

Symbol Clustering: Resolving ambiguous symbol references in the context of large-scale C / C++ projects based on linkage information

Szalay Richárd

Programtervező Informatikus BSc, 3. évfolyam
szalayrichard@inf.elte.hu



Témavezető: **Dr. Porkoláb Zoltán**, egyetemi docens
Eötvös Loránd Tudományegyetem, Informatikai Kar,
Programozási Nyelvek és Fordítóprogramok Tanszék,
1117 Budapest, Pázmány Péter sétány 1/C.

Contents

I	Introduction	1
I.1	Motivation	1
I.2	Results	2
II	Basics	4
II.1	C/C++ building pipeline	4
II.2	Distinguishing names in C/C++ projects	5
II.3	Description of the problem	5
III	Industrial implementation	8
III.1	Precise reference resolution in the CodeCompass tool	8
III.2	State of the Art	10
IV	Outline of the solution	12
IV.1	Nature of code comprehension	12
IV.2	Build sources and targets	13
IV.3	Clusters	13
IV.4	Calculating a cluster	14
IV.5	Weighing symbol matches	16
V	Analysis of practical implementation approaches	18
V.1	Store clusters information for every AST node	18
V.2	Store clusters for select AST nodes only	19
V.3	Store clusters for every file	19
V.4	Store clusters for only certain files calculated heuristically . .	21
V.5	Ambiguity criteria	25
V.5.1	Functions	25
V.5.2	Types	26
VI	Summary	27

VI.1	Related work	27
VI.2	Future work	27
VI.3	Conclusion	28

Acknowledgements

First and foremost I would like to express my gratitude to my advisor, Zoltán PORKOLÁB for his support in my research project, his patience, motivation and insight to the field. I would also like to thank Dániel KRUPP, my colleague, who helped me in the time of this research with his approach to the topic, along with my colleagues Tibor BRUNNER and Gábor HORVÁTH, for offering their counterarguments and questions during the research.

My sincere thanks goes to Ericsson Hungary, Ltd. who helped this research by giving the industrial setting for the problem described in this paper, and the opportunity of an internship, access to their facilities, and software.

Abstract

Resolving symbol references is an important part of many application areas from development environments to various static analyser tools. Different occurrences of the same program elements, like function definitions and their call sites, variable declarations and their usage, or type definitions and their applications should be connected. In case of the C++ programming language, the most current tools use mangled names to correlate symbols, e.g. when implementing actions like “go to definition” or “list all references”. However, for large projects, where multiple binaries are created, symbol resolution based on mangled names can be, and usually is, ambiguous. This leads to inaccurate behaviour even in major development tools. In this paper we explore the reason of this ambiguity, and propose a method to improve the accuracy of symbol resolution for large C and C++ projects. We introduce our clustering algorithm based on essential build information of the project and discuss various implementation approaches to minimise run-time and storage overhead of the method. We implemented our method as part of the CodeCompass open source code comprehension tool and measured its efficiency on various large C++ projects. Although our method still leaves some ambiguity in case of the dynamically used symbols (e.g. via `dlopen`), in most of the practical cases it eliminates the uncertainty connected to symbol resolution and can significantly increase the quality of development environments.

Chapter I

Introduction

I.1 Motivation

The size and scale of software systems have grown rapidly over the course of the recent years. Large-scale projects amounting up to million lines of code aren't uncommon along with the usage of different build configurations across hardware architectures – the source code of the Linux kernel, with drivers, is around 17 million lines of code. This presents a challenge when it comes to understanding and navigability of the project. It is always essential to understand the precise behaviour of a software system when we are fixing a bug, or extending the system with a new functionality, and the importance of this understanding is preeminent when a major refactoring task is being undertaken.

One of the most important parts of such a task is identifying the impact the changes have: when we are about to modify, e.g. a function, we are interested in the call sites of that function; for variables, we are looking for the read and write locations of that variable. These occurrences change as the project is built with different settings: behind a function of a given name, multiple implementations are present, depending on the environment of the build.

To enhance understanding of the project, we have to correlate the occurrences of symbols in the source code and how these symbols are present in the built binaries. This helps developers to understand the project they are working on better via code comprehension tools while helping automated defect detection via static analysis tools. Consider, though, that this correlation usually can't be discovered using simple name identity: in most programming languages, the same name could mean different symbols based on the occurrence's context, e.g. attributes or methods with the same name in

different classes, overloaded version of functions, or functions with the same name in different namespaces/packages or build configurations.

Modern software tools, like various development frameworks [1, 2, 3, 4] and code comprehension tools [5, 6] provide discovery functionality: the user of the tool can jump to the place of the definition or can iterate all places of reference of a given symbol. Naturally, we expect such tools to work not on simple name identity, but on exact symbol correlation. Unfortunately, we found that current development and comprehension tools fall behind and into ambiguity with the increase of the codebase’s size. The reason is – as we explain in detail in Section II.3 – that *mangled names* (see Section II.2) are allowed to be ambiguous when multiple executables are produced to form the released product.

I.2 Results

To alleviate this issue, we describe a new method to *clusterise* the mangled names found in the project into distinct groups, based on the information available about how the project is built, which translation units are used and in what exact build configuration and environment are they used to form the resulting binaries. Mangled names belonging to the same cluster should refer to the same symbol; and they are distinct from symbols with identical mangled names classified to different clusters. This way, we were able to significantly reduce the possible ambiguity in symbol resolution.

We implemented the method as part of CodeCompass [7] – an open source software comprehension tool – where symbol correlation is used in a number of functionalities. We evaluated our method using CodeCompass to parse various open source C++ projects and measured different implementation variants on the run-time and storage expense. Our final solution has a minimal overhead on the discovery process, therefore it is applicable for various development and code comprehension tools.

Although, some uncertainty remains in the rare cases when a symbol is used dynamically, e.g. when a shared library opened via `dlopen` and a symbol is retrieved with API functions [8], our method can dramatically increase the quality of C and C++ symbol resolution for large systems. This may further improve essential functionalities of development frameworks and code comprehension tools.

This paper is organised as follows: In Chapter II we describe the problem in details using examples. The implementation context of our solution, the CodeCompass code comprehension tool, is described in Chapter III. We discuss current well-known tools and their approach to the problem in Sec-

tion III.2. In Chapter IV we outline the concepts needed for our solution and then define our precise, linkage-safe symbol reference resolution method. Possible implementation approaches are evaluated in Chapter V with measurement results present. Possible extensions for our solution is discussed in Section VI.2. Our paper concludes in Section VI.3, where we reflect on the accomplishments discussed in this paper.

Chapter II

Basics

II.1 C/C++ building pipeline

Software products written in C or C++ language take multiple steps to be built. First of all, a translation unit, usually a source file is handled by the preprocessor which, amongst other things, includes header files containing more symbols (usually declarations and inlinable definitions) to the to-be-compiled code. The preprocessor can add and remove blocks of the source code (conditional compilation) and textually change tokens, inplacing different code parts (preprocessor macros). This preprocessed code, which is still a simple text file, is then *compiled* into an object file. Thus, at this point, multiple object files could be created from the same initial source file, albeit with possibly different contents based on build configuration parameteres, such as using different include paths for the headers, or defining different macros – the most trivial example would be enabling log emission from certain functions in a debug build.

A set of object files is then *linked* together by the linker to form a binary executable or a library. The linker takes object files and other libraries to form this output binary. Built libraries, which are results of a linking command, can also be used in the creation of other binaries. Thus, the potentially vast number of different object files could be used for multiple binaries, e.g. using 32-bit and 64-bit architectures or built for different system libraries (such as different versions of networking stacks or databases).

Various tools exist which allow the instrumentation of the pipeline in a way that the actual executed commands are available for other software to read and understand, such as the JSON compilation database format [9].

II.2 Distinguishing names in C/C++ projects

In the case of the C++ programming language, usually the *mangled name* [10, 11] is used to distinguish between different symbols of the same name. The mangled name is constructed using (possibly multiple) namespace and class information by concatenation, but its exact form is compiler-dependent. Compilers use mangled names to enable *operator overloading* [12], i.e. they generate different mangled names for functions with the same name but different parameter lists. Linkers use these mangled names to resolve symbols [13].

In the case of the C language, there is no such thing as namespace, class, or function overloading [14]. Still, in certain cases, such as specific optimisations on how a method is called, *name decoration* occurs [15]. In this paper we will refer to the name visible to the linker as *mangled name*, both in the context of C and C++ for the sake of simplicity.

While mangled names must be unique for all translation units linked into a single executable, this does not stand for large-scale projects where multiple executables are typical.

II.3 Description of the problem

In a software comprehension activity the most frequent questions users ask are “Where is this method called, or type referenced?” and “What does the declaration or definition of this look like?” [16]. A software comprehension tool should be able to answer these questions as precisely as possible, as accuracy ensures more optimised usage of the developers’ time spent working. Both questions lead to the fundamental problem of correctly resolving references to the definition and usages of a type, a function, or variable, and other language components.

According to the *One Definition Rule* (ODR) of C++ [17], only one definition of any variable, function, class type, enumeration type, or template is allowed in any one translation unit. When resolving references to ordinary C functions, static and non-virtual C++ member functions, type names or non-polymorphic variables, the unique definition within a single translation unit can be found based on static information. Specifically, the function definition of non-virtual functions or ordinary C functions can be looked up based on function signature, which – according to [18] – contains the name of the function, the enclosing namespace, class of which the function is member of (in the context of C++), the type of the parameters, template parameter list (in case of function templates in C++), *cv*- and *ref*-qualifiers, unique

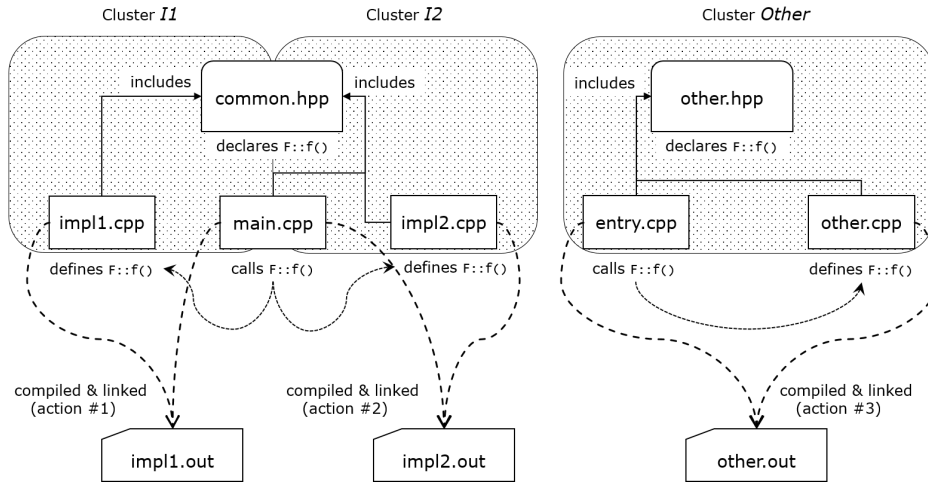


Figure II.1: Developers searching for the definition referenced by the $F::f()$ call in `main.cpp` should be present with an indication that the definition of a function with this signature in `other.cpp` has no static linkage connection to `main.cpp` – thus it is highly unlikely that the call in `main()` actually refers to this function.

type names (qualified with namespace) and scope-correct variable names.

There can be, however, more than one definition belonging to the same unique-name or signature, but defined in different translation units that are not linked together. This is a typical scenario in large-scale programs consisting of multiple separate executables and build configurations, e.g. every executable having a `main()` function as entry point.

Since the translation unit containing the reference and the set of translation units linked together is known for the linker, it is possible for the linker to look up the correct, unique definition for any given reference.

In contrast, for a software development or comprehension tool, while the user is browsing a source file, the linkage context (the set of translation units linked together) where the definitions should be resolved is usually unknown [19]. This leads to ambiguous type, function or variable references. In some cases this ambiguity can be resolved automatically, by taking into consideration the linkage information. For example in Figure II.1, the reference $F::f()$ in `entry.cpp` can be resolved to the definition in `other.cpp`, since it is known that `entry.cpp` is only linked with `other.cpp` in any target binary.

However, this is particularly problematic if the same source file is compiled into several binaries, such as `main.cpp`, which is compiled into `impl1.out` and `impl2.out`. When, for example the user queries the definition of $F::f()$

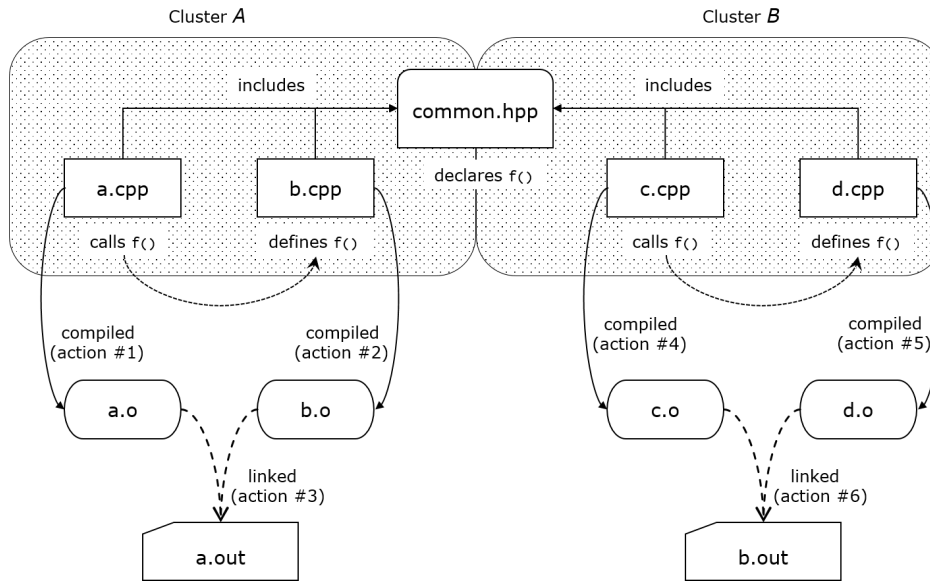


Figure II.2: Jumping to the definition of $f()$ from a call is ambiguous without knowing how translation units are linked together – forming a cluster of symbols and considering it in disambiguation solves this problem.

in `main.cpp`, it is impossible to decide whether to present `F::f()` in `impl1.cpp` or `impl2.cpp`. In these cases the ambiguity can only be resolved by asking the user to specify the linking context, but asking the linking context should only be done when it is absolutely necessary, to avoid superfluous user interaction that causes disorientation in the comprehension process [20].

The prevalence of the problem was tested on a group of projects with measurement results presented in Tables V.3a and V.3b. It appears that, statistically, a considerable ratio of symbols (0.5 – 10%) are affected – thus the problem is not negligible for its impact on code comprehension and development time when developers have to resolve symbol references using their internal and external knowledge instead of a proper tool solving the problem for them.

Chapter III

Industrial implementation

III.1 Precise reference resolution in the CodeCompass tool

CodeCompass [7] is an open source web-based comprehension tool that helps in understanding large-scale software systems written in C/C++, Java or Python. It uses static analysers to parse the source code and build a database of the collected symbols and their relationships. It provides a read-only web user interface where queries can be executed and visualisations (such as component diagrams, or class diagrams) can be rendered.

To test our algorithm in an industrial setting, we implemented the solution outlined in Section V.4 in the CodeCompass tool. This tool is widely used within Ericsson for large projects and allows us to collect user feedback. Due to the efficient clusterisation algorithm, the parsing time increased only by 2% even for large (over 10 million lines) software projects – see Table V.2b for measurements. The run-time performance of definition and reference (function callers, a type reference) lookup did not deteriorate in any noticeable extent. This means that the new reference disambiguation feature could be introduced without any real performance penalty.

CodeCompass utilises the LLVM/Clang [21] compiler infrastructure to analyse C and C++ source code. It gives exact information about complex C/C++ language elements like overloading, inheritance, read or write locations of variables, possible calls on function pointers, virtual functions, or automatically generated methods – features that various existing tools support only partially.

Comprehending call hierarchies is the most frequent activity developers perform for bug investigation. During this process, they query the definition for function calls and query callers of functions. To find the definition of a

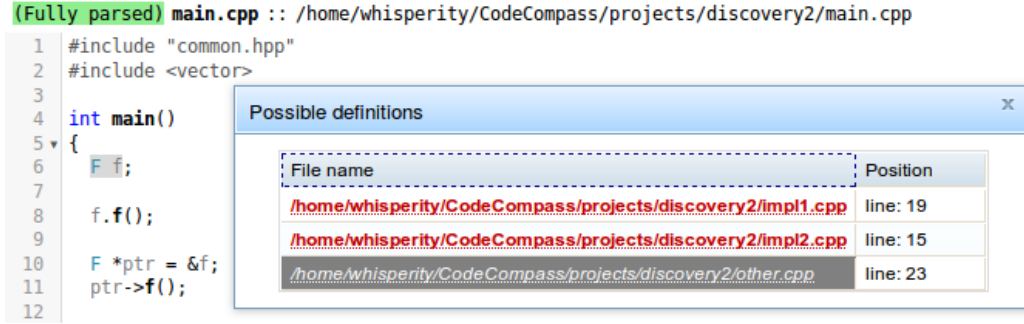


Figure III.1: Query results are shown by CodeCompass indicating their weight in a disambiguation page – *strong matches* are shown with a bold font while *weak matches* have a background colour attached.

function reference, the tool needs to look up all function definitions with a matching signature. Unfortunately, in many cases, this is not sufficient. As described in Section II.3, reference to a function definition can not always be resolved thoroughly without linkage or runtime information. Utilising linkage information for symbol reference resolution greatly improves the caller and callee resolution precision in CodeCompass.

We found the most pertinent examples of ambiguous function references in the implementation of a proprietary Ericsson middleware [22] that was compiled for Linux and DICOS operating systems. The functions of the middleware interface was implemented for both OSes. Each function had the same signature but completely different implementation – one for Linux and one for DICOS. The two middleware implementations were compiled into different binaries. Let us assume that a developer starts to analyse a call chain starting out in a Linux user program which eventually contains a call to the *init()* function of the middleware. Using the refined resolution procedure described in this paper, the developer can smoothly traverse from the Linux user program code into the Linux implementation of *init()*, without erroneously jumping into its corresponding DICOS implementation.

When the user clicks to jump to the definition of a function call that definitions with the same signature, the user is offered a new reference disambiguation pop-up window, as shown in Figure III.1: most likely the function call ends up in the strongly matching definition (indicated by bold), the weakly matching definition has less possibility. Based on user feedback, developers found this feature particularly useful in case the same interface is implemented for different platforms, as it is with TSP [22].

III.2 State of the Art

Current well-known development, engineering, and code comprehension tools rarely provide good support for this problem, because engineering tools were not created with large-scale, long-term projects with many different build configurations in mind. We have analysed some common tools in terms of their reaction to a “jump to definition” query:

- *Microsoft Visual Studio* [1] shows a disambiguation page when encountering the problem described in this paper. If the entire *solution* (a group of projects handled together) is configured for a certain dependency (i.e. only one of the ambiguous definitions is compiled when the solution is built) and the user changes the internal settings of the solution, Visual Studio decides which symbol a “jump to definition” query jumps to. This fine-tuning on the users’ end seemingly does not affect “get calls” / “get usage”-like queries, which still show results with every possible option present, including those which are clearly not valid in the solution’s current state.
- *JetBrains CLion* [2] analyses and builds symbol information when a project is configured, and one project having multiple separately configured executable targets misleads the IDE. A certain file is designated as location where function $f()$ is defined – our understanding currently reveals that the file which has a larger name in lexicographical order. The ambiguity is present even when a certain executable is being debugged by the IDE with “Step Into” showing the proper implementation being executed while “Jump to Definition” opening an entirely distinct source file.
- *NetBeans* [3] does not show a disambiguation page at all, seemingly jumping to a file first detected for a certain symbol after the last build – this can be overridden by manually setting certain files to “Exclude from Build” after which the file’s symbols won’t contribute to the set of potential results. The file to which NetBeans jumps for a definition varies between client restarts, seemingly in a non-deterministic fashion; the only exception is when the symbol is *defined* in the same file where it is *used*: in this case **all** queries jump to this location in particular.
- *Eclipse* [4] properly prioritises symbols explicitly defined in the same source file, but if the definition is not found in the file where a query is issued (see examples Figure II.1 and II.2), a disambiguation page is shown. Putting different builds into entirely different projects with

their own `Makefile` solves this issue, but search queries does not traverse project boundaries.

- *Woboq Code Browser* [5] shows the locations where a symbol is defined when viewing information about a particular symbol, but the problem of Section II.3 is present. Jumping to definition by clicking on a usage location jumps the user to the definition that has been first (in the order of build commands) discovered by the `codebrowser_generator` tool of *Woboq*. It binds usages and definitions in the same source file together, but further “clustering” capabilities are not present.
- *OpenGrok* [6] uses Apache Lucene to index source code and it considers source files as pure textual input with no knowledge of build relationships. This prevents the recognition of “clusters”.

It is seen from this empirical test on currently existing, commonly available and widely used tools that developers need to invest their own knowledge – which may or may not exist, e.g. when venturing into a different module of a large system – to essentially figure out how symbols affect each other.

Chapter IV

Outline of the solution

IV.1 Nature of code comprehension

Software engineering and comprehension tools help their users by providing a platform which can be used to execute queries. Such queries include but are not limited to: finding a symbol's (e.g. a function's) definition or listing where a symbol is used (e.g. a function is being called). Conceptually these queries involve filtering the set of symbols $\mathcal{S}$ until a deemed answer is reached.

The first step in such filtering is to consider an equivalence relation which selects the equivalence set $\mathcal{O}$ of symbol set $\mathcal{S}$ which contain only symbols that are related to each other in terms of meaning, e.g. all $f()$ functions. A trivial approach to do this is to perform a **string match** on the projects' source files for the symbol's name and tokens that “look like” a function in accordance with the language's ruleset.

A more elaborated approach involves maintaining a symbol table (in a database or in the tool's memory) and using *qualified* names and type information that are more descriptive about a symbol's nature. For example, in LLVM/Clang, a function $f()$ has a mangled name `_Z1fv`.

The mentioned *jump to definition* and *get function calls* queries from this point involves further filtering this $\mathcal{O}$ set for a certain definition symbol or a set of symbols which are of *function call* kind.

Searching based on a symbol's **mangled name** is usually considered “good enough” because of how the Language Specification lays out rules for symbol interaction in the compilation and linkage process. E.g. a function may only have strictly one definition and every call of this function is bound to that certain definition.

However, the aforementioned queries produce ambiguous results when executed on large-scale projects. This stems from the fact that these tools

were developed with the language rules in mind, considering the entire project as a single ultimate binary. E.g. if a symbol is defined multiple times for a binary, – a clear violation of the language rules – the compilation toolchain issues an error for the programmer. Software engineering and comprehension tools could, at most, show a disambiguation page and put weight on the developer’s shoulders to make out which is the “real” definition said call is using.

Consider our example in Figure II.2. If we contemplate that all files are sources for exactly one binary, the $f()$ calls become ambiguous between the two definitions, while in reality, the programs are well-formed and each call is properly bound to executing the implementation of a single definition, within the usage’s binary. The function *defined* in `b.cpp` is called at one location, while the *declaration* of $f()$ in `common.hpp` is part of two binaries: from the declaration’s point of view, considering the entire project, multiple definitions exist.

IV.2 Build sources and targets

We can view build actions as the propagation of symbols from one file to another. A build action always has one or more sources and one or more targets. We can express a program’s or a whole project’s build chain by chaining the usage of the “file is source of” and “file is target of” relations.

Consider the example depicted in Figure II.2. We have `a.cpp` and `b.cpp` as our source translation units which are *compiled* into `a.o` and `b.o` (action #1 and #2), respectively. These objects are then *linked* together, forming `a.out` (action #3). A second binary is created by first compiling (action #4 and #5) `c.cpp` and `d.cpp`, then linking the resulting objects together into `b.out` (action #6). In this example, the compilation commands have a “one source–one target” layout, while linkage commands have two sources.

IV.3 Clusters

A *cluster* is a set of symbol occurrences which are referenceable by each other. It is deducible from the language rules (most importantly ODR) that every built binary well-defines its own cluster: every symbol which had been compiled and linked into a given binary is available to each other (in context of “physical” availability and not considering the concept of visibility), and thus part of the cluster formed by the given binary itself. Thus, we can identify the cluster by the ID of the build action that generated the binary which

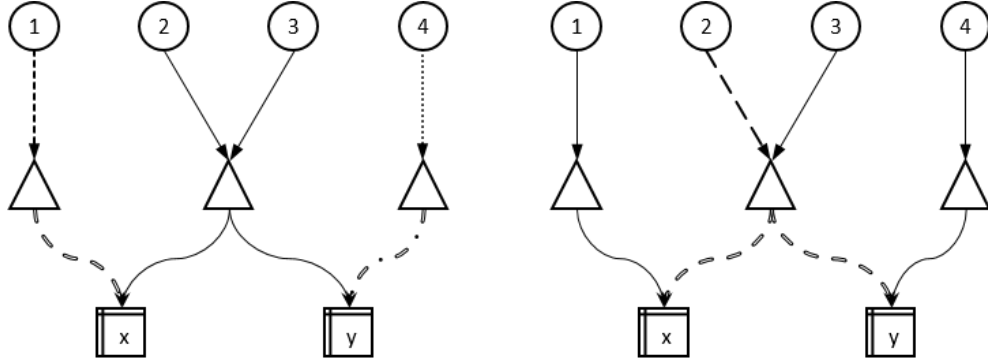


Figure IV.1: Starting the algorithm with the solid dashed (1) and solid dotted (4) actions gives hollow dashed (x) and hollow dash-dotted (y) as result, respectively. However, starting it with solid long-dashed (2) gives a result $\mathcal{C}$ with two elements: both x and y – indicating that symbol occurrences in the sources of 2 are propagated into both binaries.

defines a particular cluster. A symbol occurrence can be part of multiple clusters (just as how the declaration of function $f()$ in Figure II.2 is), and in almost every case a cluster contains multiple symbol occurrences.

Symbols are propagated from one file to another via subsequent build actions, forming a chain. The intermediate files in this chain do not carry extra information in terms of symbol availability: if `a.o` and `b.o` is linked into `a.out`, the target will contain the symbols from every source. Thus these intermediate files can be omitted without the loss of generality. The solution to the problem described in this paper requires the consideration of *ultimate clusters*, a set of clusters into which symbols are propagated, and from where they are not propagated any further – the leaf nodes of the graph formed from the aforementioned build relations.

For our example in Figure II.2 we consider two ultimate clusters, while for the one in Figure II.1, we consider three.

IV.4 Calculating a cluster

Assuming that we know the build actions that were used in the compilation of a given project, we can calculate the ultimate clusters for each build action by using the following algorithm. This algorithm takes a single build action B' – usually a *compile* command of a translation unit/source file – and it calculates a set of build actions that represent the ultimate clusters for the symbols found in the input build action's sources.

In Figure IV.1, we present an example of output clusters for some inputs on a small project: we had a common object file which is used in both binaries, and executed our algorithm to calculate where symbols of this “common object” are present.

```

Input:  $B'$  build action
let  $\mathcal{C}$  be a set of build actions
 $\mathcal{C} \leftarrow \{B'\}$ 
expand  $\leftarrow$  true
while expand is true do
  expand  $\leftarrow$  false
   $\mathcal{C}' \leftarrow \mathcal{C}$ 
  for all  $\beta$  in  $\mathcal{C}$  do
    for all  $T$  such that  $T$  is a target of build action  $\beta$  do
      if exists a build action for which  $T$  is a source file then {the build
        chain can be continued}
         $\mathcal{C}' \leftarrow \{x \in \mathcal{C}' \mid x \neq \beta\}$ 
         $\cup \{x \mid x \text{ build action for which } T \text{ is source}\}$ 
        expand  $\leftarrow$  true
      else {the build chain reached its end, no further expansion can
        happen for  $\beta$ }
        noop
      end if
    end for
  end for
   $\mathcal{C} \leftarrow \mathcal{C}'$ 
end while
Output:  $\mathcal{C}$  ultimate clusters

```

The clusters into which a symbol s is propagated are the collection of all $\mathcal{C}$ sets calculated by calling the algorithm on every build action B for which the file containing s is a source.

In the case of symbols found in header files, the previous sentence considers “files containing s ” as the files which include – via any number of hops in the header inclusion chain – the header file where s is found.

In this paper, we propose a way of *enhancing* currently existent search queries by taking advantage of build information and relations described in Section IV.2. By calculating the clusters for each symbol, we can walk in the footsteps of the compiler and recreate how the compiler bound and linked symbols together using the record of an executed build.

IV.5 Weighing symbol matches

Consider a set of symbols σ which has been appropriately pre-filtered by a “good-enough” filtering method, e.g. based on *mangled names* described in Section IV.1. For each of these symbols, we calculate the clusters in which a symbol is found, and if two symbols have at least one common cluster, i.e. they appeared in the same binary at least once, the relation between them is considered a **strong match**. A strong match definitely indicates that the compiler – in accordance with the language rules – did emplace a relation between the two symbols, e.g. a function call was linked to use the definition which was found in the same cluster.

On the contrary, if the two symbols never appear in a common cluster, the relation is considered a **weak match**. A weak match indicates that no *discovered* relationships enforce strength to the match, but the symbols could still be related to each other. A weak match could, e.g., indicate a function which is called from a dynamically loaded library.

We also define a third category called **indefinite match** for matches that are indefinite in their strength – no extra information could have been retrieved from calculating the clusters or the clusters do not exist at all. This could happen from the fact that certain translation units were not compiled or we have no build information (see Section IV.2) available. We will see in Section V.4 that certain matches will deliberately fall into the latter category due to optimisation reasons. The fact that a match does not have any strength indication proves not to hinder search efficiency and accuracy any further if a query is unambiguous based on previously available properties, such as mangled name equivalence.

The algorithm which categorises symbol references into the match weights is as follows:

0. A query (e.g. a “definition search”) is based on an input parameter (ξ), a particular occurrence of a symbol, e.g. a single function call in some translation unit.
1. The query is executed, and produces a set of symbols, which are *candidate results* – this process usually encompasses a mangled-name-based search. In the context of the previous “definition search” example, candidate results are definitions which match the call symbol’s mangled name.
2. **If and only if** the query results are ambiguous (based on language rules and the context of the query), we take clustering information into consideration for each candidate s and define *match strength* based

on the relation below. (Note that this relation is symmetrical and is applicable between every pair of symbols.)

Let $\mathcal{C}_\xi$ be the set of clusters in which ξ is found, and $\mathcal{C}_s$ the set of clusters for a particular candidate result s .

$$\mathcal{R}(\xi, s) = \begin{cases} \text{strong,} & \mathcal{C}_\xi \neq \emptyset \text{ and } \mathcal{C}_s \neq \emptyset \text{ and } \mathcal{C}_\xi \cap \mathcal{C}_s \neq \emptyset \\ \text{weak,} & \mathcal{C}_\xi \neq \emptyset \text{ and } \mathcal{C}_s \neq \emptyset \text{ and } \mathcal{C}_\xi \cap \mathcal{C}_s = \emptyset \\ \text{indefinite,} & \mathcal{C}_\xi = \emptyset \text{ or } \mathcal{C}_s = \emptyset \end{cases}$$

Consider the example depicted in Figure II.1. For this project, assume that a user searched for the definition of $F::f()$ from a function call in `main.cpp`. Having the symbols in the project filtered by the mangled name, there are three *candidate* functions, from which two are *strong matches* and one is a *weak match*.

Chapter V

Analysis of practical implementation approaches

Developing a practical solution for the problem described in this paper took many iterations for us. We decided to calculate symbol clusters at “parse time” to ensure that user interactions are only marginally slower. With each iteration we striven for further enhancing the practical benchmarks of the solution, most importantly to keep database size and parse time increase to a minimum and ensure that it scales well for even larger projects.

A common part of each solution was the method of *presenting* our findings to the user. The extra calculation practically involved lazily checking if a set of ID numbers – of `sizeof(size_t)` size – have at least one common element and did not increase search times in a measurable amount, but a naïve approach’s parse times were, initially, too high. We discuss our steps in refining our solution in the upcoming sections, with measurements showing their highlights and pitfalls.

V.1 Store clusters information for every AST node

CodeCompass uses LLVM/Clang to parse C / C++ projects and create an Abstract Syntax Tree (AST), which is then to extract information about the code itself. Our first approach was to store which AST node is found in which cluster for every AST node visited by the parser. Parsing a project involves registering the build actions (see Section IV.2) and then visiting the AST and storing information for every source file in the project. We calculated the clusters for each parsed source file and saved this set for each and every node in a new database table.

This made the previously described search enhancement very straightforward as we could instantly read the cluster information for the candidate nodes, but the time it took for a project to be parsed increased drastically, and so did the database's size.

Apache Xerces [23] has 123 882 AST nodes discovered by the parser which are related to function and type usage. The total count of different symbols appearing in different built binaries is 2 934 570, which is the number of new rows in the database. Calculating this information took 7 minutes (an increase of 100% compared to parse time without this new feature) and storing this extra data increased the database size from 190 MiB to 403 MiB, a 112% increase – this was considered to put an enormous strain on the process for larger projects by wasting processor time and thus we decided to discard this idea.

V.2 Store clusters for select AST nodes only

We have attempted to lessen the size requirement of storing this data by performing a cleanup routine after a project was parsed. To this end, we defined certain ambiguity criteria (see Section V.5) for symbols of a certain type and our cleanup procedure only kept information about AST nodes where the calculated information could *actually* contribute to enhancing the accuracy of queries.

For Xerces, we found that from the almost 3 million extra rows in the database, only 31 949 rows had the aforementioned property. This post-processing step decreased the final, persistent database size to a 35% increase (compared to the size without using this feature), however, this post-processing step incurred an extra 2 hours – after a 7 minute parse of Xerces – of time increase whilst it also institutionalised the fact that 99% of the rows were calculated only to be deleted in the end.

It was obvious to us that such increase in either space usage and time requirement would not scale well at all for any software project beyond a few mebibytes, like CodeCompass [7] itself or LLVM/Clang [21] or LLVM/Linux kernel [24].

V.3 Store clusters for every file

Utilising the fact that the compiler treats source code as a set of files interconnected instead of a set of AST nodes we have decided to shift our base scale to encompass this view. The parser stores the containing file for

Table V.1: Parsing performance of file-based clustering (Section V.3)

	LLVM/Clang	LLVM/Linux	Xerces
Build time	18.5m	1m 36s	37.5s
Parse time w/o clusters	267m 20s	118m 17s	3m 53s
~ w/ clusters	350m	124m	4m
DB size w/o clusters	14 120 MiB	1 706 MiB	193 MiB
# of files	47 931	54 213	3 336
# of build actions	2 274	2 017	648
# of cluster info rows	734 332	835 724	46 273
Database overhead	40 MiB (.0028%)	43 MiB (.025%)	2.5 MiB (.013%)

every AST node in the database. This change reduced the time it took to persist information about one translation unit greatly as there were only one set of cluster-information rows for each file instead of *num of nodes * num of clusters calculated*. Calculating a file’s cluster information in the context of a build action from which it is visited (see Section IV.4) stayed the same and there were no impacts on query performance either.

The shifting of base scale introduced the problem of deduplication: CodeCompass already efficiently stores AST node information only once instead of every time a certain node is visited (e.g. declarations in header files which are included in a myriad of locations) and for this purpose, it uses a hash-based ID caching system which was used to prevent duplication in our first approach described in Section V.1. We created a similar system for files and the parsing routine was changed to the one described below – IDs are unique `sizeof (size_t)` numeric scalars:

1. For each build action handled by the C/C++ parser, we calculated the $\mathcal{C}$ cluster (Section IV.4) into which symbols of the current build are propagated. This was represented as an `std::set` of cluster IDs.
2. The parser runs the parsing routines on the current build action, and we store every visited file’s unique ID in another `std::set` – this is the first step in deduplication if the same file is encountered multiple times in a build action.
3. Each file could have already been parsed at least once and thus have a set of clusters ($\bar{\mathcal{C}}$) already discovered and persisted. $\mathcal{C} - \bar{\mathcal{C}}$ is calculated with `std::set_difference` as the set of not-yet-persisted cluster information.

Table V.2a: Parsing performance of heuristical file-based clustering (Section V.4) – smaller projects

	Xerces	CodeCompass
Source size	26 MiB	103 MiB
Build time	37.5s	3m 7s
Parse time w/o clusters	1m 48s	25m 16s
~ w/ clusters	1m 50s	25m 38s
# of files	1 753	4 451
~ involved in clustering	45 (2.57%)	82 (1.84%)
# of build actions	648	663
# of cluster info rows	445	6 425
Database overhead	64 KiB (.0004%)	448 KiB (.000204%)

4. This difference is persisted in the database and cached in memory in a `std::map<FileID, std::set<ClusterID>>` data structure which is a direct mirror of the persisted database entries grouped by file ID.

We present the measurements of this process in Table V.1. The time requirement increase is clearly visible as we consider larger projects, with LLVM/Clang showing a 33% increase in parse times. We have profiled our implementation and discovered that even comparing only integers, the administration of the aforementioned data structures contribute to large amounts of the time increase. Our measurement tests were ran in a 24-core environment (with ~ 40 GiB of RAM on enterprise solid-state drives) so there were a maximum of 24 parsing threads allowed to run at any given time. We had to carefully make the implementation thread-safe as the information of clusters is a project-level one. Due to locking, on many occasions in the process of parsing LLVM/Clang as much as half of the threads were waiting for one thread to persist its clustering data as described in Step 3 and 4.

V.4 Store clusters for only certain files calculated heuristically

Drawing the conclusions from our previous approaches' efficiency, we developed our final solution which combines the ideas described in Section V.2 and V.3. We decided to keep the clusters' on file-level scale, but instead of calculating and storing the cluster information of every file we stored information for files which contain certain symbols.

Table V.2b: Parsing performance of heuristical file-based clustering (Section V.4) – larger projects

	LLVM	Linux	TSP [22]
Source size	328 MiB	642 MiB	1 109 MiB
Build time	18.5 m	1m 36s	—
Parse time w/o clusters	3h 32m 24s	36m 56s	2h 25m 17s
Parse time w/ clusters	3h 34m 16s	37m 9s	2h 26m 45s
# of files	8 346	6 676	103 601
~ involved in clustering	486 (5.82%)	1 163 (17.42%)	11 072 (10.69%)
# of build actions	2 262	2 017	65 558
# of cluster info rows	54 053	92 014	276 174
Database overhead	2 816 KiB (.000205%)	4 824 KiB (.00321%)	13 670 KiB (.00263%)

Taking into account that certain queries cannot benefit from clustering information for certain inputs – using a set of criteria (see Section V.5) we established in Section V.2 – files that only contain references to such non-ambiguous nodes need not have their clusters calculated and persisted. We have decided not to modify the currently implemented parsing process at all, but instead implement a project-level post-process routine which could be expressed with the algorithm as follows:

```

1:  $\Sigma \leftarrow \emptyset$ 
2:  $\mathcal{S} \leftarrow$  set of ambiguous mangled names
3: for all  $s$  in  $\mathcal{S}$  do
4:    $\Sigma \leftarrow \Sigma \cup \{f \text{ file} \mid f \text{ contains a symbol with mangled name } s\}$ 
5: end for
6: for all  $f$  in  $\Sigma$  do
7:    $\mathcal{B} \leftarrow \{\beta \text{ build action} \mid f \text{ is a "source-of" } \beta\}$ 
8:    $\mathcal{B}' \leftarrow \text{calc\_cluster}(\mathcal{B})$ 
9:    $\text{persist}(f, \mathcal{B}')$ 
10: end for

```

The algorithm takes the following steps to calculate the clustering information:

- (Line 2) A database query is executed which generates a set of mangled names which are definitely ambiguous by certain criteria (see Section V.5) – we need the entire project to have been successfully parsed to discriminate on this property.
- (Line 4) We create the list of files which contain references to symbols

having mangled names generated in Step 1.

3. (Line 7) The *entry build action* set $\mathcal{B}$ is calculated for every marked file f . The entry build action set for a file is the set of build actions for which the file was first discovered by the parser: it was either a “source-of” or a header included into a “source-of” file (see Section IV.2).
4. (Lines 8,9) These sets are formed into clusters with the procedure described in Section IV.4, and the resulting clusters are persisted in the database.

Calculating clusters for multiple build actions one-by-one and then taking the union of the result is the same – in terms of the result – as starting with a $\mathcal{C}$ containing all entry build actions, so we modified the `calc_cluster` algorithm (see Section IV.4) to accept a set of build actions instead of exactly one build action and initialise its inner $\mathcal{C}$ with this input set, which increased the efficiency of our solution.

The calculations of Σ , $\mathcal{S}$ and $\mathcal{B}$ (for each file) were joined together into a single SQL query which returns (FileID, ClusterID) pairs. Queries of this magnitude (containing multiple Common Table Expressions [25]) can be heavily optimised by database engines which gave our solution another

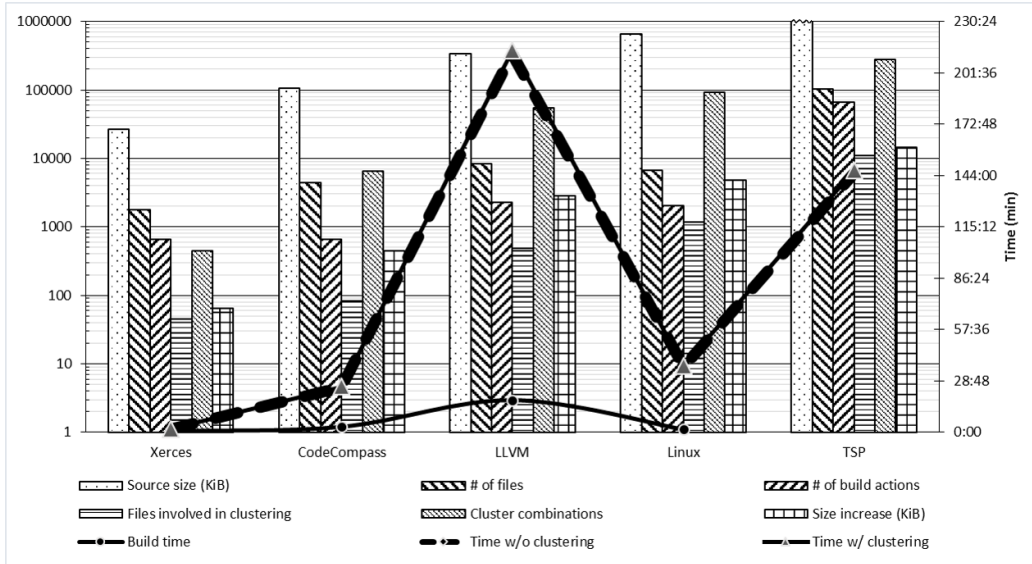


Figure V.1: Project measures and parsing performance of the solution described in Section V.4. The most important result is that the black solid and black dashed lines overlap each other – calculating the clusters virtually does not impact performance at all.

improvement in efficiency. Executing the equivalent algorithm written natively in C++ took CodeCompass 2 hours for Xerces (see Section V.2) while a single query produced the results in under 10 seconds. The `calc_cluster` algorithm is then executed for every $\mathcal{B}$ in an independent, multi-threaded setup which further increased efficiency.

We measured the parse time increase compared to without this feature on some sample projects and were satisfied with the results: the impact our feature did on parse time is virtually nonexistent despite the increase in the project’s complexity. The amount of files that needed the calculations described in this paper were considerable, but – as seen in Tables V.2a and V.2b – the number correlates with the inner layout and complexity of the source code. The measurements are depicted in the form a diagram in Figure V.1. Our measurements were ran focusing on the impact of the implementation described in this paper and we purposefully disabled other orthogonal capabilities (e.g. metrics calculation, Python parser, etc.) of CodeCompass.

As discussed previously in Section IV.5, certain information is lost in this solution compared to previous ones. Files which only contain symbols which are unique for the entire project (see Section V.5) are not considered at all for clustering as the *mangled-name*-based search is enough to produce necessary results. This is the reason behind the *indefinite match* category defined in

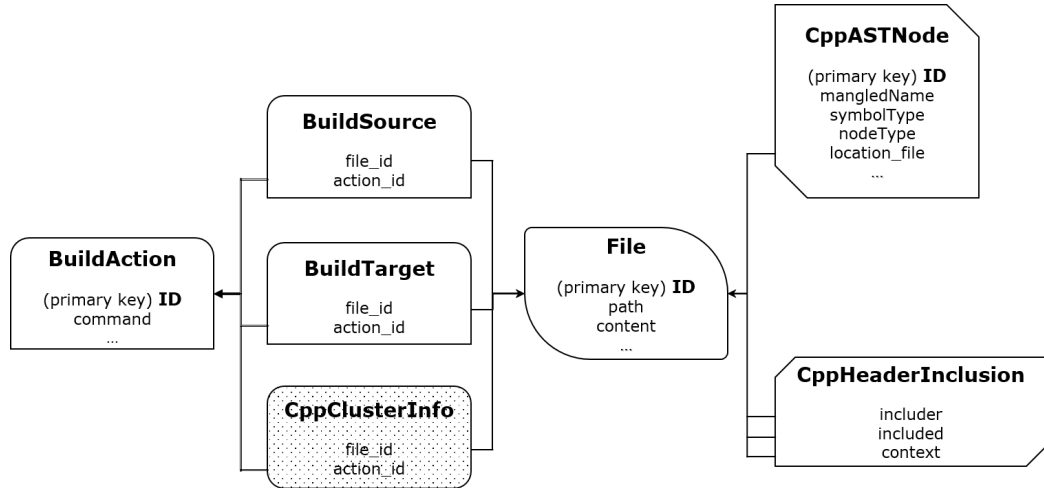


Figure V.2: The layout of how the information required by our solution is present in the `CppClusterInfo` table, connected with other tables in the CodeCompass database. – The relations described in Section IV.2 are stored in `BuildSource` and `BuildTarget`; the “header is compiled in a translation unit” relation (see Section IV.4) is stored in `CppHeaderInclusion`.

Table V.3a: Number of *functions* discovered and found ambiguous in our tests projects (in parantheses the number of ambiguous names and nodes)

	Functions				
	Names	Nodes	<i>Decl.</i>	<i>Def.</i>	<i>Usage</i>
Xerces [23]	18 547 (19, 0.1%)	63 167 (348, 0.5%)	14 933 (23%)	11 123 (18%)	37 111 (59%)
CodeCompass [7]	337 833 (143, 0.04%)	622 220 (473, 0.07%)	229 737 (37%)	127 864 (20%)	264 619 (43%)
LLVM [21]	1 793 978 (334, 0.02%)	4 545 086 (16 469, 0.4%)	1 213 123 (27%)	708 156 (16%)	2 623 807 (57%)
Linux [24]	70 579 (503, 0.7%)	411 691 (11 074, 2.7%)	33 746 (8%)	66 228 (16%)	311 717 (76%)
TSP [22]	410 946 (9 721, 2.3%)	2 091 578 (194 118, 9.3%)	377 393 (18%)	321 964 (15%)	1 392 221 (67%)

the aforementioned section: a function call to $f()$ – assuming that $f()$ is a uniquely defined function in the entire project – is most definitely linked with the only definition, any other combination would be considered a violation of the Language Specification and produce a compile/link error. Storing this information would be superfluous and we have thoroughly discussed and reviewed our criteria and decided that this loss is negligible compared with the efficiency gain.

The database layout of how the clustering information is connected with other tables in a project’s database of CodeCompass can be seen in Figure V.2.

V.5 Ambiguity criteria

The solution described in Section V.4 requires well-formed criteria by which it considers a mangled name ambiguous. We have researched the following criteria to be sufficient in our C++ context of large projects for enhancing search accuracy.

In Tables V.3a and V.3b, we show our measurements on how many of such problematic symbols were found for our test projects.

V.5.1 Functions

A function symbol $f()$ (mangled name in LLVM/Clang: $_Z1fv$) is considered ambiguous if neither of the following conditions hold true for all symbols having this particular mangled name in the entire project:

Table V.3b: Number of *types* discovered and found ambiguous in our tests projects (in parantheses the number of ambiguous names and nodes)

	Types				
	Names	Nodes	<i>Decl.</i>	<i>Def.</i>	<i>Usage</i>
Xerces [23]	2 849 (9, 0.3%)	60 715 (329, 0.5%)	2 248 (4%)	1 402 (2%)	57 065 (94%)
CodeCompass [7]	346 387 (1 315 (0.4%))	779 523 (9 461, 1.2%)	182 413 (23%)	169 124 (22%)	427 986 (55%)
LLVM [21]	1 628 232 (7 192, 0.5%)	4 206 743 (73 375, 1.8%)	860 245 (20%)	789 886 (19%)	2 556 612 (61%)
Linux [24]	19 883 (188, 1%)	159 581 (579, 0.3%)	1 990 (1%)	20 029 (13%)	137 562 (86%)
TSP [22]	196 374 (2 478, 1.2%)	2 108 217 (74 511, 3.5%)	62 914 (3%)	162 226 (8%)	1 883 077 (89%)

- The function has exactly one *definition* node – in this case, this definition would be *the* definition
- The function does not have a *definition* node, but has exactly one *declaration* node – in this case, *the* “definition” becomes this single declaration node
- The function has neither a *definition* node nor a *declaration* node – in this case, we do not present a “definition” to the user as none were discovered

V.5.2 Types

A type T (mangled name in LLVM/Clang: T) is considered ambiguous if it is *defined* more than once in the entire project. Types can be *declared* as many times as needed.

A header discovered to be included multiple times in the same translation unit does not constitute as multiple definitions in CodeCompass – however, without a *header guard*, it would constitute as such and violate the language rules in a single compile/link action.

Chapter VI

Summary

VI.1 Related work

Producing clusters by decomposing large-scale software and observing the build toolchain of a project is not unknown. Richard C. Holt emphasises on the importance of linkage information when it comes to extracting software models in [26]. Their approach uses the information retrieved from the linker to resolve dangling references in software models created by extractors, such as [27]. Our research was centered on the code comprehension of large-scale systems with a well-scaling algorithm on the source code level, using hard facts retrieved from the toolchain.

Van der Burg *et al.* implemented a way of checking the license compliance of software products that use a variety of open-source software components by recording the steps taken when the product was built [28]. Our approach is similar to theirs in terms of using exact build informations, however, our goal was to show the connection between symbols in the built components.

VI.2 Future work

We would like to increase our data by testing our solution on more projects. In terms of further developing our solution, a few routes are available and yet to be taken by us.

The approach we have taken has proven robust enough to effectively be used in CodeCompass. We aim to consider the development of methods which aid in the discovery and modelling of dynamic type information and polymorphic reference bindings, which will further enhance the accuracy of our clustering method. The consideration of dynamically referenced symbols goes further than dynamic polymorphism in the project, we also aim for

increasing the accuracy by discovering symbols in shared libraries and pure dynamically loaded objects, e.g. those referenced through *dlopen* ().

We have also discussed a potential way to reduce the size of the extra data stored by excluding code with known semantics, e.g. parts of the Standard Template Library from the ambiguity check process.

The development of further heuristics to ensure a finer-grained way of weighing matches (see Section IV.5) is expected to decrease the number of false positive *weak matches*.

A problem similar to that described in Section II.3 can be applied to other programming languages, e.g. in *Java*, different `classpath`s could be considered as “linkage” information and the question of “How symbols reference each other?” is applicable for packages built with different configuration in a large-scale project.

VI.3 Conclusion

In this paper we investigated the ambiguity of symbol resolution in large C/C++ programs. Resolving symbol references is the base of many essential functionalities implemented in software development tools, code comprehension, and static analysis systems. These actions include finding the exact definition of a function call, listing all the call sites of a given function or the place of usage of a given variable, or where we used a certain type.

Most of the available development environments and comprehension tools implement these features for C/C++ based on mangled names. Mangled names are used by the linker to distinguish different symbols with the same name, e.g. in case of overloading resolution. Mangled names are much more specific than the name of the object.

Unfortunately, using mangled names only may lead to ambiguity when different symbols with the same mangled name are defined in different translation units and assembled to different binary executables. This is not a violation of the One Definition Rule of C++ – that forbids the multiple definition for a single translation unit only. In fact, this problematic scenario happens quite often for large systems compiled into different platforms (e.g. both 32 and 64 bit) as well as systems with extensive binary set of examples and test cases. Current state of the art tools are ignoring this problem which we found to affect a considerable ratio (0.5 – 10%) of symbols.

We defined a clustering algorithm based on build actions – essential information on how the system was built. Based on these clusters, we can reason whether symbols with the same mangled name are identical or likely different. Although, the method leaves some uncertainty for cases where symbols are

resolved dynamically – e.g. via `dlopen` – in practice, it dramatically increases the accuracy of the symbol resolution.

We implemented our method as part of the open source CodeCompass code comprehension tool. CodeCompass utilises the LLVM/Clang compiler infrastructure and capable to analyse, and present to the user, systems in the gigabyte scale. Various implementation methods were analysed to achieve the minimal run-time and storage overhead. As our method is based only on information commonly available for other tools too, it can be used to increase the quality of most development environment and code comprehension frameworks.

References

- [1] Microsoft Visual Studio, version 14.0.24720.00-Update1
<https://www.visualstudio.com/>, 08/15/2016.
- [2] CLion: a cross-platform IDE for C and C++, version 2016.2.1
<https://www.jetbrains.com/clion/>, 08/15/2016.
- [3] The Netbeans IDE, version 8.1
<https://netbeans.org/>, 08/15/2016.
- [4] The Eclipse project, Eclipse CDT (C++ Development Tools), version *Neon* 4.6.0
<http://www.eclipse.org/home/>, 08/15/2016.
- [5] Woboq code browser, version 2.0.1
<https://woboq.com/codebrowser.html>, 08/15/2016.
- [6] OpenGrok home page, version 0.13-rc2
<https://opengrok.github.io/OpenGrok>, 08/15/2016.
- [7] The CodeCompass project,
<https://github.com/Ericsson/CodeCompass>, 08/15/2016.
- [8] James Norton: *Dynamic Class Loading for C++ on Linux*, Linux Journal, 2000, 73es, Article 38 (May 2000).
<http://www.linuxjournal.com/article/3687>
- [9] JSON compilation database format specification,
<http://clang.llvm.org/docs/JSONCompilationDatabase.html>,
08/15/2016.
- [10] Bjarne Stroustrup: *The C++ Programming Language*, 4th ed.
Addison-Wesley, ISBN 978-0321563842, 2013.

-
- [11] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Section 7.2.1c. ISBN 0-201-51459-1, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 1990.
 - [12] Bjarne Stroustrup: *Design and Evolution of C++*, Addison-Wesley, ISBN 0-201-54330-3, 1994.
 - [13] The C++14 Standard, ISO International Standard, ISO/IEC 14882:2014 — Programming Language C++, §1.3 “Terms and definitions”, 2014.
 - [14] The C11 Standard, ISO International Standard, ISO/IEC 9899:2011 — Programming Language C, 2011.
 - [15] Format of a C decorated name,
<https://msdn.microsoft.com/en-us/library/x7kb4e2f.aspx>, 08/15/2016.
 - [16] Jonathan Sillito, Gail C. Murphy, Kris De Volder: *Asking and Answering Questions during a Programming Change Task*, IEEE Transactions on Software Engineering, VOL. 34, NO. 4, July/August 2008.
 - [17] The C++14 Standard, ISO International Standard, ISO/IEC 14882:2014 — Programming Language C++, §3.2 “One Definition Rule”, 08/17/2016.
 - [18] The C++14 Standard, ISO International Standard, ISO/IEC 14882:2014 — Programming Language C++, §3.4 “Name lookup”, 08/17/2016.
 - [19] Rudolf, Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy: *Columbus Reverse Engineering Tool and Schema for C++*, In Proc. 18th Int’l Conf. Software Maintenance (ICSM 2002), pp. 172–181., Oct. 2002.
 - [20] B. De Alwis and G.C. Murphy: *Using Visual Momentum to Explain Disorientation in the Eclipse IDE*, Proc. IEEE Symp. Visual Languages and Human Centric Computing, pp. 51–54., 2006.
 - [21] Clang: a C language family frontend for LLVM,
<http://clang.llvm.org>, 08/15/2016.
 - [22] Ferraro-Esparza, Victor, Michael Gudmandsen, and Kristofer Olsson: *Ericsson Telecom Server Platform 4*, Ericsson Review 3 (2002) pp. 104–113., accessed 08/15/2016.

-
- [23] Apache Xerces C++, part of Apache Xerces Project, git version 26f8004
<http://xerces.apache.org>, 08/15/2016.
 - [24] LLVM/Linux: GNU/Linux kernel compatible for LLVM/Clang builds,
git version 230b22d
<http://llvm.linuxfoundation.org>, 08/15/2016
 - [25] M. Burzańska, K. Stencel and P. Wiśniewsky: *Pushing Predicates into Recursive SQL Common Table Expressions*, Advances in Databases and Information Systems (ADBIS), 13th East European Conference, Sept. 2009.
 - [26] Jingwei Wu, Richard C. Holt: *Resolving Linkage Anomalies in Extracted Software System Models*, Proceedings of the 12th IEEE International Workshop on Program Comprehension, pp. 241–, 2004.
 - [27] R. Ferenc, Á. Beszédes, F. Magyar and T. Gyimóthi: *A short introduction to Columbus/CAN*, Technical Report, 2001.
 - [28] van der Burg, *et al.*: *Tracing Software Build Processes to Uncover License Compliance Inconsistencies*, Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 731–742., 2014.