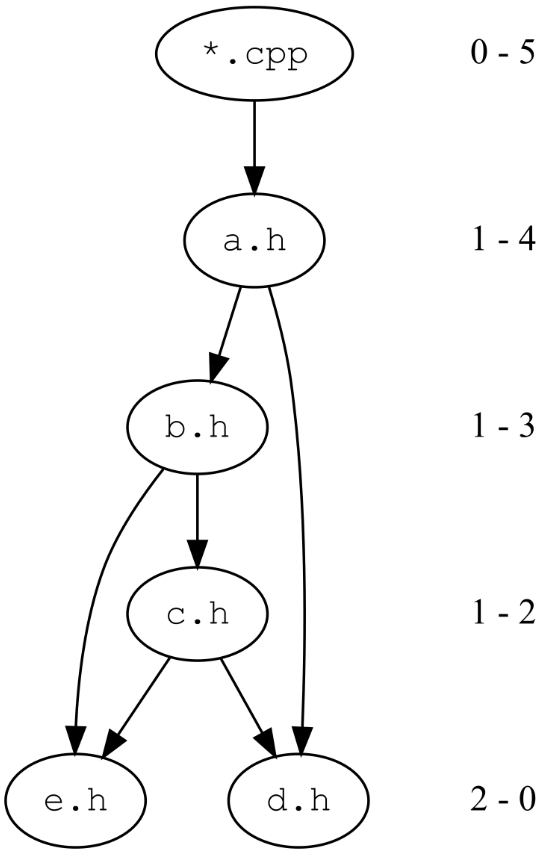


Figure 2.1: Representation of an example program with a total of seven `#include` directives. The left numbers represent the amount of includes including that file, whereas the right numbers represent the amount of transitive includes of that file.

a *header* contains is not quite so straightforward. It is calculated by iterating through all included files, the files they include, and so on, and counting the total number of uniquely included files. Fig. 2.1 shows the situation for an example of concatenating includes. For each *header* the numbers state the amount of files that include the *header* and how many transitive includes the *header* has. So the *source* file, marked as `"*.cpp"` here has no including files, but five transitive includes. Even though multiple include paths lead to `"d.h"` and `"e.h"`, each included file in the graph is only counted once.

The motivation behind this rule is, that the preprocesser, as outlined in 1.1 replaces any preprocessing directives it finds with the referenced code. Therefore, *header* files are not only included into any file that targets it via an `#include` directive, but also transitively included throughout the entire chain to the primary *source* file. In the above example in Fig.

2.1, `*.cpp` contains all the code from all the *header* files it transitively includes, from `a.h` to `e.h`.

Multiplying a *header's* transitive includes by the amount of times that specific *header* is included throughout the code, yields how much complexity this *header* alone introduces into the project. If a certain user-determined threshold is breached, the rule reports a violation in the dashboard. To remedy this issue, the programming team might consider restructuring the code base and removing `#include` directives from heavily used *header* files.

Algorithm 1 Busy Headers Rule

```
procedure FIND INCLUDES(node: ir.Node, trans: dict, direct: dict)
    if node previously visited then return                  \\ break cyclic includes
    else
        add node to previously visited
    for all incl in User and System include-declarations do
        FIND INCLUDES(incl.File, trans, direct)        \\ call with target file of include
        trans[node] += trans[incl]     \\ concatenate current node's with child's includes
        trans[node].append(incl)               \\ append current child to node's includes
        direct[incl].append(node)      \\ append current node to child's list of includers
    trans = set(trans)                 \\ make list of includes unique via set() method
    return

procedure EXECUTE(graph: ir.Graph)
    initialize trans, direct
    for all source-files in graph (ir.Physical) do          \\ operate on physical part of IR
        FIND INCLUDES(source, trans, direct)
    for all node in header-files do
        if trans[node] · direct[node] > threshold then
            report node
```

The working principle of the rule is shown in Alg. 1. The algorithm is split into two procedures, the first of which finds all transitive includes and direct includers for each file. The second procedure iterates over all *source* files, calls the first procedure, and subsequently calculates the product of transitive includes and direct includers for all *source* and *header* files, as outlined above. If the product of transitive and direct includes for a file rises above a user-defined threshold, the file will be reported. The programmer should consider rewriting the program to break *source* file dependencies on a large number of *header* files and mitigate long chains of `#includes` that cascade throughout the program. Since the algorithm iterates over all includes but doesn't contain nested loops, the time complexity is in $\mathcal{O}(n)$.

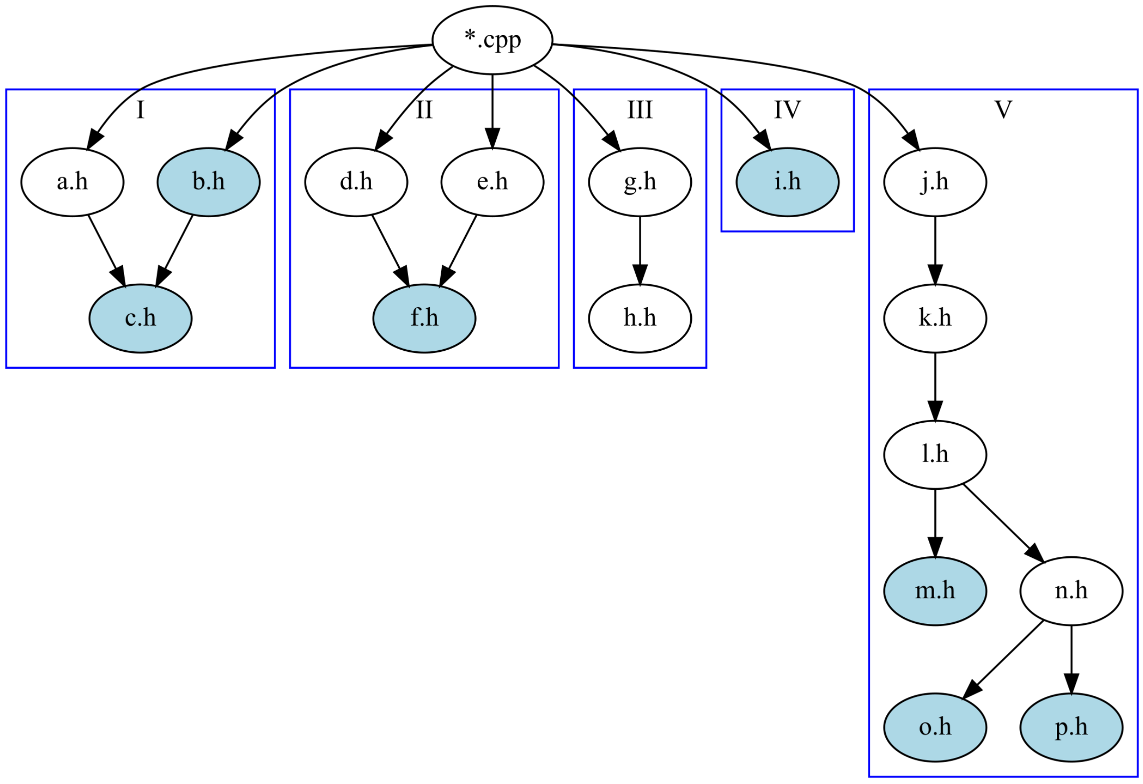
It should be noted here that this implementation of *AB - Busy Headers* and the *Axivion Generic-BusyHeaders* rule diverge in their results. *Axivion Generic-BusyHeaders* consistently reports one more transitive include than this implementation. When manually counting through the includes of test code, e.g. as detailed in Fig. 2.1, the *AB - Busy Headers* rule's count seems to be correct and is supported by counts provided by the RFG, which points to a bug in the *Axivion Generic-BusyHeaders* rule.

2.2 Chained Includes

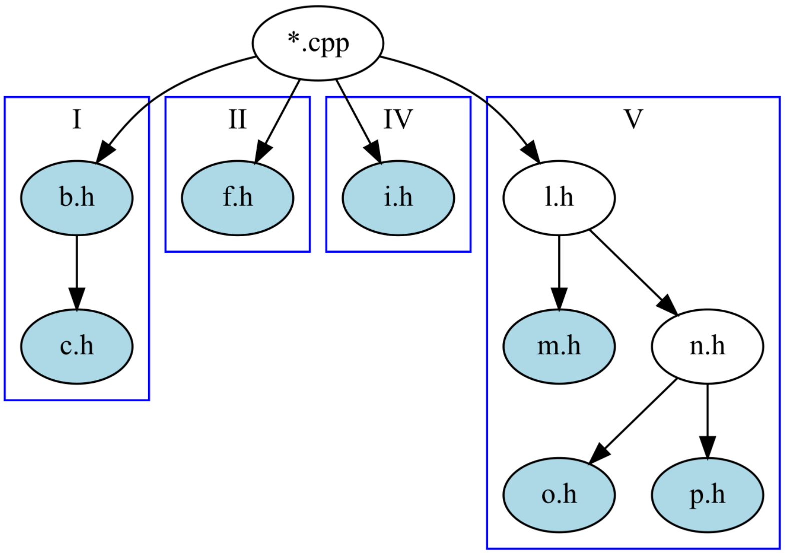
The rule *AB - Chained Includes* was inspired by **Include What You Use**, as outlined in section 1.2. Under the same philosophy, a file adhering should always include the headers for its tokens directly. An example of how includes might be chained to enable access to a token for the original *source* file is shown in Fig. 1.2. For an explanation on the veracity of mitigating chained includes, the reader is referred to section 1.2.

However, strictly adhering to including every single *header* file that is required for all tokens in a *source* file without allowing for chained includes may lead to an extreme number of necessary includes. This directly clashes with the rule against too many includes, discussed in section 2.8. Therefore, an alternative representation is discussed here: each *source* file should include the **most specific** *header* file it can include without having to split include trees. Furthermore, unnecessary *header* files should be flagged for removal.

The rule, as detailed in Alg. 2, first searches for all transitive includes of all `#include` directives and the declarations and definitions of all tokens in the *source* file. It then breaks all loops and ambiguous include paths. The algorithm tries to determine which is the include with the least source files in it, to acknowledge efforts made to already provide the most accurate includes. It also checks which direct ancestor *headers* are themselves declaring tokens and are therefore indispensable. The script then checks for newly empty include paths, which can be removed. Finally, the algorithm checks which *header* file in the include-path has the least transitive includes, while still containing all the descending tokens necessary headers. The algorithm iterates of all *source* and *header* files and contains a double-nested `for`-loop again iterating over all *header* files as source files and compares them to all included files. Therefore, the time complexity is in $\mathcal{O}(n^3)$.



(a) An example of include dependencies before the rule is applied.



(b) The same example of include dependencies after the rule is applied.

Figure 2.2: These are five examples of how concatenating includes may be arranged before and after applying the *AB - Chained Includes* rules. The blue *header* files contain tokens the *source* file uses and are therefore essential. A discussion of how the algorithm handles this situation is given in the text.

Algorithm 2 Chained Includes Rule

```
procedure FIND INCLUDES(node: ir.node, trans: dict, direct:dict)    \\ same as Alg. 1
    ...
procedure DECLARATIONS AND DEFINITIONS(node: ir.Node, sources: list)
    for all children of node do               \\ recursively search for all declarations
        DECLARATIONS AND DEFINITIONS(children, sources)
    if node has part in ir.Logical, has a name, and is not a namespace then
        DeclDefs = all files that are declaring and defining node (from ir.Logical)
        DeclDefs = set(DeclDefs)                              \\ ensure uniqueness
        for all DeclDefs do
            if DeclDef is a header file then
                sources.append(DeclDef)                  \\ save this token's header

procedure EXECUTE(graph: ir.Graph)
    initialize trans, direct
    for all source and user header-files in graph (ir.Physical) do
        FIND INCLUDES(source, trans, direct)
        initialize source-list
        DECLARATIONS AND DEFINITIONS(source, source-list)
        include-files = header files of all include-directives in source

        \\ now have all transitive includes and all declarations and definitions of all tokens
        initialize src-inc, inc-src                  \\ Dictionaries referencing lists
        for all src in source-list and incs in include-files do
            \\ store which includes have which source files as descendants and which
            \\ includes precede the source file. Due to forks these may be different
            if src in trans[inc] then
                src-inc[src].append(inc)
                inc-src[inc].append(src)

        \\ flag includes for removal, leaves includes without source-descendant
        to-remove = include-files - inc-src

        \\ check whether each src header has a unique include path, otherwise snip paths
        for src, incs in src-inc.items() do
            if len(incs) > 1 then    \\ multiple include paths to src, see Fig. 2.2a, Case I
                for all inc in incs do        \\ must choose one inc of all available incs
                    choose first inc which is also a source-include     \\ Fig. 2.2a, Case I
                    otherwise choose inc with least source-includes  \\ Fig. 2.2a, Case II
                delete all other sources from inc-src                 \\ ensure uniqueness
                src-inc[src] = inc                                    \\ ensure uniqueness
```

Algorithm 2 Chained Include Rule

```
41:     \\ paths now cleaned up, now search for empty paths and most accurate includes
42:     \\ that still cover their respective paths
43:     initialize to-replace, pot-remove
44:     for inc, srcs in inc-src.items() do
45:         if srcs is empty then
46:             pot-remove.append(inc)  \\ flag all includes NOW empty (after cleanup)
47:     min-trans = len(trans[inc])     \\ current amount of transitive includes for inc
48:     for all trs-inc in trans[inc] do
49:         skip, if not all items in srcs descend from trs-inc
50:         if len(trans[trs-inc] < min-trans then
51:             min-trans = len(trans[trs-inc]
52:             to-replace[inc] = trs-inc   \\ most accurate header for descending tokens
53:
54:     \\ report all rule violations
55:     for all header in to-remove, pot-remove, to-replace do
56:         report header
```

To better explain this rather complicated algorithm, the reader might consider Fig. 2.2a. Five different examples for potential include paths show how the rule works. The *source* file has a variety of `#include` directives, whereas only the *headers* with light blue coloring contain tokens used by the *source*:

Case I Even though `a.h` is included first, the algorithm notices that `b.h` also provides a token to the *source* file. Therefore `a.h` is flagged for removal.

Case II In contrast to Case I, the algorithm assigns `f.h` to the include path of `d.h`. Therefore, `e.h` is flagged for removal, whereas it is suggested to replace `d.h` by `f.h`.

Case III The algorithm cannot find a token used in `g.h` or `h.h`. Therefore, the directive `#include "g.h"` is flagged for removal.

Case IV Tokens in `i.h` are used. Nothing is flagged.

Case V The algorithm goes through all *headers* in the include path and checks how many transitive includes it has and whether all *headers* with tokens used by the *source* file are still included. Specifically, `l.h` has fewer transitive includes than `j.h` and `k.h` and still precedes all token-containing *headers*. `n.h` has fewer transitive includes than `l.h`, but doesn't contain `m.h`. The same limitation holds for all other *headers* in this subgraph. Therefore, the algorithm suggest to replace `j.h` with `l.h`

An updated results of how the five cases are handled when the programmer adheres to the suggestions made be the rule is displayed in Fig. 2.2b. As is directly evident, the amount of includes that need to be processed is significantly reduced by using the rule on this given example. It should be noted that the algorithm only flags includes for removal, or suggests to replace an include by a more accurate one. Therefore, using this rule never leads to more `#include` directives being proposed. Rather the rule will only reduce the amount of includes a user is suggested to use. This is in contrast to **Include What You Use**, which, according to the project's website, requires the user to always use the most accurate include they can for any given token. Furthermore, includes in *header* files are only flagged for removal with the caveat that an including *source* or *header* file might use the include. Therefore, the user can try to remove these files, but the algorithm does not guarantee that removal of `#include`-directives in *header* files may not break the program.

The Axivion Suite provides *Axivion Generic-LocalInclude* with similar functionality. The explanation provided with the rule reads: "Replace `#include` with forward declaration or more precise `#include` where possible". Where superfluous `#include`-directives are detected it provides the same suggestions for removal, but the rules diverge somewhat where suggestions for other `#include`-directives are concerned. When analyzing a real-world project, as outlined in Chapter 3, one receives a lot of suggestions. Particularly, both rules agree on flags for superfluous includes in *most* cases, but there are some where the two rules flag different includes for removal. That is most likely due to the `#include` dependencies being built differently, which cannot be confirmed though, since the source code for *Axivion Generic-LocalInclude* is not accessible. Lastly, the *Axivion Generic-LocalInclude* rule suggests to add multiple further `#include`-directives, which may increase the amount of includes in a file, in contrast to the rule described in this section.

2.3 Discouraged Content

The `C++` standard makes no mention of what is allowed to be in a header and what is absolutely included. Conceivably, a user could write the entire code of a program into a single *header* and only have the main compilation unit include that *header*. Therefore, as with many other topics discussed in this work, the decision what a *header* file should contain is informed by convention and ultimately decided by the programming team.

In Table 2.1 a compilation of what Stroustrup considers allowed and discouraged *header* content is shown. The rules for what is allowed and discouraged are, according to Stroustrup: "simply a reasonable way of using the `#include` mechanism to express the physical structure of a program." [21, p. 426]. Therefore, adherence to the guidelines presented here is not strictly necessary, but can definitely be adhered to as good practice.

The *AB - Discouraged Content* rule is implemented here by a simple script. The algorithm iterates over all *header* files it can find and checks for each token whether it is discouraged content. If so, the token is reported and should be moved to a *source* file by the programming team. The Axivion Suite does not seem to provide similar functionality. Since each *header* file is touched exactly once, the complexity of the algorithm is in $\mathcal{O}(n)$.

2.4 Include Guards

Once a project acquires a certain size, different parts of the program start to include the same *header* files. From the perspective of the program as a whole, this can lead to *header* includes and their declarations being made multiple times within the same compilation unit. Multiple types of *header* content, such as class definitions and inline functions, cause errors when used multiple times [21, p. 440 f.].

There are two alternatives for the programmer to follow. They can either reorganize the entire program to ensure that there are no redundancies, which is tedious for small and nearly impossible for large projects. Or they can use a strategy to ensure that the multiple inclusion of *header* files will not lead to errors. This is what *include guards* are designed to do. An *include guard* for a *header* of name `example.h` may be defined as follows:

```
#ifndef EXAMPLE_H
#define EXAMPLE_H

... // The complete code of example.h

#endif
```

The first time the preprocessor encounters the leading `#ifndef`, the token `EXAMPLE_H` is not known to it. Therefore, the token gets defined by the following line and the entire code of

Allowed Content	
Named namespaces	`namespace N {...}`
`inline` namespaces	`inline namespace N {...}`
Type definitions	`struct Point(int x, y);`
Template declarations	`template<class T> class Y;`
Template definitions	`template<class T> class Z{...};`
Function declarations	`extern int decl(const char*);`
`inline` function definitions	`inline char indecl(char* p){...}`
`constexpr` function definitions	`constexpr int sqr(int n) {return n**2;}`
Data declarations	`extern int a;`
`const` definitions	`const float pi = 4;`
`constexpr` definitions	`constexpr int foo=13;`
Enumerations	`enum class Starter {Bulbasaur, Charmander, Squirtle};`
Name declarations	`class Matrix;`
Type aliases	`using value_type = long;`
Compile-time assertions	`static_assert(4<3);`
Include directives	`#include "awesome_header"`
Macro Definitions	`#define MATRIX_VERSION 42.0.1`
Conditional compilation directives	`#ifdef __bachelorThesis`
Comments	`// this is the end of this list`

Discouraged Content	
Ordinary function definitions	`int answer(char* universe) {return 42;}`
Data definitions	`int a;`
Aggregate definitions	`char** choice[] = {"red_pill", "blue_pill"};`
Unnamed namespaces	`namespace {...}`
`using`-directives	`using namespace Skywalker;`

Table 2.1: An overview over which content is allowed and discouraged in a header file according to *The C++ Programming Language* by Bjarne Stroustrup (slightly adapted) [21, p. 425 f.]

the *header* within the *include guard* is made available to the source file. Any subsequent time the preprocessor encounters the line `#ifndef EXAMPLE_H`, the token is already defined and the check fails. The entire code until the `#endif` statement is ignored. Therefore, the *include guard* ensures that the code from the *header* file, even though included multiple times, is only compiled and executed once.

A couple of facts should be pointed out:

- First of all, the C++ ISO standard [17] does not require or even mention *include guards* at all. Any *header* file that does not contain an *include guard* is therefore fully compliant with the standard and will be compiled successfully. Nevertheless, any programmer not using *include guards* has to deal with the problems stated above.
- Secondly, as *include guards* are not enforced, there is no naming requirement. Convention dictates that an *include guard* should use the name of the *header* file it is contained in, but there is no requirement to do so. Conceivably, the programmer may use any name they feel like. It should only be unique within the entire project, so that crucial headers with the same *include guard* are not mistakenly ignored.
- Finally, the preprocessing directive `#pragma once` is an often used alternative to *include guards*. This directive is not mentioned in the ISO standard and is highly contentious in the C++ coding community, as evidenced by various threads on Stackoverflow[2] and Reddit[3]. According to the Wikipedia[4] article on `#pragma once`, there are a number of advantages and caveats of using the directive over *include guards*, even though all major compilers support the directive. In the *AB - Include Guards* rule `#pragma once` directives will be identified and the use of *include guards* suggested instead.

The algorithm implemented for this rule identifies the *include guards* of all header files in the project and reports any duplicates. It also identifies `pragma once` directives and missing *include guards* and suggests implementation of an *include guard*. As detailed in Alg. 3 the rule first gathers all *include guards* and `#pragma once` directives. It then compares all *include guards* and first reports all user-written *guards* that are the same as *guards* used by system *headers*. The rationale is that the user can change their own *include guard* but not that of a system header. The rule also reports all *header* files that have no *include guard* or

[2]https://stackoverflow.com/questions/1143936/pragma-once-vs-include-guards
[3]https://www.reddit.com/r/cpp/comments/4cjjwe/come_on_guys_put_pragma_once_in_the_standard/d1j04te/
[4]https://en.wikipedia.org/wiki/Pragma_once

Algorithm 3 Include Guards Rule

```
 1: \\ Identifies all include guards of requested type
 2: procedure IDENTIFY GUARDS(graph: ir.Graph, include-type: string)
 3:     initialize guards
 4:     for inc in descendants of include-type in graph do
 5:         initialize found-guard = False, found-pragma = False
 6:         for all child in inc do
 7:             if child is Preinclude-Directive then              \\ Axivion specific directive
 8:                 continue
 9:             else if child is Pragma-Directive then              \\ possible "#pragma once"
10:                 found-pragma = True
11:                 continue                           \\ Could still contain an include guard
12:             else if child is #ifndef then                              \\ include guard found
13:                 guards[inc] = child.Name
14:                 found-guard = True
15:                 break
16:             else if child is macro then              \\ Axivion might identify this instead
17:                 possible-guard = child.Name
18:                 \\ Check whether the macro found has an #ifndef attached to it
19:                 for all #ifndef in inc.children do
20:                     if #ifndef macro name = possible-guard then
21:                         guard[inc] = possible-guard
22:                         found-guard = True
23:             else
24:                 break                   \\ If nothing else breaks, no guard has been found
25:         if not found-guard and not found-pragma then
26:             guards[inc] = None
27:         else if not found-guard and found-pragma then
28:             guards[inc] = #pragma
29:     return guards                  \\ returns all include guards of specified type to caller
30:
31: procedure EXECUTE(graph ir.Graph)
32:     sys-includes = IDENTIFY GUARDS(ir.Graph, System-Includes)
33:     usr-includes = IDENTIFY GUARDS(ir.Graph, User-Includes)
34:     for all usr in usr-includes and sys in sys-includes do
35:         report system-include guard used by usr-include                    \\ skip #pragma
36:     for all usr-1 in usr-includes do
37:         report missing include guard
38:         report "#pragma once" instead of include-guard
39:         for all usr-2 in usr-includes do
40:             skip if usr-1.node == usr-2.node             \\ skip if this is the same include
41:             if usr-1 == usr-2 then
42:                 report duplicate include guard in different user includes
```

use `#pragma once` directives. Finally, the algorithm identifies all uniquely different user *header* files with the same *include guard* and reports them. Since the algorithm compares all *header* files against all other *header* files, its time complexity is $\mathcal{O}(n^2)$. It should be noted that the Axivion Suite does not identify *include guards* or `#pragma once` directives, but rather shows macro definitions, pragma directives, and #ifndef statements. Therefore, logic dictates that these should represent *include guards* or `#pragma once` directives in the code, but ultimately, the user will have to determine whether the given statement was identified correctly and the reported error is valid.

With *Axivion Generic-DuplicateIncludeGuard* and *Axivion Generic-MissingIncludeGuard*, the Axivion Suite provides two rules with a similar functionality to Alg. 3. While the former works similarly in detecting duplicate include guards as the rule presented here does, the second Axivion rule, *Axivion Generic-MissingIncludeGuard*, does not flag `#pragma once` directives. It does flag the same missing include guards as this rule does though.

2.5 No Absolute Paths

Include paths, according to the ISO standard [17, p. 16] can be any combination of source characters except newline, `'>'`, and `'"'`. Therefore, any manner of paths are allowed to be used in the `#include` directive, including absolute paths. However, absolute paths are in no way portable to other machines and can wreck havoc on a programming teams' efforts to share and work on distributed code. Consider code on Bob's machine that he wants to share with Tracy. Bob may write an include directive as

```
#include "C:\Users\Bob\OurProject\Headers\foo.h"
```

Tracy clones the git repository to her programming directory and has the *headers* lying in

```
C:\Users\Tracy\Programming\NewProject\OurProject\Headers\foo.h
```

Even though the end of the path is the same, the compiler is not going to find the *header* file targeted by Bob's include statement. Therefore, absolute paths should always be avoided.

Even if the user only has small local builds, the use of absolute paths may lead to bad habits and subsequently requires the programmer to relearn their ways.

The algorithm used for this *AB - No Absolute Paths* rule is extremely straightforward. It iterates over all `#include` statements and checks, whether an absolute include path was used, using Python's inbuilt functionality. If so, that node and path will be reported for alteration by the programmer. As all `#include` directives are touched only once, the time complexity is in $\mathcal{O}(n)$. The Axivion rule *Axivion Generic-NoAbsoluteInclude* has the same functionality.

2.6 No Conditional Preprocessing

The `C++` ISO standard defines a number of conditional preprocessing directives: These are `#if` and `#elif` which get activated when the following conditional statement evaluates to non-zero, `#ifdef` if the following macro has been defined, and, conversely, `#ifndef` if the following macro has not yet been defined [17]. The last directive has already been used in section 2.4 to build a framework for *include guards*. Conditional preprocessing can often be encountered to allow compilation and different definitions for different target platforms.

However, according to NASA's 10 Rules for Developing Safety-Critical Code, the use of the preprocessor is highly problematic. Specifically, Rule 8 states: "The use of the preprocessor must be limited to the inclusion of header files and simple macro definitions. [...] The use of conditional compilation directives is often also dubious, but cannot always be avoided." [15]. The document further elaborates that the use of conditional preprocessing tools is highly obfuscating and should therefore be avoided at all cost.

The Axivion rule *Axivion Generic-NoIfdefInHeader* takes a similar approach and prohibits all of the conditional preprocessing directives mentioned above except for *include guards*. The rule presented here, *AB - No Conditional Preprocessing*, uses the "Identify Guard" procedure in Alg. 3 to find a *header* file's include guard. It then finds all other conditional preprocessing tokens and reports each in turn for removal or alteration by the programmer. Again, the algorithm touches each *header* file once, setting its time complexity at $\mathcal{O}(n)$.

2.7 Primary Includes

The code organization outlined in section 1.1 and the allowed and discouraged content detailed in Table 2.1 underline the concept of using *header* files as an interface for *source* files. Nevertheless, as an `#include` directive will allow the programmer to include any kind of file, the preprocessor will easily copy all the code from the included file to the position of the `#include`. Yet, in order to keep large codebases understandable, it is advised to strictly adhere to the separation of *source* and *header* files and not to include any *source* files.

The algorithm presented here in Alg. 4 checks exactly that. It aggregates all *source* files and even checks whether any *source* files might have been missed. Particularly, the algorithm can identify all compilation entry points using the LIR (see section 1.3) and find their respective *source* files. This might be necessary if an entry point, i.e. a `main()` function, is, for some very obscure reason which deeply clashes with the concepts brought forth in this work, located in a *header* file.

Lastly, it can happen that two different includes in different files target the same *header* file "a.h" with `#include "a.h"` and `#include <a.h>`. Since this rule already compiles all user and system include directives and their targets, this behavior is easily identified and can be reported. The programmer is encouraged to work through the reported includes and to remedy wrong user and system includes and to remove all *source* files as include targets. Even though the rule operates on each `#include` directive multiple times, there are no nested for loops, bringing its time complexity to $\mathcal{O}(n)$.

This is also the behavior of the Axivion provided rule *Axivion Generic-FileKindDifference*. However, in contrast to the Axivion rule, reporting in the *AB -Primary Includes* rule has been altered such that the user may see exactly which files have clashing system and user includes. This is not provided by *Axivion Generic-FileKindDifference*, which only states which *header* file is targeted by diverging `#include` directives, but not where these directives are located.

Algorithm 4 Primary Include Rule

```
 1: procedure EXECUTE(graph: ir.Graph)
 2:     \\ get all source and all header files
 3:     for all files in project do
 4:         primary: list all source files
 5:         system: list all included system header files
 6:         user: list all included user header files
 7:
 8:     \\ go through the entry points to check whether any source files have been missed
 9:     use LIR to find compilation unit's entry point
10:     find source file associated with entry point
11:     add that source file to primary
12:
13:     \\ check whether any primary file is an include target and whether user and system
14:     \\ includes are not mixed
15:     for all prim in primary do
16:         if any prim in system or user then
17:             report system or user node and include
18:     for all sys in system do
19:         if any sys in user then
20:             report system and user node and include
```

2.8 Too Many Includes

This rule is rather straightforward, as is it's counterpart *Axivion Generic-TooManyIncludes*: The rules tallies up the amount of `#include` directives in each *source* and *header* file and reports files with an amount of includes larger than a preconfigured threshold. Each *header*-file is analyzed once, therefore again leading to time complexity $\mathcal{O}(n)$. A large number of includes can lead to excruciatingly long compile times, as a large number of dependencies have to be resolved. The programming team is therefore advised to rethink their code structure, to consolidate different *header* files into more meaningful units, and to drastically cut the amount of `#include` directives each file uses. This rule is closely related to **AB - Busy Headers** (see section 2.1) and fixing the code base to adhere to one rule may already resolve issues with the other.

2.9 Wrong Include Casing

Different file systems and operating systems can be either case-sensitive or case-insensitive. This determines whether the *header* names "FOO.h" and "foo.h" are considered distinct or equivalent. When switching between different file systems, care must be taken to ensure that case sensitivity is not an issue. In general, Windows systems tend to be case-insensitive, while Unix systems operate case-sensitive [11]. However, different version handle casing differently and in some installations casing behavior can be altered.

In order to avoid problems when porting code from a case-insensitive to a case-sensitive environment, it is encouraged to have the same casing in the `#include` statement as in the *header* filename. This also aids legibility of a codebase. The rule *AB - Wrong Include Casing* iterates over all `#include` directives in all *source* and *header* files and checks whether the directive and include target have the same casing. If not, the directive is reported and should be amended by the programmer. This is the same behavior as is exhibited by *Axivion Generic-WrongIncludeCasing*. The *AB - Wrong Include Casing* rule operates by iterating over each `#include` exactly once and accessing it's target. Thus the rule operates in time complexity $\mathcal{O}(n)$.

2.10 Wrong Include Type

The `C++` ISO standard defines two different kinds of `#include` directives, the `<...>` and `"..."` sequences [17, ch. 5.8]. The space inbetween the delimiters is reserved for the filename of the *header* to be searched for. The standard does not however define what these different delimiters should be used for. It notes: "An implementation can provide a mechanism for making arbitrary source files available to the < > search. However, using the < > form for headers provided with the implementation and the `" "` form for sources outside the control of the implementation achieves wider portability." [17, p. 465]. Therefore, any standard compliant compiler will compile any combination of includes, regardless of which include type is used and whether the include is a user-defined *header* or one provided by the implementation, what is generally called a Standard Library or system *header* (see section 1.1).

`<cassert>`	`<cctype>`	`<cerrno>`	`<cfenv>`	`<cfloat>`
`<cinttypes>`	`<climits>`	`<clocale>`	`<cmath>`	`<csetjmp>`
`<csignal>`	`<cstdarg>`	`<cstddef>`	`<cstdint>`	`<cstdio>`
`<cstdlib>`	`<cstring>`	`<ctime>`	`<cuchar>`	`<cwchar>`
`<cwctype>`				

Table 2.2: `C++` Standard Library implementations of `C` Standard Library *headers* used for backwards compatibility. [17, p. 485]

It is recommended to use `<...>` includes for Standard Library and `"..."` includes for user defined *header* files. Furthermore, the ISO standard supports a number of `C` Standard Library *headers* for backwards compatibility, but also provides current alternatives for `C++` that provide the same functionality but ensure static type safety [17, p. 478]. It is therefore also recommended to use the `C++` instead of the `C` Standard Library *headers*. Furthermore, there are a couple of *header* files that are empty or have since been deprecated.

The `C++` replacements for `C` Standard Library *headers* are listed in Table 2.2. The `C` *header* is of the form `name.h` and is replaced by the `C++` *header* `cname`, where `name` is the basename of the file. Therefore, it is simple to identify `C` system *headers* and flag them for replacement. Furthermore, through the evolution of the ISO standard from `C++98` to `C++20` and soon to `C++23`, several *headers* have become deprecated and/or empty. A compact overview of these, which is easier to read than the standard, can be found in the unofficial C++ reference[5]:

- The *headers* `<ccomplex>` and `<complex.h>` only include the *header* `<complex>` and should be replaced by it.
- The *headers* `<ctgmath>` and `<tgmath.h>` only include the *headers* `<complex>` and `<cmath>` and should be replaced by them.
- The *headers* `<ciso646>`, `<cstdalign>`, `<cstdbool>`, and the `C` *header* files `<iso646.h>`, `<stdalign.h>` `<stdbool.h>` are meaningless, since the macros they define are now `C++` keywords or their functionality is no longer required.

The algorithm for this rule searches through all `#include` directives in the project once, therefore being in time complexity $\mathcal{O}(n)$, and reports issues according to the following logic:

[5]https://en.cppreference.com/w/cpp/header

- A user-defined *header* included with the system include `<...>` should be included with the user include `"..."` instead.
- A system-defined *header* included with the user include `"..."` should be included with the system include `<...>` instead.
- When a user include `"..."` is used for a `C` Standard Library *header*, the programmer should use the system include `<...>` and `C++` Standard Library *header* instead.
- A `C` Standard Library *header* included with the correct system include `<...>` should use the `C++` header instead.
- If the *headers* `ccomplex`, `complex.h`, `ctgmath`, or `tgmath.h` are included, they will be flagged for replacement as outlined above. The use of the system include `<...>` syntax will also be recommended, should a user include `"..."` be used.
- If a meaningless *header* as outlined above is used, it will be flagged for removal.

The Axivion Suite provides the rule *Axivion Generic-NoCHeaderInclude* which only checks whether a `C` Standard Library file is being used and suggests to replace it with the appropriate `C++` Standard Library file. This also includes (now) meaningless *headers* such as `<iso646.h>` which aren't flagged for removal but suggested to be replaced with the `C++` *header* `<ciso646>`. Furthermore, the rule *Axivion Generic-IncludeKind* checks whether the correct type of include (`"..."` vs. `<...>`) is being used. That rule seems to perform exactly as this rule here does, where the wrong kind of include directive was used.

2.11 Analysis not implemented

Apart from the rules mentioned until here, there are several further concepts that can be targeted with static include analysis. Two of these, Precompiled Headers and Template analysis, will be briefly explained here and an explanation provided as to why they were not explored within this work.

Precompiled Headers

Precompiled Headers (PCH) are actually not part of the `C++` ISO standard [17], but they are supported by the Visual Studio Suite [12], GCC [5], and Clang [3]. The idea behind PCH files is that multiple includes of a *header* in different compilation units within a project greatly increase compilation time [12]. Depending on the compiler, a project can have a single or multiple chained PCH with an unlimited number of includes [3, 5]. However, since the PCH needs to be recompiled after every change made, only sufficiently stable *headers* should be added to it.

Precompiled Headers have a number of caveats though, some of which are acknowledged in different compilers' documentation [3, 5, 12], others of which are discussed within the `C++` community[6]

- The PCH has to be produced by the same compiler binary file as is currently being used for compilation [5]. This can lead to problems, if a distributed programming team is using slightly different versions of the same build-system.
- PCH can easily become bloated, as it is simply to add content, but requires some house-keeping to remove content. This is turn can obfuscate the benefits of PCH if frequent recompiles are required.
- In line with the point above, PCH need to be kept up to date and therefore might introduce additional maintenance.
- Should inconsistent `#define` directives occur inside and outside of PCH, they could be missed by the compiler and linker. This would result in a broken build, which might lead to inconsistent runtime errors that can be excruciatingly difficult to debug.
- Portability of code is pretty much excluded without recompiling the PCH.

Precompiled Headers should only be used by programming teams that are experienced in their application and only for *headers* that are not prone to further change. This is tough to adhere to for a programming team and difficult to analyze in static include analysis. The Axivion rule *Axivion Generic-PCHIncludes* suggest to move often used `#include` directives to PCH. However, this does not take into account how often each *header* file might have been changed in past versions of the project and is prone to change in future versions. This requires access to the projects history, necessitates statistical analysis of past changes, and subsequent prognosis of further changes. On top of that, PCH-files need to be compared to the current state of the project and adequate conclusions drawn from that analysis. Even the Axivion Suite does not provide functionality to investigate PCH-files. Therefore, Precompiled Headers were deemed to be out-of-scope for this thesis.

[6]https://www.reddit.com/r/cpp_questions/comments/gw7vrs/what_are_the_downsides_to_precompiled_headers

Templates

Templates are a construct that allow for further abstraction within the class framework of C++. They are defined in the ISO standard [17] and are therefore a regular part of the C++ programming language. Stroustrup provides an excellent overview over how templates are used [21, ch. 23]. A template is constructed by programming

```
template<typename T> class someClass{...};
```

The type T is then available as a general object within the class. It can be used as a type, struct, or even as a template object itself. Stroustrup provides a small example which serves to illustrate the usefulness of templates here as well [21, p. 668]

```
template <typename T>
class String{...};

String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;

struct Jchar { /* ... */ }; // Japanese character
String<Jchar> js;
```

The example shows that the defined `class String` can be used with any character as well as newly implemented characters, such as a `struct` of a Japanese character set. This is all using one class definition and therefore greatly eases development of the class for a whole host of applications.

However, when it comes to static include analysis, templates raise a host of issues. The freedom and ambiguity in their writing means that T could be any object in the project and could be defined in any *header* file used. Even more confounding, C++ allows function template overloading, meaning that the following code-snippet is allowed [21, p. 689]

```
template<typename T>
    T sqrt(T);
template<typename T>
    complex<T> sqrt(complex<T>);
double sqrt(double);

void f(complex<double> z) {
    sqrt(2);     // sqrt<int>(int)
    sqrt(2.0);   // sqrt(double)
    sqrt(z);     // sqrt<double>(complex<double>)
}
```

Furthermore, the `C++` Standard Library makes heavy use of templates, which provides it with an enormous range of functionality. For instance, the class `unordered_map` uses templates in the following way [21, p. 914]

```
template<typename Key,
         typename T,
         typename Hash = hash<Key>,
         typename Pred = std::equal_to<Key>,
         typename Allocator =
             std::allocator<std::pair<const Key, T>>>
class unordered_map {...};
```

Therefore, the argument for constructing an `unordered_map` object itself contains nested template classes. All of the information above is known during compile time and can therefore conceivably be statically analyzed. The work necessary would basically require the analysis to work through the program as a `C++` compiler does and would easily require more development than the rest of the rules presented here combine. Not even the Axivion Suite provides functionality to analyze Templates and flag potential issues. Therefore, this aspect of the `C++` language was also deemed out-of-scope.

Chapter 3

Analysis of an Existing Project

The rules above check out on the test cases provided to each unit and can therefore be used to analyze a real-world project. For this chapter, the text editing program "Notepad++" was chosen as a test case [8]. The github page describes the program as follows: "Notepad++ is a free (free as in both "free speech" and "free beer") source code editor and Notepad replacement that supports several programming languages and natural languages." [7]. Notepad++ presents an interesting test case, since it makes heavy use of the "Boost C++ Libraries" [2] and the editing component "Scintilla" [13]. Therefore, analysis carried out on the entire source code also find issues associated with the versions of these libraries used in the project. Notepad++ is authored and maintained by Don Ho [8].

3.1 Analysis Results

The rules presented in Chapter 2 as well as the respective rules for Axivion's *Generic* Stylechecks were implemented and "Notepad++" analyzed. The number of violations, as reported by the rules and subdivided into each subaspect of each rule are presented in Table 3.1. Furthermore, the results were split into the core of "Notepad++", which was programmed by Don Ho, and the Scintilla and Boost `C++` libraries. The table lists the rules developed in this thesis as *AB* (Aiko Bernehed) to discern them from the *Axivion Generic* Stylechecks. Following is an overview over each rule and an interpretation of the analysis results.

Rule	**Notepad++ Core**	**Scintilla**	**Boost**
AB - Busy Headers	34	2	-
Axivion Generic-BusyHeaders	34	2	-
AB - Chained Includes			
redundant headers	171	410	32
more accurate include	59	83	25
may be removed	14	719	24
may be removed, might be used by includers	26	-	44
Total	*270*	*1,212*	*125*
Axivion Generic-LocalInclude			
`#include` moved to other header	20	179	34
`#include` to be removed	28	517	35
redundant `#include`	200	183	30
`#include` replaced by other `#include`	13	11	14
`#include` replaced by forward declaration	11	1	2
unused `#include` that includers may need	16	3	8
add `#include`	71	170	99
add declaration of struct or (template) class	-	34	17
Total	*359*	*1,098*	*239*
AB - Discouraged Content	6	-	-
AB - Include Guards			
`#pragma once` detected	116	-	-
missing include guard	11	-	9
Total	*127*	-	*9*
Axivion Generic-DuplicateIncludeGuard	-	-	-
Axivion Generic-MissingIncludeGuard			
guard does not cover complete file	1	-	1
guard incomplete	-	-	1
missing include guard	11	-	9
Total	*12*	-	*10*
AB - No Absolute Paths	-	-	-

Axivion Generic-NoAbsoluteInclude	-	-	-
AB - No Conditional Preprocessing			
`#if` in header	297	6	324
`#ifdef` in header	74	9	162
`#ifndef` in header	16	4	152
Total	*387*	*19*	*638*
Axivion Generic-NoIfdefInHeader	503	26	739
AB - Primary Includes	1	-	-
Axivion Generic-FileKindDifference	1	-	-
AB - Too Many Includes	-	-	-
Axivion Generic-TooManyIncludes	24	-	-
AB - Wrong Include Casing	71	9	-
Axivion Generic-WrongIncludeCasing	71	9	-
AB - Wrong Include Type			
use Std. Lib. `C++` header instead of `C`	29	641	3
unnecessary (deprecated) `#include`	2	-	-
use `" "` instead of < >	1	-	86
use < > instead of `" "`	1	-	-
Total	*33*	*641*	*89*
Axivion Generic-NoCHeaderInclude	31	641	3
Axivion Generic-IncludeKind			
use `" "` instead of < >	1	-	86
use < > instead of `" "`	1	-	-
Total	*2*	-	*86*
AB Total	**929**	**1,883**	**861**
Axivion Generic Total	**1,037**	**1,776**	**1,077**

Table 3.1: The number of Stylecheck violations for the rules presented in Chapter 2 and Axivion's *Axivion Generic* Stylechecks, separated into the Notepad++ core programming and the Scintilla and Boost libraries.

AB - Busy Headers

The *AB - Busy Headers* rule and *Axivion Generic-BusyHeaders* agree exactly, suggesting that both rules work correctly or have roughly the same bugs. As already pointed out in Section 2.1, the transitive includes calculated for the *Axivion Generic* rule are off by one include in comparison to the rule developed here. In this project, the Scintilla and Boost libraries are notably quite clean, whereas the core program could be redesigned to have fewer transitive includes and direct includers in some files.

AB - Chained Includes

As explained in Section 2.2, it is not known how *Axivion Generic-LocalInclude* iterates through subsequent includes and decides how to assign ambiguous include paths. It is also not clear how the rule decides to implement forward declaration, which `#include` could be replaced how by another `#include`, how or why includes should be added, or how the declaration of structs, classes, and template classes could improve the program.

In direct comparison to the *AB - Chained Includes* rule, *Axivion Generic-LocalInclude* is a lot more granular in its recommendations. The former is a lot more aggressive in suggesting includes to be removed and flagging headers as redundant. This indicates either, that the *Axivion Generic* rule is too careful in determining a better `#include` path, or, more likely, that the *AB - Chained Includes* rule misses tokens in various headers. The rule, as presented in Alg. 2 is rather complex and might miss some of the more obscure directives `C++` supports. Unfortunately, the Axivion Suite is also highly complex in its naming convention of `C++` tokens. In order to confidently ensure that the algorithm works as intended, extensive testing, far surpassing the rudimentary unit tests used in this project, must be employed.

However, both rules show that the majority of style violations to these rules are committed in the Scintilla Library. The library has been under development and maintenance since 1999 [13]. To ensure backwards compatibility, it is likely that new styles of code organization have been added to an old, existing project. This potentially leads to bloated projects with a multitude of unnecessary and redundant `#include` directives, which in turn lead to a large number of style violations in this analysis. Unfortunately, these violations are not trivial. In a small project, such as "Notepad++", this inefficient code organization may be acceptable, but in large codebases compilation time can be increased greatly just by using the Scintilla Library. This might be unacceptable to programming teams, which therefore might opt for alternative solutions.

In contrast, the Boost Library is alright, but could use some overhauling of its `#include` paths. The core programming of "Notepad++" could also use some work, particularly where redundant and unnecessary includes are concerned.

AB - Discouraged Content

Only six style violations in three different files were detected, and these are all part of the core "Notepad++" program. These violations are quickly remedied by moving the respective code snippets to *source* files instead. The Scintilla and Boost libraries exhibit no issues. As mentioned in Section 2.3, there is no counterpart to this rule within the *Axivion Generic* Stylechecks of the Axivion Suite.

AB - Include Guards

Neither the rule presented here nor the *Axivion Generic* equivalent detect any duplicate *include guards*. Furthermore, both rules agree on the amount of missing *include guards* throughout the project. This suggest that both rules were implemented correctly. Furthermore, the *Axivion Generic-MissingIncludeGuard* provides interesting functionality that was not considered in *Include Guards* presented here, i.e. checking for incomplete guards and checking whether an *include guard* covers the entire file. On the other hand, *Include Guard* identified 116 `pragma once` directives, which were not investigated by *Axivion Generic-MissingIncludeGuard*. However, since the latter did not report these 116 *headers* as having a duplicate or missing *include guard*, the rule must be aware of the `#pragma once` directive. An optional reporting mechanism of `#pragma once` for the provided *Axivion Generic-MissingIncludeGuard* might be beneficial for users of Axivion's Software.

AB - No Absolute Paths

It is not surprising that neither this rule nor its *Axivion Generic* counterpart flagged any absolute paths in any headers. As soon as an absolute path is used in a project, it becomes nearly impossible to successfully compile such a program on any other but the original host machine. The project paths would have to be rebuilt exactly as they are programmed into the code. Since "Notepad++" has been in successful deployment for many years and Scintilla and Boost are established `C++` libraries, all mistakes pertaining to absolute paths have already been corrected.

AB - No Conditional Preprocessing

A couple of things are directly obvious when looking at this rule's part of Table 3.1. First of all, the rule *Axivion Generic-NoIfdefInHeader* consistently reports many more `#if`, `#ifdef`, and `#ifndef` than the *AB - No Conditional Preprocessing* rule suggested in section 2.6. Unfortunately, since the *Axivion Generic* rule does not display data for different types of conditional preprocessing tokens, it is not clear what the *Axivion Generic* rule picks up that the *AB* rule misses. Since the *AB* rule passes the rudimentary unit test, the discrepancy could either be due to a bug in the *Axivion Generic* rule, or some unknown type of preprocessing token being handled by the *Axivion Generic* rule that the *AB* rule does not.

Furthermore, most style violations occur in the Boost library. This is surprising, since the library covers a wide range of topics and was designed with eventual integration into the `C++` standard in mind [2]. However, the behavior uncovered by these rules directly contradicts NASA's "10 Rules for Developing Safety Critical Code" [15]. In contrast, the Scintilla library is mostly clean.

Finally, the "Notepad++" core program exhibits too many conditional preprocessing directives for such a, comparatively, small project. It is not clear what all of these directive are intended to provide. The project's download page [8] only has three different software versions (x86, x64 and ARM64) which does not explain why so many different build versions might be necessary to be supported.

AB - Primary Includes

Both, the *AB - Primary Includes* rule and its *Axivion Generic* counterpart only find a single header that is used as a user `"..."` and a system include `<...>`. As mentioned in section 2.7, the *Axivion Generic* rule does not provide both headers, which would be useful for debugging.

AB - Too Many Includes

The logic behind this rule is simply: count all includes in each file, report files with a large number of `#include` directives. However, here the *AB - Too Many Includes* rule developed in this thesis and *Axivion Generic-TooManyIncludes* diverge. At the same time, it is not clear how the *Axivion Generic* rule reaches the conclusion that 24 files in the project have more than 400 includes, the predefined threshold. For instance, `Notepad_plus.cpp` is reported as: "unit includes more than 400 files.", even though a manual count of the file only returns 27 `#include` directives. The rule description reads: "This rule will report compilation units that include more than the configured number of files."

There are two possible ways to explain this behavior. The *Axivion Generic* rule is actually referring to transitive includes, in which case the rule's description is rather poor, or the rule itself is badly implemented and returns false results. Should the former be true, the *Axivion Generic-TooManyIncludes* rule would have very similar behavior to *Axivion Generic-BusyHeaders*, which would make the rule itself redundant. In both cases, the rule needs to be checked and maintained and the description of the rule updated.

AB - Wrong Include Casing

In this case, the *AB* and *Axivion Generic* rules agree perfectly, suggesting consistent behavior. Most violations occur in the core source code, while the libraries seem to be well implemented. This makes intuitive sense, since "Notepad++" was written for case-insensitive Windows systems, whereas Scintilla and Boost are supposed to be available to `C++` programmers on all platforms.

AB - Wrong Include Type

Although a little more convoluted than previously discussed rules, the *AB* and *Axivion Generic* rule here agree perfectly. The use of `C` vs. `C++` Standard Library includes is handled a little differently. The rule presented in section 2.10 also checks whether *header* files might have become unnecessary, which *Axivion Generic-NoCHeaderInclude* does not do. Therefore, there is a slight discrepancy, in this case of 2, of reported `#include` directives that should be exchanged for their `C++` equivalent.

All other metrics match up exactly, which once again speaks to a good implementation of both rules. It is remarkable that the Scintilla Library shows 641 violations pertaining to the use of `C` instead of `C++` Standard Library files. This speaks to the origins of Scintilla, which came out when the `C++98` standard was just established [16]. It also shows how important continuous maintenance and updates to a codebase are.

3.2 Execution Time

Apart from comparing how the rules developed here match up against Axivion's *Axivion Generic* rules, the execution time is of particular interest. If the rules take an extraordinary long time to execute, their value to the programming team might be diminished, since a team might opt not to use the rules regularly due to project time constraints.

The analysis for this work was carried out on an Intel Core i5-8365U CPU with four cores of 1.60 GHz per core with 16 GB of RAM. The resulting runtimes for each rule are shown in Table 3.2. The table also shows the worst-case time complexity constraints as estimated according to the algorithms presented in Chapter 2. The time complexity is presented in terms of the total number of `#include` directives n.

"Notepad++" has roughly 230,000 source lines of code, which puts it on the low-end for software projects. Nevertheless, the code base size is already sufficient to show the impact of the algorithms' time complexity on execution time. As expected, complex algorithms exhibit a higher execution time than less complex algorithms. Furthermore, all *AB* rules, except for *AB - No Conditional Preprocessing*, execute slower than their *Axivion Generic* counterparts. Seeing as most rules are about two seconds slower, this would suggest that the

Rule	Execution Time (s)	Time Complexity
AB - Busy Headers	2.96	$\mathcal{O}(n)$
Axivion Generic-BusyHeaders	0.19	-
AB - Chained Includes	10.60	$\mathcal{O}(n^3)$
Axivion Generic-LocalIncludes	7.92	-
AB - Discouraged Content	0.82	$\mathcal{O}(n)$
AB - Include Guards	2.12	$\mathcal{O}(n^2)$
Axivion Generic-DuplicateIncludeGuard	0.54	-
Axivion Generic-MissingIncludeGuard	0.76	-
AB - No Absolute Paths	0.83	$\mathcal{O}(n)$
Axivion Generic-NoAbsoluteInclude	0.02	-
AB - No Conditional Preprocessing	1.26	$\mathcal{O}(n)$
Axivion Generic-NoIfdefInHeader	4.39	-
AB - Primary Includes	1.66	$\mathcal{O}(n)$
Axivion Generic-FileKindDifference	0.01	-
AB - Too Many Includes	2.02	$\mathcal{O}(n)$
Axivion Generic-TooManyIncludes	0.03	-
AB - Wrong Include Casing	1.01	$\mathcal{O}(n)$
Axivion Generic-WrongIncludeCasing	0.1	-
AB - Wrong Include Type	2.00	$\mathcal{O}(n)$
Axivion Generic-NoCHeaderInclude	0.04	-
Axivion Generic-IncludeKind	0.02	-

Table 3.2: Execution time of all rules used for this project analysis and the corresponding worst-case time complexity estimates for the rules developed in this thesis.

Axivion Generic rules are, unsurprisingly, more deeply integrated into the Axivion framework and may even use a quicker programming language such as `C` or `C++`. One peculiarity should be noted though: Axivion's *Axivion Generic-NoIfdefInHeader* is significantly slower than *AB - No Conditional Preprocessing*, which is supposed to have the same functionality. This points to the *Axivion Generic* rule either working through a lot more possibilities than were considered in the *AB* rules, or to the rule being poorly implemented.

Chapter 4

Conclusion and Future Work

In this work static include analysis schemes using the proprietary Axivion Suite by the Axivion GmbH were investigated. The rules implemented were inspired by previous work carried out in other projects (**Include What You Use**), best practice suggestions by Stroustrup [21] and NASA [15], and the Axivion Suite's own analysis rules. Overall, ten rules were implemented, compared to Axivion's *Generic* rules, the rules implemented on the "Notepad++" code base, and the different rules' execution times analyzed.

In Section 2.1 a rule to identify busy *header* files, i.e. a *header* with a large product of direct includers and transitive includes, was developed. It was shown that the respective Axivion rule, *Axivion Generic-BusyHeaders*, overreports the amount of transitive includes by one, which points to a possible bug in the rule.

In contrast, for the *AB - Chained Includes* rule it was shown in Chapter 3 that there is a large discrepancy in reporting in comparison to *Axivion Generic-LocalInclude*, whereas the latter exhibits a lot more granular reporting. As the *AB - Chained Includes* rule also suggests significantly more `#include` directives for removal than *Axivion Generic-LocalInclude*, it also likely misses several `C++` tokens in its analysis. Furthermore, the rule sometimes suggest the use of `C` instead of `C++` Standard Library *headers*. That is a result of the tokens being defined by the `C` header, but the `C++` header providing further static type safety. The rule needs to consider this kind of dependency. Therefore, the *AB - Chained Includes* rule requires significantly further development and testing on a wide variety of projects for it to reach maturity, much more than could be carried out for this thesis.

Lastly, it was shown with the *AB - Discouraged Content* rule and the `#pragma` implementation in the *AB - Include Guards* rule that the *Generic* rules miss certain functionality that may be of interest for users. As detailed in Chapter 3 in the discussion of the *Axivion Generic-TooManyIncludes* rule, there are also misleading rule descriptions or even bugs in the *Axivion Generic* rules that make the reports of rules confusing. It is suggested here that the user friendliness of all *Axivion Generic* rules is reviewed and updated.

Of further interest are the analysis of the Scintilla and Boost libraries used by "Notepad++". Table 3.1 shows that most errors in the Scintilla library pertain to include organization and the use of `C` instead of `C++` Standard Library Headers. This probably is a results of the Scintilla Library being comparatively old, originally developed in 1999, and therefore still using a lot of the original `C` functionality that hadn't yet been transferred to `C++`. However, in the meantime the `C++` ISO standard has been significantly developed and a redesign of the Scintilla library might be called for.

On the other hand, the Boost library has a declared goal of eventual standardization of the library [2]. This shows in the "false" use of the `<...>` include type within the library as flagged by *AB - Wrong Include Type*. Also, the library has a lot of conditional preprocessing directives, which is in violation of NASA's 10 Rules for Developing Safety Critical Code [15] and should be reconsidered. The Boost Libraries code organization could be amended to provide more accurate includes as well.

Topics of interest but not analyzed here are the static analysis of Precompiled Headers and Templates, as outlined in Section 2.11. If the work carried out in this thesis is further developed, the advantages, drawbacks, and potential analysis of Precompiled Headers might be interesting. Templates, due to their ubiquitous use in the Standard Library and subsequent migration into larger projects need to be investigated for any static include analysis to be complete and should be considered in future work.

Overall, it is clear that static include analysis has extremely important applications for existing projects. The implementation carried out in Chapter 3 shows that the "Notepad++" core implementation and each of the two libraries used have a host of different style issues associated with them. Authors and maintainers of new and existing projects need to be aware of hidden issues within their projects and new developments in the `C++` ISO Standard that may impact their work. Finally, it should be pointed out that the Axivion Suite itself, including the *Axivion Generic* rules, needs to be protected from software erosion.

Bibliography

[1] Axivion Documentation (closed source).

[2] Boost C++ Libraries, available: https://www.boost.org/, accessed: 08-02-2022

[3] Clang 15.0.0 Documentation - Precompiled Header and Module Internals, available: https://clang.llvm.org/docs/PCHInternals.html, accessed: 06-02-2022

[4] cppclean, available: https://github.com/myint/cppclean, accessed: 27-01-2022

[5] GCC Documentation - 3.22 Using Precompiled Headers, available: https://gcc.gnu.org/onlinedocs/gcc/Precompiled-Headers.html, accessed: 06-02-2022

[6] Include What You Use, available: https://include-what-you-use.org/, accessed: 27-01-2022

[7] Notepad++ git repository, available: https://github.com/notepad-plus-plus/notepad-plus-plus, accessed: 08-02-2022

[8] Notepad++ Homepage, available: https://notepad-plus-plus.org/, accessed: 08-02-2022

[9] Official AUTOSAR Guidelines, available: https://www.autosar.org/, accessed: 01-02-2022

[10] Official MISRA Guidelines, available: https://www.misra.org.uk/, accessed: 01-02-2022

[11] Adjust case sensitivity, available: https://docs.microsoft.com/en-us/windows/wsl/case-sensitivity, accessed: 06-02-2022

[12] Microsoft C, C++, and Assembler Documentation - Precompiled Header Files, available: https://docs.microsoft.com/en-us/cpp/build/creating-precompiled-header-files?view=msvc-170, accessed: 06-02-2022

[13] Scintilla - A free source code editing component for Win32, GTK, and OS X, available: https://www.scintilla.org/, accessed: 08-02-2022

[14] A. Mallia and F. Zoffoli, *C++ Fundamentals*, Packt Publishing, Mar. 2019, available: https://www.ebook.de/de/product/36449192/antonio_mallia_francesco_zoffoli_c_fundamentals.html

[15] G. J. Holzmann, *The Power of 10: Rules for Developing Safety-Critical Code*, NASA/JPL Laboratory for Reliable Software, 2006, available: http://pixelscommander.com/wp-content/uploads/2014/12/P10.pdf

[16] ISO/IEC 14882:1998 - Programming Languages — C++, International Organization for Standardization, 1998, available: https://www.iso.org/standard/25845.html

[17] ISO/IEC 14882:2020 - Programming Languages — C++. International Organization for Standardization, 2020, available: https://www.iso.org/standard/79358.html

[18] R. Koschke, *Zehn Jahre WSR - Zwölf Jahre Bauhaus*, Workshop Software Reengineering (2008) pp. 51–65.

[19] S. Pramanick, *History of C++*, available: https://www.geeksforgeeks.org/history-of-c/, accessed: 26-01-2022

[20] A. Raza, G. Vogel, and E. Plödereder, *Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering*, Reliable Software Technologies – Ada-Europe 2006, Springer Berlin Heidelberg (2006) pp. 71–82.

[21] B. Stroustrup, *The C++ Programming Language*, Addison Wesley, 2013, available: https://www.ebook.de/de/product/19365406/bjarne_stroustrup_the_c_programming_language.html

[22] TIOBE Software BV, *Tiobe Programming Language Index*, available: https://www.tiobe.com/tiobe-index/, accessed: 26-01-2022

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Clark, CO, 11. Februar 2022 Aiko Bernehed, M. Sc.