



Eötvös Loránd University

Faculty of Informatics

Department of
Programming Languages
and Compilers

Towards Modularisation of C++ Software Systems: Semi-Automatic Black-Box Module Organisation

Zoltán Porkoláb
associate professor, PhD

Richárd Szalay
computer science MSc, 2nd year

Supported by the ÚNKP-18-2 “New National Excellence Program” of the Ministry
of Human Capacities.

Budapest, 2019

Abstract

The componential design of software projects is a desired property both for development, ease of code analysis, and optimising compiler performance. Programming languages and paradigms have long allowed componential development (e.g. Ada and Java packages, Python modules), however, other widely adopted languages, especially C and C++, had stuck to the “translation unit” model where every source file is an individual input with no knowledge of other sources except those explicitly expressed via headers. The Modules system of C++ draws from the success of other languages’ packaging concepts and is expected to allow cleaner encapsulation of concern and ease of performance requirements on the compiler. However, for existing systems, Modules brings issues with trying to move away from the traditional “translation unit” and “small source files” approach. In this work, we investigate the effort of a (semi-)automatic, black-box modularisation of C++ projects. As we experienced, the properties of existing software systems that were inherited from the manner programming had to be done for decades in C and C++ makes it infeasible to move already existing code to Modules mostly due to coupling issues, and implementing real transition would require significant redesign of both inner and user-facing interfaces of libraries.

Chapter 1

Introduction

1.1 Motivation

Good composability and organisation of software projects is a desired property for software systems to ensure that both development effort and program performance is close to optimal. Various programming languages and paradigms over the years have supported different approaches for developers to organise their code in some logical fashion. Developer rules of thumb have also evolved on top of the rules expected by language specifications and tooling expectancies. The logical organisation of code might at first glance seem to offer no tangible difference on the actual runtime performance as compiler optimisation techniques have evolved greatly, although this is not the case especially for C++. Compared to the comfortable packaging solutions offered by most modern programming languages, C++ was a long way coming in the introduction of such a solution.

Compiling large-scale C++ software projects is a computationally intensive task due to the nature of separate compilation where each source file containing some code is compiled independently from one another. The requirement for type safety is satisfied through the use of special C++ codes called *header files* which get copied and pasted into every source file in which types and functions are used from that particular header. There are multiple issues with the usage of this – from the point of view of many programmers and other programming languages, legacy – technique, which includes both design and software technology problems and compiler performance issues. A solution to these issues emerged with the *Modules* feature accepted into the current newest version of C++, C++ 20. However, there are multi-million lines of code in C++ software systems which might be considered legacy with the adoption of this feature, even though there are no direct aims of deprecating the classical translation unit-based approach. The question of lifting projects to the new world of proper logical packages of code naturally arises from the trend of the

language’s evolution. We have investigated the history and the current proposal for *Modules*, along with what prototype implementations were made by compiler developers.

We have devised a method that attempts to further ease this possible transitioning of software projects that work by analysing the inner dependencies and the code’s structure. This method does not use domain knowledge on software projects’ behaviour. Our method is given a formal description and an experimental implementation, which is useful for identifying pain points inside the projects’ structure which might prevent useful automatic modularisation, and also generates a modularisation, which is a clustering of translation units into unity build-like compilation groups. We have performed an empirical study on various open-source projects.

The rest of the thesis is organised as follows: Logical code organisation solutions in some influential programming languages are discussed in Section 1.2 along with approaches directed specifically at improving compilation of C++. The current compilation model of C++ is presented in Chapter 2, with also discussing the issues with the current model. C++ *Modules* is detailed in Chapter 3. Our approach to semi-automatic modularisation is proposed in Chapter 4. The characteristics of our method and empirical results are presented in Chapter 5. The thesis concludes in Chapter 6 with the discussion of future approaches.

1.2 Related Work

Various programming languages – e.g. Java, Ada, Python – have been designed with the ability to organise code into logical packages, while some which originally had no concept as such following suit in a later version, most notably Fortran’s major release dubbed *Fortran ’90* [11].

In Java, *packages* are logical groups of source files that are somewhat logically related together [32]. A package can be created by putting the source files for the classes in the package into a folder and adding the statement `package org.example.package_name;` to the beginning of the source files. Name resolution is performed with the package name in mind for not fully qualified names, both at compilation and at run time: the root folders for the package directories can be changed via the `CLASSPATH` configuration variable. Using a symbol from a package either through fully qualified name or via an `import` statement will instruct the bytecode compiler to search for symbols in the hierarchy specified by the package name. A special semantic rule called *package-private visibility* exists for class members which specify that the member might only be accessed from any other symbol in the same package. In this sense, packages in Java combine a logical code organisation method and a namespacing capability.

Ada supports the notion of *packages* too, where a package is defined by two files: a package specification (interface) and package body (implementation) file [20]. The package specification is responsible for describing the contract of the package, which may contain **private** types, constants. A private type is like a *forward declaration* in C++ – see Section 2.3 – naming that the type in the package exists but does not allow client code to call operations on the type. The package’s body contains the implementation code from which the executable is created. Usage of the package – with the **with** statement – introduces the symbols defined in the package’s interface or contract to the client code, and it is up to the build system’s configuration to ensure linkage, as Ada is also a language with *separate compilation*. The **with** statement is analogous to the **#include** directive – see Section 2. Ada’s approach to packaging code is similar to Java’s in terms of encapsulating code and providing namespacing, with the added constraint on having to put every symbol in a single pair of files.

In Python, *packages* are groups of *modules*, where “module” stands for a single Python source file – contrary to the definition of *Modules* used in this dissertation, see Section 3. A Python module is a single source file that can contain an arbitrary number of definitions interleaved with code, and packages are represented as directories in the file system [43]. The **from package_name import module as alias** statement is used in Python to load external code into the interpreter, where packages might be referred by absolute name or via relative name. At the encounter of such statements the interpreter will check if the module from the package is loaded already, and if not, will search for the module’s code and execute the found code including executing the lines in the “global” part of the script. The search for modules belonging to a package to be imported starts from a configuration variable called `sys.PYTHONPATH`, in a similar manner to Java’s behaviour with `CLASSPATH`. While partial imports from a package or even partial imports of a program unit from a module are possible in Python, this is only partial in the sense that the name is hidden from the client module. Python packages form part of the qualified name of a symbol only through the way the interpreter is instructed to find the implementation, and due to the **as** keyword, the user can rename any symbol to be usable under an alias in the local code.

Various other programming languages also support a package manager or code organisation effort through similar means, either offering namespacing or even distribution of code on top of such.

There is no way to organise source code into logical groups supported by the language itself in C, and up until the introduction of *Modules*, neither was one in C++.

Also, the compilation model of C++ causes developers to face issues which are

prevented by design in other, usually younger languages. One such is the possible subtle bleed of tokens between source files. Examples to this are given in Section 2.1. The author of C++, Bjarne Stroustrup has proposed a preprocessor scoping mechanism in 2004 in [34]. Stroustrup’s work recognises the issues that developers have commonly faced with the inability to prevent accidents related to macro use. The suggestion was a new preprocessor directive pair, `#scope` and `#endscope` which may constrict the lifetime of preprocessor macro definitions. At the encounter of a `#scope` directive the preprocessor would isolate and ignore previously defined macros, thus creating a “scope”. A scope lasts until `#endscope` is found in the input buffer at which point the macros defined inside the scope will be undefined and everything returns to the preprocessor state before the `#scope` directive. Scopes were suggested to be nestable in each other.

The beginning of a new scope hides every macro before it, which might not be beneficial for some of the headers which, for example, perform conditional compilation based on architecture. This is “fixed” by introducing another two new preprocessor directives, `#import` and `#export`, which turn off the behaviour of `#scope` and `#endscope`, respectively, for the macro named in the directives. A macro explicitly imported into a scope will still be defined, and a macro explicitly exported will not be undefined at the end of the scope. While these fixes seemed to cater to the library developers and ensure expressibility of the preprocessor doesn’t decrease for real-world code, the hidden behavioural change possibilities introduced by the `#export` directive was under scrutiny. This proposal was discussed briefly by the C++ community with counter-arguments expressed by Standards Committee members and submitters and even by the original proposer himself. Due to the arguments against and the lack of clear answer or interest, no implementation support was introduced for this proposed feature.

The architectural decay and course of development of long-living software systems have been studied before by Yu, Homayoun, et. al [47] by introducing the concept of *false dependencies* between files or program entities. A false dependency is created between two files when a symbol declared in a header is in the input buffer – see Section 2 – but is not referenced by any use site. The underlying approach behind program code or symbol dependency graphs stem from similar concepts introduced in our work in Section 4.1. Their approach is to restructure the layout of the project by organising symbols into different files than originally to eliminate the false dependencies as much as possible. This has proven in their experiments to reduce Vim’s build size by 90% for incremental and by 30% for clean builds, along with 33% build time reduction in the incremental case. The researchers’ work was designed and proven for ANSI C, however, it could be applied to C++ with few modifications – most necessarily due to templates. Their algorithm outputs

new automatically generated headers and restructured `#include` directives, which might be committed with the source code or the algorithm can be run before every build to “fix up” the structure of the project, as the authors agree that automatic partitioning might not result in clusters which are sane from a code organisation, design, ownership or team responsibility viewpoint just because it is more optimal for compiler performance.

Similar to one half of the approach in [47] is implemented in the prototype tool *Include-What-You-Use* [40] first created by Google, which attempts to reduce the set of headers included in the translation unit. The tool has been criticised for its lack of precision and needs for heavy manual configuration for most projects.

The authors of the paper discussed above further continued their work in [4], where they investigated the deterioration of build system architecture through project evolution, as changes to a software project seems to naturally cause an increase in component inter-dependencies and abolishment of organisational principles that were set when the project was conceived. The authors’ goal was conceptually similar to ours in semi-automatically generating component hierarchies. The motivation behind it was to lessen the coupling between identifiable components of a C/C++ software project, with the primary aim of reducing compilation time requirements. Their approach was to try to synthesise a model from the program files and then try to transform the model – and thus, the component layout – into a different approach, either automatically or by hand. For the Vim [41] editor, this resulted in a *Model-View-Controller* architecture emerging. The solution was creating `make`-transparent new header files which group the component interfaces in a way that the dependencies – and thus, the load on the preprocessor and the compiler, see Section 2 – is minimised. The key difference between their work and our discussion is that while automatically generated headers and clustered groups of files are capable of reducing build time and the transformation of the project might be executed before every major compilation, transforming a project to a new approach of organising code due to a new language feature is a change that will be added to the project’s living history forever. On Vim, the authors found 10 times speedup in incremental, and 2 times increase in the fresh build case.

Various other techniques, implementations, research and proposals have been made that aimed at discussing or attempting to solve the issue with large-scale project compilation, including disk or network storage template repositories [28], storing parts of compilation in memory during a build [1], *precompiled headers* [21] saving entire subtrees generated from headers, with tools that allow generation of such transparently to the build system through compiler modifications [18]. *Unity build* has been investigated in [26, 27] to be a valuable solution barring some difficulties to ensure compilation generates valid code in case pure compiler performance

is needed. These works are discussed in detail with the reasoning behind the issues also given in Section 2.2.

Chapter 2

The Classical C++ Compilation Model

Understanding the effort of automatic modularisation first requires a thorough definition of how dependencies are formed during compilation for programs written in the C/C++ language family. Almost all projects are broken up to individual source files on which the compiler is invoked separately. Invocations of the compiler are handled by some *build system* – such as **make**, configured by higher-level abstractions tools **autoconf** or *CMake* [24] – which ensure that compilation of the project is run in order, and that source files are recompiled when necessary, e.g. due to changes.

Source files may contain preprocessor directives such as *#define* and *#ifdef* etc. for *conditional compilation*, and *#include* which brings in the text of another file into the current compilation’s input. These files are commonly known as *headers* and are used, amongst many other reasons, to ensure that every file of the project follows the same static semantics, symbol table, evaluate the name lookup results, etc. Under the concept of *separate translation*, each compiler invocation is independent of each other and has no knowledge of what transcends during the compilation of other files. When an *#include* directive is encountered, the *preprocessor* performs a lookup in directories configured by the build system and the encountered headers are also preprocessed transitively until no preprocessor directives are left in the input.

The resulting wholly preprocessed textual input buffer – called a **translation unit** – serves as the input for the compiler’s syntactical and semantic analyses. After semantic analysis succeeds, binary object code is generated. These *object files* are then joined and potentially ordered into binaries by the *linker*. Language rules and semantic checks ensure at multiple stages that the resulting program is sensible. The most important of these rules is the *One Definition Rule* (ODR) [13] which dictates that in every translation unit composing a particular binary exactly one definition must exist for each used symbol, e.g. called functions, accessed variables. Exceptions

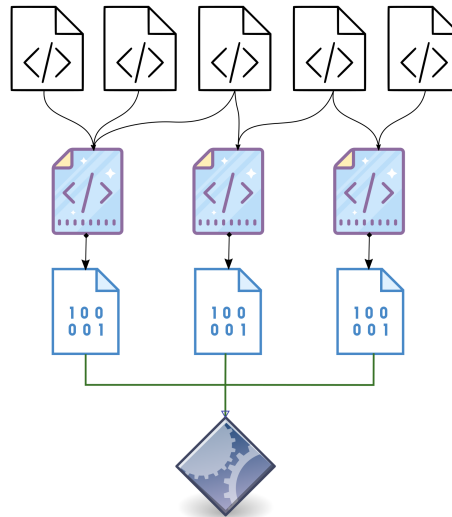


Figure 2.1: Overview of the compilation model for C/C++ software projects. The first row consists of *header files* which are – often transitively through other headers – included into source files. The source files together with their includes form an *input buffer* (second row) which is compiled into *object code* (third row). The object code files are joined and organised by the linker to produce an *executable binary* (fourth row).

to this rule are made for **inline** symbols, **templates**, and in general any record type: multiple definitions may exist, but they must be the same. One practical tool for *ODR* is *name mangling*: individual object code of functions are labelled with a name that is automatically generated by the compiler using a rule-set that varies between compilers but generally contain information about the functions’ name, scope and parameters. These *mangled names* are used by the linker to find where to jump the execution in case a function is called.

2.1 Software technology issues with the translation unit-based model

There are no restrictions on *what code* might appear in a source file or a header as long as it is semantically valid C++ code. Due to this, it is entirely up to the developers on how they organise their systems in the source code level, resulting in great variety and conflicting “best practises” on nomenclature and code organisation. Although such “rules of thumb” exists, there are issues which can lie dormant in a project and become additional pain points at a later development or refactoring point.

lib.hpp <pre> <i>#ifndef LIB_HPP</i> <i>#define LIB_HPP</i> <i>/* Licensing date of project */</i> <i>#define APP_DATE "2017. Oct. 20."</i> <i>const char*</i> LicenseStartDate(); <i>#endif</i></pre>	header.hpp <pre> <i>#ifndef HEADER_HPP</i> <i>#define HEADER_HPP</i> <i>/* Will expand to build date */</i> <i>#define APP_DATE __DATE__</i> <i>#endif</i></pre>
lib.cpp <pre> <i>#include "lib.hpp"</i> <i>const char*</i> LicenseStartDate() { <i>return</i> APP_DATE; }</pre>	main.cpp <pre> <i>#include <iostream></i> <i>#include "header.hpp"</i> <i>int</i> main() { <i>std::cout</i> << APP_DATE; }</pre>

Listing 1: A project consisting of two translation units with a header related to each. Compiled separately, the library and the user-facing binary does what is expected: one prints the build date to the standard output, while the library user can obtain the licensing date string.

2.1.1 Token leak in the single buffer

The first such issue is that tokens, specifically *preprocessor directives* leak as the compilation is done on a singular input buffer. This can cause problems with composability of software systems – program units that are producing a particular behaviour may subtly break when joined together.

Listing 1 shows a minimal example for such an occasion. The macro symbol `APP_DATE` is used in both translation units but with a different meaning. The project in its current state works and does what is expected, however, if later the main binary’s developers decide they wish to call `LicenseStartDate` to print the licensing information, a dependency on the header where this function is declared is created, and the developer will perhaps write the code shown in Listing 2.

This kind of error due to a string being type-safe with another string is a problem because depending on compiler and build tool settings, there may or may not be a *warning* diagnostic emitted. Another case would be that a previously defined function’s *name* is defined to a macro with different meaning but without violation of the type system – this will result in no diagnostics.

main.cpp

```
#include <iostream>
#include "header.hpp"
#include "lib.hpp"

int main()
{
    std::cout << "My App built at " << APP_DATE;
    std::cout << "\n Licensing FancyLib purchased "
                << LicenseStartDate();
}
```

Listing 2: (Compare with Listing 1) The new user-facing *main* binary is subtly broken: both dates printed will be the same. Due to `lib.hpp` being included to *main*’s buffer *after* `header.hpp`, `APP_DATE` will be defined to its string in `lib.hpp` which is the licensing date.

```
template <class T>
class X
{
    /* ... */
private:
    T example(T t);
};
```

Listing 3: Example template class usually defined in a header file which contains some **private** function. This function may only be called from other functions of the same class.

2.1.2 Inability to hide details due to symbol leak

Another issue with the translation unit semantics is that there isn’t an always usable way to hide implementation detail symbols. Every symbol that is declared in the input buffer of the compilation is seen by the compiler and by developer and analysis tools and thus can be used assuming that the semantic rules of the languages are adhered to. Conventions of hiding library details exist – such as enclosing the symbol in the anonymous **namespace** or declaring it **static**. These semantic markers set the symbol’s *linkage* to *internal* – to the translation unit it is defined in –, which makes other translations unable to *name* the symbol and thus unable to call it, while also enabling certain optimisations [2]. These tools may only be used on non-generic functions and only for symbols that are implementation details and are compiled to a library.

Template code is often defined in *headers* and code generation is performed on their bodies in every translation. Consider the example in Listing 3. Referring to an instance of this generic type’s function – e.g. `X<int>{}::example(1)` – refers

to a *valid* function name and the inability to call the function is only produced as a semantic rule violation due to the `private` visibility specifier. Previous research in [25] has shown that clever abuse of the language rules may still allow calling of such functions. Also, there isn't any option to control the visibility of free functions.

2.2 Compiler performance issues

Various steps taken by the compiler during its work has been emphasised as a major bottleneck when it comes to fast (re-)compilation of C++ software systems. First, compiler processes running independently from each other means that every disk operation of opening the headers used in preprocessing happens has to be done for each process. Preprocessor inclusion of dependency headers are a huge impact on the amount of code that might be useless to recompile: it has been discussed in [26, 27] that performing *unity build* could reduce the total *input buffer* lines of code of a project up to 10% of the *LOC* of separate compilation.

A larger input buffer implies more semantic analysis that has to be done for each compilation [16]. Indeed, *template instantiations* has been identified as a major consumer of processing power, with multiple instances of proposed solutions such as non-volatile caches like *template repositories* [29] – which store instantiated generic code in a local folder or a network-accessible storage – and volatile [1] caches like *ZapCC* – which caches the template code in memory during a single build system run.

A large chunk of the generated template code is superfluous work done by the compiler as template function bodies are emitted as *weak references*. These symbols are linked in a fashion that the first occurrence – in some unspecified order, however, the *One Definition Rule* requires the versions to be the same – is kept in the resulting binary, with all other implementations thrown away.

The issue with the increased debt on compiler performance with the project's growth is the continuous waste of resources lessening the time efficiency of the development process, while also being potentially detrimental to computer efficiency [8].

Various tools aiming to speed up compilation has been made as part of compiler feature extensions or build system additions, such as precompiled headers.

2.2.1 Precompiled headers

Precompiled headers (*PCH*) [21] is a non-standardised compilation performance optimisation technique that was first introduced in the *Microsoft Visual-C++ Compiler* in 1998. The goal is to ease the aforementioned issues by storing the internal representation of the input buffer for a header, thus allowing client code to not re-run

example.cpp

```
#include "myheader.h"
#include <iostream>

void example() { /* ... */ }
```

Shell

```
# Pre-compile "myheader":
clang++ -xc++-header myheader.h -o myheader.h.pch

# Compile "example.cpp" with using the precompiled header:
# (If 'header.h.pch' exists, it will be used,
# if not, header is loaded as text.)
clang++ -include header.h example.cpp -o example.out
```

Listing 4: Compiling `example.cpp` can be executed faster if the contents of `header.h` is unchanged by creating a *Precompiled Header* from the include. However, in this setup, the contents of `iostream` will always have to be loaded textually and parsed.

the semantic analysis for the code body. “Compiled” can be a misleading term as, depending on the compiler, actual code generation might not even take place – e.g. in Clang, precompiled headers store the *Abstract Syntax Tree* (AST), Clang’s internal code representation in a binary dump format. When used to compile a translation unit, a header can be *prefixed* to the compilation, and in case the prefixed header has an associated `.pch` dump next to it generated, the binary representation will be used. An example invocation of compilation is shown in Listing 4.

Every compiler invocation of MSVC, GNU/GCC and Clang may have up to 1 precompiled header argument in its configuration. This includes compiler invocations whose output is a precompiled header itself: precompiled headers may be chained together to form larger units. Due to this feature being a pure compiler enhancement and not regulated by the Language Specification, it is implementation-defined how these chains are formed – e.g. in GCC, the larger *PCHs* are emitted as full bodies, while Clang emits a backwards link into the file that depends on another unit. Chaining of two precompiled headers is depicted in Figure 3.1.

Two issues arise from the usage of precompiled headers. While usage does not require changing the source code of the software project, it is a build system requirement and maintenance burden to ensure that dependencies of code bodies are recompiled properly if the developer changes the code. The other issue with this technique is the concern of *granularity*. Because precompiled header chains form a directed acyclic graph, and only one precompiled header is allowed per compilation, it is an optimisation problem to select the granularity that might be appropriate from client translation units’ perspective.

Krishnaswamy [18] introduced a solution to the granularity problem of pre-compiled headers. Their work in the **aCC** compiler of Hewlett-Packard involved a **Makefile**-transparent automatic precompiled header generation, which created a single precompiled header for each compilation in the project. This was done by selecting the “preamble” of the source file, which was from the beginning up until the first non-comment non-preprocessor token – the header inclusion part of a typical C/C++ source file –, and creating a precompiled header from the symbols and transitively included headers found. This allowed for easy and quick recompilation in case only the “body” of the source code changed, while also automatically tracking and rebuilding the precompiled preamble of source codes when included headers change. The average reduction in compile time in their study was near 50%.

Precompiled headers is not a solution for the aforementioned *template instantiation* problem, as “compilation” of a header file does not know what template specialisations will be required by client code.

2.3 Interface and Implementation Dependencies

Due to *separate translation* and the step of linking in the process of building C++ software systems, dependencies spanned between program units can be defined on multiple layers: one for the *interface* and one for the *implementation*. It must be noted that these layers are different from the conventions of *Application Programming Interface* and *Application Binary Interface* [48] as we only consider parts of the same project’s files.

This distinction on the scope of dependency is necessary from the programmer’s point-of-view for separation of concern and quicker compilation of client code. Perhaps the best-known case of this is the *p_impl Idiom* or “*Opaque Pointer*” [35] which allows decoupling client code from the inner layout of a type at the expense of a pointer indirection at call sites — the client only has interface dependency to a wrapper to the actually executed third-party library instructions.

Remark 1. In the definitions below, a **program unit** is a reasonably sized block of code that can be named via a symbol. A single statement or a group of compound statements is not a program unit, whereas e.g. a function or a class’ definition is one. We expect and only consider codes a *program unit* if they are at least syntactically valid.

Definition 1. The **interface dependency** D_{if} of a program unit $\mathcal{S}$ is the smallest set of program units that are required to be known to validate the static semantics of $\mathcal{S}$.

For any function prototype $T\ f(U\ u, V\ v, \dots, Z\ z);$, the interface dependency consists of type **names** $T, U, V, \dots, Z$, and their interface dependencies transitively.

Definition 2. The **implementation dependency** (D_{imp}) of a program unit $\mathcal{S}$ is the smallest set of program units that are required to be known to generate the executable for or usages of $\mathcal{S}$.

Corollary 1. If the interface of a program unit is semantically invalid, the program unit is *ill-formed*, and thus code generation should not be attempted. Thus, $D_{\text{if}} \subseteq D_{\text{imp}}$.

Program units are closed to the operation of set union and are monotone: the dependency set of the union is created by forming the union of dependency sets.

The program unit dependency sets above are defined within the scope of a single translation. A third dependency set could be defined for the entirety of the project in case binary generation and run-time is concerned:

Remark 2. The **linkage dependency** D_{ld} of a symbol in the object code is the set of other named symbols which need to be resolved either by the static or dynamic linker so that the binary is well-formed and executable.

Third-party non-template libraries are often provided in the form of some headers and a shared library: the headers are used so compilation of client code can validate against its implementation dependencies, and the shared library is searched and used by the dynamic linker to load the linkage dependency symbols' machine code into memory at run time.

Certain language constructs of C++ compile in a way that the dependency sets are not trivial at first glance:

2.3.1 Forward declarations

Different level of detail is required to be known for types depending on the context they are used in. The smallest of such detail is only a (*forward*) *declaration*, where the name – and for function *prototypes*, their signature – is introduced in the source code. In case of non-**inline** and non-**static** functions, declaring the prototype – the interface – allows the function to be called and postpones finding object code for the function on the linker.

For types, a forward declaration only introduces the type's signature: its name, and in case of templates, the number of template type-arguments into the symbol table. This allows for a limited set of operations which include defining a pointer or a reference variable to said type, using the type as a template argument to a pointer or reference to a template type, and naming the type in a signature. No operation

which requires knowledge of the type’s members or inheritance hierarchy is allowed on forward declared types.

2.3.2 Templates

Due to *Two-Phase Lookup* [44] in case of template instantiations, the interface and implementation dependency sets for templates are skewed:

- *Interface dependency* for a generic template consists only of inner references to the type parameters, and the possible *concepts* – type parameter bounds, another feature to be introduced in C++20 – that constrict what type parameters might be used. For example, a `template <class T>` function must not accept a variable of (undefined) type U as an argument.
- *Implementation dependency* and *linkage dependency* for a generic template are not defined.

Templates are only checked for syntax when the generic code is encountered, and semantic checks are only performed at the *point-of-instantiation*, at which the specialised template – where the type arguments are replaced by concrete types – can have its interface and implementation dependency calculated.

2.3.3 Example

Listing 5 shows an example small C++ project consisting of two files. Building this project requires the *math* library (for `fabs`), and object code for `create_arc_from` to be available. In this example, the compilation of `main.cpp` requires no knowledge on how `struct arc_data` is laid out, only that it exists – this usage of a *forwarding header* is a common pattern to achieve this. Due to passing concrete instances of `tuple` and `pair`, these template types need to be defined for the function’s body, which is done through the inclusion of the `<utility>` header and *template instantiation*.

The lengthy chain of checking whether the dependency sets are satisfied is done by the compiler at semantic analysis. We only show some example calculation steps below. In these examples, $\mathcal{S}_{decl}(T)$ indicates that the type name T has to be **declared**, while $\mathcal{S}_{def}(T)$ indicates that T must be a fully **defined** type. $T<_, _>$ indicates a generic type T with two unbound type parameters – these correspond to the unspecialised template.

arc_forward.h

```
#ifndef ARC_FORWARD_H_
#define ARC_FORWARD_H_
struct arc_data;
arc_data * create_arc_from(float, float);
#endif // ARC_FORWARD_H_
```

main.cpp

```
#include <cmath>
#include <iostream>
#include <tuple>
#include <utility>
#include "arc_forward.h"

std::tuple<float, float, arc_data *>
envelope_arc_at_point(std::pair<float, float> point,
                     arc_data * arc)
{
    return {point.first, point.second, arc};
}

int main()
{
    float x, y;
    std::cin >> x >> y;

    foo(envelope_arc_at_point(
        std::make_pair(fabs(x), fabs(y)),
        create_arc_from(x, y)
    ));
}
```

Listing 5: An example project showing usage of forward declarations included to a translation unit, separating interface, implementation and linkage dependency sets.

$$\begin{aligned}
D_{\text{if}}(\text{envelope...}) &= D_{\text{if}}(\{\mathcal{S}_{\text{decl}}(\text{struct pair}\langle\text{float}, \text{float}\rangle), \\
&\quad \mathcal{S}_{\text{decl}}(\text{struct tuple}\langle\text{float}, \text{float}, \text{arc_data *}\rangle), \\
&\quad \mathcal{S}_{\text{decl}}(\text{arc_data *})\}) = \\
&= \{\mathcal{S}_{\text{decl}}(\text{struct tuple}\langle_ , _ , _ \rangle), \mathcal{S}_{\text{decl}}(\text{float}), \\
&\quad \mathcal{S}_{\text{decl}}(\text{struct arc_data}), \mathcal{S}_{\text{decl}}(\text{struct pair}\langle_ , _ \rangle)\}
\end{aligned}$$

$$\begin{aligned}
D_{\text{imp}}(\text{envelope...}) &= D_{\text{if}}(\text{envelope...}) \bigcup \\
&\quad \bigcup \{\mathcal{S}_{\text{def}}(\text{float}), \mathcal{S}_{\text{def}}(\text{struct pair}\langle_ , _ \rangle::\text{first}), \\
&\quad \mathcal{S}_{\text{def}}(\text{struct pair}\langle_ , _ \rangle::\text{second}), \\
&\quad D_{\text{if}}(\text{constructor for tuple}), \\
&\quad D_{\text{if}}(\text{pass-by-value copy constructor for pair}), \dots\}
\end{aligned}$$

The interface dependency for function `main` is trivial, it only requires `int` to be available – which always is, due to `int` being a keyword referring a *fundamental type*. The implementation (body) of `main` depends on a lengthy list of types and interface of operators – the implementation of the latter will become *linkage dependencies*, such is the case for `fabs` and `create_arc_from`. Function `envelope_arc_at_point` is resolved immediately, and maybe even *inlined* (a form of optimisation), due to being in the same translation unit. However, the code as it is does not compile: $D_{\text{imp}}(\text{main})$ contains `function foo;`. Other programming languages, most notably mainly functional languages such as Haskell or Clean [30] deduce the interface for the `foo` missing (it is an overload of `foo` which takes a `tuple<float, float, arc_data *>` argument by value), however, C++’s static semantics and binding rules are much more strict: the lack of `foo` being declared results in a compilation error.

Chapter 3

C++ Modules

Various programming languages – e.g. Java, Ada, Python – have been designed with the ability to organise code into logical packages, while some which originally had no concept as such following suit in a later version, most notably Fortran’s major release dubbed *Fortran ’90* [11]. The issue with preprocessor token leak had solutions proposed to it in 2004 by the creator of C++, Bjarne Stroustrup [34]. A considerable effort of potential implementations, conference talks and standard proposals followed parallel to the evolution of the language and software development world, most notably the work of Gabriel Dos Reis et al. [5]. *Modules* has been discussed and revised with the semi-final evolution state discussed in [33], and the feature has officially been accepted into the upcoming *C++20* edition of the Standard.

The aim of *Modules* is to solve some of the concerns outlined in Section 2.1 regarding scoping rules and hiding of symbols, while also enabling C++ software developers to better organise their code, mostly attempting to eliminate the reliance on the preprocessor and the maintenance burden of knowing what headers are necessary [40] for the current source file.

A **module unit** is a *translation unit* or a group of translation units – called *module partition* – that contains a logical package of symbols under a custom name. Every module unit has at least a single source translation unit which comprises the module’s interface, listing the symbols that are **exported**. Implementation details private and internal to the module can be compiled in separate translations – compiler invocations – and splitting up the module to multiple files are allowed to help resolve cyclic dependencies, however, these *fragment files* may only be referred to in the file that describes the “*primary module interface unit*”.

Compare the example shown in Listing 6 with Listing 3. The cardinal difference between **private** and not **exported** is that semantic checks are performed on a different level: **private** symbols exist but are hidden, while symbols without the **export** specifier should be regarded in client code as if the symbol was **never** defined. The *One Definition Rule* has also been changed with Modules being accepted

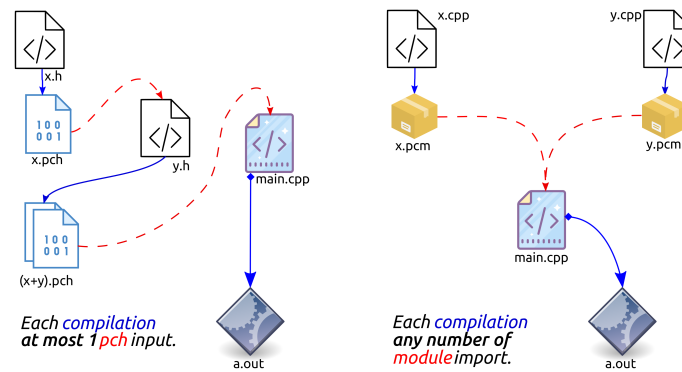


Figure 3.1: *Precompiled headers* allow using some internal representation of a header’s code in a compilation instead of loading the text and performing semantic analysis on it. In a compilation chain, precompiled headers must also be chained. Compared to it, *Modules* allows the creation of a logical package from a C++ source file or multiple files and then importing their contents to a translation unit. Modules may act as if the chains of precompiled headers were changed to a directed acyclic graph. **Red dashed lines** indicate effects of precompiled header usage and **import module** statements, while **solid blue lines** indicate compilation outputs.

```
my_module.cppm
```

```
export module MyModule; /* 'export module' defines that this file is
                        * the primary module interface file of the
                        * module. */
```

```
// Internal detail function.
int four() { return 4; }
```

```
// Publicly exported function.
export int six() { return four() + 2; }
```

```
main.cpp
```

```
import MyModule;

int main()
{
    int s = six();
    int f = four(); // <- Compile error:
                   // no function named four().
}
```

Listing 6: The “Hello World!” example of C++ modules. The `MyModule` module only exports the function `six()`. Thus, calling `four` in `main` is not valid semantically, as the symbol name outside the module does not exist.

into the language specification: symbols defined with the same name and type *may* collide in the client code – be defined multiple times to conflicting definitions across different imported modules – as long as exactly one of the definitions is declared **exported**.

The highlights of *Modules* are that better compartmentalisation of software projects can be achieved. The standard proposal text leaves it up to the compiler and build tooling developers what exactly to do at an **import** statement – in terms of finding files and how exactly to “load” symbols –, but generally **import M**; means that the *translation unit* of **M** is loaded into the current context. The exact source code for **M**’s body need not be available.

Modules has been criticised for various drawbacks, however. First, it can turn out that compilation of a module requires more source code to be present and results in more downstream dependencies to be recompiled when working on a library compared to code written with the traditional approach, which is particularly problematic in distributed build environments [17]. A quote from the talk in [17] reads at 38:50 minutes: “*The only solution here we can think of is actually to make people split up libraries.*”.

While Modules was designed with peaceful coexistence with “old-style” **#include**-based code in mind, the inertia – or the outright technical or legal impossibility – of changing large existing codebases to a whole new approach might make it so existing projects stay with the traditional approach, and only new projects are developed with Modules in mind. Cases of language evolution like this have the tendency of resulting in a seemingly permanent divide, as is the case with Python 2 vs. Python 3 [23].

Chapter 4

Semi-Automatic Modularisation

In this section, we propose and discuss a process which performs a semi-automatic black-box modularisation of C++ projects. The goal of this method is to create a clustering relation of source units to modules on an arbitrary codebase. To this end, we employ deep fact extraction and creating a dependency graph, on which parts of the algorithm can iterate.

The method uses an “initial clustering” as its input along with build system configuration. This initial clustering is an easily created mapping which we create from the physical file layout of the project, e.g. every folder under `src/` constitutes a *module*. The initial input does not necessarily constitute a semantically valid – i.e. successfully compiling – grouping of source files. It is a reasonable expectation, however, that the software project at hand successfully compiles as it is, written with the “traditional translation unit semantics” in mind.

The output modules are singular translation units formed from concatenating source files. The goal of our effort with the algorithm was to provide an output based on static analysis, taking the language specification rules and what is expressed in the code into account. This output can then be reviewed by a software engineer with domain and implementation knowledge of the particular project. The joined source text of module bodies can be viewed as a *unity build* which is created not for the entire code base but a subset thereof. Due to this, certain constraints must be satisfied so the validity of the codes’ semantics is kept.

Theorem 1. *Concatenation of lexically valid C++ files – where “file” may stand for both headers and sources – is also lexically valid. In this context, lexical validity means it produces a valid token sequence at lexical analysis.*

Proof. No source file may end in a partial lexical token or a partial comment, as per the *Phases of Translation* [14]. Thus every file included into a buffer after a previous properly terminated file will have its first token disambiguated from the last token of the previous file. □

Theorem 2. *Concatenation of syntactically valid, already preprocessed C++ input buffers – with regards to the context-free grammar – result in a syntactically valid larger source file.*

Proof. Statements and blocks of code in the input buffer 2 must be terminated by `;` and `}` respectively form a valid syntactical unit. Line breaks and white-space surrounding a syntactically valid statement – such as `T t = T{}`; – does not matter from the compiler’s perspective. For this proof, the syntactic rules of the C++ language may be reduced to the grammar below, where `stmt(i)` denotes the kind of statements or code bodies which might appear in some scope i . The rest follows from scopes’ enclosure in each other requiring fulfilment of the language of valid parenthesisising.

$$\begin{aligned}
\langle \text{statement-list} \rangle &::= \langle \text{global-statement-list} \rangle \langle \text{statement-list} \rangle \\
&\quad | \quad \langle \text{scope}(i) \rangle \langle \text{statement-list} \rangle \\
\langle \text{global-statement-list} \rangle &::= \langle \text{global-statement} \rangle \text{' ; ' } \langle \text{global-statement-list} \rangle \\
&\quad | \quad \langle \text{global-statement} \rangle \\
\langle \text{scope}(i) \rangle &::= [\langle \text{scope-header}(i) \rangle] \text{' { ' } } \langle \text{stmt-list}(i) \rangle \text{' } \text{' } [\langle \text{scope-suffix}(i) \rangle] \\
\langle \text{stmt-list}(i) \rangle &::= \langle \text{stmt}(i) \rangle \text{' ; ' } \langle \text{stmt-list}(i) \rangle \\
&\quad | \quad \langle \text{stmt}(i) \rangle
\end{aligned}$$

□

We give the skeleton of the modularisation method in Algorithm 1, with the individual steps explained in detail in later sections.

Data: build information, project layout and initial module mapping

Result: module mapping clusters and rewritten source files

synthesise dependency map by fact extraction;

while *module-to-module cyclical dependency found* **do**

 | split cycle by moving *interface files* (“headers”) to new module;

end

move implementation file of program units to the module where their interface is;

while *module-to-module cyclical dependency found* **do**

 | join the modules that form the cycle;

end

rename conflicting symbols;

remove and organise *#include* directives from source files;

Algorithm 1: Skeleton description of the modularisation algorithm outlining the steps taken.

4.1 Fact extraction and handling “unity build concerns”

The first step in the algorithm is to perform deep fact extraction on the source code. This is done for two goals, firstly to create the dependency graph needed to perform later parts of the algorithm, and the second being to search for the list of symbols which need to be touched to handle *unity build concerns* – issues which must be solved without simply concatenating source files would not produce valid and compiling code – discussed in the literature, such as in [27].

Various code schema extractors exist for the C language family and other languages, e.g. Datrix [19] or Columbus [9]. In our practical implementation, we used the LLVM/Clang [39] C-family compiler infrastructure’s `LibTooling` capabilities, which offer an object-oriented *Abstract Syntax Tree* of the translation units’ code, which is the same representation from which code is generated, thus it allowed us to truly use what a particular compiler uses as an input. Clang has been the forerunner of the implementation of the newest standard features or proposals and is a well-known testing bed for the creation of new features. This *AST* can be traversed or even altered from client code written in C++, with the *AST Matcher Library* [6] and `clang-query` offering a declarative syntax for filtering and searching nodes in the tree.

4.1.1 Synthesis of the dependency graph

The algorithm creates a dependency graph which contains the dependencies that are present in the input project. Files may depend on each other through the `#include` macro, transitively. For the purpose of modularisation, two kinds of dependencies have to be distinguished from each other: *usage* relation and *implements* relation, which correspond to the D_{if} and D_{imp} dependency sets discussed in Section 2.3. Projects are found often to be laid out in a way that a logical segment of the software – a library function or function group, or a class – has its own single header and, excluding `templates`, an implementation file for these. During fact extraction, the bindings in the compiler representation can be used to find exactly which symbol is used from which other files, and which definition implements a symbol which was declared in a different file, usually a header. While traditionally the preprocessor logic of the compiler was used to extract the list of files the compilation depends on, this logic can be more fine-tuned with upcoming tools such as `clang-scan-deps` [22] to be able to distinguish the kind of dependency spanned on the symbol granularity.

The dependency graph consists of labelled file nodes and directed, labelled edges between the nodes. The files of the project are always part of the dependency graph,

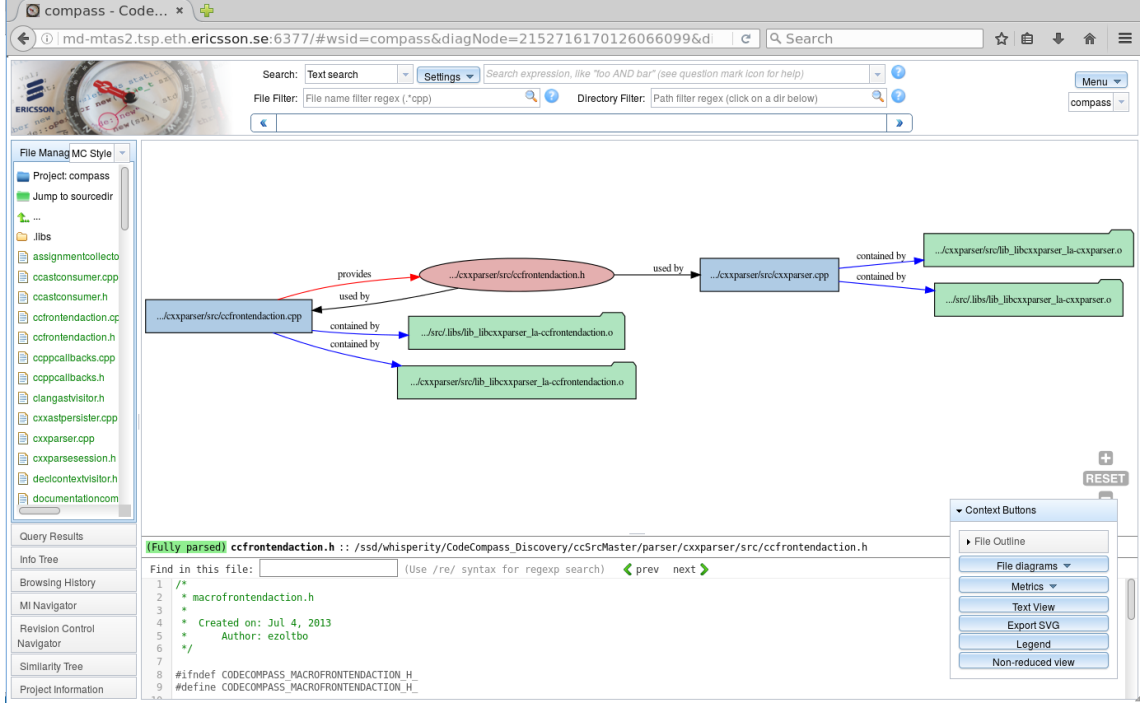


Figure 4.1: Dependency graphs are often used by code comprehension and visualisation tools such as *CodeCompass* [31] to aid developers in positioning a source file related to the full build’s architecture. (Image source from cited paper.) For the purposes of this work, the *translation unit* “contained by” an object is ignored.

while files that are external to the project may also be added to the graph on-demand in later steps of the algorithm.

A file may belong to at most one module. Every file is annotated to which module it belongs in the initial cluster input. The graph of module-to-module dependencies can be synthesised from the file-to-file dependency graph at any time using the following rule. For practical purposes in the implementation, we have kept the module-to-module dependency graph always in memory and updated when files are moved between modules.

Definition 3. A module $\mathcal{M}$ **depends on** module $\mathcal{N}$ if exists a program unit $u \in \mathcal{M}$ for which there is a program unit $w \in \mathcal{N}$ such as $w \in D_{\text{if}}(u) \cup D_{\text{imp}}(u)$.

Remark 3. Module-to-module dependencies are expressed explicitly by placing an **import** statement in the *dependent* module’s code. Modules may not cyclically depend on each other.

4.1.2 Analysis and rename of conflicting symbols

Symbol definitions can be local to a particular translation unit and thus cannot be named and linked against from other translation units. This allows extracting implementation details or helper functions into callable entities without making

them part of the API. Another such local symbol example could be the using of a `typedef` with the same aliased name but defining a different type in two separate translation units. These names collide when files are joined together. As a solution, we opted to apply an automatic rename on these potentially colliding symbols by prefixing the source translation unit's filename to their definition and usage points.

Various other kinds of conflicts can be discovered by deep analysis of the bindings in the compiler's representation. For example, `using some::symbol_name;` declarations introduce a potentially qualified name into the current scope. The effects of these symbol introductions cannot be undone in situ. Possible solutions are to rename the usage sites to the qualified names, which works if the symbol binding is not reliant on *argument-dependent lookup* (ADL) [44], a special and complicated case of lookup rulesets by which C++ name resolution works. Under ADL, a call written as `begin(C)` – in case a direct free function with the overload for type `C` does not exist in the scope the call site is written – resolves to functions `C.begin()` or a `begin` in any `namespace` that contains `C`. ADL is often used in template libraries such as the STL or Boost. In case ADL is used, we can only warn the user that there is an unsolvable case and point them to the source code where it was found.

The uniqueness of names and the *One Definition Rule* must also be respected for end binaries, not just individual translation units. ODR allows certain symbols – such as `inline` functions and classes – to be defined multiple times across translation units as long as the definition is the same. It was shown that names are not necessarily unique when considering an entire software project [36]. By traversing the build graph which we have access to in the first step, the set of symbols belonging to a binary can be formed through the process described in [37]. This way, the target's name can form part of a renaming scheme. The renaming done on the symbols are trivially reversible.

4.1.3 Handling macro bleed

While symbols collision results in an error diagnostic and unsuccessful compilation, macros might bleed over between joined translation units in a more subtle manner. The redefinition of a macro to another value results in a *warning* with most modern compilers. The implemented solution is extracting the list of macros introduced by the file at the end of its token sequence, and performing a `#undef` on the macros. Macro definitions introduced by transitive inclusions of external headers into a translation unit can be ignored as the modularisation only concerns files that are part of the project being analysed. However, this may only be done if the file including the macro definition does not depend on the macro being defined or what the particular value is. In case a “read” of the macro value is performed, the undefinition must

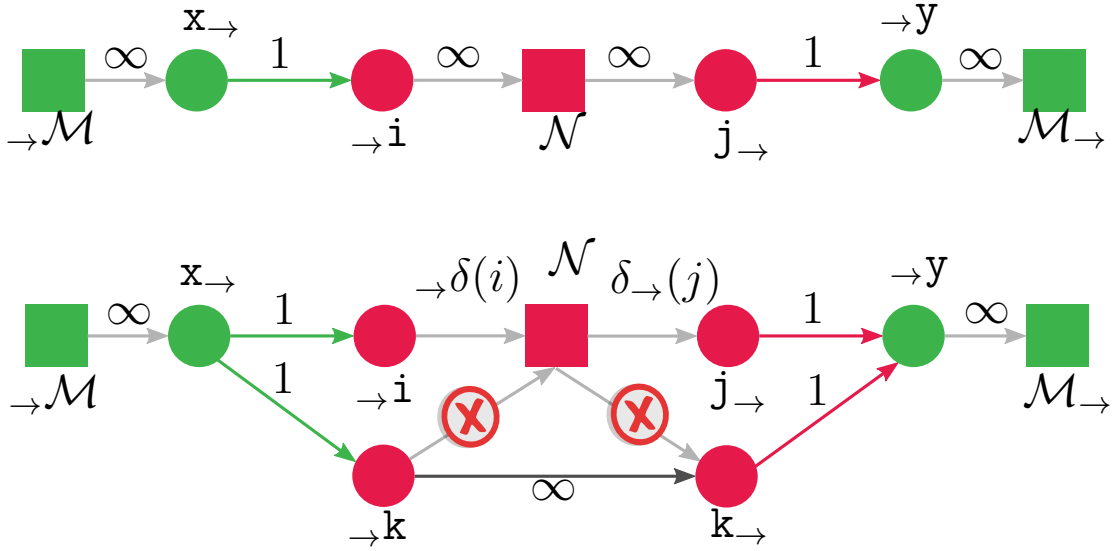


Figure 4.2: The abstract model that describes how the *edge capacities* in the flow graph for a module-to-module dependency cycle are formed. In this example, $\mathcal{M}$ and $\mathcal{N}$ co-depend on each other through files x , i , j and y , and in the bottom example, also through k . ($\mathcal{M}$'s two sides has been split for clarity.) The first case shows when the modules in cycle each module and file have a clear *dependent* ($\rightarrow i$) and *dependency* ($j \rightarrow$) side in the flow. The second case shows a setup in which file k appears both as a dependency – of i – and a dependent – of j .

not be done before the call site. If two translation units which are to be merged read the value of the same macro and the macro is undefined at the end of the first dependent, it must be redefined to the same meaning before the second one.

Theoretically the algorithm could support tracking this, however, we opted to ignore creating a practical implementation for such granularity of tracking. Macros were common for C programs with almost 50% of macro definitions used to prevent multiple definitions of a macro or handling architecture- or library-specific conditional compilation [7]. Template metaprogramming techniques including *substitution failure is not an error (SFINAE)* [44] have allowed C++ software projects to depend less heavily on the preprocessor and macros.

4.2 Creating the flow graph

As long as there are cycles in the module-to-module dependency graph that can be generated from the file-to-file dependency graph, we create a *flow* graph using the model described below. The abstract skeleton of the flow graph's construction model is presented in Figure 4.2. This flow graph is created for each cycle $\mathcal{C}$.

$$\begin{aligned}
 V_{\mathcal{C}} = & \{\mathcal{M}_{\rightarrow}, \mathcal{N}, \mathcal{O}, \dots, \rightarrow \mathcal{M}\} \cup \\
 & \bigcup_{\text{module } \mathcal{I}} \{f_{\rightarrow} \text{ for } f \text{ file } \in \mathcal{I} \mid \exists (f, x) \in E_{\Delta} \text{ such that } x \in \mathcal{J} \wedge \mathcal{J} \neq \mathcal{I}\} \\
 & \bigcup_{\text{module } \mathcal{I}} \{\rightarrow f \text{ for } f \text{ file } \in \mathcal{I} \mid \exists (x, f) \in E_{\Delta} \text{ such that } x \in \mathcal{J} \wedge \mathcal{J} \neq \mathcal{I}\}
 \end{aligned} \tag{4.1}$$

A cycle always contains some modules $\mathcal{M}, \mathcal{N}, \dots$, but for clarity purposes, the cycle is split into a path, and the module where it is split becomes two nodes representing the same module with an annotation of the *side*, one being the source and the other being the drain. The cycle might be split at any module, but for practicality purposes, the implementation splits the cycle at the module with the lexicographic smallest name. The other nodes in the flow graph are the files that *span* the dependency between each module that is part of the cycle. File nodes f are annotated based on what *side* they take in the span resulting in – potentially – nodes labelled $\rightarrow f$ and $f_{\rightarrow}$.

Before constructing the edges of the network flow graph's edges the vertices must be checked to determine whether the first or second case depicted in Figure 4.2 is applicable for each module in the current cycle's spanned dependency graph. If exists any file f in a module $\mathcal{N}$ for which both $\rightarrow f$ and $f_{\rightarrow}$ vertices are created, the second case applies, otherwise the first one. Applicability of the second case will be denoted by the predicate $\mathbb{B}(\mathcal{N})$ in the model.

Let $f, g, \dots$ denote files and $\mathcal{M}, \mathcal{N}, \dots$ denote modules in the cycle. In case a module $\mathcal{I}$ the module where the cycle was split into a source and sink node, sides are connected by edges accordingly: the “incoming” – dependency, $\rightarrow f$ – side of a file is connected to the dependency side of the module: $\rightarrow \mathcal{I}$. We construct the edges as follows:

1. Every file-to-file dependency constitutes a flow capacity of **1**.

$$\exists (f, g) \in E_{\Delta} \implies c_{\mathcal{C}}(f_{\rightarrow}, \rightarrow g) = 1 \tag{4.2}$$

2. In case $\mathbb{B}(\mathcal{N})$ does not hold, every file in $\mathcal{N}$ is connected to the module it belongs to with a capacity of ∞ .

$$\forall \mathcal{I} \text{ module} : \forall f \in \mathcal{I} : \begin{cases} c_{\mathcal{C}}(\mathcal{I}, \rightarrow f) = \infty & \rightarrow f \in V_{\mathcal{C}} \\ c_{\mathcal{C}}(f_{\rightarrow}, \mathcal{I}) = \infty & f_{\rightarrow} \in V_{\mathcal{C}} \end{cases} \tag{4.3}$$

3. In case $\mathbb{B}(\mathcal{N})$ holds, the files in $\mathcal{N}$ are connected in the following fashion:

- Files i , j , etc. which only appear on one side of the dependency span – either as a dependent or a dependency – are linked to the module’s node, but with a capacity that corresponds to their incoming and outgoing dependency degree in the software project, denoted by $\rightarrow\delta(i)$ or $\delta_{\rightarrow}(i)$ for dependencies and dependents, respectively.

$$\forall \mathcal{I} \text{ module} : \forall f \in \mathcal{I} : \begin{cases} c_c(\mathcal{I}, \rightarrow f) = \rightarrow\delta(f) & \rightarrow f \in V_c \wedge f_{\rightarrow} \notin V_c \\ c_c(f_{\rightarrow}, \mathcal{I}) = \delta_{\rightarrow}(f) & \rightarrow f \notin V_c \wedge f_{\rightarrow} \in V_c \end{cases} \quad (4.4)$$

- Files which are both dependencies and dependents in the current span are linked not to the module’s node but to each other, with a capacity of ∞ .

$$\forall f \in \bigcup_{\text{module } \mathcal{I}} : \rightarrow f \in V_c \wedge f_{\rightarrow} \in V_c \implies c_c(\rightarrow f, f_{\rightarrow}) = \infty \quad (4.5)$$

4. No other edge is constructed in the network flow graph.

In case a file f which appears both as a dependent and a dependency belongs to the module $\mathcal{M}$ where the cycle was cut into a path, adding an ∞ capacity edge between the two sides of the file would result in the possibility infinite flow, breaking the algorithm. In case $\mathbb{B}(\mathcal{M})$ holds on the source/drain module, the cycle is re-cut at the next module instead, and a new flow graph is constructed. In case every module in the cycle satisfies $\mathbb{B}$, no file move may solve the cyclic dependency, and the algorithm has to be terminated, with the prototype implementation showing the user a detail on this error. The algorithm can be restarted with a different *initial clustering* where the two co-dependent files belong to the same *initial module*.

Commodity flow algorithms such as Edmonds-Karp [3] can be used to calculate the minimal cut of the graph constructed.

Corollary 2. Due to the construction of this graph, the cutting edges will be dependency edges between two file, or between a file and a module node, as every other edge has an ∞ capacity.

For each cycle, the files on the end of the cutting edges are moved to a new ephemeral module. This iteration is performed until no cycles can be calculated between D_{if} of modules. Due to the number of source files being limited and originally known, the algorithm must terminate.

```

module1.cppm

export module module1;

/* The below example does not compile, as Private's name
 * is not exported, thus cannot be depended upon by
 * an exported symbol. */
class Private;
export void f(Private* priv_ptr);

/* Compiles. */
export class MyClass;
export void g(MyClass* mc_ptr);

```

```

module2.cppm

import module1;

export module module2;

/* Symbol collision: MyClass is defined multiple times in different
 * modules, neither this file, nor any client code will compile. */
export class MyClass { int i; };

```

Listing 7: Issues with *forward declarations* when using *Modules*.

4.3 Moving implementations to the module of the interface

At the termination of the previous step of the algorithm, every header file has been moved to modules which no longer depend on each other in a cyclic manner. However, to satisfy that the generated code will be valid, more steps need to be taken, as *headers and translation units* differ fundamentally from *modules* due to how symbol visibility works. A module must be a concise translation which *defines* the symbol it **exports** to client code and as such, (forward) declaration of symbols across module boundaries is ill-formed, while headers could forward declare and name symbols which another header – or in extreme cases, only a particular translation unit’s “source code” – defined. Listing 7 shows an example of what is not possible with modules.

The algorithm looks at the dependency graph, searching *implements* relations. Every file *g* implementing a symbol declared in “header” *f* is reassigned to the module where *f* belongs at the end of the previous step. Performing this operation presents another issue, as **new cycles** can be introduced due to it. Dependency through a *forward declaration* is also joined in this step.

```

#define AUTOMATIC_MODULE

export module Module_55b5fc;
// Tag is automatically generated from hash of files in the module.

/* Here comes the list of the source files ("fragments") that got
clustered into the current module. */
#include "src/something/Something.hpp"
#include "src/other/MyType.hpp"
#include "src/something/Something.cpp"

```

Listing 8: An example result of the prototype implementation for the algorithm discussed in this paper.

4.4 Fixing implementation-spanned module-to-module cycles by merging

We re-synthesise the module-to-module dependency graph from the current state of the clustering produced. The cycles introduced here are solved by a greedy approach: the cycles are checked with the shortest ones analysed first, and in each cycle, the two modules with the least amount of files in it are merged. The cycles can be generated by a quadratic iteration over the adjacency matrix of the graph. This is iterated until no cyclic co-dependencies between modules remain in the project. *Merging a module* is performed by creating a new – still ephemeral – module which contains all files that were contained in the original ones. Due to the number of modules lower-bound by 1, the algorithm must terminate, with the end case being the project’s collapse into a *true unity build*.

4.5 Code cleanup and emitting the result

After the modules are calculated by the algorithm, a few clean-up and result writing steps need to be taken to ensure the generated files are valid C++ code and optimally organised. We will consider previous source files and headers that were moved into modules as *fragments*.

- First, of such, `#include` directives from fragments are removed due to the contents already being either present in the module file, or the symbols from the header are now part of another module. This also means that inclusions of *system headers* and third-party libraries external to the project are left untouched.
- *Usage* dependencies are rewritten as a module-to-module dependency, and an

`import` statement is generated.

- Fragments of the generated modules are topologically sorted based on their *usage* and *implements* relations to fragments in the same module to ensure that the *One Definition Rule* within the module is respected.

An abstract example of the result is shown in Listing 8. Modules in the *initial module clustering* might have had all their fragments moved to other modules, which results in the now empty module's deletion. The modules containing fragments are emitted as C++ code in the prototype implementation.

Chapter 5

Evaluation

We have created a prototype implementation in Python using the *NetworkX* library [45] in which the graph representations and standard graph algorithms such as Edmonds-Karp are implemented. Due to the recent acceptance, neither *Modules* nor build system for it is supported at the time of writing of the thesis, we have utilised the experimental implementation in Clang [39] with an experimental build system support [46] in the *CMake* build configuration tool. While these components are experimental in their nature, Clang has been one of the most used tools as a proof-of-concept in the discussion that shaped *Modules*, and any change in the exact implementation of how the compiler and the build system interact does not change the fundamentals of our findings. Our first test project was *Apache Xerces* [38], with the implementation also tested on *LLVM/Clang* itself. Various other open-source projects have been used as test corpus for tools relying upon Clang, such as the tester tooling described in [10], however, on most of them, we failed to actually compile the result of the implementation due to the requirement of *CMake*-driven build configuration not satisfied by the projects. We, however, ran the algorithm to the point where the possible layout of the resulting modules is visible.

5.1 Cost of the algorithm

Asymptotically, the cost of the algorithm can be summarised in the following points:

- Analysis of problematic symbols and build relations require a deep fact extraction which is done on the internal representation produced by a compiler, which has roughly the same cost as **producing a build of the project** without performing linking.
- The splitting of modules is run on at most as many modules as there are files in the project, as many times as these modules might co-depend on each other in a full graph.

- The joining of modules is done by once again searching for cycles, but a joining of the module always lessens the number of modules left.

Searching for simple cycles in the graph can be done efficiently by utilising Johnson's algorithm [15] – which is also the method implemented in *NetworkX*. The cost for this sub-algorithm is $\mathcal{O}((n + e)(e + 1))$. Cost of the Edmonds-Karp algorithm is $\mathcal{O}(n * e^2)$, where n denotes the number of vertices in the graph, and e denotes the edge cardinality. The rest of the components of the algorithm, such as renaming symbols and writing output result scale linearly with the number of modules and files. The number of modules is at most as much as the number of files, and in the flows graph, each module and file appears, thus $n \leq f^2$. e can be upper-bound by n^2 , the number of edges in the full graph.

In total, the algorithm's cost can be expressed as

$$\mathcal{O}(((n + n^2) * (n^2 + 1)) * (n * n^4) + (n + n^2) * (n^2 + 1))$$

which simplifies as

$$\mathcal{O}((n^5 + 1) * (n + n^2) * (n^2 + 1))$$

and with further bound by

$$\mathcal{O}((n^5 + 1) * (n^4))$$

from which $\mathcal{O}(n^9)$ results. In practice, the polynomial's factor is very generous over-estimation on the upper bound of the algorithm's complexity as most real-world software projects contain a small number of files and rarely form a complete graph as their dependency graph.

What is more important, the above estimation leaves *build time* out, and empirical measurements show that compilation with all the semantic analysis taking place is orders of magnitude more time-consuming than executing the aforementioned graph algorithms. For example, on *Xerces*, the algorithms took 15 seconds on a particular machine, while the analysis of problematic symbols took more than 4 minutes.

5.2 Discussion of results

On *Xerces* [38], a popular XML parser library, we have started execution with 14 initial modules as input, which corresponds to the directory structure of the project. At the end of the first – split-up – phase described in Section 4.1, 74 individual modules were created. After moving the implementation together – see Section 4.3 – and joining the modules based on the D_{imp} span – see Section 4.4 –, 2 modules were emitted by the program. One of the modules was a single header which defined

a single `enum`, while the other module had all the remaining 710 source files joined together of Xerces. The resulting module was, essentially, a *unity build* of Xerces, except a trivial enumeration type. The `enum` left in a separate module is because an enum is just a list of constants, a purely “interface type” which does not have an implementation and thus was never joined with other modules.

Our results show that to move an existing C++ software project over to *Modules* the inner layout of the project has to be changed with greater granularity than file level. This will necessarily break client code because different types and function calls will have to be called. Due to this the overview analysis obtained from purely considering the language rules for a project is not enough to demarcate sensible boundaries where a project might be split in a sufficient way if it is even possible to produce such a cut. Development costs suggest it is reasonable to expect enormous inertia with projects and libraries written with “traditional TU-semantics” in mind. Usage of *Modules* could be expected from brand new projects, assuming these projects will be willing to tie themselves to the newest standard even when the feature is clearly supported by multiple compilers.

The results tend to indicate that software libraries already show well-defined boundaries for client code and thus inter-library or inter-binary usage is well-designed out of necessity. Program units within the library show a level of coupling – mostly through forward declarations – which break the ability to be split into distinct modules. These findings are consistent with those obtained from running the algorithm on LLVM’s codebase: while Xerces – excluding auxiliary code – is a single library, LLVM is composed of many libraries and binaries and the generated modules tended to line up with the binaries defined by the project even if the initial clustering started from the whole project. We found some problematic cases where two or three of the smaller binaries ended up being built from the same generated module, which resulted in linkage issues. We understand that both the build system and the compiler implementation for a brand new feature is not finalised yet so we did not pursue investigating exactly how considerable effort would it take to tweak the build system to handle this so that working binaries are produced.

Similar results have been obtained from attempting a modularisation on Vim [41]’s 8.0 version. To compare with the findings in [4], it should be noted that Vim is rather top-heavy in terms of just two headers, `global.h` and `vim.h` declaring almost all symbols. Most of the connections between the symbols in the project was formed via linkage dependencies – `extern` symbols – without static implementation dependencies formed. This, and the fact that Vim is written in C to be a cross-platform project, which means lots of macro definitions and conditional compilation is present in the code, results in the modularisation algorithm staying roughly close to the input. In practice, such a project would also collapse to a single module, if

the algorithm's implementation is extended with hard resolution of linker-handled dependencies. This means that a granularity deeper than that of file level is needed to create a meaningful new organisation of the code, necessarily requiring changes to the inner architecture.

We have created a tool which can identify, and through Python's graph visualisation libraries also plot pain points where these subtle couplings exist.

Chapter 6

Summary

6.1 Future Work

Our algorithm created single module translation units as its output, however “unity build concerns” are perhaps fixable through the use of *module partitions*. Extending the algorithm to create such partitions in some way should be investigated, but is not expected to enable serious benefits: without domain knowledge, it is hard to identify truly implementation detail files. Module partitions are a build system level ability to split a single module into more translation units, but from the client code’s perspective a module is a *single* unit – and it is an extra maintenance step to list every partition, and manage the exported symbols’ list. When a client compilation import `module T` every translation belonging to the T has to be loaded, and the client sees every exported symbol of T without knowing which partition they belonged to. Thus, while investigating the cuts one could achieve with handling partitions, the results and findings discussed in this work will not be affected.

A useful comparison or enhancement of the algorithm could be made by investigating the approach in [47] and first performing the break-up of headers and includes, and then our modularisation technique. Our approaches considered the input project “well-organised” as per the developers’ intent and did not attempt changing the inner structure, only towards the build system – the assignment of original source files to modules does not change the association of symbols to files.

As explained above for the “loading” of translation units belonging to modules, there are tangible issues around how a practical compiler implementation will handle the feature. Pseudo-implementations for Modules exist in the LLVM/Clang compiler through a modified precompiled headers – see Section 2.2 – mechanics, however, loading entire compilation units the size of a single library is detrimental to the compiler’s performance and memory use, and template instantiations and various other semantic analysis operations are still left to the compiler process that

is compiling the client code’s translation unit. Due to this, the C++ community with compiler and build system developers must investigate optimisation possibilities that the Language Specification will allow to further ease the load on build computer hardware.

6.2 Conclusion

Componentialisation and organisation of software projects had been supported by various programming languages through similar means recently, with C++ only following suit in its most recent version of the language. In this work, we discussed multiple types of issues with the compilation of C++ projects using the traditional, translation unit-based approach. A new language feature, *Modules*, has been shown, which aimed to solve some of the problems mentioned. We have presented our approach which clusterises source files of a project into groups which form a module on the build system’s level and can be used in other parts of the project. This approach creates a *unity build* of parts of a project, and thus necessarily addresses the concerns which might subtly break the resulting application when unity build is formed. The method was tested on open-source projects of different granularity, and we discussed the results found. Our method can be used to identify pain points in the design of the project which prevents good automatic clustering. The key takeaway from our effort is that “upgrading” existing C++ software projects to Modules will necessarily require white-box knowledge and API-breaking changes in almost all cases. It should be noted that the efforts for upgrading code to modules is an ongoing one in various open-source projects, such as the Debugger of LLVM, *LLDB* where the aforementioned requirement for deep domain knowledge has been expressed by the community [42].

Bibliography

- [1] Ceemple Software Ltd.: **ZapCC** – a (much) faster C++ compiler (06 2018), <http://zapcc.com>, (accessed 27 Feb. 2019)
- [2] Cooper, K., Torczon, L.: Engineering: A Compiler, chap. 8, pp. 405–475. Morgan Kaufmann, 2nd edn. (02 2011)
- [3] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd Edition. MIT Press (2009), mitpress.mit.edu/books/introduction-algorithms
- [4] Dayani-Fard, H., Yu, Y., Mylopoulos, J., Andritsos, P.: Improving the build architecture of legacy C/C++ software systems. In: Cerioli, M. (ed.) Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3442, pp. 96–110. Springer (04 2005). https://doi.org/10.1007/978-3-540-31984-9_8, https://doi.org/10.1007/978-3-540-31984-9_8
- [5] Dos Reis, G., Hall, M., Nishanov, G.: P0142R0 – a module system for C++. Tech. Rep. 4, ISO/IEC JTC1/SC22/WG21 - Open Papers (02 2016), <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0142r0.pdf>
- [6] Duffy, E.B., Malloy, B.A., Schaub, S.: Exploiting the Clang AST for analysis of C++ applications. In: Proceedings of the 52nd annual ACM southeast conference (03 2014)
- [7] Ernst, M.D., Badros, G.J., Notkin, D.: An empirical analysis of C preprocessor use. IEEE Trans. Softw. Eng. **28**(12), 1146–1170 (12 2002). <https://doi.org/10.1109/TSE.2002.1158288>, <https://doi.org/10.1109/TSE.2002.1158288>
- [8] Fakhar, F., Javed, B., ur Rasool, R., Malik, O., Zulfiqar, K.: Software level green computing for large scale systems. Journal of Cloud Computing: Advances,

- Systems and Applications **1**(1), 4 (05 2012). <https://doi.org/10.1186/2192-113X-1-4>
- [9] Ferenc, R., Beszédes, Á., Tarkiainen, M., Gyimóthy, T.: Columbus - reverse engineering tool and schema for C++. In: 18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3-6 October 2002, Montreal, Quebec, Canada. pp. 172–181. IEEE Computer Society (10 2002). <https://doi.org/10.1109/ICSM.2002.1167764>, <https://doi.org/10.1109/ICSM.2002.1167764>
- [10] Gábor, H., Réka, K., Péter, S.: Towards proper differential analysis of static analysis engine changes. In: Attila, K., Balázs, B., Tamás, G., Zoltán, K. (eds.) The 11th Conference of PhD Students in Computer Science – Volume of Short Papers. pp. 75–78. University of Szeged, Institute of Informatics, Szeged, Hungary (06 2018), <http://www.inf.u-szeged.hu/~cscs/pdf/cscs2018.pdf>
- [11] International Organization for Standardization, International Electrotechnical Commission: ISO/IEC 1539:1991 International standard: information, technology, programming languages, Fortran. International Organization for Standardization, Geneva, Switzerland, second edn. (1991)
- [12] ISO/IEC JTC 1/SC 22: ISO/IEC 14882:2017 Information technology — Programming languages — C++, version 17 (C++17). International Organization for Standardization, Geneva, Switzerland (dec 2017), [iso.org/standard/68564.html](https://www.iso.org/standard/68564.html)
- [13] ISO/IEC JTC 1/SC 22: One-definition rule, chap. 6.2, p. [basic.def.odr]. In: [12] (dec 2017), [iso.org/standard/68564.html](https://www.iso.org/standard/68564.html)
- [14] ISO/IEC JTC 1/SC 22: Phases of translation, chap. 5.2, p. [lex.phases]. In: [12] (dec 2017), [iso.org/standard/68564.html](https://www.iso.org/standard/68564.html)
- [15] Johnson, D.B.: Finding all the elementary circuits of a directed graph. SIAM Journal on Computing **4**(1), 77–84 (03 1975)
- [16] Keith, M.J., Martin, M.C.: Genetic programming in C++: Implementation issues. Advances in genetic programming **1**, 285–310 (04 1994)
- [17] Klimek, M.: Deploying C++ modules to 100s of millions of lines of code. CppCon – The C++ Conference (09 2016), <https://www.youtube.com/watch?v=dHFNpBfemDI>, (accessed 1 Mar. 2019)

- [18] Krishnaswamy, T.: Automatic precompiled headers: Speeding up C++ application build times. In: Milojicic, D.S. (ed.) Proceedings of the First Workshop on Industrial Experiences with Systems Software, WIESS 2000, October 22, 2000, San Diego, CA, USA. pp. 57–66. USENIX (01 2000), <http://www.usenix.org/events/wiess2000/krishnaswamy.html>
- [19] Lapierre, S., Laguë, B., Leduc, C.: Datrix(tm) source code model and its interchange format: lessons learned and considerations for future work. ACM SIGSOFT Software Engineering Notes **26**(1), 53–56 (01 2001). <https://doi.org/10.1145/505894.505907>, <https://doi.org/10.1145/505894.505907>
- [20] Ledgard, H.: Reference Manual for the ADA Programming Language. Springer-Verlag, Berlin, Heidelberg (1983)
- [21] Litman, A.: An implementation of precompiled headers. Software: Practice and Experience **23**(3), 341–350 (03 1993). <https://doi.org/10.1002/spe.4380230308>, <http://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380230308>
- [22] Lorencz, A.: RFC: prototype of `clang-scan-deps`, faster dependency scanning tool for explicit modules and `clangd` (10 2018), <http://lists.llvm.org/pipermail/cfe-dev/2018-October/059831.html>, (accessed 6 Mar. 2019)
- [23] Malloy, B.A., Power, J.F.: Quantifying the transition from Python 2 to 3: An empirical study of Python applications. In: Bener, A., Turhan, B., Biffl, S. (eds.) 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9–10, 2017. pp. 314–323. IEEE Computer Society (11 2017). <https://doi.org/10.1109/ESEM.2017.45>, <https://doi.org/10.1109/ESEM.2017.45>
- [24] Martin, K., Hoffman, B.: Mastering CMake: a cross-platform build system. Kitware Inc., 3.1 edn. (01 2015)
- [25] Márton, G., Porkoláb, Z.: Unit testing in c++ with compiler instrumentation and `friends`. Acta Cybernetica **23**(2), 659–686 (01 2017). <https://doi.org/10.14232/actacyb.23.2.2017.14>, cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/3947
- [26] Mihalicza, J.: How `#includes` affect build time in large systems. In: Egri-Nagy, A., Kovács, E., Kovásznai, G., Kusper, G., Tómacs, T. (eds.) Proceedings of the

- 8th International Conference on Applied Informatics (ICAI). 8, vol. 2, pp. 343–350. Eszterházy Károly College, BVB Nyomda és Kiadó Kft., Eger, Hungary (01 2010), icai.ektf.hu/pdf/ICAI2010-vol2-pp343-350.pdf
- [27] Mihalicza, J.: Analysis and Methods for Supporting Generative Metaprogramming in Large Scale C++ Projects. PhD dissertation, Eötvös Loránd University, Doctoral School of the Faculty of Informatics (2014)
- [28] Oracle: Sun Studio 12: C++ User’s Guide. Sun Microsystems, Inc. (2008), <https://docs.oracle.com/cd/E19205-01/819-5267/index.html>, (accessed 27 Feb. 2019)
- [29] Oracle: The Template Repository, chap. 7.4. In: [28] (2008), <https://docs.oracle.com/cd/E19205-01/819-5267/index.html>, (accessed 27 Feb. 2019)
- [30] Plasmeijer, M.J., van Eekelen, M.C.J.D.: Functional Programming and Parallel Graph Rewriting. Addison-Wesley (04 1993)
- [31] Porkoláb, Z., Brunner, T., Krupp, D., Csordás, M.: CodeCompass: an open software comprehension framework for industrial usage. In: Khomh, F., Roy, C.K., Siegmund, J. (eds.) Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018. pp. 361–369. ACM (05 2018). <https://doi.org/10.1145/3196321.3197546>, <https://doi.org/10.1145/3196321.3197546>
- [32] Schildt, H.: Java: The Complete Reference. McGraw-Hill Education, 8th edn. (07 2011)
- [33] Smith, R.: P1103R2 – merging modules. Tech. Rep. 2, JTC1/SC22/WG21 - Papers Mailing List (11 2018), <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1103r2.pdf>
- [34] Stroustrup, B.: N1614 – #scope: A simple scoping mechanism for the C/C++ preprocessor. Tech. rep., ISO/IEC JTC1/SC22/WG21 - Open Papers (04 2004), <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1614.pdf>
- [35] Sutter, H.: Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
- [36] Szalay, R., Porkoláb, Z., Krupp, D.: Measuring mangled name ambiguity in large C / C++ projects. In: Budimac, Z. (ed.) Proceedings of the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, Belgrade, Serbia, September 11-13, 2017. CEUR Workshop Proceed-

- ings, vol. 1938. CEUR-WS.org (09 2017), <http://ceur-ws.org/Vol-1938/paper-sza.pdf>
- [37] Szalay, R., Porkoláb, Z., Krupp, D.: Towards better symbol resolution for C/C++ programs: A cluster-based solution. In: 17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017, Shanghai, China, September 17-18, 2017. pp. 101–110. IEEE Computer Society (09 2017). <https://doi.org/10.1109/SCAM.2017.15>, <https://doi.org/10.1109/SCAM.2017.15>
- [38] The Apache Foundation: Xerces: A processor for parsing, validating, serializing and manipulating XML (1999), <http://xerces.apache.org/>, (accessed 8 Mar. 2019)
- [39] The LLVM Project: *Clang*: C language family frontend for LLVM (2003), <http://clang.llvm.org>, (accessed 13 Aug. 2018)
- [40] The LLVM Project: Include-What-You-Use (02 2011), <http://github.com/include-what-you-use/include-what-you-use>, (accessed 28 Feb. 2019)
- [41] The Vim Community: Vim: the ubiquitous text editor, <http://vim.org>, (accessed 18. Mar 2019)
- [42] Turner, Z.: *Host* is now dependency free. *lldb-dev* mailing list (03 2019), <http://lists.llvm.org/pipermail/lldb-dev/2019-March/014846.html>, (accessed 20 Mar. 2019)
- [43] Van Rossum, G., Drake, F.L.J.: The Python Language Reference Manual. Network Theory Ltd. (03 2011)
- [44] Vandevorde, D., Josuttis, N.M., Gregor, D.: C++ Templates: The Complete Guide (2nd Edition). Addison-Wesley Professional, 2nd edn. (09 2017)
- [45] Various open-source authors: NetworkX (2014), <http://networkx.github.io/>, (accessed 8 Mar. 2019)
- [46] Whisperity: Add experimental support for C++ Modules TS if the compiler supports it. Feature extension to CMake (10 2018), https://gitlab.kitware.com/cmake/cmake/merge_requests/2482, (accessed 8 Mar. 2019)
- [47] Yu, Y., Dayani-Fard, H., Mylopoulos, J.: Removing false code dependencies to speedup software build processes. In: Stewart, D.A. (ed.) Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative Research, October 6-9, 2003, Toronto, Ontario, Canada. pp. 343–352. IBM (10 2003), <https://dl.acm.org/citation.cfm?id=961375>

- [48] Zucker, J.S.: Application binary interface and method of interfacing binary application program to digital computer (01 1996), european Patent: EP 0 690 74 A2