EXTRACTION AND VISUAL EXPLORATION OF CALL GRAPHS FOR LARGE SOFTWARE SYSTEMS

HESSEL HOOGENDORP

university of groningen / faculty of mathematics and natural sciences / computing science

February 2010

Hessel Hoogendorp: *Extraction and visual exploration of call graphs for large software systems,* © February 2010

SUPERVISOR:
prof. dr. A. C. Telea

SECOND SUPERVISOR:
prof. dr. M. Aiello

LOCATION:
Groningen

ABSTRACT

Oftentimes, developers need to understand a software system they are unfamiliar with, for instance, to perform maintainance or refactoring work. Since large software systems are hard to understand, having proper tooling can significantly reduce the time a developer needs to get a firm understanding of the system.

Understanding the dependencies among the different components of a software system is one of the most important and one of the most challenging tasks in software (re)engineering. Function calls from one function to another are important in this respect, because they represent direct, functional dependencies between different components of the system. Having a correct and complete call graph of a software system can be a powerful aid, since it makes these call relations explicit and, to some extend, models the structure and behaviour of the system.

There is a lack of robust, scalable and effective support for call graph computation and visual analysis for the C++ programming language. The complex nature of C++ and the relatively large size of C++ industrial code bases makes static analysis difficult and the fast extraction and visualization of their corresponding call graphs challenging. In particular, C++ allows a complex range of semantics for function calls (operators, virtual functions, implicit calls and explicit calls). All these have to be extracted and suitably presented to the developer for optimal understanding.

A design and implementation is given of a new system that automatically extracts call graphs from large C++ code bases and the problems that one faces when building such a system are discussed. Also, a comparison is made between three existing ways to visualize the resulting call graphs and the application of the toolchain using the most suitable of these visualization methods is presented to the reader.

CONTENTS

1 INTRODUCTION 1
- 1.1 Definitions 1
- 1.2 Problem statement 3
- 1.3 Structure of this thesis 4

I CONSTRUCTION OF CALL GRAPHS 5

2 INTRODUCTION TO CONSTRUCTION OF CALL GRAPHS 7
- 2.1 Graph construction requirements 7
- 2.2 Existing call graph constructors 9
- 2.3 Overview of the call graph construction pipeline 11

3 EXTRACTION OF CALL INFORMATION 13
- 3.1 Preprocessing and parsing 15
- 3.2 Extraction 15
- 3.3 Filtering 28
- 3.4 Name mangling 36
- 3.5 Validation 39
- 3.6 Serialization 40
- 3.7 Complexity 41

4 CALL GRAPH CONSTRUCTION 43
- 4.1 Deserialization 45
- 4.2 Function mapping 46
- 4.3 Constructing call graphs 48
- 4.4 Serialization 68
- 4.5 Complexity 70

5 AUTOMATING EXTRACTION AND CONSTRUCTION 73
- 5.1 Retrieving preprocessor parameters and build targets 73
- 5.2 A solution using compiler wrapping 74

II VISUAL EXPLORATION OF CALL GRAPHS 79

6 CALL GRAPH VISUALIZATION CANDIDATES 81
- 6.1 Graph visualization requirements 81
- 6.2 Visualization candidates 81

7 APPLICATION OF THE TOOL CHAIN 89
- 7.1 Running CCIE and CCC 89
- 7.2 Visual exploration using SolidSX 89

III CONCLUSION 99

8 CONCLUSIONS AND FUTURE WORK 101
- 8.1 Evaluation 101
- 8.2 Future work 102
- 8.3 Final words 104

IV APPENDIX 107

A APPENDIX A 109
- A.1 The g++ wrapper script 109
- A.2 Source code of the *fib* program 111
- A.3 Source code of the *precalc* program 112

BIBLIOGRAPHY 125

LIST OF FIGURES

Figure 1 The static call graph of the example program. Nodes are function definitions and edges are function calls. 2
Figure 2 The complete call graph construction pipeline. 12
Figure 3 The preprocessing, parsing and extraction steps that will be covered in this chapter. 13
Figure 4 The preprocessing and parsing steps. 15
Figure 5 The extraction step. 16
Figure 6 The representation of a nested class. Nested classes are 'flattened' in the hierarchy. 19
Figure 7 The call graph resulting from the simple function call. 22
Figure 8 The call to the virtual m yields two potential call targets. 23
Figure 9 The call via pointer-to-function yields three potential call targets. 24
Figure 10 The filtering step. 28
Figure 11 The call graph one might expect for the Hello World program. The blue nodes are folders and files, the blue edges are containment relations, the green nodes are function and the green edges are function calls. 29
Figure 12 The actual call graph for the Hello World program without filtering. The blue and green nodes and edges form the expected call graph from figure 11. All other nodes and edges are unexpected: Red nodes represent function, red edges represent calls, purple nodes represent containment nodes (directories, files and classes) and purple edges represent containment relations. 30
Figure 13 The call graph for the Hello World program after filtering. The blue and green nodes and edges form the expected call graph from figure 11. The coloring of the other nodes and edges is as it is in figure 12. 31
Figure 14 The name mangling step. 37
Figure 15 The validation step. 39
Figure 16 The serialization step. 40
Figure 17 The call graph construction steps of the pipeline. 44
Figure 18 The deserialization step. 45
Figure 19 The building of functions maps. 46
Figure 20 The call graph construction step. 48
Figure 21 The call graph including the fictional Root node. 54
Figure 22 The partial call graph starting in the `main` function of the Hello World program. 60
Figure 23 The containment nodes and edges of the Calculator program (in blue). The function nodes and function call edges are shown in green. 65
Figure 24 The serialization step. 68
Figure 25 The tiny Fibonacci program layed out with the dot algorithm. 83
Figure 26 The small precalc program layed out with the dot algorithm. 83

Figure 27 The small precalc program (left) and the average/large Bison program (right). Both call graphs are layed out with the fdp algorithm. 84
Figure 28 The tiny Fibonacci program (left) and the small Precalc program (right). Both call graphs are laid out with the Improved Walker algorithm 85
Figure 29 The average/large sized Bison program. On the left the call graph layed out with the Improved Walker algorithm. On the right the call graph layed out with the GEM algorithm. 85
Figure 30 The very large Mozilla Firefox program. The call graph has been layed out with the Improved Walker algorithm. 86
Figure 31 The tiny Fibonacci program (left) and the small Precalc program (right). 86
Figure 32 The average/large sized Bison program. On the left the call dependencies between all functions. On the right the calls made in the main function. 86
Figure 33 The very large Mozilla Firefox program. On the left, the call dependencies of all directories of the code base's root directory. On the right the call dependencies of the main function. 87
Figure 34 Left: The initial presentation of the call graph of Mozilla Firefox. Center: The first subdirectories of the root directory. Right: The contents of the root directory of the Firefox code base. 90
Figure 35 The connectedness of the `XPCOM` (left), `parser` (center) and `security` (right) subsystems. 90
Figure 36 All implementations of `QueryInterface`. 92
Figure 37 All edges coming from the subsystems and going to the various instances of the `nsCOMPtr` template class. 93
Figure 38 Left: All edges between the subsystems and the various instances of the `nsCOMPtr` template class. Right: Only the edges coming from the subsystems and going to the various instances of the `nsCOMPtr` template class. 94
Figure 39 The usage of `XPCOM` objects by the `security` (left), `rdf` (center) and `widget` (right) subsystems. 94
Figure 40 The usage of `XPCOM` objects by the `editor` (left), `db` (center) and `parser` (right) subsystems. 95
Figure 41 All nsCOMPtr instances that are used by the subsystems are selected. 96

LIST OF TABLES

Table 1 The worst-case and expected-time complexities of the different phases of the extraction process. 41
Table 2 The different types of functions calls that exist and their properties. 50
Table 3 The complexities of the possible scenarios of `linkToFunctions`. 53

Table 4 The worst-case and expected-time complexities of the different phases of the graph construction process. Here, N_F is the number of functions, N_C is the number of function calls and $N_{N,Cont}$ is the number of containment nodes in the graph. 70
Table 5 The relevant GNU compiler tools and their corresponding wrapper scripts. 76

ACRONYMS

CCIE C/C++ Call Info Extractor

CCC C/C++ Call graph Constructor

AST Abstract Syntax Tree

FQN Fully Qualified Name

EFT Extended Function Type

HEB Hierarchical Edge Bundles

1 INTRODUCTION

Oftentimes, developers need to understand a software system they are unfamiliar with, for instance, to perform maintenance and refactoring tasks. Since large software systems are hard to understand, having proper tooling can significantly reduce the time a developer needs to get a firm understanding of the system and do maintenance work.

Call graphs are well-known instruments for understanding complex software systems and can be of great help in maintaining a system (see [28] and [33]). They can, for instance, assist in identifying modularity problems, which in turn help identify possible maintenance bottlenecks. Or, as another example, call graphs can aid the porting of a system, by visualizing the components that depend on code that must be rewritten or removed.

Call graph construction tools are used to compute such graphs from the source code of existing systems. After extraction, several visualization tools can be used to enable the interactive exploration of the extracted graphs.

However theoretically well understood, there is significant lack of robust, scalable and effective support for call graph computation and visual analysis for the C++ programming language. The very complex nature of C++, and the relatively large size of C++ industrial code bases, makes static analysis difficult and the fast extraction and presentation of such call graphs challenging.

In the next section (1.1) we will define more precisely what type of graph we are dealing with in this thesis. After that, in section 1.2 we will describe what we aim to obtain and why this is such a challenging problem. The last section (1.3) of this chapter will give an overview of the structure of the thesis.

1.1 DEFINITIONS

Before we move on to explain the difficulties that arise when attempting to construct a static call graph, we first explain the concept of a static call graph. A static call graph is a graph in which the nodes are function definitions (or declarations) and the edges are static call relations. For example, consider the following trivial program:

```
void foo()
{
    bar();
}

void bar()
{
}

int main()
{
    foo();
    bar();
    return 0;
}
```

It is easy to see that the above program contains three function definitions and three function calls. Now, the static call graph corresponding to this program is depicted [1] in figure 1.

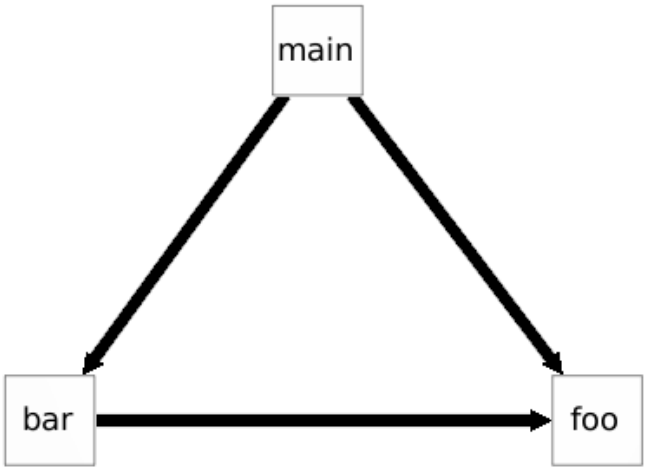

Figure 1: The static call graph of the example program. Nodes are function definitions and edges are function calls.

Since this thesis is concerned with *static call graphs*, it is important to highlight a number of implications that follow from that fact:

- We are interested in *static* call graphs, which means that the call relations we extract correspond to occurrences of function calls *in the source code*.

  This is opposed to a *dynamic* call graph, in which call relations are extracted from actually *running the program*. The result is a call graph containing only edges for those function calls that have been made during the execution of the program. The resulting edges may be annotated with information, such as the number of times that a call has been made, or temporal information indicating when or in what order calls have been made. It is important to note that in this thesis we study only *static* call graphs, not dynamic call graphs.

  The relevance of studying static call graphs becomes clear when one considers its applications. A static call graph of a system can be an invaluable tool to aid in, for instance, reverse engineering and software maintenance. A significant advantage of static call graphs over dynamic call graphs is that they can be constructed even when we are not able to run the system, which might be the case if we do not have access to the right (hardware or software) platform, or when we do not have a good idea of the input parameters to run the system with. Also, a static call graph contains *all* call relations, not just those that were actually executed. Having all call relations available is ofcourse important in refactoring and re-engineering tasks and obtaining such a complete call graph is very hard using runtime analysis, unless we achieve a code coverage of 100%.

- In our call graph definition, we consider all types of function calls that exist in C/C++. These include classic function calls (stemming from C), calls to virtuals, operators, constructors and destructors. Next to that, we are

1 All images of graphs throughout this thesis have been created using Tulip [23], unless stated otherwise. Tulip is discussed in 6.2.2.

interested in both explicit calls (the calls that are visible at the syntactic level) and implicit calls (which are added by the compiler but have no explicit syntax, such as calls to default constructors from inherited classes).

- A pure call graph would consist only of function (definition and declaration) nodes and call edges. However, C/C++ programs generally have much more structure than just a set of functions calling each other. Functions are typically grouped in a hierarchy of folders, files, namespaces and classes. Virtually all understanding problems that involve call graphs hugely benefit from a combination of the call relations with hierarchical information. As such, we extend the model of our call graph to include this hierarchical information. Formally, our target graph is thus a *compound, directed graph*, or a graph in which two types of relations exist: call relations and containment (hierarchical) relations. In this thesis, whenever we refer to a *call graph*, we are talking about such a *compound, directed, call-and-containment graph*.

- A call graph in itself only describes (part of) the *structure* of a given system. However, for understanding tasks, more is needed than just the call and containment relationships between functions, files, classes and folders. For example, necessary additional information about functions and function calls that should be available in the graph includes the name, full signature, location in the source code, visibility (public, protected, private) and the type of declaration (such as static, virtual or inline). Such information is invaluable when performing different types of analyses. We call such information *attributes*. Both the nodes and the edges of the call graph can be annotated with attributes.

1.2 PROBLEM STATEMENT

The aim of this thesis is twofold:

1. Build a (set of) program(s) that is able to construct directed, compound containment-and-call graphs, annotated by a rich set of static attributes, from a given C/C++ code base. The precise requirements that this (set of) program(s) must satisfy are stated in section 2.1 and the requirements that must be satisfied by the constructed graphs are stated in chapter 4.

2. Present a visualization method that allows a developer to visually explore the graphs constructed by the above (set of) program(s) in an intuitive way. The exact requirements to this visualization method are stated in section 6.1.

There are three primary reasons why constructing and visualizing such graphs is a difficult and challenging task, namely:

1. *Industrial-sized code bases are very large*. Because of this, it can take a long time to generate a call graph for such a system. Then, when the call graph is available, there is the challenge of being able to visualize and navigate the graph in a timely fashion. Obviously, the speed of a call graph constructor/visualizer is an issue.

   Also, generating a call graph for large code bases tends to be a very memory consuming and processor intensive task. It follows that stability and scalability are important properties of a call graph constructor. The same goes for the program that visualizes the graph.

   The most common method to keep a large software system manageable is to split it up into smaller components. Such components can be static libraries,

dynamic libraries or translation units. So, to be able to generate a call graph of the complete system, the call graph constructor must be able to operate across such component boundaries. This implies that the scope of the call graph constructor is an important factor.

Lastly, a large code base is by itself hard to understand. Adding to that is the fact that it is non-trivial to visualize large graphs in an understandable fashion. So, whether the graph visualizing program is able to produce an understandable view of the graph is a big issue.

2. *C++ is a complex language*. Call dependencies between functions can occur in many different forms. Some of these types of function calls are even hidden from sight, but they exist nonetheless. So, for a good understanding of the program, it is important that the call graph constructor is able to detect all of the different kinds of function calls.

   A powerful feature of C++ is its ability for run-time binding of function calls to functions. The drawback of this is that it can be very hard, and sometimes even impossible, to pin-point call targets using only static analysis. This shows that a call graph constructor's completeness and accuracy in finding the call targets of function calls are two more non-trivial aspects that must be dealt with.

3. *Call graph construction and visualization tools are hard to utilize*. The reason for this comes from a combination of factors. For instance, programs using static analyzer technology with limited capabilities, and the variations between C/C++ dialects make it hard to get accurate and complete results. Also, it is often difficult to successfully apply the tools to incorrect and incomplete code bases and complex build schemes make it hard to determine the settings with which the tools should be run. This illustrates that the usability of a call graph constructor is another important and complex issue.

1.3 STRUCTURE OF THIS THESIS

Excluding the appendix (A), this thesis is comprised of three parts.

Part i of this thesis deals with the construction of call graphs. In chapter 2 we will first discuss what the requirements are for a call graph constructor and what call graph constructing programs currently exist and how well they satisfy the stated requirements. The next chapter (3) deals with the extraction of all the information that is needed to construct a call graph and chapter 4 discusses how to actually construct a call graph from that information. Part i is concluded by chapter 5, which shows how we can automate the entire call graph construction process to build graphs for systems build with the GNU compiler tools.

Part ii is concerned with the visual exploration of the call graphs we constructed in part i. Again, we begin with discussing the requirements that any visualization method should satisfy and investigating some well known candidates for visualization (chapter 6). Then, using the most suitable visualization candidate, we will show the application of the entire tool-chain on a non-trivial, real-world software system (chapter 7).

Part iii of this thesis will reflect on what has been done, what requirements of the call graph construction and visualization systems have been satisfied and what requirements remain unsatisfied. We finish with a discussion on the future work that remains to be done.

Part I

CONSTRUCTION OF CALL GRAPHS

2 INTRODUCTION TO CONSTRUCTION OF CALL GRAPHS

For a developer to be able to quickly get a proper understanding of a software system, the call graphs presented to him must be of good quality. Intuitively, this means that each function call that is made in the source code, is represented in the call graph by two nodes and a connecting edge. The first node would represent the function *from which* the call is made and the second node would represent the function *to which* the call is made. The edge should be directed from the *calling function* to the *called function*. As will become clear in a later chapter, constructing a call graph that strictly adheres to these properties is not always easy. Even worse, since C++ supports dynamic binding ([29]), and we will only be performing static analysis, it is sometimes even impossible to construct such a call graph. However, as call graphs become less complete, or more inaccurate, they become less useful for a developer. So, for call graphs to be as useful as possible, they should be as complete and accurate as possible.

As we stated before, the C++ programming language is a relatively complex language compared to other languages. Its standard has evolved over the years and many different dialects are in widespread use. To be able to construct call graphs for a large part of the C++ source code that exists in the wild, a parser is needed that accepts a large set of the C/C++ dialects that are used in practice. Next to that, the parser needs to have full support for the C/C++ language; it needs to be able to handle all of the language's features.

From these implicit requirements, it becomes clear that the call graph construction system must be robust, fast, scalable, easy to use and must deliver call graphs that are as complete and accurate as possible. The next section (2.1) will make these requirements explicit. That section is followed by section 2.2, giving a short enumeration of existing call graph constructing programs and an evaluation of how well they satisfy the listed requirements. Lastly, this introductory chapter is concluded by section 2.3 with an overview of the steps needed to obtain a full call graph construction system.

2.1 GRAPH CONSTRUCTION REQUIREMENTS

The requirements for the call graph construction component of the software are as follows:

1. *Scalability*. The program must be able to extract call graphs of large, real-world code bases having hundreds of thousands up to millions of lines of code and it must be stable in doing so (i.e., it must not crash on large input).

2. *Efficiency*. The program must be able to extract the call graph in reasonable time. As a rough quantification, the program must take no longer than the order of time required to do a compilation of the code base with an efficient C/C++ compiler.

3. *Completeness*. The program must be able to find all function calls. More specifically, it must be able to find:

    - Standard function calls: plain C-style function calls and C++-method calls.

- Constructor and destructor calls: Calls to constructors and destructors by explicitly creating or destroying an object.
- Implicit constructor and destructor calls: For example, calls to constructors and destructors caused by passing an object instance as a parameter, or returning an object instance as a return value, or an object instance going out-of-scope.
- Pointer calls: Calls via pointer-to-function or pointer-to-member.
- Virtual calls: Calls to C++ virtual methods via a pointer to an object or a reference to an object.
- Intializing and finalizing calls: Calls made before and after the execution of `main`, such as the constructor and destructor of a global object variable.
- Operator calls: Calls to C++ overloaded operators.

Next to that, the program must be able to resolve each function call to a function, or a set of functions in case it is not possible to determine the exact function that is called. The latter can happen, for instance, when making a call via a pointer-to-function. The resolved set of functions must be as complete as possible, i.e., whenever possible it should be guaranteed that the function that is actually called is present in the set of call candidates. Lastly, resolving function calls to functions must be possible across translation unit and library (both static and dynamic) boundaries.

4. *Correctness*. The program must resolve each function call to the correct (set of) function(s). When there are multiple call candidates, that set of candidates must be as small as possible. Also, all extracted call relations should actually take place in the source code and should not be inferred by heuristics.

5. *Robustness*. The program must be able to accept incomplete or incorrect source code as part of its input. That is, although it is obvious that syntactically or semantically erroneous code will produce gaps in the call graph, the program should proceed as much as possible in delivering a correct and complete call graph from the information that is available.

6. *Source code based*. The program must only depend on source and header files. That is, the program should not be dependant on executables, object files or debug information files to be able to perform its analysis.

7. *Genericity*. The program must be able to handle a wide range of C/C++ dialects.

8. *User friendliness*. The program should be as easy to use as an equivalent build system on the target platform. That is, a developer should be able to run the system on large projects consisting of hundreds of files, organized in different build targets (i.e., executables or static or dynamic libraries) and compiled by complex build processes (such as makefiles). Running the system on such code bases should be no more difficult than performing a regular build of the code base.

9. *Open source*. The program should, preferrably though not mandatory, be open-source. This will allow us to make modifications where required and will easily allow further development of the tool-chain in the future.

It should be noted that, although these different types of requirements live in isolation, none of them *alone* is sufficient for a program that fits our goals. To have a truely effective and efficient call graph construction solution, *all* requirements should be satisfied. It should also be noted that these requirements are very similar to the ones discussed in [25].

2.2 EXISTING CALL GRAPH CONSTRUCTORS

Static call graph extraction is an old problem. Tens of different solutions exist for this process, of which many also work for the specific context of the C/C++ languages. However, we argue that it is hard to find a solution that complies with *all* the requirements that we stated in section 2.1 for our goal. In this section we review a number of well-known C/C++ development environments and C/C++ static analyzers. Each of these tools is capable of constructing call graphs, to some extend. They are very relevant for our review, because the requirements of a call graph constructor will be part of the requirements of each of these tools. Here, we will outline the limitations of these tools in the light of our requirements.

2.2.1 *Eclipse CDT*

The Eclipse C/C++ Development Toolkit [4] is part of the Eclipse project [3] and a fast growing tookit for C/C++ syntactic and semantic analysis. It quickly provides developers with local information from large code bases, while parts of that code base undergo editing. Primarily, it provides interactive search features for C/C++ developers in Eclipse.

Eclipse CDT is fast and stable in analyzing large code bases, it is able to resolve function calls to functions over translation unit and library boundaries and it is easy to use. However, it does not satisfy the completeness (3) and correctness (4) requirements. For example, it is not able to find all types of constructor calls and it does not find any implicit function calls (e.g., caused by passing an object as a parameter). Also, it redundantly lists all overriding functions in the case of a call to virtual function on a non-pointer, non-reference object instance.

Next to that, Eclipse CDT utilizes a limited preprocessing step which causes a header file to be preprocessed only the first time that it is encountered in a project. Hence, header files whose context depends on macros, which may have different values before inclusion, and which may be included multiple times with different macro values set, will not be analyzed correctly. Obviously, this produces potentially incorrect call graphs.

2.2.2 *Visual Studio 2008*

Visual Studio 2008 [18] is a comprehensive IDE for a range of programming languages, including C/C++. Like Eclipse CDT, it provides developers with local information from code bases, while they are being edited.

Also like Eclipse CDT, it is fast and stable when processing a large code base, able to look across translation unit and library boundaries and easy to use. Unfortunately, Visual Studio 2008 also does not satisfy the completeness and correctness requirements (3 and 4). For example, it is not able to find any constructor calls at all. Its limitations are inherent to the way Visual Studio 2008 extracts call graph information: Rather than using the compiler's parser and type checker, it uses a second, lightweight, parser to extract call information. Although advantageous as this makes it very fast, since it does not require full parsing and type checking, this technique is limited to finding only a subset of the syntactic constructs which correspond to function calls. Next to that, Visual Studio 2008 is closed source software, which makes it hard for us to adapt it to our needs.

2.2.3 *Doxygen*

Doxygen [2] is a documentation system for a number of programming languages, including C/C++. Its primary functionality is to generate understandable technical documentation from source code. Part of this functionality is the ability to generate call graphs for individual functions. Doxygen uses a simple parser which cannot handle the complex lookup and scoping rules of C++, which is why it often delivers incomplete (e.g., no constructor and destructor calls at all) and incorrect information.

2.2.4 *Rigi*

Rigi [12] is a well-known toolkit for reverse engineering, with plugins for C/C++ (among others). It has a tool which extracts call graphs from C/C++ programs, but it is rather limited in correctness and completeness, for the same fundamental reasons that Doxygen is limited: It does not implement a full semantic analyzer. Moreover, the C++ plugin uses the parser from the IBM VisualAge C++ [17] compiler, which is proprietary software and is only supported on the AIX platform and a small number of Linux distributions.

2.2.5 *MCC*

MCC [37] is a relatively good fact extractor for C/C++ and it works very fast. However, it does not fully support the C++ language. For example, it cannot analyze the sometimes more complex constructs present in typical system headers. Also, MCC seems to have some semantic analyzer limitations. These restriction make that it does not deliver a correct and complete call graph.

2.2.6 *Columbus*

Columbus [24] is an industry-strength parser and fact extractor for C/C++. It appears to support the entire C++ language quite well, and as such it is able to deliver correct and complete information. However, the tool is closed source, which makes it impossible for us to adapt it in case we want to further analyze and filter its raw output.

2.2.7 *KDevelop*

KDevelop [10] is a C/C++ development environment available for a variety of platforms and it comes with a standalone C/C++ parser. The goal of the KDevelop parser is relatively similar to that of the CDT parser: To provide information to developers during the development cycle, such as code completion and symbol-to-definition relations. Overall, the KDevelop parser is fast, relatively well documented, supports a wide class of C/C++ dialects and is robust against incorrect and incomplete code. However, at the time of inception of this project, the KDevelop C/C++ parser did not provide sufficient semantic analysis information to extract a call graph. Although recent additions added such functionality, the semantic analysis is still under heavy development and it is expected that a mature and stable C/C++ semantic analyzer will not available in KDevelop in the short term.

2.2.8 *Elsa/Oink*

The Elsa [35] C/C++ parser is part of the Oink [36] static analysis framework. Elsa provides scalable, robust, correct, and complete static analysis for a wide family of C/C++ dialects and is carefully engineered for high performance. It also supports, up to some extend, analysis of incomplete and/or incorrect code bases. Elsa comes with a complete and stable semantic analyzer as an open source toolkit. It is properly documented and maintained.

2.2.9 *Choosing a suitable system*

The review above confirms our statement that a suitable, ready-to-run call graph construction system for C/C++ that meets all of our requirements is not available at this point in time. Even though it does not satisfy all of our requirements, the system that is most suitable to our goals is Elsa/Oink. In [25], Boerboom and Janssen give a comprehensive description of the advantages and disadvantages of Elsa, as do Telea and Voinea in [40].

Since Elsa/Oink is not a ready-to-run call graph construction system, but rather a general purpose C/C++ static analysis framework, we will use Elsa/Oink as the basis of our new call graph construction system. The next section will provide an overview of the steps that we will need to take to build a complete C/C++ call graph construction system, based on the Elsa/Oink framework.

2.3 OVERVIEW OF THE CALL GRAPH CONSTRUCTION PIPELINE

Figure 2 depicts the entire process of constructing a call graph. The white blocks represent the individual steps that are required to get from source code to call graph and the arrows between them represent the type of data flowing out of and into the different steps. Each of the individual steps belongs to one of the four major phases, each of which has its own color. The four phases are performed by separate programs or program components, one after another:

1. The purple phase deals with preprocessing source code and is performed by a C/C++ preprocessor.
2. In the blue phase the preprocessed source code is parsed by the Elsa C/C++ parser.
3. During the green phase all information that is required to construct a call graph is extracted. The system responsible for the extraction of call information is a new program (the C/C++ Call Info Extractor) presented in this thesis.
4. Finally, in the red phase, a call graph is constructed, also using a new program (the C/C++ Call graph Constructor) presented in this thesis.

The four different phases will be discussed in the next two chapters. The first section (3.1) of chapter 3 is concerned with preprocessing and parsing the source code. The remainder of chapter 3 then deals with the *extraction of the information* required to construct a call graph. Then, the actual *construction of call graphs* from the extracted information is dealt with in chapter 4.

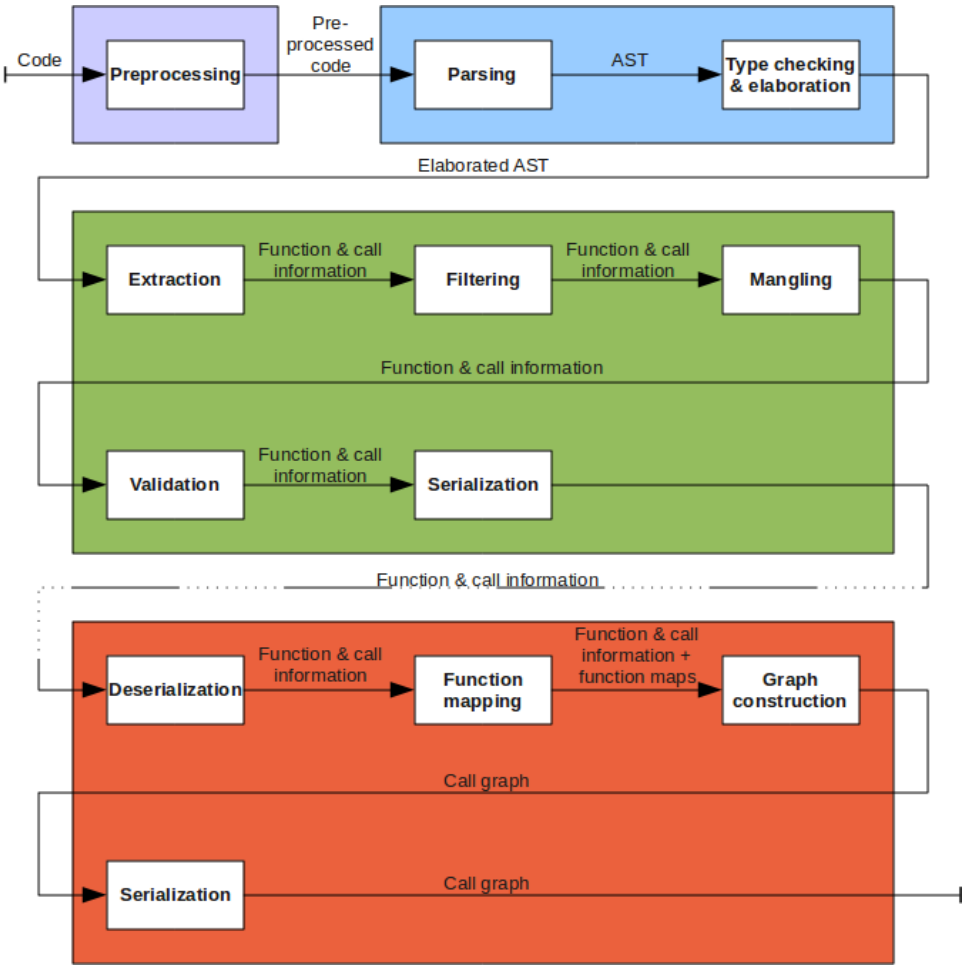

Figure 2: The complete call graph construction pipeline.

3 EXTRACTION OF CALL INFORMATION

This chapter is concerned with *gathering the information* from the source code required to construct a call graph and the information that we aim to extract can be summarized as follows:

- All functions and function calls. What exactly we mean by *function* is explained in 3.2.1. Not surprisingly, we need this information to construct a graph containing function nodes and function call edges.
- Attribute data, describing the functions and function calls. Next to the functions and function calls themselves, we are also interested in additional information about the functions and function calls, such as their location in the source code, the visibility of a function (i.e., public, protected or private), and so on.
- Containment information. To be able to construct our compound graph we need the folder, file and class hierarchy by which the functions are contained.

Please note that the actual *construction of call graphs* is not discussed until chapter 4.

As indicated by figure 3, this chapter is concerned with the first three phases of the call graph construction pipeline (see figure 2). The first two of these phases will be discussed only briefly in section 3.1, since they will be handled by existing, third-party software. However, the extraction phase (in green) which is concerned with the extraction of information needed to construct a call graph, is done by a new program presented in this thesis. As such, the extraction phase will be discussed in great detail. Below is a brief introduction to the individual steps of the extraction phase, which constitute the majority of this chapter.

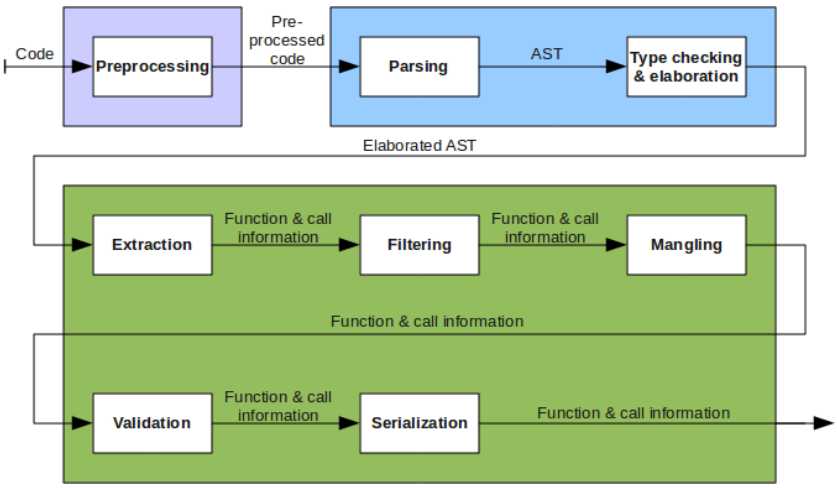

Figure 3: The preprocessing, parsing and extraction steps that will be covered in this chapter.

Extraction

After preprocessing has been done and the preprocessed source code has been parsed using Elsa, the first thing that we will to do is gather all the information that we need to construct a call graph that is as complete and correct as possible. To this end, all function declarations, function definitions, function calls and all their attribute information (such as location in the source code) are extracted from the parsed source code using the AST traversal system provided to us by Elsa. Section 3.2 describes in detail what information is to be extracted, where it can be found and how we can obtain it using Elsa.

Filtering

A rather trivial fact about large software systems is that they tend to contain a large number of functions and function calls. This makes the visual presentation and exploration of such systems much harder, so the final call graph should ideally only include those functions that are relevant with respect to the questions to be answered on the system at hand. Now, in virtually all common reverse engineering tasks, unused functions from system libraries are not of interest for analyses, so we would like to be able to filter these out. The final call graph should ideally only include those functions that are used by the system and discard all other functions from system libraries. The process of filtering such unused functions is described in section 3.3.

Name mangling

Large systems are nearly always split up into multiple files. Such a division allows for a proper structuring of the system and is commonly regarded as good practice. As a consequence, it is often the case that a function defined in one source file is called from another source file. Obviously, the function call must somehow be associated with the correct function definition. During a normal build of the system, the linker is responsible for this association. That means that, in case the parser processes one translation unit at a time, as Elsa and most other C/C++ parsers do, this information is not directly available from the parser. We will therefore need to do this linking ourselves. Eventhough linking itself is not discussed until the next chapter (in section 4.3), we will be doing some preparation for the linking process in this chapter. Namely, to make sure that functions with static linkage are not linked to function calls in another translation unit, the names of these functions will be mangled. This process of name mangling will be discussed in section 3.4.

Validation

After all the necessary information gathering and manipulation steps have been performed, section 3.5 will deal with validating the extracted information. During validation we make sure that all function calls have a corresponding function definition or function declaration. In the ideal case, this step is not necessary, but it has proven its worth many times during the development of the system.

Serialization

The last step of the extraction process is serializing the output to a binary format. Section 3.6 will describe the approach that was chosen and the options that are available.

Complexity

At the end of each section describing a step in the pipeline, the running-time complexity of that respective step (in terms of number of functions and number of function calls), is presented. As a summary, the total running-time complexity of the entire information extraction process will be presented in section 3.7.

3.1 PREPROCESSING AND PARSING

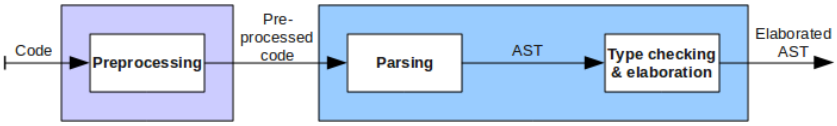

Figure 4: The preprocessing and parsing steps.

As was illustrated in [25], Elsa does not have its own preprocessor. Instead it expects a single preprocessed source code file (translation unit) as its input. Hence, we need to preprocess source code files ourselves before we pass them to Elsa. Luckily, most mainstream C/C++ compilers provide readily available preprocessing functionality (either as a standalone program or integrated into the compiler) and we can simply use that to preprocess the source code files.

It is worth mentioning that Elsa features a mechanism that allows us to retrieve the location of a code fragment *before* it was preprocessed. This is convenient, since this location information is very relevant information that we would like to be able to present to developers using our system.

When preprocessed, each source code file is transformed into an AST during the parsing step and this AST is then further refined during the type checking and elaboration step: During the type checking step the AST is annotated with type information.

Then, during the elaboration step AST nodes are inserted which correspond to implicit (i.e., invisible) syntax, such as the invocation of a destructor when an object goes out of scope. Also, to simplify further analysis, it normalizes the AST so that syntactically different, but semantically equivalent constructs (such as `a + b` vs. `a.operator+(b)`) are rendered in the AST in the same way.

At the end of the preprocessing and parsing phase, a source code file (and the files that it includes) will have been transformed into a type-checked, annotated, augmented and normalized AST. It is this elaborated AST that will subsequently be used in the extraction phase.

3.1.1 *Complexity*

Not surprisingly, the Elsa parsing system is a very large and complex software system. As such, it would be very difficult to determine a simple running-time complexity for it and therefore we will simply refer to its complexity as C_P from now on. Regarding Elsa's *performance*, we state, from experience, that parsing using Elsa is roughly similar to that of compilation using an efficient C/C++ compiler.

3.2 EXTRACTION

The call graph we aim to construct will consist, for one part, of *function nodes and call edges*. Apart from the functions and function calls themselves, extra information about the functions and function calls will provide a developer with valuable

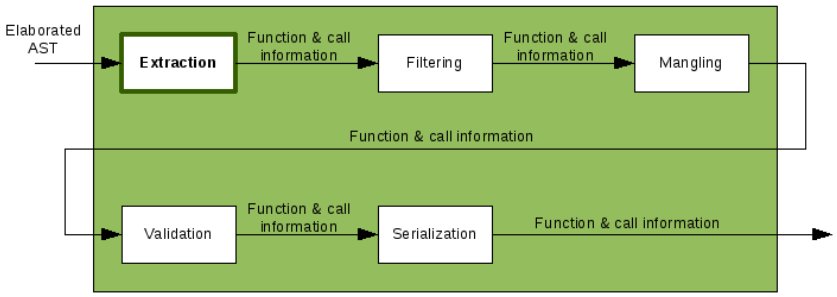

Figure 5: The extraction step.

information needed during various types of analyses. This additional information will be stored as attributes of the functions and function calls.

It is easy to understand why such attributes are essential for a developer to perform his analyses. As an example, location information can be used to locate a function or function call in the source code and knowing whether a function is a C-style function or a C++-method can be used to determine the 'object-orientedness' of a system. As another example, if we know whether a method is virtual we can use that information to identify chains of calls to virtual functions, which are potential performance bottlenecks.

The other major part of our compound graph will consist of *containment nodes and edges*. These containment nodes can be folder nodes, file nodes and class nodes and the edges between them will indicate their containment hierarchy. These nodes and edges provide the functions and function calls in the graph with a very natural and intuitive origanization and will allow a developer to easily navigate to a function or function call based on the directory, file and class that contain it.

This section will first define precisely what we mean when we use the term 'function'. Next we will describe, for both functions and function calls, respectively, what information will be extracted and where in the source code it can be found. Third, this section will describe how the information can actually be retrieved using the Elsa parser and finally the complexity of extracting the information will be discussed.

3.2.1 *Definition of function*

In many cases, the use of the term *function* will not lead to much confusion as its meaning is straightforward. However, in our context it is useful to be precise about what we mean by *function*, since it might not always be clear whether we are talking about a *function declaration*, a *function definition*, or both.

Whenever we use the term *function* in the remainder of this thesis, we mean either:

1. A function declaration only, or
2. A function declaration together with its definition.

To be a bit more formal, we regard a *function* to be an entity that:

- Always has exactly one *function declaration* associated with it, and
- Always has zero or one *function definitions* associated with it.

This is exactly how functions are represented in our implementation: An object of type `Function` always has one reference to an object of type `FunctionDeclaration` and it has zero or one references to an object of type `FunctionDefinition`. Lastly, there are three important remarks that we must make regarding function declarations and function definitions:

1. Whereas a definition of a function can occur only once throughout all translation units, a function declaration can occur any number of times, even within the same translation unit. If we encounter multiple declarations of the same[1] function within a single translation unit, we disregard all but the first declaration that we find. The subsequent declarations, although legal, do not provide us or the developer with any relevant information that we cannot retrieve from the first declaration. If, within a translation unit, a function *definition* exists, then that definition is also used as the function's *declaration* (i.e., one could think of a function definition as a function declaration plus its implementation). Hence, a function always has exactly one declaration associated with it.

2. Since, in the source code, a function definition can have any number of function declarations associated with it, there exists a many-to-zero-or-one relation from declaration to definition. However, since we just stated that, per translation unit, we will only associate a single declaration with a function, in our case there exists a one-to-zero-or-one relation from declaration to definition.

3. Although one could argue that the declaration of a function pointer is, in some sense, a declaration of a function, we do not regard such function pointer declarations as function declarations. As such, the `FunctionDeclaration` reference in a `Function` object will never refer to a function pointer declaration.

3.2.2 *Function attributes*

To be able to construct the call graph and be able to present the developer with enough relevant information to perform his analyses, quite some information is needed for every function. All the required function attributes are described below.

- *Fully qualified name*. A function's fully qualified name (FQN) consists of its return type, its name including any namespaces and classes in which it is contained, and the types of its parameters. The following table gives some examples of functions and their corresponding FQN:

Function	**FQN**
`float max(float a, float b)`	`float (max)(float, float)`
`int Tree::insert(int* v)`	`int (Tree::insert)(Tree&, int*)`
`int List<int>::insert(int v)`	`int (List<int>::insert)(List<int>&, int)`
`static void A::s()`	`void (A::s)()`

  Note that in case of non-static C++-methods, the first parameter in the FQN is a reference to an object of the class containing the method. This stems from the `this` pointer that is being passed to the method under the hood. In line with this, you can see that such an object reference is not passed in case of a static C++-method, since a `this` pointer does not exist in that context. A second thing to note from the above table, is that for classical

1 We consider two function declarations to be the same when they are considered the same according to the rules of the C/C++ standard [29]. Elsa/Oink supports checking for this.

C-style functions and C++-methods, the name of the function returns in its FQN. For C++ constructors, however, a slighlty different name is used in the FQN, as is illustrated in the table below. For completeness, this table also lists the FQNs generated for C++ destructors and operators.

Function	FQN
`A::A()`	`(A::constructor-special)()`
`A::~A()`	`(A::~A)(A&)`
`A & A::operator=(const A & other)`	`A& (A::operator=)(A&, const A&)`

During the construction of the call graph, function calls need to be linked to functions across several translation units. To be able to do this, we need a way to uniquely identify functions throughout the system. The fully qualified name of a function does exactly that. Furthermore, this information is very valuable to the developer, since it tells him exactly what function he is dealing with.

- *Unqualified name*. A function's unqualified name contains the name of the function, but does not include the function return type and parameters types. The following table lists the unqualified names of the functions from the previous example:

Function	Unqualified name
`float max(float a, float b)`	`max`
`int Tree::insert(int* v)`	`insert`
`int List<int>::insert(int v)`	`insert`

Unqualified names are not required during the construction of the call graph and exist merely for the convenience of the developer performing analyses: FQNs easily become large and difficult to read, so it is convenient to also have a short version of the name available.

- *Extended function type*. A function type consists of the combination of return type and parameter types of a function. In case the function is a non-static C++-method we extend this notion by also including the names of any containing classes and namespaces. The table below then lists the *extended function types* (EFTs) of the example functions:

Function	EFT
`float max(float a, float b)`	`float ()(float a, float b)`
`int Tree::insert(int* v)`	`int (Tree::)(int*)`
`int List<int>::insert(int v)`	`int (List<int>::)(int)`

This property is used during the linking process to associate a call via a pointer-to-function or pointer-to-member with a set of potential call candidates. The matching of call candidates is done based on the EFT of the called function and the EFTs of the call candidates. All the details of this process of call candidate resolution using EFTs will be discussed in section 4.3.1. EFTs are merely required for the construction of the call graphs; they are most likely not relevant for end users.

- *Path to the file that contains the function*. To be able to include folder and file containment nodes in the call graph, we obviously need the name of the directory and file in which the function resides. In case the *definition* corresponding to the function is available, the directory and file that contain that definition are used. Otherwise the directory and file of the function *declaration* are used. A simple example of such a path is: `/home/hessel/development/program/main.cpp`.

- *Position within the file*. Next to knowing in what file the function resides, it is interesting for a developer to know at what position in that file the function is located. The position consists of a line number and a column number and points to the start of the function's definition or declaration (whichever is relevant). The position is not required for the construction of the graph.

- *Class name of the function*. Similar to the file path, the class name is required to include class containment nodes in the call graph. Of course, a class name is only available for C++-methods, not for C-style functions. Again, we use the example functions to illustrate:

Function	Class name
`float max(float a, float b)`	
`int Tree::insert(int* v)`	`Tree`
`int List<int>::insert(int v)`	`List<int>`

  To be precise, we should mention that the value of this attribute actually contains the *compound scope* of the function. That is, if we have a method `m`, which is a member of class `A`, which is an inner class of class `B`, which is an element in namespace `N`, then the name of the class of function `m` is set to `N::B::A`. Its parent class will end up in the graph as a class name `N::B`. The result of this is that nested classes are effectively flattened in the graph hierarchy, as illustrated in figure 6. The namespace `N` is included in the name of the class solely to prevent name clashes; namespaces themselves do not appear in the hierarchy.

  Since C-style function do not have a containing class, this attribute is empty for such functions. In the case such a C-style function is contained in a namespace we still have no reason to fill in this attribute, since its containing namespace does not represent a class (which is what this attribute is for) and since namespaces are not part of the hierarchy.

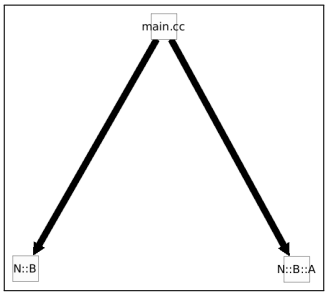

Figure 6: The representation of a nested class. Nested classes are 'flattened' in the hierarchy.

- *Linker visibility*. Using the 'static' keyword, C-style functions can be made invisible for any translation unit other than the one in which the function is defined. In other words, a function defined with static linkage can never be called by a function defined in another translation unit. It is not hard to see that this information is required to prevent function calls from one translation unit to be linked to functions with static linkage from another translation unit. Section 3.4 contains more information on this subject. Apart from being used during the construction of the graph, the linker visibility might be of some interest to the end user.

- *C-style function or C++-method*. A function is either a C-style function or a C++-method. This property is used to some extend during the construction

of the graph, mainly in resolving the call candidates of a function call. Next to that, this can be valuable information for the developer, for instance, to determine the 'object-orientedness' of the system.

- *Virtuality*. C++-methods can be declared in a class as virtual to allow them to be overridden in derived classes. This property is very important during the resolution of call candidates of a function call. Next to that, a developer might also find this property interesting, for instance, when doing a performance analysis (calls to virtuals tend to be slower than calls to non-virtuals).
- *Static instance*. C++-methods can be declared as static, which causes the method to be accessible without an instance of the class. This information is briefly needed when constructing the call graph and it might be relevant information for end users.
- *Access specifier*. Access specifiers make C++-methods visible for either: everyone (public), only the defining class and its derivatives (protected) or only the defining class (private). This property is not used during construction of the graph, but might be interesting for end users.
- *Overridden methods*. The set of methods M that this method overrides is used to resolve the set of overriding methods, for all methods $m \in M$. That information, in turn, is used during the resolution of call candidates. Although this might be very relevant information for an end user, it is currently not included explicitly in the final call graph.
- *Declared inline*. This property indicates whether the function is *declared* as inline, either implicitly or explicitly. Note that this property *does not* indicate whether a particular compiler *actually inlines* the function. During construction of the call graph, this property is not used. However, it is interesting for developers that are, for instance, doing performance analyses.

3.2.3 *The location of function attributes*

A *function declaration* can always provide us with all of the function attributes described above, whereas there are situations in which a *function definition* cannot. For instance:

```
class A
{
    virtual void m(); // Declaration of m
};

void A::m() { }       // Definition of m
```

It is clear that the definition of `m` does not tell us that `m` is a virtual function. Its declaration does tell us this. So, whenever possible, we extract a function's attributes from its declaration. Only when a declaration is not available will we use its definition to retrieve the needed information. Do note, however, that whenever only a definition is available, this does not give us less information than a declaration would be able to give us (in the above example this would mean that `m` would be defined in its class and that it would thus be declared as virtual by its definition). Also note that not having a function's definition available (e.g., in the case of a system library) is never a problem, since all required information can be retrieved from the function's declaration.

3.2.4 *Function call attributes*

It was stated at the beginning of this section that function calls will be represented in the call graph by edges. As is the case with functions, it will be useful, from the developer's perspective, if these call edges are annotated with relevant attribute information. The attribute information that must be extracted from function calls is described now.

- *The type of call*. This property identifies what type of function call this is. Eight different types of function calls have been distinguished, each of which is illustrated with an example.
  1. *Direct function call*. Represents a call to a C-style function, e.g.:

```
{
    f();
}
```

  2. *Direct method call*. Represents a call to a C++ method, on an object instance, e.g.:

```
{
    Object o;
    o.m();
}
```

  3. *Object pointer call*. Represents a call to a C++ method, on an object pointer, e.g.:

```
{
    Object* o = new Object;
    o->m();
}
```

  4. *Object reference call*. Represents a call to a C++ method, on an object reference, e.g.:

```
{
    Object o1;
    Object & o2 = o1;
    o2.m();
}
```

  5. *Constructor call*. Represents a call to a C++ constructor, e.g.:

```
{
    // a constructor call.
    Object o1;
    // another constructor call.
    Object* o2 = new Object;
}
```

  6. *Destructor call*. Represents a call to a C++ destructor, e.g.:

```
{
    Object* o1 = new Object();
    // destructor call.
    delete o1;

    // d-tor call because o goes out of scope.
    {
        Object o;
    }
}
```

7. *Pointer-to-function call*. This is a call to a C-style function, through a pointer-to-function, e.g.:

```
{
    void (*f)(int) = &g;
    f();
}
```

8. *Pointer-to-member call*. This is a call to a C++ method, through a pointer-to-method, e.g.:

```
{
    Object o;
    void (Object::*m)(int) = &Object::m;
    (o).*(m)(1);
}
```

- *Name of the call target*. The name of the call target is used to identify the function (or set of functions) that is the target of the function call. There are two possibilities for what this attribute can contain, depending on what type of function call this is. Depending on the call type, we will store and use either the FQN or the EFT of the call target, so *name* in this context means one of these two. The two possibilities are discussed below:

  1. **This is not a pointer-to-function or pointer-to-member call**. In this case, the name of the call target *will be set to the FQN of the called function*. Obviously, the FQN is obtained from the call site, not from the call target. Since the name of the call target contains an FQN, it will always uniquely *identify* a single function. This does not mean, however, that there will always be exactly one *call target*. The first example below illustrates the case in which there is exactly one call target and the second example illustrates the case in which there is more than one call target.

     Consider the following code snippet, which shows the scenario of a C-style function call with exactly one call target:

```
void f() { }
void g()
{
    // A plain and simple function call:
    f();
}
```

     The FQN of the called function, and thus the name of the call target, for this function call is '`void (f)()`'. The call graph resulting from the small program above is depicted in figure 7. It shows that there is indeed exactly one call target: the function `f`. Please note that the call graph has been stripped of containment nodes and edges for the sake of clarity.

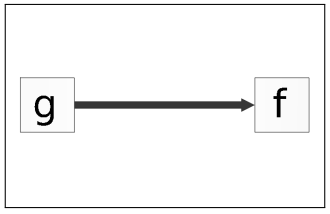

Figure 7: The call graph resulting from the simple function call.

As said, it is not always the case that there is exactly one call target when not calling a function via a pointer: When a call is made to a virtual function (i.e., the FQN of the called function identifies a virtual function) and the call is made on an object pointer or object reference, then there can be more than one call target. This is illustrated using a the following code snippet:

```
class A
{
public:
    virtual void m() { }
};
class B : public A
{
public:
    virtual void m() { }
};

void h(A* a)
{
    a->m();
}
```

In this case, the FQN of the called function extracted from the call site is '`void (A::m)()`'. However, since this is a call on an object pointer to a virtual method, the function that is actually called can be either '`void A::m()`' or '`void B::m()`'. Since we do not know which of the two methods is actually going to be called, both methods are considered a call target. Figure 8 shows the (simplified) call graph belonging to the code snippet above.

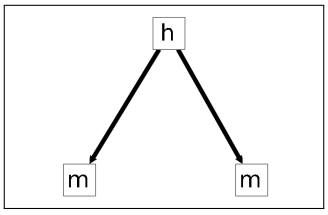

Figure 8: The call to the virtual m yields two potential call targets.

The exact details on how function calls are linked to their corresponding set of call targets is discussed in 4.3.1.

2. **This is a pointer-to-function or pointer-to-member call**. When this is the case, the name of the call target *will be set to the EFT of the call target*. Like the FQN in the former case, the EFT is retrieved from the call site and not from the actual call target. In the case of a call via a pointer-to-function or pointer-to-member, the EFT of the call target might identify more than one function, and, consequently, there might be more than one call target. Consider the following example:

```
void f() { }
void g() { }

void h()
{
    void (*p)();
    p = &f;
```

```
    p();
}
```

As said, the name of the call target is equal to the EFT extracted from the call site: '`void ()()`'. It is not hard to see that all functions in this program have a matching EFT. Therefore, all these functions, including `h`, are potential call candidates, as illustrated in figure 9.

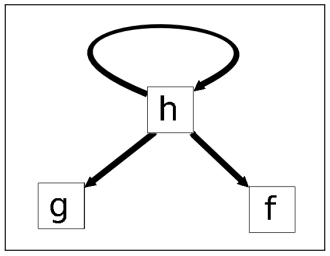

Figure 9: The call via pointer-to-function yields three potential call targets.

Again, the exact details on how function calls are linked to their corresponding set of call targets is discussed in 4.3.1.

- *Path to the file that contains the function call*. For each function call, the fully qualified path to the file in which the call is made is available. Although not required during construction of the graph, this might be relevant information for the end user.
- *Position within the file*. Apart from the file in which the call is made, the position in that file is also available to the end user. The position consists of a line number and a column number and points to the start of the function call. The position is not required for the construction of the graph.

3.2.5 *The location of function call attributes*

The function call attributes presented above must obviously be retrieved from the function calls themselves, so we need to know where to look for function calls. This is fairly straightforward, because function calls occur only in two different types of places.

First and foremost, function calls occur *within function definitions*. Consider this very trivial example:

```
void printHelloWorld()
{
    printf("Hello World!\n");
}

int main(int argc, char** argv)
{
    printHelloWorld();
    return 0;
}
```

As you can see, the call to `printHelloWorld` occurs neatly within the definition of the `main` function. This is the most common type of place where function calls occur and there is little obscurity about it. The above case demonstrates function calls made from within function definitions that have an explicit syntax. Function calls can however, also be made from within function definitions without having

such an explicit function call syntax. Please consider the second example in which a constructor and destructor are implicitly called from within a function definition:

```
class A
{
};

void f(A a)
{
}

int main(int argc, char** argv)
{
    A a;
    f(a);
    return 0;
}
```

In the above code snippet, first a call to the constructor of a is made. Then, since instance `a` is passed as a parameter to function `f`, a copy of object `a` is made, causing `A`'s copy-constructor to be called. Lastly, when function `f` returns, the object `a` goes out of scope and `A`'s destructor is called. This illustrates how function calls can be made from within a function definition without having an explicit function call syntax.

The second type of place where calls can occur is somewhat more concealed. Consider again a little example:

```
class A
{
public:
    A() { }
    ~A() { }
};

A a;

int main(int argc, char** argv)
{
    return 0;
}
```

The above code snippet defines a class `A` and then declares an instance of that class in the global scope. The `main` function does nothing and immediately returns to the operating system with return value `0`.

At first sight, it might seem that no function calls occur in this code. A closer look, however, reveals that the constructor and destructor of `A` must be called, since an global instance of `A` is declared. These calls occur just before and just after the call to `main`, respectively, which can be explained as follows: It can be seen from the code that the constructor and destructor calls do no occur within the body of `main`. However, object `a` is available throughout the entire body of `main`. So, that must mean that `a` is constructed before `main` is called and is destroyed after `main` returns.

Function calls that occur before the call to `main` are referred to as *initializing* function calls and calls that occur after the call to `main` are referred to as *finalizing* function calls.

3.2.6 *Obtaining the information from Elsa*

The Elsa parser reads the source code of a translation unit (i.e., a preprocessed source code file) and creates an annotated Abstract Syntax Tree (AST) from that. All the information that we need resides in this annotated AST.

To retrieve the information from the AST, Elsa provides us with the `ASTVisitor` class, which is an implementation of the Visitor pattern described in [30]. It allows one to traverse the entire AST and take appropriate action when a node of interest is visited.

Elsa defines a large number of node types, so only the ones that contain information that is relevant in this context will be described.

- The *TranslationUnit* node. The TranslationUnit node is the top node in the AST. It is the ancestor of all other nodes in the AST and, as its name suggests, it represents the translation unit as a whole. Traversal of the AST always starts at this node.
- The *Function* node. The Function node represents a function definition.
- The *Declarator* node. A Declarator node can represent anything that is declared. For instance, declarations of variables, functions and members are all represented by a Declarator node.
- The *Expression* node. All expressions in the source code are represented by an Expression node. For instance, a function call is represented by an Expression node.
- The *TopForm* node. A TopForm node represents an entity in the global scope, or an entity at the top level of a namespace. Examples of such entities are declarations and function definitions.
- The *Initializer* node. An Initializer node occurs as a descendent node of a Declarator node. For instance, the constructor call in an object declaration is represented by an Initializer node.

Now that the relevant types of AST nodes are known, we can explain what information is retrieved from what nodes. The full details on Elsa's C++ grammar and AST node structure can be found in [35].

Function definitions are, not surprisingly, retrieved from Function nodes and function declarations are retrieved from Declarator nodes.

Function calls are primarily retrieved from Expression nodes within Function nodes, but also from TopForm nodes. TopForm nodes may contain Declarator nodes representing, for instance, object declarations. Such an object declaration can in turn contain an Initializer node, from which initializing and finalizing function calls will be retrieved.

The process of extracting the required information is split up into three traversals of the AST.

1. In the first traversal, all *function definitions* are retrieved from Function nodes and at each Function node the definition's *function calls* are retrieved from the Expression nodes within the Function node. For each retrieved function definition, a new `Function` object (see 3.2.1) is created and its declaration and definition references are both set to the retrieved definition.
2. During the second traversal, all *function declarations of functions for which no function definition was found during the first traversal* are retrieved from the

Declarator nodes. For each retrieved function declaration, a new `Function` object (see 3.2.1) is created and its declaration reference is set to the retrieved declaration, while its definition reference is left empty.

3. Finally, in the last traversal, all *initializing and finalizing function calls* are retrieved from the TopForm nodes.

After these traversals, all functions (as defined in 3.2.1) and function calls will have been retrieved, since we will have searched all locations that can contain function declarations or function definitions (3.2.3) and all locations that can contain function calls (3.2.5).

3.2.7 *Complexity of extraction*

During the three traversals of the AST, information about all functions and function calls is retrieved. The complexity of this process depends for one part on the complexity of collecting the information and storing it in the appropriate data structures. For the other part, however, it depends on the complexity of a tree traversal.

Before we begin analyzing the complexities, let us first define a few symbols:

N_{Defn} : The number of function definitions in the input.

N_{Decl} : The number of function declarations in the input.

N_F : The number of functions in the input, with $0 \leqslant N_F \leqslant N_{Defn} + N_{Decl}$.

N_C : The number of function calls in the input.

The complexity of the first part, collecting and storing information, is rather straighforward. Information will be collected for all function definitions and function declarations. Collecting information of a definition or a declaration can be done in constant time, since Elsa has prepared the required information for us; we merely need to get it from the AST. So, collecting all information will take at most $O(N_{Defn} + N_{Decl})$ time.

Then, the information needs to be stored. We will store information for each function and each function call. The functions will be stored in a hash table and the function calls will be stored in lists. According to [27], insertion into a hash table takes $O(N)$ in the worst case, but can be expected to take $O(1)$ on average. Insertion into a list can be done in $O(1)$ time in the worst case. So, we can conclude that the worst-case complexity of storing the information is equal to $O(N_F^2 + N_C)$ and that the expected complexity is $O(N_F + N_C)$.

The complexity of the second part is, unfortunately, not so straightforward. Because of the intricacies of Elsa, it is extremely difficult to discover a simple complexity for the traversal process. The same was true for the complexity C_P of the parsing process (3.1). So, instead of trying to specify the dependency of the complexity of a traversal on the size of the input, we will simply refer to the complexity of a tree traversal as C_T.

From here, calculating the total complexity of extraction C_E is simply a matter of adding the individual complexities. Thus, the worst-case complexity $C_{E,W}$ is:

$$C_{E,W} = C_T + O(N_{Defn} + N_{Decl} + N_F^2 + N_C)$$

Likewise, the expected-time complexity $C_{E,E}$ becomes:

$$C_{E,E} = C_T + O(N_{Defn} + N_{Decl} + N_F + N_C)$$

However, since $0 \leqslant N_F \leqslant N_{Defn} + N_{Decl}$, this can be simplified to:

$$C_{E,E} = C_T + O(N_{Defn} + N_{Decl} + N_C)$$

3.3 FILTERING

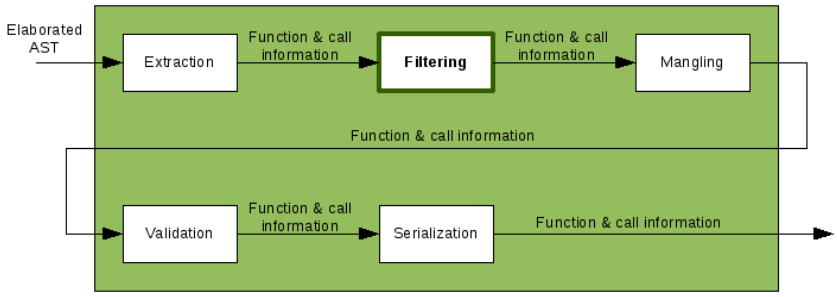

Figure 10: The filtering step.

Any real life software system will make use of libraries that help solve many basic problems. The use of such libraries is a good thing, since it provides developers with stable, proven and often efficient implementations of commonly needed functionality.

However, in our case the use of such libraries has some consequences. Whenever a library header is included from a file containing user code, the code from that header file becomes part of the same translation unit as the user code. In particular, all code from that header file becomes part of the translation unit and thus part of the call graph, regardless of whether it is used by the user code. For example, consider the following simple Hello World program:

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("Hello World!\n");
    return 0;
}
```

In the example the system header `stdio.h` is included so the `printf` function can be used to print `Hello World!` to the screen.

Now, one might expect a call graph of this system to include only a few nodes and edges: A node for the `main` function, a node for the `printf` function, an edge between these two functions and, finally, a few nodes and edges for the directories and files in which the functions are contained. Figure 11 depicts such a call graph.

However, as was stated earlier, it is not just the `printf` function that becomes part of `main`'s translation unit. Instead, *all* of the functions in `stdio.h` become part of the same translation unit as `main`. The consequence of this is that the call graph of the Hello World program will also contain all of the functions from `stdio.h`.

As it turns out, the actual call graph for the Hello World program contains 180 nodes and 192 edges, which is much more than the 9 nodes and 9 edges depicted in figure 11. Remember that only a single system header file was included; the number of nodes and edges will continue to grow if more system headers are included.

Figure 12 shows the actual call graph for the Hello World program. Note that the nodes and edges from figure 11 are shown in green.

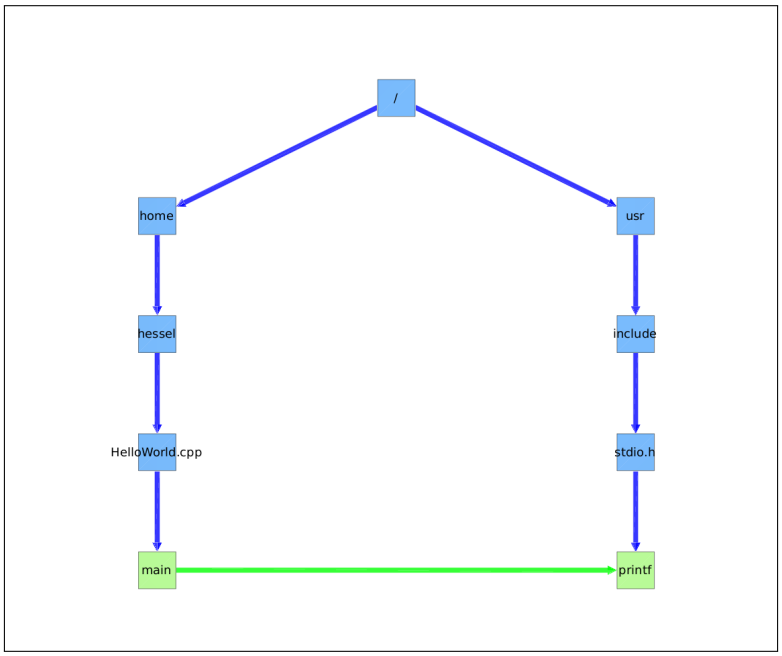

Figure 11: The call graph one might expect for the Hello World program. The blue nodes are folders and files, the blue edges are containment relations, the green nodes are function and the green edges are function calls.

If the developer's goal is to investigate the structure of the Hello World program, then all these extra functions from `stdio.h` will only distract from the analysis. So, to avoid the call graph from getting cluttered by a large amount of uninteresting functions, we need a way to filter out functions that are not used by user code and are not user code themselves.

Unfortunately, it is not possible to determine what is user code and what is not without input from the user. It is possible, however, to determine a conservative set of functions that *can be called from* user code. Given such a set, we could safely filter out all functions that *can never be called* from user code.

Although this solution is not ideal and we will still be left with uninteresting, cluttering functions in the call graph, it does reduce the size of the graph by 50-80%, which is still pretty good. To illustrate, the number of number of nodes from the Hello World call graph was reduced by 70%, from 180 to 54 nodes. Figure 13 depicts this reduced call graph. Again, the nodes and edges from figure 11 are shown in green.

The remainder of this section will describe the process of filtering function declarations (3.3.1) and filtering function definitions (3.3.2) and their respective running-time complexities. This section will conclude with the restrictions that apply when one wants to guarantee that no functions are inappropriately filtered out. conclude with an analysis of the worst-case run-time complexity of the filtering process.

3.3.1 *Filtering declarations*

To understand what function declarations can be safely filtered out, it is important to notice two things.

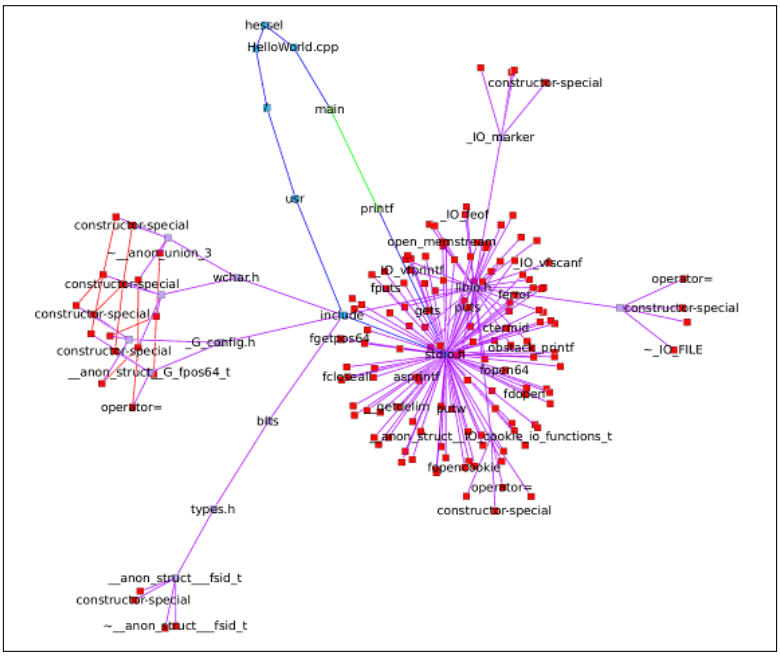

Figure 12: The actual call graph for the Hello World program without filtering. The blue and green nodes and edges form the expected call graph from figure 11. All other nodes and edges are unexpected: Red nodes represent function, red edges represent calls, purple nodes represent containment nodes (directories, files and classes) and purple edges represent containment relations.

1. The first thing is the fact that a function declaration is nothing more than a claim saying that a specific function definition exists somewhere. Obviously, it is not a definition itself. So, no matter how many declarations we filter out, we never loose any definitions by doing so.

2. The second thing to notice is that to be able to call a function from within a translation unit, that function must, at least, be declared in that translation unit. In other words, a function must be redeclared in every translation unit in which it is called.

These two observations lead us to conclude that if a function is *declared*, but is *not called* within a translation unit, then that declaration can safely be filtered out. This is exactly what happens during declaration filtering and the implementation of the filtering process is illustrated by the pseudo-code below.

We should note, however, that we leave some room for improvement here: In short, we want to filter out all declarations that are unused and we attempt to do so by filtering out all declarations that are never called. However, it would be better to define the criterium for filtering as follows: All declarations that are not called, either directly, or indirectly, from a function that is publicly visible (i.e., does not have static linkage), can safely be filtered out.

```
void filterDeclarations(Functions & F)
// F: The set of all functions extracted from the AST.
{
    // Make a working copy of F.
1:  F' = CopyOfSet(F);

    // Remove all functions that have a definition from the
    // from the working copy.
```

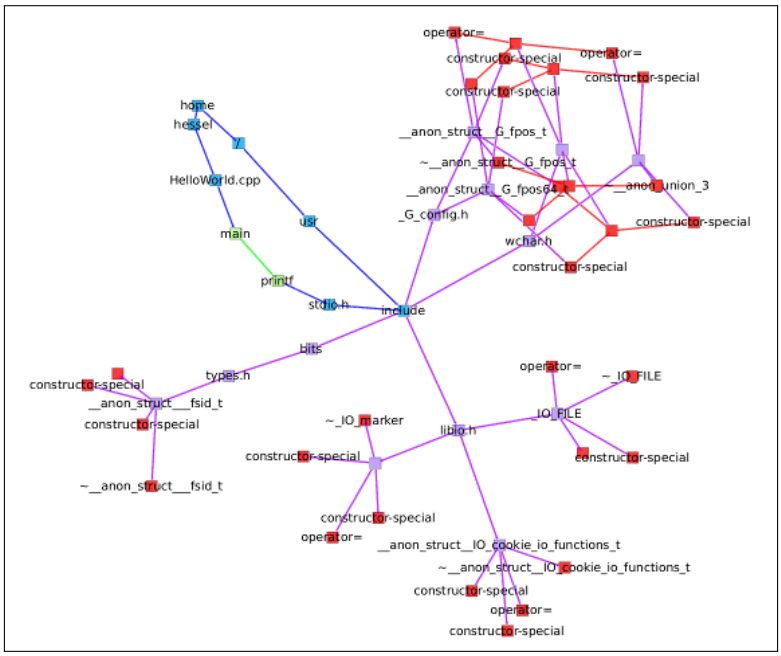

Figure 13: The call graph for the Hello World program after filtering. The blue and green nodes and edges form the expected call graph from figure 11. The coloring of the other nodes and edges is as it is in figure 12.

```
2:  RemoveFunctionsWithDefinitionsFromSet(F');

    // Remove all functions that are called in this
    // translation unit from the working copy.
3:  RemoveCalledFunctionsFromSet(F');
    // The set F' now contains all the function declarations
    // that can safely be removed.

    // Remove all functions in F' from F.
4:  RemoveFunctionsFromSet(F', F);
}
```

To determine the worst-cast run-time complexity of this procedure, we first define the following symbols:

N_F : The number of functions in F.

N_C : The total number of function calls.

For line 1 of the code it can be seen that it takes $O(N_F)$ to complete, since it needs to make a copy of every function in F.

Line 2 potentially removes all of the functions from F'. According to [27], deleting an element from a hash table can be done in $O(1)$ time. So, to remove N_F functions from a hash table takes at most $O(N_F)$ time.

In the case that no functions were removed from F' in line 2 and that all functions are called in the translation unit, line 3 would need to remove all functions from F'. However, `RemoveCalledFunctionsFromSet` iterates over all *function calls* and, at each iteration, removes the called function from F'. Now, the term stemming from the deletions from F' is easy: Since each function can be removed only once, the maximum number of deletions is $O(N_F)$. Moreover, if each function would be called exactly once, then this term would also reflect the number of function calls we need to iterate over. Obviously, there is no such guarantee: It may very

well be that functions are called more than once. For this reason, we introduce another term that will reflect the extra iterations coming from functions that are called more than once: $O(N_C - N_F)$. When we add the two terms, the equation becomes $O((N_C - N_F) + N_F)$, which can be reduced to $O(N_C)$. It is interesting to see that the worst-case complexity of `RemoveCalledFunctionsFromSet` does not depend on the number of functions removed; instead, it depends solely on the number of function calls made in the translation unit.

Line 4 must, in the worst case, remove all functions from F. As before, this can be done in $O(N_F)$ time.

When we add up the complexities of the individual lines, we come to a worst-cast run-time complexity of $O(N_C + N_F)$ for `filterDeclarations`.

3.3.2 *Filtering definitions*

Before we start explaining the problem of filtering definitions, first recall what functions we would like to filter out. All functions that are not used by user code, either directly or indirectly (i.e., transitively), should, ideally, be filtered out. All such functions are imported from libraries, but are never actually used, so they are not of interest.

Filtering function definitions is somewhat more challenging than filtering declarations. Whereas function declarations are only visible within their own translation units, function definitions are potentially visible to all translation units. So, a function defined in one translation unit may very well be called from another. Because of this, we need to be a bit more careful when filtering definitions. We cannot simply throw out all definitions that are not used within the current translation unit, since they could be called from within another translation unit.

Therefore, we would like to have a way to determine whether a function is used, either from within the current translation unit, or from within any other translation unit. Unfortunately, translation units are processed one at a time, and we cannot see beyond the boundaries of the current translation unit; at any time, our scope is limited to a single translation unit. This means that, at the moment of processing individual translation units, we have no way to determine whether a function is called from another translation unit.

Although it is not possible to determine for every function whether it *is* called from another translation unit, it is possible to determine for every function whether it *can* be called from another translation unit. Namely, functions that have static linkage are only visible within the current translation unit and hence cannot be called from another translation unit.

Obviously, when a function *cannot* be called from another translation unit, it *is* not called from another translation unit. Unfortunately, this property is not symmetric: It is not true that all functions that *are* not called, *cannot* be called. By this, we mean that not all functions that we would like to filter out - the ones that *are not called* - can be detected as 'filterable' using our method. Only the ones that *cannot be called* (because they have static linkage) are detected as 'filterable' using our method.

So, eventhough this method does not make it possible to filter out *all* unused functions, it does make it possible to safely filter out *some* unused functions.

Recall that the goal is to filter out function definitions that are not called from user code. The approach will be to first construct the set of function definitions that *can be* called from user code. Then, the functions definitions that exist in the set of all function definitions, but do not exist in the set we just constructed, can

safely be removed. The algorithm that implements this approach is printed in pseudo-code below.

```
void filterDefinitions(Functions & F)
// F: The set of all functions extracted from the AST, minus
//    the functions filtered during declaration filtering.
{
    // Begin with the set of function definitions from F
    // that do not have static linkage and are thus callable
    // from other translation units.
1:  C = SetOfFunctionDefinitionsWithNonStaticLinkage(F);

    // Next, calculate the transitive closure over the
    // 'calls' relation of C. That is, add all functions
    // to C that are (transitively) called by the functions
    // in C.
2:  C = TransitiveClosureOverCallsRelation(C);

    // Finally, remove all functions from F that are in F,
    // but not in C.
3:  RemoveFunctionsNotInSet(F, C);
}
```

In order to determine the worst-cast run-time complexity of this procedure, we (again) define the following symbols:

N_F : The number of functions in F.

N_C : The total number of function calls.

The function `SetOfFunctionDefinitionsWithNonStaticLinkage` in line 1 of the algorithm needs to inspect all functions in F and, in the worst case, needs to insert all of those functions into the ordered set C. Inserting an element into an ordered set can be done in at most $O(\log(N_F))$ time. However, [27] shows that when using a hash table, inserting an element takes on *average* $O(1)$ time. It should be noted, though, that the *worst-case* complexity of insertion into a hash table is $O(N_F)$. So, whereas the expected running time is $O(N_F)$, the worst-case running time of line 1 is $O(N_F^2)$.

In line 2, all functions that are called from globally visible code are added to the set C. In the worst case, C contained only a single function before the call to `TransitiveClosureOverCallsRelation` and contains all functions when the call is finished. That would mean that all functions and all function calls have been traversed, and that all functions have been added to the ordered set C. So, when using hash tables, the expected running time of this function is $O((N_C - N_F) + N_F)$, which reduces to $O(N_C)$. Again, as was the case with declaration filtering, the term $(N_C - N_F)$ stems from the fact that the same function may be called multiple times. The worst-case running time, on the other hand, is $O((N_C - N_F) + N_F^2)$.

The last line of the algorithm potentially removes all functions from the set F. As was stated in 3.3.1, this will take at most $O(N_F)$ time.

When we add up all the individual complexities, we conclude that `filterDefinitions` has:

- An *expected* running-time complexity of $O(N_C + N_F)$, and
- A *worst-case* running-time complexity of $O((N_C - N_F) + N_F^2)$.

3.3.3 *Restrictions in filtering definitions*

The algorithm for filtering definitions presented above can, in most cases, be safely applied. It makes sure that all functions that are callable from another translation are not filtered out. However, there are still two possible scenarios in which functions that can be called from another translation unit are filtered out. An interesting note is that these functions can be called from another translation unit, even though they are not *visible* from another translation unit. This can occur in case of:

- A call via a pointer-to-function, or
- A call to a virtual method on an object reference or object pointer.

Call via a pointer-to-function

The first of the two scenarios concerns a function call via a pointer-to-function. Consider the following collection of files.

```
// ----- File: library.h -----
typedef void (*FunctionPtr)();
FunctionPtr GetAddressOfFunction();

// ----- File: library.c -----
#include "library.h"

static void Function() { }

FunctionPtr GetAddressOfFunction()
{
    return &Function;
}

// ----- File: main.c -----
#include "library.h"

int main(int argc, char** argv)
{
    FunctionPtr f = GetAddressOfFunction();
    f();
    return 0;
}
// ----- End
```

There are two source files in the above example, so that means there are also two translation units:

1. The Library translation unit, which consists of the contents of `library.c` plus the contents of `library.h`.
2. The Main translation unit, which consists of the contents of `main.c` plus the contents of `library.h`.

Now, the function named `Function` is defined in the Library translation unit. Its address is 'exported' to the Main translation unit and from there the function is called via a pointer-to-function. Thus, we see that even though all of the criteria formulated earlier for definition filtering are satisfied, the function can still be called from a translation unit different from the one in which it was defined.

If we want to be sure that absolutely no functions that can be called from another translation unit are filtered out, the solution is to not filter out any C-style functions at all. That is a rather extreme solution for a problem that is probably hardly ever going to occur. Therefore, the filtering of C-style functions was made optional and a warning will be printed during the construction of the call graph whenever a call via a pointer-to-function is detected. That way, whenever no such warnings are printed, the user knows that no potential call candidates were wrongfully filtered out and still have the full benefits of filtering. Whenever such warnings are printed, the user has the option to enable conservative filtering and then no C-style functions will be filtered out.

Call to a virtual method on an object reference or object pointer

The second of the two scenarios involves a call to a virtual method on an object reference or object pointer. Again, consider the following illustrating example.

```
// ----- File: library.h -----
class VisibleBaseClass
{
public:
    virtual void m() { }
};

VisibleBaseClass* GetBaseClassPtr();
VisibleBaseClass& GetBaseClassRef();

// ----- File: library.cpp -----
#include "library.h"

class InvisibleDerivedClass : public VisibleBaseClass
{
public:
    virtual void m() { }
};

InvisibleDerivedClass c;
VisibleBaseClass* GetBaseClassPtr() { return &c; }
VisibleBaseClass& GetBaseClassRef() { return c; }

// ----- File: main.cpp -----
#include "library.h"

int main(int argc, char** argv)
{
    VisibleBaseClass* p = GetBaseClassPtr();
    VisibleBaseClass& r = GetBaseClassRef();

    p->m();
    r.m();

    return 0;
}
// ----- End
```

As in the previous example, we have two translation units: The Library translation unit and the Main translation unit. In the Library translation unit, both

the `VisibleBaseClass` and the `InvisibleDerivedClass` classes are defined. However, in the Main translation unit, only the `VisibleBaseClass` class is defined. Regardless, whenever the calls to the method `m` are made in the `main` function, `InvisibleDerivedClass::m` is the method that is actually being called.

Now, the problem here is that `InvisibleDerivedClass::m`

1. Has static linkage: It is defined within its class and therefore has static linkage.
2. Is never called within the translation unit in which it is defined.

Therefore, even though it is being called from the Main translation unit, `InvisibleDerivedClass::m` will be deleted during definition filtering. The solution to this problem is similar to the solution for the first scenario: Do not filter out any virtual methods. Again, instead of never filtering virtual methods, the filtering of C++ virtuals is made optional. During the construction of the call graph a warning is printed whenever a call to a virtual on an object reference of object pointer is detected. This way, the user can turn conservative filtering on or off, depending on what is appropriate.

3.4 NAME MANGLING

During the linking process, which is described in 4.3.1, function calls are linked to functions across translation unit boundaries. As was stated in 3.2, functions that have static linkage are not visible outside the translation unit in which they are defined and must therefore not be linked to function calls from another translation unit.

Now, one might argue that the most obvious time to solve this problem is during the linking process itself: When trying to link a function call to a function definition, we could test and see whether the definition has static linkage and whether it is defined in a translation unit different from the one in which the call was made. If that is the case, the function call must not be linked to that function.

However, a big drawback of that approach would be that we would not have the guarantee that an FQN identifies a *single* function: An FQN could very well identify *two or more functions with static linkage* that have the same name and function type. Whenever we would be using an FQN to identify a function, we would need to check whether the function has static linkage and whether it actually identifies more than one function. To overcome this another approach was chosen: name mangling.

To make sure that functions with static linkage can only be linked to function calls in the same translation unit, the FQN of these functions and all references made to them in their translation unit will be mangled. As a result, function definitions with static linkage from a translation unit different than that of the call site will no longer be considered during linking.

As an illustration, consider the following code sample:

```
// ----- Translation unit: A -----
static void F()
{
}

// ----- Translation unit: B -----
static void F()
{
```

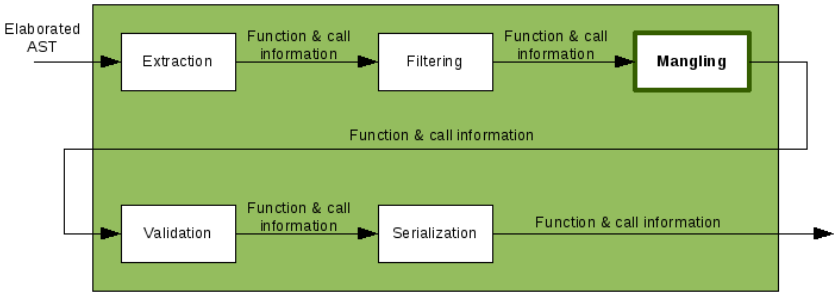

Figure 14: The name mangling step.

```
}

void FunctionCallingF()
{
    F();
}
// ----- End
```

In the sample code there are two translation units, A and B, and they both contain a definition for `F`. In translation unit B, a call is made to `F` and that call must obviously be linked to the `F` defined in B, and not to the `F` defined in A. However, the FQNs of the function definitions and function call are as follows:

Element	**FQN**
`static void F()` in A	`void (F)()`
`static void F()` in B	`void (F)()`
Call to `F()` in B	`void (F)()`

This means that, currently, the call to `F` would be linked to both the definition of `F` in A and the definition of `F` in B. To resolve this, the FQNs will be mangled per translation unit. More specifically, the FQN of each element will be mangled to include the file name of the translation unit in which it occurs. The result is as follows.

Element	**FQN**
`static void F()` in A	`void (F)(){sl:/path/to/A}`
`static void F()` in B	`void (F)(){sl:/path/to/B}`
Call to `F()` in B	`void (F)(){sl:/path/to/B}`

As a result of this mangling, the FQN of the function call now no longer matches the FQN of the definition of `F` in A and will thus only be linked to the definition of `F` in B.

The major advantage of the mangling of FQNs is that a function's FQN now uniquely identifies it system-wide. After name mangling, there are no two functions with the same FQN and we no longer need to worry about functions that have static linkage during the linking process.

Finally, it is important to note that only the FQNs of functions that have static linkage will be mangled. FQNs of functions that do not have static linkage will never be mangled.

The algorithm that performs name mangling is depicted in the pseudo-code below.

```
void mangleNames(FunctionCalls & C, Functions & F)
// C: The set of all function calls.
// F: The set of all functions.
{
    // Mangle the FQNs of calls to functions that have
    // static linkage.
1:  for(int i = 0; i < length(C); i++)
    {
        FunctionCall c = C[i];
        if((!c.isCallViaPointerToFunction() &&
            !c.isCallViaPointerToMember())
        {
            // Retrieve the function belonging to this
            // function call.
2:          Function f = F.getFunctionByFQN(c.FQN);

            // If the called function has static linkage,
            // the the FQN of the function call must be
            // mangled.
            if(f.HasStaticLinkage())
            {
                mangleFQN(c.FQN);
            }
        }
    }

    // Mangle the FQNs of function definitions that have
    // static linkage.
3:  for(int i = 0; i < length(F); i++)
    {
        Function f = F[i];
        // If this function has static linkage, then its FQN
        // must be mangled.
        if(f.HasStaticLinkage())
        {
            mangleFQN(f.FQN);
        }
    }
}
```

Determining the worst case run-time complexity of `mangleNames` is rather straightforward. First, consider the following definitions:

N_C : The number of function calls in C.

N_F : The number of functions in F.

Line 1 of the algorithm iterates over all function calls made in the translation unit and line 3 iterates over all functions present in the translation unit. Both make at most one call to `mangleFQN` per iteration and the first also makes at most one call to `getFunctionByFQN` on line 2.

One could argue that the running-time of function `mangleFQN` depends on the length of the FQN that is being mangled. However, the average length of all FQNs will be a constant, so, the average running time of `mangleFQN` is also constant.

The function `getFunctionByFQN` needs to retrieve an element from a hash table, which will take $O(1)$ *expected* time and $O(N_F)$ *worst-case* time.

Thus, we conclude that the total running-time complexity of `mangleNames` depends on the number of functions and the number of function calls as follow:

- The *expected* running-time complexity is $O(N_C + N_F)$.
- The *worst-case* running-time complexity is $O(N_C N_F + N_F) = O(N_C N_F)$.

3.5 VALIDATION

The validation phase is concerned with making sure that, after filtering and name mangling, a function declaration or function definition has been found for every function call that is made. It should be noted that this process exists solely for debugging purposes, since any target (i.e., executable or library) that does not satisfy this requirement is not a valid, compilable, target. However, having a validation phase increases confidence in the correctness of the implementation of the extraction, filtering and mangling phases.

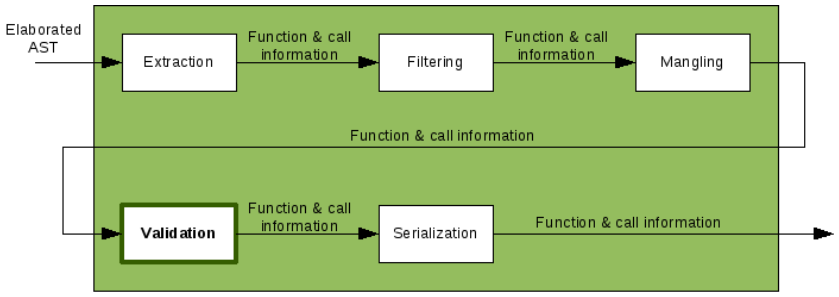

Figure 15: The validation step.

Before the validation algorithm is given, it should be noted that not all function calls need to have a corresponding function declaration or function definition. For instance, a function that is not declared nor defined in the current translation unit, can still be called via a pointer-to-function. So, such function calls are not validated.

```
int validateFunctionCalls(FunctionCalls & C, Functions & F)
// C: The set of all function calls.
// F: The set of all functions.
{
    // Iterate over all function calls and validate each
    // call, if possible.
1:  for(int i = 0; i < length(C); i++)
    {
        FunctionCall c = C[i];
        if(!c.isCallViaPointerToFunction() &&
           !c.isCallViaPointerToMember())
        {
            // Attempt to retrieve the function from the set.
            // If this fails, the function is missing and we
            // have a validation error.
2:          Function f = F.getFunctionByFQN(c.FQN);
            if(f == NULL)
            {
```

```
                reportValidationError();
            }
        }
    }
}
```

The loop in the above algorithm has a structure similar to the first loop of the `mangleNames` function. That is, the loop iterates over all function calls and, at most, needs to find an element in a hash table for each iteration. Consider once again the following definitions:

N_C : The number of function calls in C.
N_F : The number of functions in F.

Now it is easy to conclude that the function `validateFunctionCalls` has:

- An *expected* running-time complexity of $O(N_C)$, and
- A *worst-cast* running-time complexity of $O(N_C N_F)$.

3.6 SERIALIZATION

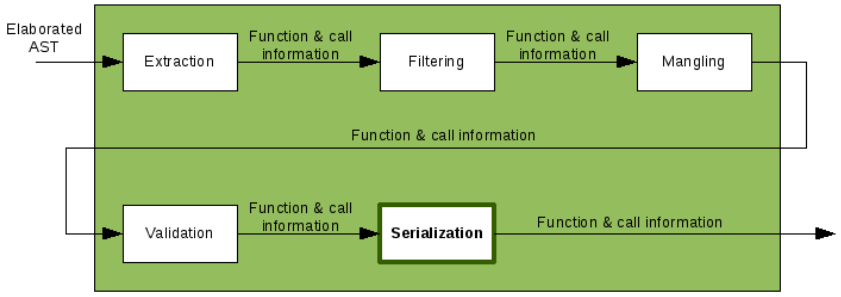

Figure 16: The serialization step.

The last step in the process of extracting call information is to serialize the information extraction from the translation unit. Since translation units are processed one at a time, the results of processed translation units must be stored. That way, when all translation units have been processed, the stored information can be used to construct the final call graph.

During serialization really only two sets of information need to be stored:

1. All extracted *functions and their attributes* (3.2.2), and
2. all extracted *function calls and their attributes* (3.2.4).

The information is serialized to a straightforward, size-prefixed, binary format. That is, before all function calls are serialized, first the *number of function calls* is serialized. Then, when serializing the attributes of an individual function, the *size of an attribute* is serialized before the attribute itself is serialized. Such a format is easy to serialize to, easy to deserialize from and relatively efficient in terms of storage size and serialization/deserialization speed when compared to, for instance, an XML format.

The algorithm that does this is depicted below.

```
void serialize(FunctionCalls & C, Functions & F)
// C: The set of all function calls.
// F: The set of all functions.
{
    // Serialize all function calls.
1:  for(int i = 0; i < length(C); i++)
2:      serializeFunctionCall(C[i]);

    // Serialize all functions.
3:  for(int i = 0; i < length(F); i++)
4:      serializeFunction(F[i]);
}
```

The complexity of this algorithm is as straightforward as the algorithm itself. Consider the definitions of N_C and N_F to be as before.

Line 1 iterates over all function calls, while line 2 serializes the function call in the current iteration. Serializing a function call will take, on average, constant time. Therefore, performing the first loop will take $O(N_C)$ time in the worst case.

Lines 3 and 4 are similar to lines 1 and 2, only they deal with functions instead of function calls. So, performing the second loop will take at most $O(N_F)$ time.

In all, the worst case running-time complexity of `serialize` is $O(N_C + N_F)$.

3.7 COMPLEXITY

The last thing that remains to be done is to determine the total complexity of the extraction of call information. Luckily, the complexity of the individual phases has already been determined. Furthermore, all the different phases of extraction of call information are performed consecutively, which means that the complexity of the entire process is equal to the sum of the complexities of the individual phases.

As a reminder, the complexities of the different individual phases are:

Phase	**Worst-case complexity**	**Expected complexity**
Parsing	C_P	C_P
Extraction	$C_T +$ $O(N_{Defn} + N_{Decl} + N_F^2 + N_C)$	$C_T +$ $O(N_{Defn} + N_{Decl} + N_C)$
Filtering decls	$O(N_C + N_F)$	$O(N_C + N_F)$
Filtering defs	$O((N_C - N_F) + N_F^2)$	$O(N_C + N_F)$
Name mangling	$O(N_C N_F)$	$O(N_C + N_F)$
Validation	$O(N_C N_F)$	$O(N_C)$
Serialization	$O(N_C + N_F)$	$O(N_C + N_F)$

Table 1: The worst-case and expected-time complexities of the different phases of the extraction process.

When we add up a these individual terms we end up with two equations: one total worst-case complexity and one total expected-time complexity.

3.7.1 *Total worst-case complexity*

Adding all the worst-case complexities in Table 1 gives the following equation for the total worst-case running-time complexity $C_{E,W}$ of the entire extraction process:

$$\begin{aligned} C_{E,Worst} &= C_P \\ &+ C_T + O(N_{Defn} + N_{Decl} + N_F^2 + N_C) \\ &+ O(N_C + N_F) \\ &+ O((N_C - N_F) + N_F^2) \\ &+ O(N_C N_F) \\ &+ O(N_C N_F) \\ &+ O(N_C + N_F) \end{aligned}$$

Some quick simplification brings us to the final equation for the worst-case running-time complexity of the extraction process:

$$C_{E,Worst} = C_P + C_T + O(N_{Defn} + N_{Decl} + N_F(N_F + N_C))$$

3.7.2 *Total expected-time complexity*

When we add all the expected-time complexities in Table 1, we end up with the following equation for the total expected-time complexity $C_{E,Exp}$:

$$\begin{aligned} C_{E,Exp} &= C_P \\ &+ C_T + O(N_{Defn} + N_{Decl} + N_C) \\ &+ O(N_C + N_F) \\ &+ O(N_C + N_F) \\ &+ O(N_C + N_F) \\ &+ O(N_C) \\ &+ O(N_C + N_F) \end{aligned}$$

Removal of duplicate terms now yields:

$$C_{E,Exp} = C_P + C_T + O(N_{Defn} + N_{Decl} + N_F + N_C)$$

If we recall that $0 \leqslant N_F \leqslant N_{Defn} + N_{Decl}$, the equation can be simplified even further to:

$$C_{E,Exp} = C_P + C_T + O(N_{Defn} + N_{Decl} + N_C)$$

This is a very acceptable expected complexity: When we disregard Elsa's complexities of parsing (C_P) and AST traversal (C_T) the expected running time of the extraction process depends only linearly on the number of function definitions, function declarations and function calls.

4 CALL GRAPH CONSTRUCTION

In the previous chapter it was discussed how all the information that is required to construct a call graph is gathered. This chapter will explain how a call graph is constructed from that information. But before that is done, we first determine what properties the resulting call graph should satisfy and what the capabilities of the call graph construction program should be.

The final call graph should satisfy the following *call graph requirements*. It should:

1. Contain one node for each function in the source code and contain one or more edges for each function call in the source code. Function nodes should be annotated with function attributes (3.2.2) and function call edges should be annotated with function call attributes (3.2.4). These annotations will make a wealth of information available to the developer, which can be used to perform analyses.

2. Contain edges for function calls *across translation units and libraries*, not just for function calls made to functions within the same translation unit. Being able to see function calls across translation unit and library boundaries will give developers extremely valuable information on dependencies between different components of a system. Satisfying this requirement is non-trivial, since functions and function calls are extracted *per translation unit*. So, we will need a way to associate function calls to functions across translation unit boundaries.

3. Contain as little false positive function calls as possible, while remaining conservative. In other words, an effort must be made to make sure that the actual call target is present in the set of call candidates, while, at the same time, the size of the set of call targets is kept as small as possible. The reason for this requirement is twofold:

   a) A function node in the graph that only has function call edges to the wrong function nodes is useless, and maybe even misleading, to the developer. This is because the graph will not contain the information the developer is looking for.

   b) A function node which does have a function call edge to the correct function, but which also has a very large number of function call edges to wrong function nodes, is also useless to the developer. This is because the graph will contain so much clutter that it becomes very difficult for the developer to find the information he is looking for.

   It should be noted that it is more important for the graph to contain the correct function call edge, than it is for the graph to be very small. The reason for this is straightforward: Whenever the graph does not contain the correct function call edge, it immediately becomes less valuable to the developer. On the other hand, the graph can very well contain some amount of clutter without becoming significantly less valuable to the developer. 2

4. Contain the relevant containment nodes for every function node in the graph. That is, for every function node, nodes should be present in the graph that represent the function's containing class, file, and directories. Containment edges should be present between these containment nodes and also between

the function node and the containment node that directly contains it. The great advantage of this for the developer is that he will be able to find functions, and groups of functions, according to the way they are organized on the filesystem and according to the way they are organized into classes.

The call graph constructor should satisfy the following *call graph constructor requirements*. It should:

1. Be able to construct a call graph, with the aforementioned properties, of a complete build target. That is, whenever a target consists of multiple translation units, a graph must be generated of all functions and function calls in those translation units (excluding any filtered functions, obviously). The result may be a graph consisting of multiple, disconnected sub-graphs. Having a complete view of the system allows the developer to inspect the function call relations between the different components of the system.

2. Be able to construct a call graph, with the aforementioned properties, of part of a build target, starting in a particular function. As an example, the user should, for instance, be able to generate a call graph starting in the target's `main` function. The resulting graph must then contain all functions that are called, either directly or indirectly, by that `main` function. In any case, the result is a connected graph. Being able to generate a partial call graph allows the developer to more specifically investigate a particular part of the system, with as little clutter as possible.

Now that it is clear what properties the call graph constructor and the graphs that it constructs should satisfy, it is time to elaborate on the different steps that will be taken. As was the case with the extraction of call information, the construction of a call graph is divided into a number of steps. Figure 17 gives an overview of these steps.

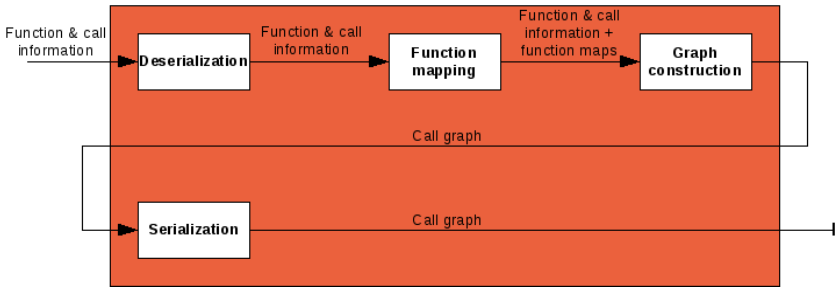

Figure 17: The call graph construction steps of the pipeline.

The first step is deserialization, as is depicted in figure 17. The needed functions and function calls (and their attributes) that were written to file in 3.6, are read from file in this step. At the end of this step, the original function and function call datastructures that were composed during the extraction will have been restored.

In the second step, three maps are constructed that are needed in the next step. These maps translate FQNs and EFTs to functions or sets of functions and are needed during the construction of the call graph. Section 4.2 describes exactly what these maps are for and how they are created.

The third step, described in section 4.3, is the most interesting part of this chapter as it deals with the actual construction of the call graph. It will discuss

linking, how to construct full call graphs and partial call graphs and how to add the containment nodes to the call graph.

Finally, section 4.4 deals with writing out the constructed call graph, with all its fully annotated nodes and edges, to one of the available graph formats.

4.1 DESERIALIZATION

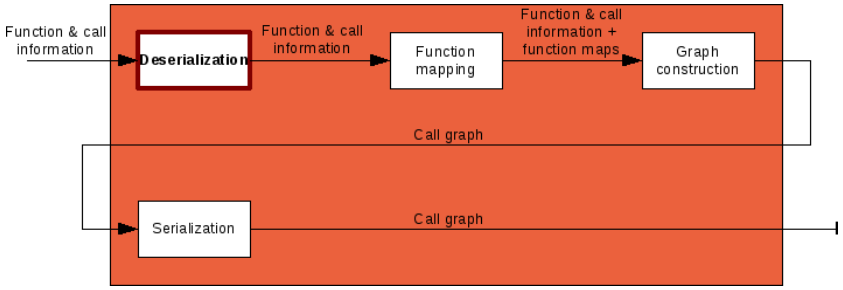

Figure 18: The deserialization step.

Functions and function calls and their attributes were written to file during the serialization step (3.6) of the extraction pipeline. Before the call graph construction program can operate on the extracted data, those data need to be read from file and the original data structures need to be restored. The deserialization step does exactly that. So, input to the deserialization step are the files containing the serialized data and output of the deserialization step are the restored datastructures.

The algorithm that performs the deserialization is rather straightforward and very similar to the serialization algorithm from 3.6. The similarity makes sense, since the data need to be read by the deserialization algorithm in the same order they were written by the serialization algorithm. The pseudo-code of the algorithm that deserializes the functions and function calls from a file is as follows:

```
void deserialize(File & f, FunctionCalls & C, Functions & F)
// C: The set that will contain all read function calls.
// F: The set that will contain all read functions.
{
    // Deserialize all function calls.
    int nC = readNumberOfFunctionCalls(f);
1:  for(int i = 0; i < nC; i++)
    {
2:      FunctionCall c = deserializeFunctionCall(f);
3:      C.push(c);
    }

    // Deserialize all functions.
    int nF = readNumberOfFunctions(f);
4:  for(int i = 0; i < nF; i++)
    {
5:      Function f = deserializeFunction(f);
6:      F.push(f);
    }
}
```

To determine the complexity of `deserialize` we declare the following symbols:

$N_{C,f}$: The number of function calls in file f.
$N_{F,f}$: The number of functions in file f.
N_C : The total number of function calls in all files.
N_F : The total number of functions in all files.

From lines 1 to 3 it can be seen that exactly $N_{C,f}$ function calls will be read from file f. We will assume that reading a function call from disk can be done, at least on average, in $O(1)$ time and we know from [27] that insertion into a hash table can also be done in $O(1)$ time. Therefore, the loop at line 1 will take at most $O(N_{C,f})$ time to complete.

Similar arguments can be made for lines 4 to 6, so we conclude that the loop at line 4 will take at most $O(N_{F,f})$ time to complete.

When we put the complexities of both loops together, the worst-case running-time complexity of `deserialize` comes to $O(N_{C,f} + N_{F,f})$. That means that the worst-case running-time complexity of deserializing all functions and function calls in all files is $O(N_C + N_F)$.

4.2 FUNCTION MAPPING

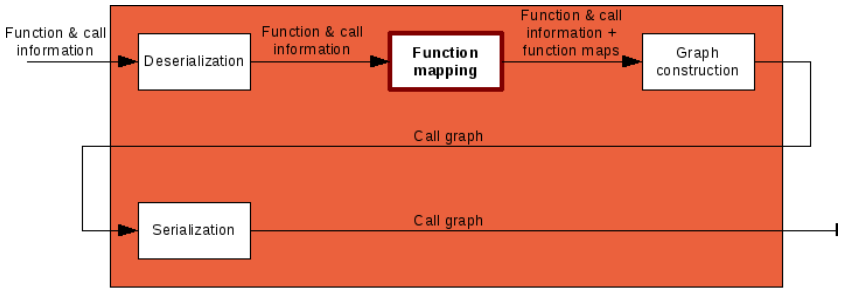

Figure 19: The building of functions maps.

To be able to properly link function calls to functions (or sets of functions) in the next step (4.3), some preparation is needed. More specifically, we will be needing the three maps listed below. Why exactly these maps are needed will become clear in 4.3.1. Before you read on, you might want to go back to section 3.2.2 to review the definitions of *fully qualified name* (FQN) and *extended function type* (EFT).

4.2.1 *FQN to matching function*

This mapping will translate an FQN to the function that it represents. Remember that an FQN always represents at most one function, so this will indeed translate each valid FQN to its corresponding function. To construct this set we merely need to iterate over all functions and add an entry to the map for each function. The following pseudo-code gives the algorithm:

```
void mapFQNToMatchingFunction(Functions & F,
                              map<FQN, Function> & M)
// F: The set of all functions.
// M: The map that will translate an FQN to a function.
{
1:  for(int i = 0; i < length(F); i++)
    {
```

```
        Function f = F[i];
2:      M[f.FQN] = f;
    }
}
```

Determining the complexity of the algorithm is about as simple as the algorithm itself. Line 1 shows that the algorithm iterates over each function in F exactly once. Line 2 then shows that each function is added to the map M. Now, we know from [27] that insertion into a hash table can be done in at most $O(1)$ time. So, the worst-case running-time complexity of `mapFQNToMatchingFunction` is $O(N_F)$, where N_F is the total number of functions.

4.2.2 *EFT to set of matching functions*

This second mapping will translate an EFT to a set of zero or more functions. Each of the functions in this set will have an EFT that is equal to the EFT key of the entry of the mapping. Composing this mapping is simply a matter of iterating over all functions and putting each function in the set behind the correct key. The following pseudo-code illustrates this.

```
void mapEFTToMatchingFunctions(Functions & F,
                               map<EFT, Functions> & M)
{
// F: The set of all functions.
// M: The map that will translate an EFT to a set of functions.
1:  for(int i = 0; i < length(F); i++)
    {
        Function f = F[i];
2:      M[f.FQN].push(f);
    }
}
```

From the little code snippet it can be seen that the algorithm iterates over all functions (line 1) and inserts each function into a hash table (line 2). This means that `mapEFTToMatchingFunctions` will run in at most $O(N_F)$, just like `mapFQN-ToMatchingFunction`.

4.2.3 *FQN to set of overriding functions*

The last mapping translates the FQN of a function f to a set of zero or more functions. Each function in this set overrides function f. To compose this mapping, we need to iterate over all the functions and, for each function, iterate over the functions that it overrides (which we know). For each of those overridden functions, we then add an entry in the mapping. The following algorithm illustrates:

```
void mapFQNsToOverridingFunctions(Functions & F,
                                  map<FQN, Functions> & M)
// F: The set of all functions.
// M: The map that will translate an FQN to the set of functions that
//    override it.
{
    // Iterate over all functions.
    for(int i = 0; i < length(F); i++)
    {
        Function f = F[i];
```

```
            // Iterate over all the functions om that the
            // function f overrides.
            for(int j = 0; j < length(f.overriddenMethods); j++)
            {
                Function & om = f.overriddenMethods[i];

                // Insert an entry in the map indicating that
                // function om is overridden by function f.
                Functions & overridingMethods = M[om.FQN];
                overridingMethods.insert(f);
            }
        }
    }
```

It should be noted that the set `f.overriddenMethods` contains the set of methods that function `f` overrides, either directly or indirectly. That means, that if there are N_F functions, in the worst case each function overrides on average $\frac{1}{2}N_F$ other functions. Adding to that the fact that insertion into a hash table takes at most $O(1)$ time, we come to a worst-case run-time complexity of the above algorithm of $O(\frac{1}{2}N_F^2)$. However, since oftentimes most functions will not be overridden nor override another function, the running-time will in practice be much lower than this.

4.3 CONSTRUCTING CALL GRAPHS

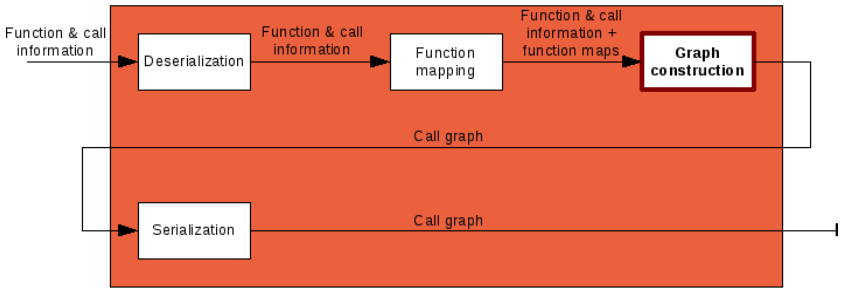

Figure 20: The call graph construction step.

After extracting the information we need from the source code in the previous chapter and doing some preparation in this chapter, finally we have come to the good part: The actual construction of call graphs.

In this section we will describe how call graphs can be constructed of a complete target or part of a target and how we can include containment nodes and edges in these graphs. The first thing that will be discussed, however, is how function calls are linked to functions.

4.3.1 *Linking*

To be able to construct a call graph we need to be able to associate function function calls with the functions they call. Furthermore, we need to do this both for calls and functions *within a single translation unit*, as for calls and functions *across translation units*. This process is called linking.

Linking is performed using two properties of function calls and functions. Firstly, for function calls that are made through a pointer-to-function or a pointer-

to-member, linking is performed on *EFTs*. Secondly, for all other function calls, linking is performed on *FQNs*.

Different linking properties of types of function calls

We can distinguish roughly three types of function calls with different linking properties, which will now be discussed. After that, we will go into more detail on exactly what situations can occur.

1. In the case of a call to a plain C-style function, the name of the call target (which can be retrieved from the function call), will contain the FQN of the function that is called (see 3.2.4). For that function call, this FQN will be used to look up the function to which the call must be linked. It is obvious that there is only a single call target, which is also guaranteed to be the correct call target. So, this situation perfectly satisfies call graph requirement 3.

2. When a call via a pointer-to-function or pointer-to-member is detected, the EFT of the called function can be extracted from the function call. All functions that have an EFT matching this extracted EFT are now potential call candidates. Call graph requirement 3 states that it is more important that the actual call target is included in the set of call targets then it is to have the smallest possible set of call targets. Therefore, every function that has a matching EFT is then linked to the function call.

3. Whenever a function call is made to a virtual method, the method matching the FQN of the call site is linked to the function call. If, however, the call is made on a pointer to an object, or a reference to an object, then there are more potential call candidates than just the method with the matching FQN. Namely, all methods that override the matching method are also call candidates. Now, we have no way to determine with absolute certainty whether a function from the set of call candidates is the actual call target. Since call graph requirement 3 states that we must make a best effort to make sure that the actual call target is present in the set of call candidates, all call candidates will be linked to the function call.

Now that we have a feeling for what will happen during linking, it is time to analyze more precisely what situations we can encounter. Table 2 below lists all the different types of call targets a function call can have. It also indicates how many functions the call can be linked to.

The last column in the table says something about the conservativeness of the set of call targets. A set of call targets is conservative when it contains the function that is actually called at the call site. Of course it is not known what function in the set that is; if we did, we would have only a single call target. So, the next best thing is to be certain that the called function is in the set of call targets. The last column in the table indicates whether or not it is certain that the called function is present in the set of call targets, or, in other words, whether or not we are certain that the set of call targets is conservative. Note however, that even if we cannot guarantee that the set of call targets is conservative, it often still is conservative in practice.

Note that the first column in table 2 refers to the three scenario's listed above and that the second column contains exactly the types of function calls that were discussed in 3.2.4.

For calls of type 'direct function' it is easy to see that they can only call C-style functions and that there is always exactly one call target. The same goes for calls of type 'direct method', regardless of whether the method is virtual.

Scn.	Type of call	Type of target	# of targets	Consv.
1	Direct function	C-style function	1	Yes
1	Direct method	C++-method	1	Yes
1	Object pointer	non-virtual C++-method	1	Yes
3	Object pointer	virtual C++-method	1 or more	No
1	Object reference	non-virtual C++-method	1	Yes
3	Object reference	virtual C++-method	1 or more	No
1	Constructor	C++-method	1	Yes
1	Destructor	C++-method	1	Yes
2	Pointer-to-function	C-style function	0 or more	No
2	Pointer-to-member	non-virtual C++-method	0 or more	Yes
2	Pointer-to-member	virtual C++-method	0 or more	No

Table 2: The different types of functions calls that exist and their properties.

It is also true for calls of type 'object pointer' and 'object reference' in case their call target is non-virtual. When their call target is virtual though, then there can be more than one call target. This is because the function that is called can be the one referred to by the FQN of the call target, or any of the functions that override it. The reason why we cannot be certain that we can find the actual call target was explained in 3.3.3.

Calls to constructors and destructors are similar to calls of type 'direct method': There is always exactly one call target and we know for sure that its the correct call target.

Calls via a 'pointer-to-function' only have C-style functions and static class methods as call targets. Any C-style function and any static class method in the system that has an EFT that matches the EFT of the call target is a call candidate. So, if no functions are found that match, then there are no call candidates. Hence, such function calls can have zero or more call targets. Also, we can never be sure that the function that is actually called can be detected. Again, the reason for this is explained in 3.3.3.

The last case is a function call via a 'pointer-to-member'. Now, to be able to declare a pointer-to-member variable, the class of the member to which will be pointed must be known. So that means, that the only call candidates are the non-static members of that class that have a matching EFT. In the case that none of the matching methods is virtual, we known for sure that the actual call target is one of the methods of that class. If one or more of the matching methods is virtual, however, then any methods overriding those methods are also call candidates. Also, we then no longer know for sure that the actual call target is detectable, for the same reason as was the case with 'object pointer' and 'object reference'.

The linking algorithm

At this point it is clear what we can expect from the different types of function calls, so it is time to investigate the algorithm that links a function call to a set of functions. The algorithm handles the three different scenarios discussed earlier, each of which handles a number of the function call types listed in table 2. Each scenario, in turn, handles a number of the function call types listed in table 2. Table 2 also shows which scenario handles which call types. Together, these three

scenarios handle all of the different types of function calls that may be encountered. The pseudo-code below illustrates the algorithm:

```
void linkToFunctions(FunctionCall & c,
                     Functions & F,
                     map<FQN, Functions> & M,
                     map<EFT, Functions> & P,
                     Functions & T)
// c: The function call for which to resolve call
//    candidates.
// F: The set of all functions.
// M: The mapping of FQN to overriding functions.
// P: The mapping of EFT to matching functions.
// T: Will contain all detectable call targets of c on
//    return.
{
    // Scenario 1
    // If we know for sure that the function call has exactly
    // one call candidate, simply insert that into T.
    if(c.isDirectFunction || c.isDirectMethod ||
       c.isConstructor    || c.isDestructor)
    {
1:      Function f = F.getFunctionByFQN(c.FQN);
2:      T.push(f);
    }

    // Scenario 2
    // If the type of function call is object pointer or
    // object reference, then insert the call target t
    // identified by the call's FQN into T. Next to that,
    // insert all call targets that override t.
    else if(c.isObjectPointer || c.isObjectReference)
    {
3:      Function f = F.getFunctionByFQN(c.FQN);
4:      T.push(f);

        // Note that if function f is not virtual, then
        // M[f.FQN] below will be empty and f will be the only
        // call target inserted into T.
5:      Functions & overridingMethods = M[f.FQN];
6:      for(int i = 0; i < length(overridingMethods); i++)
        {
7:          Function om = overridingMethods[i];
8:          T.push(om);
        }
    }

    // Scenario 3
    // If the type of function call is pointer-to-function or
    // pointer-to-member, then insert all functions that have a
    // matching EFT into T.
    else if(c.isPointerToFunction || c.isPointerToMember)
    {
9:      Functions & matchingFunctions = P[c.EFT];
10:     for(int i = 0; i < length(matchingFunctions); i++)
```

```
        {
11:         Function mf = P[i];
12:         T.push(mf);
        }
    }
}
```

Note that the function call c can, in principle, be linked to any function in F. Since the set F contains the functions from all translation units, the `linkToFunctions` algorithm satisfies call graph requirement 2.

Complexity of the linking algorithm

We will now analyze the complexity of linking a single function to its call targets. The code the three different scenarios, all of which seem to have a different complexity. So, we will simply determine the complexity of each scenario and state that the worst of those complexities is the worst-case running-time complexity of `linkToFunctions`. First, though, consider again the following symbols:

$$
\begin{aligned}
N_F &: \text{The number of functions in } F. \\
N_M &: \text{The number of elements in } M, \text{ with } N_M \leqslant N_F. \\
N_P &: \text{The number of elements in } P, \text{ with } N_P \leqslant N_F.
\end{aligned}
$$

1. *Scenario 1*. We know for sure there is exactly one call target. In line 1 a search is performed on a hash table, which takes $O(N_F)$ time in the worst case, but is expected to take $O(1)$ time. Immediately afterwards, in line 2, the found function is inserted into a list, which takes at most $O(1)$ time.

2. *Scenario 2*. We are dealing with a call on an object pointer or object reference, so there are one or more call targets. Lines 3 and 4 are similar to lines 1 and 2 and have equal complexities. At line 5, the functions overriding function f are retrieved from hash table M, taking at most $O(N_M)$ time. In the worst case, every function in F overrides f (except, of course, f itself). In that case, lines 7 and 8 will run $N_F - 1$ times. Line 7 retrieves an element from an array and line 8 inserts an element into a list. Both take at most $O(1)$ time, so the loop at line 6 will never take more than $O(N_F)$ time to complete.

3. *Scenario 3*. In this case, we are dealing with a call via pointer-to-function or pointer-to-member, which means that there can be zero or more call targets. The number of call targets depends on how many functions in F have an EFT that matches the EFT extracted from the call site. Obviously, in the worst case, all functions in F have a matching EFT. So, the complexity for this scenario comes to $O(N_P)$ for line 9 and $O(N_F)$ for the loop at line 10, since lines 11 and 12 are similar to lines 7 and 8.

Now, when we list the complexities for each of the three scenarios, we come to the following table:

The table above shows complexities that are, at first glance, hard to compare: How do we now which is bigger, N_M or N_P? However, if we refer again to the definitions of N_M and N_P it can be seen that $N_M \leqslant N_F$ and $N_P \leqslant N_F$. So, that means we can conclude that the worst-case running-time complexity of each of the three scenarios is $O(N_F)$. That, in turn, leads us to conclude that the worst-case running-time complexity of `linkToFunctions` is also $O(N_F)$, regardless of the three scenarios.

Scenario	Complexity
1	$O(N_F)$
2	$O(N_M + N_F)$
3	$O(N_P + N_F)$

Table 3: The complexities of the possible scenarios of `linkToFunctions`.

4.3.2 *Constructing full call graphs*

Now that it is clear how function calls are to be linked to functions, we will use that information to construct the algorithm that builds an actual call graph. First we focus on how to build a call graph of the complete system (call graph construction requirement 1) and then on how to build a call graph of part of the system (call graph construction requirements 2). Before the code of the algorithm that constructs a full call graph is given, we will first describe the idea behind the algorithm.

As we are constructing a graph of the *complete* system, that means *all functions* and *all function calls* from the system must be present in the call graph. So, a natural approach to make sure all functions and calls are included in the graph is to simply iterate over all functions and calls and add them to the graph one by one.

For the functions (i.e., the nodes), this is extremely simple: Just add a node to the graph for each function.

For the function calls (i.e., the edges), this is a bit more complex, since we need to figure out what pairs of nodes to connect by edges. Luckily, given a function call, we can use the `linkToFunctions` function to determine the set of functions that are possibly invoked by the function call. We can then insert an edge from the calling function to each of the possibly invoked functions into the graph.

Then, using the calling function and the set of called functions, an edge from the calling function to each of the called functions can be inserted into the graph.

Another problem that needs to be solved, comes from the initializing and finalizing function calls (see 3.2.5). These calls occur before and after the call to the `main` function, respectively. So, the question that arises is, which function makes these initializing and finalizing calls? The answer is that there is no function *in the source code* that makes these function calls, just like there is no function in the source code that calls the `main` function. However, to be able to include edges in the graph for these calls we need a source node at which these edges can start. This problem is solved by introducing a fictional node into the graph called the *Root* node. This node represents a function that is not called by any other function, but that makes calls to the initializing and finalizing functions and the `main` function. As an example, consider the following code snippet:

```
class A { };

A a;

int main(int argc, char** argv)
{
    return 0;
}
```

The small program above calls, respectively, the constructor of `A`, the `main` function and finally the destructor of `A`. The call graph, including the fictional Root node (but excluding containment nodes), is depicted in figure 21 below.

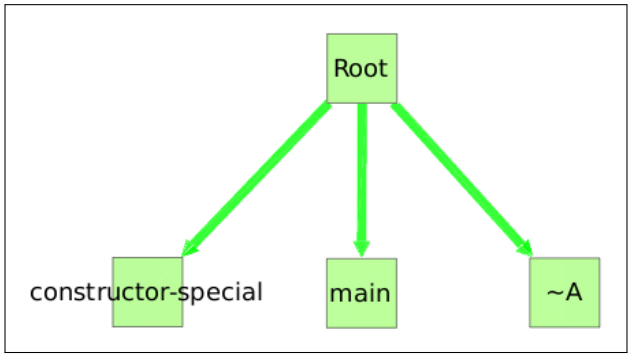

Figure 21: The call graph including the fictional Root node.

A question that might come to mind is what containment node (i.e., directory, file or class) contains this Root node? The answer is simple: The Root node is always contained by the file node representing the file that defines the `main` function. The respective initializing and finalizing call edges are, however, annotated with attributes which point the developer to the file and position where the call is made in the source code. Now, from the answer above immediately a new question rises. What happens if there is no `main` function, for instance, in case we are investigating a library? Again, the answer is simple: In that case there is no Root node and there will be no initializing and finalizing edges present in the graph. The rationale for this is that when a developer is investigating a library on its own, then that library on its own will never be able to make these initializing and finalizing calls; it needs a `main` function (or actually, a program with a `main` function linking to the library) to define where the initializing and finalizing function calls are made. In line with this is the following: As soon as the library is used by a program which does have a `main` function, then the location of the initializing and finalizing function calls is defined and they will be present in the graph.

So, in summary, to make sure that all function calls are inserted into the graph, we take the following three steps:

1. Iterate over all functions and insert a node into the graph for each function.
2. Iterate over all function nodes in the graph. Since we already inserted all functions into the graph, it is guaranteed that by iterating over the function nodes in the graph we will encounter all functions. For each function we find in the graph, the list of function calls that it makes is retrieved. Using that list, we iterate over all function calls made by the current function node and we insert edges from the current function node to each of the functions in the set calculated by `linkToFunctions`.
3. Insert the fictional Root node into the Graph, if a `main` function exists.
4. Insert a call edge from the Root node to the `main` function. That is, if there is a `main` function: Libraries, for instance, generally do not have a main function.
5. Iterate over all initializing and finalizing function calls and insert the appropriate edges, beginning in the Root node, into the graph.

The full call graph construction algorithm

Below is the pseudo-code of the algorithm that builds a full call graph.

```
void buildFullCallGraph(Functions & F,
                        FunctionCalls & Ini,
                        FunctionCalls & Fin,
                        map<FQN, Functions> & M,
                        map<EFT, Functions> & P,
                        Graph & G)
// F: The set of all functions.
// Ini: The set of all initializing function calls.
// Fin: The set of all finalizing function calls.
// M: The mapping of FQN to overriding functions.
// P: The mapping of EFT to matching functions.
// G: The graph that will contain the result.
{
    // Insert nodes into the graph for all functions in F.
1:  insertAllNodes(F, G);

    // Insert edges into the graph for the function calls made
    // by the functions just inserted into G.
2:  insertEdges(F, M, P, G);

    // Insert the fictional root node into the graph, if a main
    // function exists.
3:  Function root = insertRootNode(G);

    // Insert the call edge from the Root node to the main
    // function, if a main function exists.
4:  insertCallEdgeFromRootToMain(root, F, G);

    // Insert edges into the graph for the initializing
    // function calls.
5:  insertEdgesFromNode(root, Ini, M, P, G);

    // Insert edges into the graph for the finalizing
    // function calls.
6:  insertEdgesFromNode(root, Fin, M, P, G);
}

void insertAllNodes(Functions & F, Graph & G)
{
    // Insert a node into the graph for each function in F.
7:  for(int i = 0; i < length(F); i++)
    {
        Function f = F[i];
8:      insertNode(f, G);
    }
}

void insertEdges(Functions & F,
                 map<FQN, Functions> & M,
                 map<EFT, Functions> & P,
                 Graph & G)
```

```
{
    // Iterate over all function nodes in the graph and
    // retrieve the function calls that each function makes.
    // Then, insert edges for those function calls.
9:  for(int n = 0; n < length(G.nodes); n++)
    {
        Function source = G.nodes[n];
        FunctionCalls C = source.calls;
10:     insertEdgesFromNode(source, C, F, M, P, G);
    }
}

void insertEdgesFromNode(Function & source,
                         FunctionCalls & C,
                         Functions & F,
                         map<FQN, Functions> & M,
                         map<EFT, Functions> & P,
                         Graph & G)
{
    // Iterate over all function calls in C and add edges for
    // each of them.
11: for(int e = 0; e < length(C); e++)
    {
        FunctionCall c = C[e];
        Functions T;

        // Retrieve the set of call targets T for function.
        // call c
12:     linkToFunctions(c, F, M, P, T);

        // For each call target, add an edge from the source
        // node to the call target to the graph.
13:     for(int t = 0; t < length(T); t++)
        {
            Function destination = T[t];
14:         insertEdge(source, destination, G);
        }
    }
}

Function & insertRootNode(Graph & G)
{
    Function root = new Function("Root");

15: insertNode(root, G);

    return root;
}

void insertCallEdgeFromRootToMain(Function & root, Functions & F,
                                  Graph & G)
{
    // Try to find the main function.
16: Function main = F.findMain();
```

```
    // If there is a main function, insert an edge from the root
    // node to the main function into the graph.
    if(main != NULL)
    {
17:     insertEdge(root, main, G);
    }
}
```

Duplicate nodes for functions with static linkage

The above algorithm inserts a node into the graph for each function. However, before we conclude on how the above algorithm satisfies call graph requirement 1, we should note that our name mangling approach has some consequences at this point. Recall that we apply name mangling (3.4) to prevent functions with static linkage to be linked to function calls from other translation units. The idea is to make the FQN of the functions in question globally unique with respect to the translation unit in which they are defined. This means, however, that for every translation unit we actually generate *a new copy* of those functions. This is, in some sense, correct, since function calls are never linked to incorrect functions and one could argue that, by being visible only in their own translation units, these copies are indeed separate functions. Regardless, this is not how a developer will perceive these functions: The functions occur *only once in the source code*, so a developer will expect to see these functions *only once in the call graph*. This is in line with call graph requirement 1, which states that the call graph should contain *one node* for each function *in the source code*.

The solution to this issue is to *merge all copies of a function with static linkage into a single node* during the construction of the call graph. We will call such nodes *shared nodes*. In other words, a shared node in the call graph represents all (i.e., one or more) copies of a function with static linkage.

To realize this we need a way to determine *when* a function must be represented by a shared node, which is easy: All functions that have static linkage must be represented by a shared node. Some of these shared nodes will consequently represent only a single copy, but that poses no problems.

The second thing we need is a way to determine *by what shared node* a function must be represented. This required a bit more attention. Let us first define when two functions are copies of each other. Two functions are copies of each other iff:

- Both functions have static linkage, and
- both functions are defined in the same source file, at the same location, and
- both functions have equal unmangled FQNs.

The approach will be as follows. During construction of the graph, we will keep a mapping that translates the triplet (*source file name*, *location*, *unmangled FQN*) to the corresponding shared function node. Now, when a node is inserted into the graph for a function with static linkage, we determine its information triplet (*source file name*, *location*, *unmangled FQN*) and we use that to lookup the corresponding shared function node in the mapping. If no such node exists yet, it is created and inserted into the mapping and into the graph. If it does exist, we do nothing.

Lastly, when an edge is inserted into the graph and the edge has at least one static function node as an endpoint, we again determine its information triplet and retrieve the corresponding shared node from the mapping. The shared node is then used as the endpoint of the edge.

It is not hard to see that using shared nodes does not increase the complexity of the overall graph construction algorithm: The graph nodes are stored using hash

tables, so the use of another hash table (which will contain information of only a subset of the graph) will not increase the computational complexity.

The solution presented above ensures that for each function in the source code precisely one node ends up in the call graph and that one or more call edges are inserted into the graph for each function call in the source code (`insertEdges`, `insertEdgesFromNode`). Furthermore, function and function call attributes are easily available from the inserted nodes and edges because references to the corresponding functions and function calls which are annotated with the required attributes are included with the nodes and edges. Hence, `buildFullCallGraph` satisfies call graph requirement 1. Besides that, by being able to construct a complete call graph of a build target, `buildFullCallGraph` also directly satisfies call graph constructor requirement 1.

Complexity

The last thing to do is to determine the complexity of `buildFullCallGraph`. The code above shows that the algorithm consists of several functions, so we begin by defining the symbols for the complexities of these individual functions:

C_{BFG} : The complexity of `buildFullCallGraph`.
C_{IAN} : The complexity of `insertAllNodes`.
C_{IED} : The complexity of `insertEdges`.
C_{IRN} : The complexity of `insertRootNode`.
C_{IEM} : The complexity of `insertCallEdgeFromRootToMain`.
$C_{IEN,5}$: The complexity of `insertEdgesFromNode`, called from line 5.
$C_{IEN,6}$: The complexity of `insertEdgesFromNode`, called from line 6.
$C_{IEN,10}$: The complexity of `insertEdgesFromNode`, called from line 10.

Now, lines 1 through 6 are simply calls to each of the other functions from the algorithm, so we can calculate the complexity of `buildFullCallGraph` by adding the complexities of the functions it calls. Therefore:

$$C_{BFG} = C_{IAN} + C_{IED} + C_{IRN} + C_{IEM} + C_{IEN,5} + C_{IEN,6} \tag{4.1}$$

Finally, before we define the complexities of the other individual function, consider the following symbols:

N_F : The total number of functions.
$N_{C,I}$: The number of initializing function calls.
$N_{C,F}$: The number of finalizing function calls.
$N_{C,Avg}$: The average number of function calls per function.
N_C : The total number of function calls.

Please note that the total number of function calls N_C can be expressed in terms of N_F, $N_{C,I}$, $N_{C,F}$ and $N_{C,Avg}$ as follows:

$$N_C = (N_{C,Avg} N_F) + N_{C,I} + N_{C,F} \tag{4.2}$$

Complexity of insertAllNodes

Lines 7 and 8 show that `insertAllNodes` iterates over all functions, and inserts a node into the graph for each function. Now, there are N_F functions to iterate over and the internal datastructures used to store the graph are hash tables. That means that the worst-case running time complexity of `insertAllNodes` is:

$$C_{IAN} = O(N_F)$$

Complexity of insertEdges

From line 9 we can see that `insertEdges` iterates over all nodes in the graph. Since `insertAllNodes` just inserted all functions into the graph, we know that there are exactly N_F nodes to iterate over. Then, during each iteration of the loop, a call is made to `insertEdgesFromNode` at line 10. So, we can conclude that:

$$C_{IED} = C_{IEN,10}O(N_F)$$

Complexity of insertRootNode

From line 15 it is quickly obvious that

$$C_{IRN} = O(1)$$

Since inserting a new node into the graph can be done in at most $O(1)$ time.

Complexity of insertEdgesFromNode

The function `insertEdgesFromNode` iterates over a set C of function calls. How many function calls are in C, however, differs. When called from line 10, the size of C will be $N_{C,Avg}$, but when called from line 5 there will be $N_{C,}I$ elements in C and when called from line 6 there will be $N_{C,F}$ elements. To resolve this, we will explicitly state a different complexity for each of these cases, respectively called $C_{IEN,10}$, $C_{IEN,5}$ and $C_{IEN,6}$.

During each iteration of the loop, a call is made to `linkToFunctions`, which has a worst-case running-time complexity of $O(N_F)$. In the absolute worst-case that every function is a call target of the current iteration's function call, then the loop at 13 will also having a running-time complexity of N_F, since inserting an edge into the graph at line 14 can be done in $O(1)$ time.

So, for the different calls we have the following different complexities:

$$\begin{aligned} C_{IEN,5} &= O(N_{C,I}N_F) \\ C_{IEN,6} &= O(N_{C,F}N_F) \\ C_{IEN,10} &= O(N_{C,Avg}N_F) \end{aligned}$$

Complexity of insertCallEdgeFromRootToMain

As we noted before, [27] show us that doing a lookup operation on a hash table takes, in the worst case, $O(N_F)$ time. Since line 17 will complete in $O(1)$ time, we conclude that:

$$C_{IEM} = O(N_F)$$

Total complexity

We now know the complexities of the individual functions, which means we can now calculate the total complexity of building a full call graph using equation 4.1. When we substitute the symbols in equation 4.1 for the complexities that we calculated, we end up with the following expression for C_{BFG}:

$$\begin{aligned} C_{BFG} &= C_{IAN} + C_{IED} + C_{IRN} + C_{IEM} + C_{IEN,5} + C_{IEN,6} \\ &= O(N_F) + C_{IEN,10}O(N_F) + O(1) + O(N_{C,I}N_F) + O(N_{C,F}N_F) \\ &= O(N_{C,Avg}N_FN_F) + O(N_{C,I}N_F) + O(N_{C,F}N_F) \\ &= O(N_F(N_{C,Avg}N_F + N_{C,I} + N_{C,F})) \end{aligned}$$

Now, when we look again at equation 4.2, it becomes clear that we can simplify this equation as follows, to arrive at the final equation for the complexity of `buildFullCallGraph`:

$$C_{BFG} = O(N_F N_C) \tag{4.3}$$

4.3.3 *Constructing partial call graphs*

Partial call graphs represent only part of the build target that is being analyzed. More specifically, the partial call graph G_f is that part of the full call graph G that is reachable from a given function node f, the starting point. To illustrate that partial call graphs can be extremely useful, please consider again the code of the Hello World program presented in section 3.3:

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("Hello World!\n");
    return 0;
}
```

The problem descibed in section 3.3 was that when a build target makes use of libraries, all functions from those libraries are included in the call graph of the build target, regardless of whether the functions are actually used in the build target. As an alternative to calculating the entire graph of the Hello World program, we can calculate only that part of the graph that starts in the `main` function of the program. Figure 22 depicts the resulting partial call graph G_{main}.

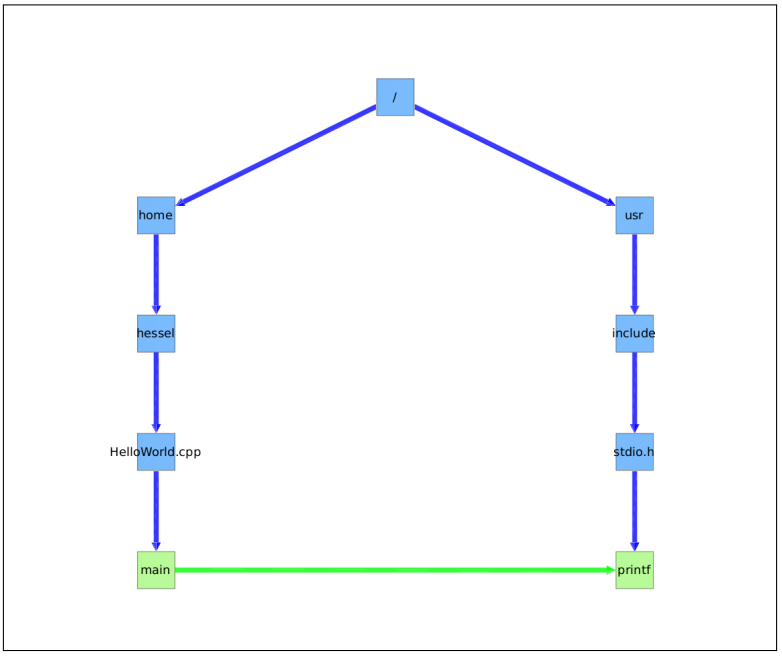

Figure 22: The partial call graph starting in the `main` function of the Hello World program.

Now compare the graph from figure 22 with the call graph shown in figure 11, which is the graph that was originally expected for the Hello World program. It can be seen that the partial call graph G_{main} exactly matches the graph that one would intuitively expect for the Hello World program.

The partial call graph construction algorithm

The algorithm for calculating a partial call graph strongly resembles the algorithm for calculating a full call graph. The only difference between the two algorithms is we do not want to add *all* functions to the graph, but only those functions that are reachable from the starting point, including the starting point itself. After the correct nodes have been added, the remainder of the algorithm is identical to the full call graph construction algorithm.

To determine what functions are reachable from the starting point f, the algorithm retrieves the function calls made by f and links those calls to their call targets. Then, all the call targets are added to the graph and each call target is recursively processed to make sure all functions reachable from those functions are also added to the graph. The pseudo-code of the algorithm is given below. Please note that the code for the functions called in lines 2 through 6 was already given in the full call graph construction algorithm.

```
void buildPartialCallGraph(Function & f,
                           Functions & F,
                           FunctionCalls & Ini,
                           FunctionCalls & Fin,
                           map<FQN, Functions> & M,
                           map<EFT, Functions> & P,
                           Graph & G)
// f: The function in which the call graph must start.
// F: The set of all functions.
// Ini: The set of all initializing function calls.
// Fin: The set of all finalizing function calls.
// M: The mapping of FQN to overriding functions.
// P: The mapping of EFT to matching functions.
// G: The graph that will contain the result.
{
    // Insert nodes into the graph for those functions in F that
    // are reachable from f, including f itself.
1:  insertNodesStartingAt(f, F, G);

    // Insert edges into the graph for the function calls made
    // by the functions just inserted into G.
2:  insertEdges(F, M, P, G);

    // Insert the fictional root node into the graph, if a main
    // function exists.
3:  Function root = insertRootNode();

    // Insert the call edge from the Root node to the main function,
    // if a main function exists.
4:  insertCallEdgeFromRootToMain(root, F, G);

    // Insert edges into the graph for the initializing
    // function calls.
5:  insertEdgesFromNode(root, Ini, M, P, G);

    // Insert edges into the graph for the finalizing
    // function calls.
6:  insertEdgesFromNode(root, Fin, M, P, G);
```

```
}

void insertNodesStartingAt(Function & f, Functions & F, Graph & G)
{
7:  insertNode(f, G);
    FunctionCalls C = f.calls;
8:  for(int i = 0; i < length(C); i++)
    {
        FunctionCall c = C[i];
        Functions T;

        // Retrieve the set of call targets for function call c.
9:      linkToFunctions(c, F, M, P, T);

        // Recursively process each call target.
        for(int t = 0; t < length(T); t++)
        {
            Function target = T[t];
            // If this function is not yet present in the call
            // graph, process it.
            if(!G.nodes.Contains(target)
            {
                 insertNodesStartingAt(target, F, G);
            }
        }
    }
}
```

The `buildPartialCallGraph` function inserts a node into the graph for each function that is reachable from function `f` (`insertNodesStartingAt`) and inserts one or more edges into the graph for each function call (`insertEdges`, `insertEdgesFromNode`). Again, function and function call attributes are easily available from the inserted nodes and edges since function and function call references (which are annotated with the required attributes) are included with the nodes and edges. Therefore, `buildPartialCallGraph` satisfies call graph requirement 1. Ofcourse, since `buildPartialCallGraph` constructs a call graph of part of a build target starting in function `f`, it directly satisfies call graph constructor requirement 2.

Determining the complexity of `buildPartialCallGraph` can be done relatively quickly, since complexities have already been determined for all but the `insertNodesStartingAt` function. So, all we need to do is determine the complexity C_{ISA} of `insertNodesStartingAt` and then replace the C_{IAN} term from equation 4.1 with C_{ISA}. We then end up with the following equation for the complexity C_{BPG} of constructing a partial call graph:

$$C_{BPG} = C_{ISA} + C_{IED} + C_{IRN} + C_{IEM} + C_{IEN,5} + C_{IEN,6} \tag{4.4}$$

To aid in calculating its complexity, consider the following definitions:

N_F : The total number of functions.
$N_{F,f}$: The number of function nodes in the partial call graph G_f, with $N_{F,f} \leqslant N_F$.
N_C : The total number of function calls.
$N_{C,f}$: The number of function calls that have at least one corresponding edge in the partial call graph G_f, with $N_{C,f} \leqslant N_C$.

Also, recall from 4.3.1 that the worst-case running time complexity of `linkToFunctions` is $O(N_F)$. There are two observations here that are important to make:

1. Function `insertNodesStartingAt` will be called once for every function that is part of the partial call graph. That means that line 7 will run exactly $N_{F,f}$ times.
2. After the algorithm finishes, line 8 will have iterated once over all function calls for which one or more edges have been inserted into the graph. Thus, it will have iterated over $N_{C,f}$ function calls, meaning that line 9 will also have run exactly that many times.

The remainder of the algorithm, that is after line 9, deals with recursively calling `insertNodesStartingAt`. Since the above two observations talk about the total number of times that lines 7 and 9 are executed, and the part of the algorithm after line 9 has a complexity of $O(1)$, we can disregard everything after line 9 from the complexity analysis.

Now, using the above two observations we can conclude that the worst-case running-time complexity C_{ISA} of `insertNodesStartingAt` is:

$$C_{ISA} = O(N_{F,f} + N_{C,f}N_F)$$

However, since $N_{F,f} \leqslant N_F$, this equation can be reduced to:

$$C_{ISA} = O(N_{C,f}N_F)$$

The last step is to plug this equation into equation 4.4:

$$\begin{aligned} C_{BPG} &= C_{ISA} + C_{IED} + C_{IRN} + C_{IEM} + C_{IEN,5} + C_{IEN,6} \\ &= O(N_{C,f}N_F) + C_{IEN,10}O(N_F) + O(1) + O(N_{C,I}N_F) + O(N_{C,F}N_F) \\ &= O(N_{C,f}N_F) + O(N_{C,Avg}N_FN_F) + O(N_{C,I}N_F) + O(N_{C,F}N_F) \\ &= O(N_{C,f}N_F) + O(N_F(N_{C,Avg}N_F + N_{C,I} + N_{C,F})) \end{aligned}$$

Just like we did before, this can be reduced to the following, because of 4.2:

$$C_{BPG} = O(N_{C,f}N_F + N_FN_C)$$

Finally, because $N_{C,f} \leqslant N_C$ we come to the following final equation for the worst-case running-time complexity of `linkToFunctions`:

$$C_{BPG} = O(N_FN_C)$$

When we compare this with the complexity of `buildFullCallGraph` (4.3), we see that both algorithms have the same worst-case running-time complexity. This makes sense, since in the worst-case for `buildPartialCallGraph` the constructed partial call graph is equal to the full call graph constructed by `buildFullCallGraph`.

4.3.4 *Inserting containment nodes*

All requirements have been satisfied, except one: call graph requirement 4. This requirement states that the resulting call graph should include the relevant containment nodes and edges for its function nodes and function call edges. There are three types of containment nodes:

1. *Class nodes*. These nodes represent classes from the source code. Every class from the source code should have exactly one corresponding class node in the graph. A class node has outgoing containment edges to the methods that belong to the class it represents.
2. *File nodes*. Not suprisingly, file nodes represent files on the file system. A file is represented as a file node in the graph if it contains at least one class or function that is represented in the graph as a node.
3. *Directory nodes*. Similar to file nodes, directory nodes represent directories on the file system. Like file nodes, directories are only included in the graph if they contain at least one file that is represented in the graph.

As an example, please consider the following calculator program, which performs some basic arithmetic operations:

```
// ----- File: Calculator.h -----
#ifndef CALCULATOR_H
#define CALCULATOR_H

class Calculator
{
public:
    float Add     (float a, float b) { return a + b; }
    float Subtract(float a, float b) { return a - b; }
    float Multiply(float a, float b) { return a * b; }
    float Divide  (float a, float b) { return a / b; }
};

#endif

// ----- File: Main.cpp -----
#include "Calculator.h"

int main(int argc, char** argv)
{
    Calculator calc;
    calc.Add     (1.0f, 2.0f);
    calc.Subtract(1.0f, 2.0f);
    calc.Multiply(1.0f, 2.0f);
    calc.Divide  (1.0f, 2.0f);
    return 0;
}
```

The calculator program consists of two files:

- `Calculator.h`, containing the implementation of the `Calculator` class.
- `Main.cpp`, containing the `main` function, which uses the Calculator class to perform some calculations.

When we look at the call graph belonging to this program (figure 23), it can be seen that it has six containment nodes (blue) and eight containment edges (also blue). Closer inspection reveals that the top three containment nodes are directory nodes: They represent the path `/home/hessel`. The two nodes below that represent the `Main.cpp` and `Calculator.h` files, which are located in the `/home/hessel` directory. Finally, there is the `Calculator` containment node, which represents the `Calculator` class, located in the `Calculator.h` file.

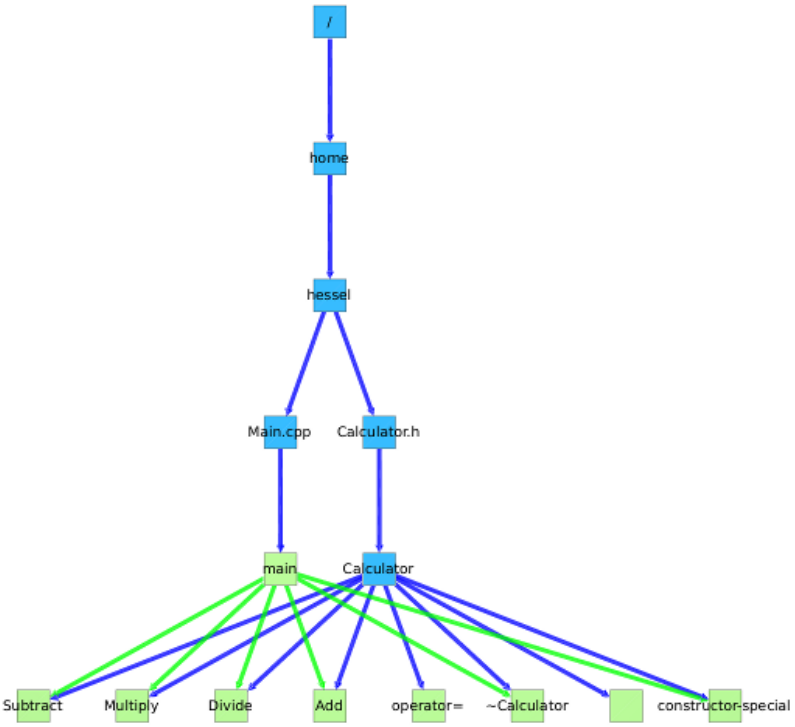

Figure 23: The containment nodes and edges of the Calculator program (in blue). The function nodes and function call edges are shown in green.

The containment node insertion algorithm

It is clear that the final graph should only contain those directories, files and classes that contain, either directly or indirectly, functions that are already present in the graph. The natural way of implementing this is to insert the appropriate containment nodes and edges whenever a new function node is inserted into the graph. The code below shows the implementation of the `insertNode` function used by the `buildFullCallGraph` and `buildPartialCallGraph` algorithms.

Next to actuall adding a function node to the graph, the implementation of `insertNode` makes sure that the appropriate containment nodes and edges are inserted into the graph. To realize that, there are four functions that help with the insertion of the appropriate nodes and edges:

- `getContainmentNode`. Returns the containment node belonging to the specified function `f`. If `f` is a C++-method, then the returned node will represent a class, otherwise it will represent a file. If the requested node does not yet exist in the graph, `getContainmentNode` will insert it into the graph.
- `getClassNode`. Returns the class node belonging to the specified function. If the requested class node does not yet exist in the graph, `getClassNode` will insert it into the graph.

- `getFileNode`. Returns the file node belonging to the specified filename. If the requested file node does not yet exist in the graph, `getFileNode` will insert it into the graph.
- `getDirectoryNode`. Returns the directory node belonging to the specified path. If the requested directory node does not yet exist in the graph, `getDirectoryNode` will insert it into the graph.

```
void insertNode(Function & f, Graph & G)
// f: The function for which to insert a node into the graph.
// G: The graph into which the node must be inserted.
{
    // Get the containment node of function f.
    Node n = getContainmentNode(f, G);

    // Insert a containment edge from the containment node
    // to the function f into the graph.
    insertEdge(n, f, G);

    // Finally, insert the function into the graph.
    G.insertNode(f);
}

Node getContainmentNode(Function & f, Graph & G)
{
    Node n;
    if(f.isMethod)
    {
        // Get the class node belonging to method f.
        n = getClassNode(f, G);
    }
    else // f is a C-style function
    {
        // Get the file node belonging to C-style function f.
        n = getFileNode(f, G);
    }

    // Return the containment node of function f.
    return n;
}

Node getClassNode(Function & f, Graph & G)
{
    // Try to get the requested class node from the graph. If
    // the class node does not yet exist in the graph, then
    // create it and insert it into the graph.
    string className = f.className;
    Node cn = G.getClassNode(className);
    if(cn == NULL)
    {
        // Insert a new class node into the graph.
        cn = new ClassNode(className);
        G.insertNode(cn);

        // Get the file node corresponding to the file in which
```

```
        // the current class is defined.
        Node fn = getFileNode(f.classFileName, G);

        // Insert an edge from the file node to the class node
        // into the graph.
        insertEdge(fn, cn, G);
    }

    // Return the requested class node.
    return cn;
}

Node getFileNode(string fileName, Graph & G)
{
    // Try to get the requested file node from the graph. If
    // the file node does not yet exist in the graph, then
    // create it and insert it into the graph.
    Node fn = G.getFileNode(fileName);
    if(fn == NULL)
    {
        // Insert a new file node into the graph.
        fn = new FileNode(fileName);
        G.insertNode(fn);

        // Get the directory node corresponding to the directory
        // in which the current file is stored.
        string path = getPathFromFileName(fileName);
        Node dn = getDirectoryNode(path, G);

        // Insert an edge from the directory node to the file
        // node into the graph.
        insertEdge(dn, fn, G);
    }

    // Return the requested file node.
    return fn;
}

Node getDirectoryNode(string path, Graph & G)
{
    // Try to get the requested directory node from the graph.
    // If the directory node does not yet exist in the graph,
    // then create it and insert it into the graph.
    Node dn = G.getDirectoryNode(path);
    if(dn == NULL)
    {
        // Insert a new directory node into the graph.
        dn = new DirectoryNode(path);
        G.insertNode(dn);

        // If we are not dealing with the root directory of the
        // file system, then make sure this directory is properly
        // contained by its parent directory.
        if(path != "/")
```

```
        {
            // Get the parent directory of the current directory.
            string parentPath = getPathFromFileName(fileName);
            Node pn = getDirectoryNode(parentPath, G);

            // Insert an edge from the parent directory node
            // to the directory node into the graph.
            insertEdge(pn, dn, G);
        }
    }

    // Return the requested directory node.
    return dn;
}
```

Complexity of the insertion algorithm

We can be quick about the complexity of `insertNode`: There are no loops in the algorithm, only a single recursive call in `getDirectoryNode`. The depth of the recursion depends on the depth d in the file system tree of the directory that holds the file in which the supplied function `f` is located. Furthermore, the algorithm performs lookup and insertion operations on a hash table, both of which can be done in $O(1)$ expected time. The worst-case running-time of a lookup in a hash tables with N_F elements is $O(N_F)$, however. Since a lookup occurs within the recursively called function `getDirectoryNode`, the worst-case running-time complexity of `insertNode` is $O(dN_F)$, whereas the expected running-time complexity is $O(d)$. When we consider d to be a constant, the complexities are reduces to $O(N_F)$ and $O(1)$ respectively.

It is obvious that `insertNode` will be called once for every function in the graph. That means, that all calls to `insertNode` collectively contribute $O(N_F^2)$ to the complexity of the call graph construction algorithms in the worst case and $O(N_F)$ in the expected case. As before, N_F is the number of functions in the graph. When we compare this to the complexity of the call graph construction algorithm we calculated before ($O(N_F N_C)$), we see that in the expected case the insertion of containment nodes does not increase the overall running-time complexity.

4.4 SERIALIZATION

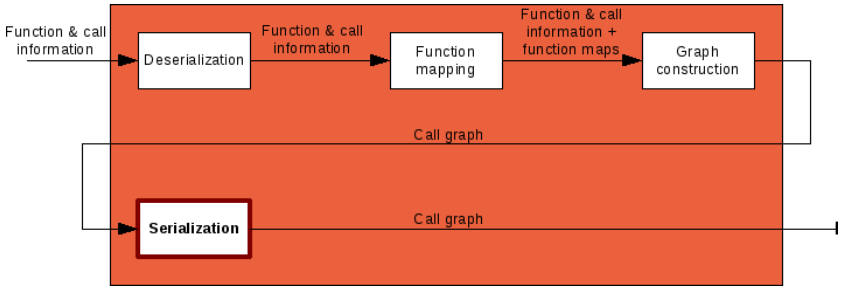

Figure 24: The serialization step.

Once the call graph has been constructed it must be serialized, so it can be visually explored using another program. The Call Graph Constructor supports three different output formats:

1. *Graphviz DOT format*. This is the format used by the Graphviz graph layout tools. Please refer to [31] for the specification of the format.

2. *Tulip format*. This is the format used by the Tulip graph visualization tool. Please refer to [23] for the specification of the format.

3. *SolidSX sqlite format*. This format is used by the DepView dependency viewer tool. Please refer to [33] for a discussion of this tool.

Even though these three different formats require three different serialization backends, their global approach is the same. All formats sequentially write all nodes and their annotations and all edges and their annotations to a file. So, in general, the graph serialization algorithm can be described by the following pseudo-code:

```
void serializeCallGraph(Graph & G)
{
    // Serialize the graph's nodes.
1:  for(int i = 0; i < length(G.nodes); i++)
2:      serializeNode(G.nodes[i]);

    // Serialize the graph's edges.
3:  for(int i = 0; i < length(G.edges); i++)
4:      serializeEdge(G.edges[i]);
}
```

The complexity of this algorithm can be determined rather easily. If N_N is the number of nodes in the graph and N_E is the number of edges in the graph, then line 1 loops N_N times and line 3 loops N_E times. This means that lines 2 and 4 are executed N_N and N_E times respectively. We assume that serializing a node or an edge can be done in constant time. So, we conclude that the worst-case running-time of `serializeCallGraph` is $O(N_N + N_E)$.

The above gives us the complexity of `serializeCallGraph` in terms of N_N and N_E. It would, however, be much more useful to have the complexity expressed in terms of the number of functions N_F and the number of function calls N_C, since this is how we expressed all complexities. Well now, nodes in the graph can be either function nodes or containment nodes. So, if $N_{N,Cont}$ is the number of containment nodes in the graph, we can say that:

$$N_N = N_F + N_{N,Cont}$$

Furthermore, in the very worst case, every single function call causes an edge to every function in the graph (this is the case for instance, when every function in the program has the same EFT and a call via a pointer-to-function is made to some function in the program). So, we can say that $N_C N_F$ is an upper bound for the number of call edges in the graph. Then, if $N_{E,Cont}$ is the number of containment edges in the graph, we obtain the following expression for the number of edges N_E in the graph:

$$N_E \leqslant N_C N_F + N_{E,Cont}$$

If we plug the above two equations back into the complexity equation, we obtain as the worst-case running-time complexity of `serializeCallGraph`:

$$\begin{aligned} O(N_N + N_E) &= O(N_F + N_{N,Cont} + N_C N_F + N_{E,Cont}) \\ &= O(N_C N_F + N_{N,Cont} + N_{E,Cont}) \end{aligned} \tag{4.5}$$

Now, the containment part of the graph (directories, files and classes) is actually a tree, so that means that if there are $N_{N,Cont}$ containment nodes, there must be exactly $N_{N,Cont} - 1$ containment edges connecting the containment nodes. Next to that, every function is also connected to exactly one containment node by exactly one containment edge. So, the total number of containment edges $N_{E,Cont}$ comes to:

$$N_{E,Cont} = N_{N,Cont} - 1 + N_F \tag{4.6}$$

Finally, when we merge equation 4.6 with equation 4.5, we end up with the final complexity of serialization:

$$\begin{aligned} O(N_N + N_E) &= O(N_C N_F + N_{N,Cont} + N_{N,Cont} - 1 + N_F) \\ &= O(N_C N_F + N_{N,Cont}) \end{aligned}$$

4.5 COMPLEXITY

At this point the only thing left to do is determine the total complexity of constructing a call graph. As before, the different phases of the construction process are performed consecutively and the complexities of the individual phases have already been determined. We merely need to add these complexities together to obtain the total complexity of constructing a call graph.

As a reminder, the complexities of the different individual phases are:

Phase	**Worst-case complexity**	**Expected complexity**
Deserialization	$O(N_C + N_F)$	$O(N_C + N_F)$
Function mapping	$O(\frac{1}{2}N_F^2)$	$O(\frac{1}{2}N_F^2)$
Graph construction	$O(N_C N_F + N_F^2)$	$O(N_C N_F)$
Serialization	$O(N_C N_F + N_{N,Cont})$	$O(N_C N_F + N_{N,Cont})$

Table 4: The worst-case and expected-time complexities of the different phases of the graph construction process. Here, N_F is the number of functions, N_C is the number of function calls and $N_{N,Cont}$ is the number of containment nodes in the graph.

When we add these individual terms we again end up with two equations: one total worst-case complexity and one total expected-time complexity.

4.5.1 *Total worst-case complexity*

Adding all the worst-case complexities in Table 4 gives the following equation for the total worst-case running-time complexity $C_{E,Worst}$ of the entire graph construction process:

$$\begin{aligned} C_{E,Worst} &= O(N_C + N_F) \\ &+ O(\frac{1}{2}N_F^2) \\ &+ O(N_C N_F + N_F^2) \\ &+ O(N_C N_F + N_{N,Cont}) \end{aligned}$$

Some quick simplification brings us to the final equation for the worst-case running-time complexity of the construction process:

$$C_{E,Worst} = O(N_C N_F + N_F^2 + N_{N,Cont})$$

4.5.2 *Total expected-time complexity*

When we add all the expected-time complexities in Table 4, we end up with the following equation for the total expected-time complexity $C_{E,Exp}$:

$$\begin{aligned} C_{E,Exp} &= O(N_C + N_F) \\ &+ O(\frac{1}{2}N_F^2) \\ &+ O(N_C N_F) \\ &+ O(N_C N_F + N_{N,Cont}) \end{aligned}$$

A little simplification yields:

$$C_{E,Exp} = O(N_C N_F + N_F^2 + N_{N,Cont})$$

Remember that the N_F^2 comes from the case that all functions are virtual functions that override, on average, $\frac{1}{2}N$ other functions. Whenever virtuals do not override such an extreme amount of functions and every function overrides, on average, r functions, then the complexity would be:

$$C_{E,Exp} = O(N_C N_F + rN_F + N_{N,Cont})$$

Whenever r does not take on an extreme value (i.e., a value close to $\frac{1}{2}N_F$), then the complexity of constructing a call graph primarly depends on the number of functions times the number of function calls in the program.

5 AUTOMATING EXTRACTION AND CONSTRUCTION

5.1 RETRIEVING PREPROCESSOR PARAMETERS AND BUILD TARGETS

Up until now, we discussed how call information can be extracted from source code and how this information can be used to construct a call graph. However, before a developer can easily use the system on his own source tree, there are a couple of problems left to solve. What these problems are will become clear when we take a closer look at what we have at this point and what exactly our goals are. At the moment, the system is comprised of two executable programs:

- *The C/C++ Call Info Extractor (CCIE)*: The CCIE program takes a single preprocessed source code file as input, and outputs a single call information file.
- *The C/C++ Call graph Constructor (CCC)*: The CCC program takes a set of call information files as input, and outputs a single graph file.

Now, when we go back to the graph construction requirements in section 2.1, in particular, to the user friendliness requirement (8) it becomes clear what our goals should be at this point: Be able to use the instruments created so far (i.e., the C/C++ Call Info Extractor and the C/C++ Call graph Constructor), in an easy manner, to construct call graphs of actual targets in a real-life software system that has a complex build configuration. Specifically, using our instruments should be no more difficult than performing a regular build of the code base.

From the program descriptions above and the stated goals we can now distill two unsolved problems:

1. The CCIE program expects *preprocessed* source code as its input. For small systems, manually preprocessing source code files using the compiler's preprocessor might not be such a big issue. Oftentimes, however, parameters are passed to the preprocessor when the system is build, to ensure that the source code is preprocessed in a way appropriate for the target being built. So, the first problem is that *we do not know the correct parameters to pass to the preprocessor*.
2. The CCC program takes *multiple* call information files as input and uses them to produce a graph file. The question that rises is what files should be given to CCC as input. Again, for simple systems this is a trivial question: Just feed all call information files as input to CCC. However, picture a code base that produces two targets: A standalone command line program and a static library. The library exposes the functionality of the standalone program for other developers to use in their own programs. In this case, it is not so obvious anymore what call information files should be fed to CCC: What kind of graph will we obtain when we feed both the standalone and library call information files to CCC? We would then attempt to generate a graph using the information extracted from the library twice, resulting in linking errors. So, the second problem is that *we do not know what call information files need to be fed to CCC when constructing the graph corresponding to a build target*.

5.1.1 *Requirements to the solution*

Before we continue to present the solution to the problems above, we first formalize the problems by translating them into a series of requirements.

The solution to problems 1 and 2 must satisfy the following requirements:

1. To solve the first problem, we need some way to preprocess the source code files with the correct parameters. The 'correct' parameters are, obviously, the parameters that the build system would pass to the preprocessor during a normal build. So, the solution must make sure that each call information file is constructed from a source file that is preprocessed in the same way that the build system would preprocess that source file.

2. The solution to the second problem must supply CCC with the correct call information files when constructing a call graph corresponding to a build target. Let us first determine what the 'correct' call information files are.

   A build target may produce different types of files as output. For us, only three types of output files are interesting: Static libraries, dynamic libraries and executables. It must be possible to construct a separate call graph for each of such output files: It would not make sense to generate one grand call graph of all produced binaries, since there is no guarantee that the binaries are related to each other.

   The question that now rises is what call information files we need to supply to CCC when constructing the call graph for a particular target. Let's consider this for each of the three types of targets that are relevant to us.

   - A static library is nothing more than a collection of object files, packed into an archive. So, the call graph of a static library should consist of all functions and function calls present in the object files of the library. Therefore, all call information files corresponding to those object files (that is, the call information files corresponding to the translation units from which the object files have been build) should be input to CCC.
   - The call graph of a dynamic library should consist of all functions and function calls present in the object files that are linked into the dynamic library. If the dynamic library uses functions from *other dynamic libraries*, then all functions and function calls present in *their* call graphs should be present in the call graph as well. Input to CCC should thus be the call information files corresponding to those object files and dynamic libraries.
   - The call graph of an executable should consist of all functions and function calls present in the object files that are linked into it. If the executable uses functions from dynamic libraries, then all functions and function calls present in their call graph should be present in the call graph as well. Input to CCC should be the call information files corresponding to those object files and dynamic libraries.

5.2 A SOLUTION USING COMPILER WRAPPING

In this section a solution to the problems stated above is presented to the reader. The implementation of this solution is, however, specific to the GNU Compiler Collection (GCC) [8] and the GNU Binutils [7] compiler utilities. Nonetheless, it is expected that it is possible, with reasonable effort, to implement a similar solution for other compiler utilities.

The key to solving the problems stated in the previous section is this observation: When doing a normal build of the system, the code base's build system is giving the GNU compiler tools the following information by passing parameters:

1. What source files are to be preprocessed and compiled, and with which parameters (using GNU gcc, g++, c++),
2. What binary object files are to be archived into a static library (using GNU ar),
3. What binary object files, static libraries and dynamic libraries are to be linked into an executable or dynamic library (using GNU ld).

This is exactly the information we need to solve our two problems. If we can somehow intercept the calls to the relevant GNU compiler tools, then the build system will give us exactly the information we need to properly preprocess the source files (requirement 1) and to associate the correct call information files with the appropriate target (requirement 2). Intercepting calls to the GNU compiler tools can be done using a technique called compiler wrapping, which is descibed in [39]. The remainder of this section will first briefly explain the concept of compiler wrapping and will then illustrate a solution to the problems using this technique.

5.2.1 *Compiler wrapping explained*

Basically, this technique works by using an executable file that impersonates the original compiler. Then, whenever the compiler is called during a build of the code base, the impersonating program will be called instead of the original compiler. This way, it is possible to intercept all the arguments that the build system passes to the original compiler, simply by looking at the arguments that the impersonating program receives from the build system.

In essence, the following steps are required to setup compiler wrapping, for say, the GNU g++ compiler:

1. *Create an executable file called 'g++'*. The impersonating program must have the same name as the program that is being impersonated. This way, the impersonating program can be called by the build system without having to make changes to the build system.
2. *Make sure the program intercepts the parameters passed to it*. The parameters passed to the program are the parameters that the build system is trying to pass to the original compiler. We will want to inspect these parameters and use them in an appropriate way.
3. *Make sure the program calls the original compiler*. While this step may not be intuitively obvious, it is important that the original compiler is called, because the build system might depend in some way on the output produced by the compiler. If after the build system makes the call to the compiler (which we intercepted) the expected output is not present, the build system may terminate (or do something else we do not want).
4. *Make sure the program exits with the exit code returned by the original compiler*. The reason why the original compiler's exit code must be returned to the build system is similar to the reason why the original compiler must be called: The build system may depend in some way on the exit code returned by the compiler. If the actual exit code is not returned, the build system

GNU gcc/g++/c++	:	Translate C/C++ source code files to binary object files.
gcc/g++/c++ wrappers	:	Translate C/C++ source code files to call information files.
GNU ar	:	Archives a set of binary object files into a static library archive file.
ar wrapper	:	Archives a set of call information files into a call information archive file.
GNU ld	:	Takes a set of binary object files, static libraries and dynamic libraries and produces a final binary executable or a dynamic library.
ld wrapper	:	Takes a set of call information files and call information archive files and produces a call information archive file.

Table 5: The relevant GNU compiler tools and their corresponding wrapper scripts.

might react in an unwanted way (it may, for instance, keep on building after the compiler produced an error, making it much harder to track the error).

5. *Modify the PATH environment variable*. The last thing to do to make sure that the impersonating program is called instead of the original compiler, is modify the PATH environment variable. The path in which the impersonating program resides should be present in the PATH variable *before* the path to the original compiler. That way, when 'g++' is invoked, the operating system will find and invoke the impersonating 'g++' instead of the original one.

5.2.2 *Wrapping the GNU compiler tools*

Earlier we stated that we need to create a wrapper script for each of the relevant GNU compiler tools. The table below lists each of the tools with their corresponding wrapper script. For both the tools and the wrapper scripts a description of their input and output is given. Note that 'input' here does not mean the parameters passed to the respective program: Each pair of tool and wrapper program receives the exact same parameters when invoked. What files they then consume as input is determined by the respective program, depending on the parameters received.

Note that the GNU compiler tools and the wrapper scripts have a high conceptual similarity with respect to what they take as input and what they produce as output.

Wrapping gcc, g++, c++

The main purpose of the scripts wrapping GNU gcc, g++ and c++ is to make sure that CCIE receives properly preprocessed source code files as input. To ensure this is the case, each of these scripts will perform the following steps:

- Each wrapper script will first invoke the original compiler with the parameters passed to the wrapper script. This will make sure that the original compiler gets called in exactly the same way that the build system intended. These is only one small difference: The `-save-temps` switch is added to the list of parameters. By adding this argument the compiler will not delete

the preprocessed source file, which is exactly what we need. If the compiler exits with an error code then the wrapper script will terminate, returning the compiler's exit code to the build system.

- If the compiler returns success, then the preprocessed source file will be fed to CCIE, which will generate the appropriate call information file. If for some reason CCIE fails, the script will still return success to the build system to make sure that the behaviour of our script exactly mimics that of the original compiler.

It is easy to see that requirement 1 is satisfied by the above solution, since we feed source files to CCIE that were preprocessed by the original build system. To illustrate, the complete source code of the g++ wrapper script is printed in appendix A.1.

Wrapping ar and ld

At this point, we have a mechanism to automatically create call information files from source files. To satisfy requirement 2, we need to associate those call information files with the correct targets. To that end, we will create wrapper scripts for the GNU ar and ld tools.

The wrapper script for ar performs the following steps:

- First, the original GNU ar tool is invoked with the arguments supplied by the build system. If ar returns an error, then the script will terminate, returning the exit code to the build system.

- If ar returns success, the name of the generated static library archive, and the names of all object files that were put into that archive, are extracted from the supplied parameters. Next, the script puts all of the call information files corresponding to the archived object files into a new call information archive. The name of that call information archive is derived from the name of the static library archive. This call information archive now contains all the call information files needed to construct a call graph of the static library that was just created. Again, the script will return success regardless of whether the call information archive was successfully created.

The wrapper script for ld is similar to the ar wrapper script, though it is slightly more complex. It performs these steps:

- Like before, the first thing the script does is call the original GNU ld program with the supplied arguments. However, the `-trace` parameter is added to force the linker to print out the names of all the files that are currently being linked. These file names are intercepted and used in the next step. If ld returns an error, then the script terminates and returns the error code to the build system.

- If ld returns success, then the name of the dynamic library or executable that was just created is retrieved from the parameters supplied by the build system. After that, all of the call information files corresponding to the object files that were printed out by ld are archived into a new call information archive A. The name of the new archived is derived from the name of the dynamic library or executable that was just created. Now, ld can be used to link three different types of files and each of these types of files has a different set of call information files corresponding to it:

1. If a dynamic library is being linked, then we need to include all the call information files in the call information archive corresponding to that dynamic library into archive A.
2. If a file f from a static library is being linked, then we need to include the call information file corresponding to file f in the call information archive corresponding to the static library, into archive A.
3. If an object file is being linked, then we need to include the call information file corresponding to that object file into archive A.

As before, the script will return success regardless of whether the call information archive was successfully created.

As was stated in table 5, the output of the wrappers of ar and ld is a call information archive file. Such an archive contains all of the call information files that correspond to the static library, dynamic library or executable that was created by ar and ld. In other words, by using compiler wrapping, we now know for each static library, dynamic library or executable produced by the build system what call information files correspond to it. That means that requirement 2 has been satisfied, by which problem 2 has been solved.

At this point there is only one small step left to be automated: We still need a way to feed all of the files in a call information archive as input to the call graph constructor. For this, we use a simple script which extracts the complete archive and then calls CCC, supplying it with the names of the extracted files.

Part II

VISUAL EXPLORATION OF CALL GRAPHS

6 CALL GRAPH VISUALIZATION CANDIDATES

At this point we have a fully functional toolset that can be used to construct large call graphs of real-world systems. The next thing we want to accomplish is to visualize these graphs to support the actual analysis of the systems. In the first section (6.1) of this chapter, we discuss what requirements the visualization system must satisfy. Then, in section 6.2, we test three readily available systems against these requirements and we conclude on which one of these systems is the most suitable for our purpose. In the next chapter (7) our entire toolset, including the chosen visualization system, will then be applied to several large, real-world code bases.

6.1 GRAPH VISUALIZATION REQUIREMENTS

The requirements for the call graph visualization component of the software are as follows:

1. *Scalability*. The program should be able to visualize very large call graphs of potentially hundreds of thousands of elements. The different operations on such graphs, like reading, navigating and searching elements or attributes should work in near real-time, even for very large datasets.
2. *Overview*. The program should be able to produce views of very large graphs in such a way that they contribute to the understanding of the program. Specifically, the visualization program should be able to display a large number of calls with limited clutter. Also, it should be able to show both the containment and the call relations in the same view and it should be able to present the different types of attributes in an intuitive way to the user.
3. *Navigation*. The program should be able to visually navigate the graph to allow many different views of the call graph.
4. *User friendliness*. The program should be able to search the call graph for specific nodes and edges by searching their attributes. Furthermore, the program should be readily usable for developers with minimal configuration. Ideally, developers should be able to simply load a dataset and then use the full capabilities of the program without having to configure any settings.
5. *Ready to use*. Our goal is not to create new visualization system, so we strongly prefer an existing system which is ready to be used and requires no or minimal modifications.

6.2 VISUALIZATION CANDIDATES

There are obviously many visualization systems around for software containment and dependency relations. Our goal here is not to provide an exhaustive overview of all these system, but to present a short list of well known and/or promising candidates. The visualization candidates that will be discussed are:

- Graphviz [31],
- Tulip [23],

- SolidSX [14]

All of the above are existing graph visualization systems and each of them will be tested for suitability for our purposes in this section. It should be noted that a more in-depth, side-by-side comparison of Tulip and SolidSX is given in [33].

We will use these three systems to (attempt to) generate overviews of the four software programs listed below. To give an impression of their relative sizes, the number of lines of code (LOC) of each of program has been calculated using the CCCC code counting system [34]. For the same purpose, the number of nodes and edges (both containment and function/call nodes and edges) in their respective call graphs is also listed.

1. *Fibonacci*. A tiny, trivial program that computes the n-th number of the Fibonacci sequence. Fibonacci has 22 LOC, 15 nodes and 19 edges. See section A.2 in the appendix.

2. *Precalc*. A small calculator program that calculates the result of simple prefixed-operator arithmatic expressions. Precalc has 413 LOC, 148 nodes and 264 edges. See section A.3 in the appendix.

3. *Bison* [1]. An average/large sized open-source general-purpose LALR(1) and GLR parser generator. Bison has 13628 LOC, 3214 nodes and 14382 edges.

4. *Mozilla Firefox* [5]. A very large open-source web browser. Mozilla Firefox has 1479817 LOC, 102969 nodes and 809272 edges. We must note, however, that the LOC metric is biased, since it was calculated from the complete Mozilla code base, which contains the source code for a number of applications, one of which is Firefox. Since Firefox is the largest of those applications and much code is shared between the applications, the order of magnitude of the LOC metric is still relevant. Nonetheless, the reader should be aware that the actual value for the LOC metric is lower than the value presented here. The number of nodes and edges is accurate, though.

Then, after using the visualization candidates to generate overviews of these programs, we conclude on what system is the most suitable for our purposes.

6.2.1 *Graphviz*

Graphviz [31] is a well known graph visualization system that has fine tuned, stable algorithms and is used by hundreds of different information visualization applications. It is well-tested, readily available and maintained on many platforms.

On the downside, it does not provide any interaction features: it simply generates static images of graphs. The figures 25, 26 and 27 show that the Graphviz programs are really only suitable for investigation of the smallest of call graphs. Even the small Precalc program is somewhat hard to understand when we look at figures 26 and 27. The call graph of Bison depicted in figure 27 clearly demonstrates that Graphviz is not suitable for the visualization of call graphs of real-world programs.

6.2.2 *Tulip*

Tulip [23] is also a well known visualization framework, with fine tuned, stable algorithms. It is actively maintained, well documented and available for a range of platforms. Tulip comes with a variety of layout algorithms and allows the user to interactively explore the graphs, using search and navigation functions that are exposed through the user interface. Tulip is a mature and ready-to-use graph

Figure 25: The tiny Fibonacci program layed out with the dot algorithm.

Figure 26: The small precalc program layed out with the dot algorithm.

visualization framework and a big improvement over Graphviz with respect to our goals.

As expected the tiny Fibonacci program is clearly visible in figure 28. On the right hand side of this figure it can be seen that the hierarchy and call relations of the small Precalc program are much more easy to understand than they were in figures 26 and 27 using Graphviz.

However, on a real-world sized program like bison, Tulip's layout algorithms start failing, as is clearly visible in figure 29: Call edges clutter the layout of both the Improved Walker algorithm and the GEM algorithm. At this point we are no longer able to easily distinguish individual call edges and we are at best able to make some comments on what components are heavily interconnected. Although it is still possible to perceive hierarchy (most notably when using the Improved Walker layout algorithm), that too starts to suffer from the clutter of the call edges.

When we take a look at the call graph of the very large Mozilla Firefox program (figure 30), we see that our display is mostly clutter. It becomes nearly impossible to base any relevant analysis on the displayed results. Only the layout using the Improved Walker algorithm is shown here, because the GEM layout algorithm does not yield results for graphs of this size in reasonable time. This clearly demonstrates that Tulip, even though it performs much better than Graphviz, is also not a viable candidate for the visualization of large call graphs.

6.2.3 *SolidSX*

SolidSX [14] is a relatively new, not so well-known, graph visualization system. In contrast to the previous visualization tools, SolidSX offers only a single type of layout. It allows the visualization of a compound directed graph, consisting of a tree representing the containment hierarchy, and a graph representing the adjacency edges (the call edges in our case). It does so by placing the containment nodes on concentric rings and allowing the user to interactively expand and collapse those hierarchy nodes. The adjencency edges are visualized using the Hierarchical Edge Bundles (HEB) technique of Holten [32], which bundles groups of edges based on their containment hierarchy. This technique significantly reduces the clutter of adjacency edges and effectively emphasizes the containment hierarchy of the nodes of those edges. Like Tulip, it enables the developer to search for particular nodes and edges based on their attributes, using the user interface.

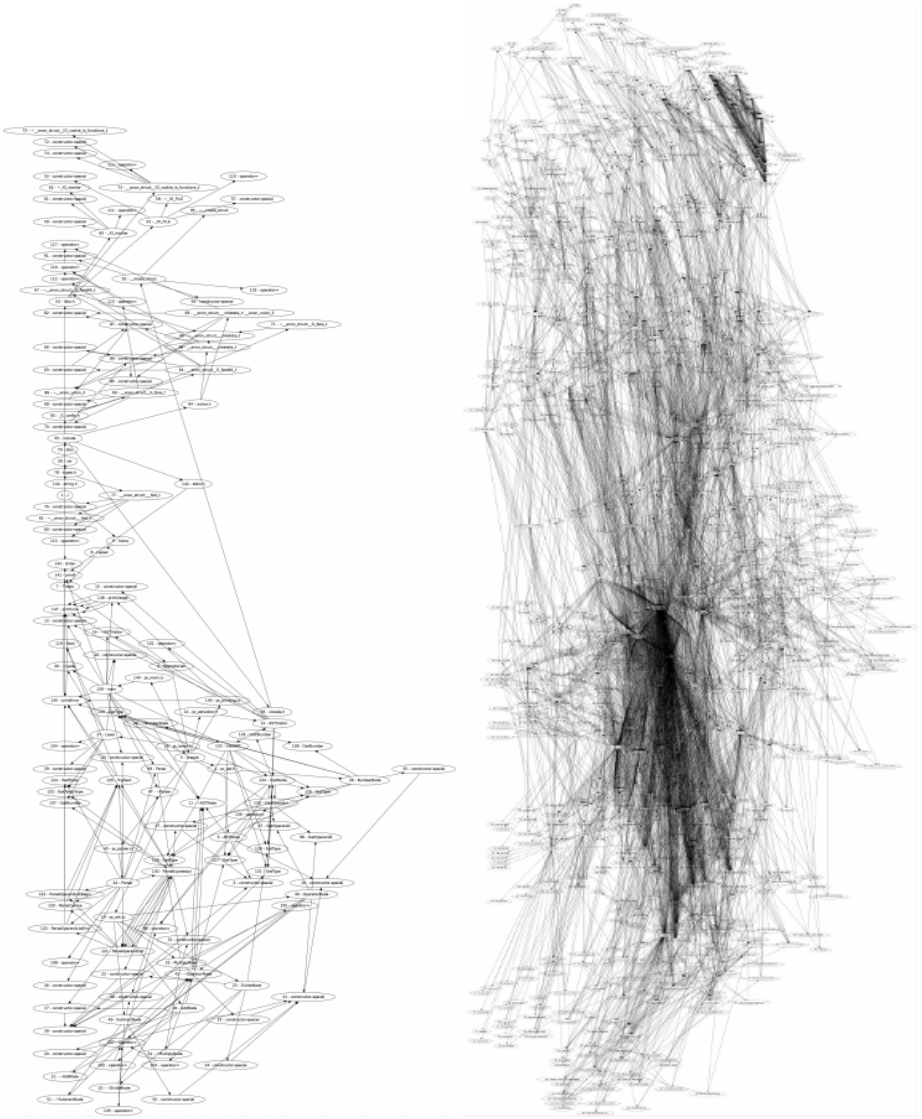

Figure 27: The small precalc program (left) and the average/large Bison program (right). Both call graphs are layed out with the fdp algorithm.

A notable difference between SolidSX and the previous candidates, is that SolidSX has been specifically designed to visualize the type of graph that we are using (compound digraphs), whereas Tulip and Graphviz have been designed to visualize a wide range of types of graphs.

When we look at figure 31, we see that small programs pose no problem for SolidSX. All nodes and adjacency edges are clearly visible. The right hand side of the figure nicely illustrates the effectiveness of the HEB approach: Edges are bundled based on containment and the bundles hierarchically fan out at the end points. The left hand side of figure 32 shows all the nodes and all call edges of the average/large sized bison program. Although it becomes rather difficult to follow individual edges, the overview still provides us with much more information than the overview of Tulip (figure 29) did. It is relatively easy to see what components are heavily used and what components are sparsely used. Furthermore, when we wish to investigate the incoming and outgoing call edges of a single node (i.e.,

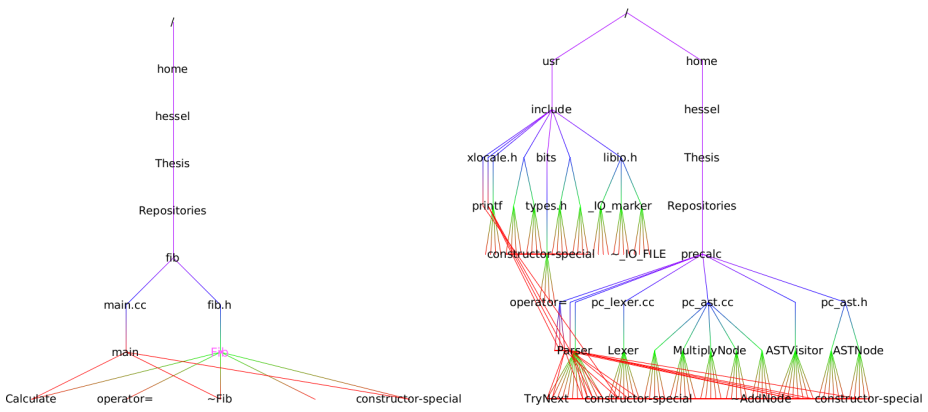

Figure 28: The tiny Fibonacci program (left) and the small Precalc program (right). Both call graphs are laid out with the Improved Walker algorithm

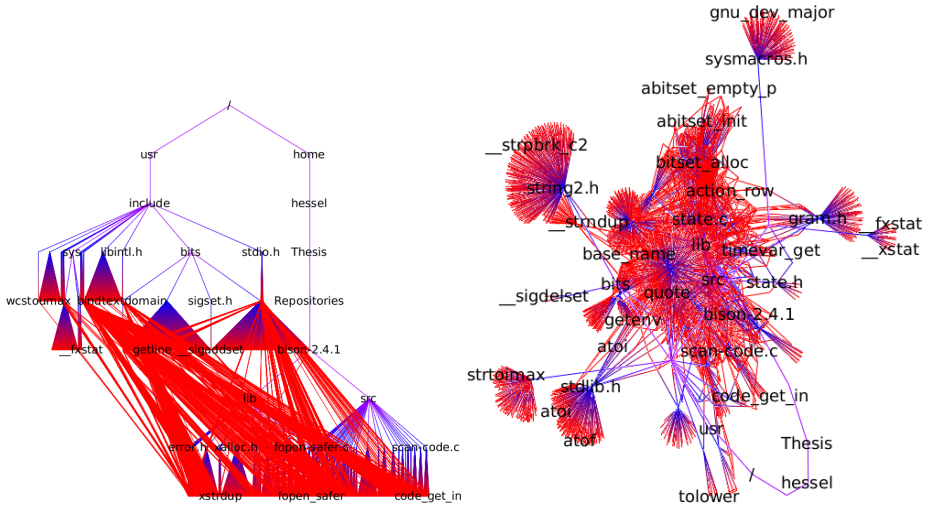

Figure 29: The average/large sized Bison program. On the left the call graph layed out with the Improved Walker algorithm. On the right the call graph layed out with the GEM algorithm.

a directory, class, file or function), all visual clutter can easily be eliminated by selecting the concerned node (right hand side of figure 32).

We are able to perform our analyses without restriction even on the largest of the code bases investigated here: That of Mozilla Firefox. The image on the left in figure 33 instantly shows us that all but a few components are heavily interconnected. When digging down the containment hierarchy a little further (right hand image of figure 33), we see that Firefox's `main` function actually makes very little function calls: Apparently, the real action happens somewhere in those called functions.

SolidSX allows us to interactively explore and visualize very large call graphs while minimizing clutter of call edges. We thus state that it is the best candidate for our purposes. This statement is confirmed in [41], which shows (by means of a user study) that SolidSX's approach is indeed perceived by developers as the most effective of the approaches investigated here.

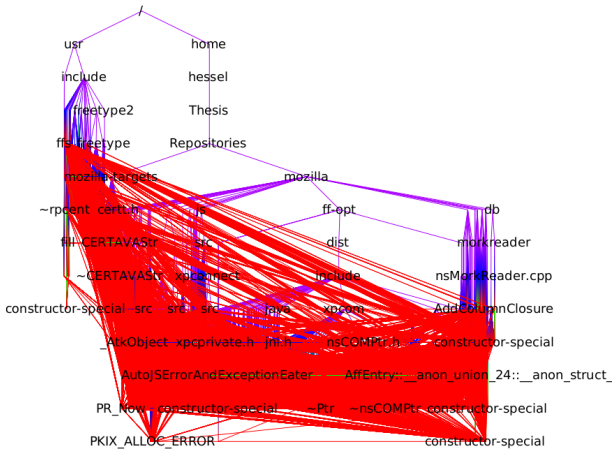

Figure 30: The very large Mozilla Firefox program. The call graph has been layed out with the Improved Walker algorithm.

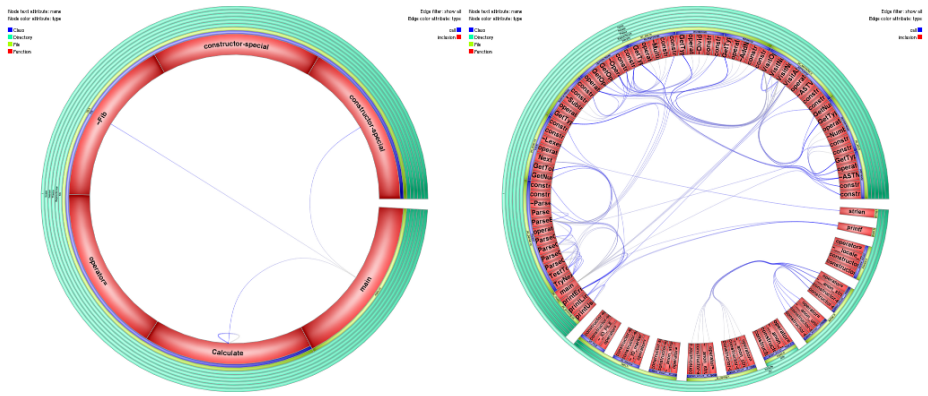

Figure 31: The tiny Fibonacci program (left) and the small Precalc program (right).

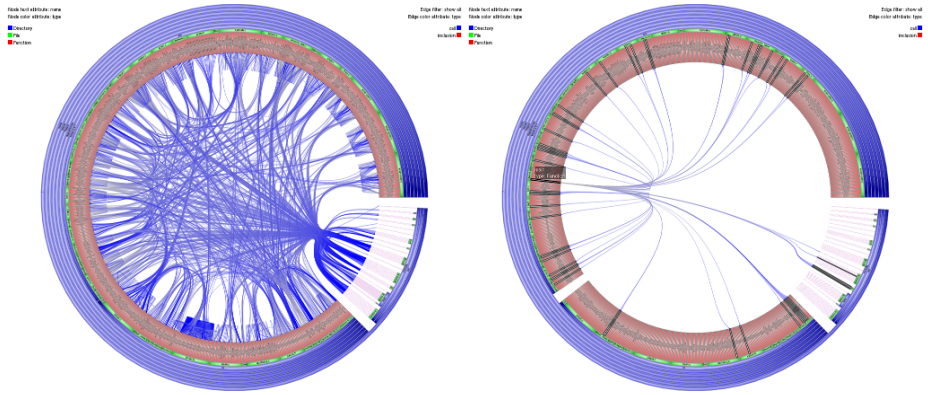

Figure 32: The average/large sized Bison program. On the left the call dependencies between all functions. On the right the calls made in the main function.

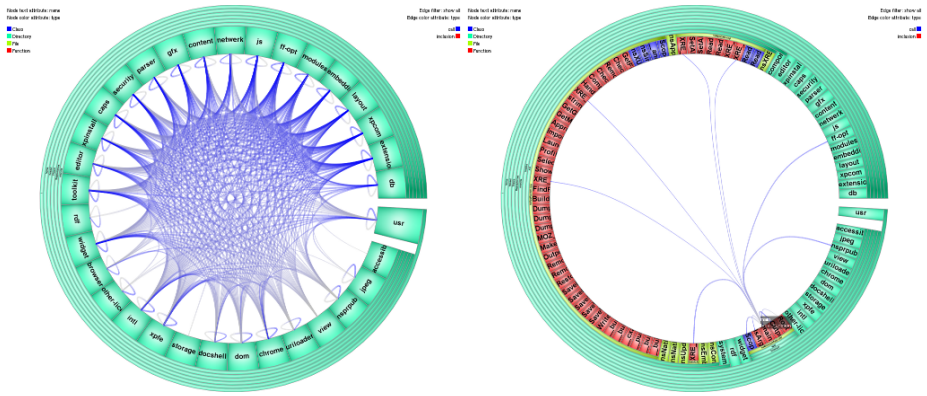

Figure 33: The very large Mozilla Firefox program. On the left, the call dependencies of all directories of the code base's root directory. On the right the call dependencies of the main function.

7 APPLICATION OF THE TOOL CHAIN

At this point we have a complete call graph construction and exploration tool chain, consisting of an information extractor (CCIE), a call graph constructor (CCC) and a call graph exploration tool (SolidSX). When we sequentially apply these tools, we have the ability to quickly and accurately investigate numereous interesting properties of a C/C++ code base.

In this chapter, we will demonstrate the application of the entire tool chain on a large, real-live C/C++ code base. We will apply the tool chain to Mozilla Firefox [5] and use it to attempt to find some interesting properties of Mozilla Firefox by exploring its call graph. We will not specifically define characteristics we are looking for up front; instead, we will freely explore the call graph and comment on what we find as we go.

7.1 RUNNING CCIE AND CCC

The first thing to do is to run the C/C++ Call Info Extractor on the source code of Mozilla Firefox. To this end, we set up our compiler wrapping environment and issue a `make` command on the code base. This will both compile Firefox and extract the information we need to construct a call graph.

On a mid-range machine, this process will take approximately three hours to complete, as opposed to the approximately one hour that is needed to just compile Firefox on the same machine. It is interesting to see that the extraction process thus takes roughly twice as long as a normal compilation run. This seems to confirm our statement in 3.1.1 that the *performance* of the Elsa parser is roughly equal to that of an efficient C++ compiler (GNU GCC in this case). Moreover, if we assume this is correct, then it also seems to confirm that our information extraction component does not add significant computational complexity.

When the extraction process is finished we have a collection of call information archive files (see table 5 in section 5.2.2), which are ready to be processed into call graph files. We are now able to generate a call graph for each of the binary targets that were build while compiling Firefox. These binary targets include, next to the Firefox executable, a collection of shared libraries representing separate subsystems of Firefox. In this case we are interested in the complete call graph of Firefox, but if, for some reason, we would like to investigate a single subsystem of Firefox in isolation, we would be able to do so by generating the call graph for the respective target.

Running the C/C++ Call graph Constructor is just as easy as running the C/C++ Call Info Extractor: Using the scripts we created we simply specify that we want to create a SolidSX compatible graph file from the call information archive file representing the `firefox-bin` executable. Creating the requested graph takes little over one minute and it results in a graph file of little under 200 megabytes in size.

7.2 VISUAL EXPLORATION USING SOLIDSX

Loading the generated call graph file in SolidSX takes approximately five minutes. After that, we are presented with a fully collapsed directory structure, as depicted in the left hand image of figure 34. When we expand the root directory two subdirectories appear: `home` and `usr` (center image of figure 34). This is expected,

since the Firefox code base resides in the `/home/hessel/Thesis/Repositories/` directory, whereas the system header files reside in `/usr/include`. At this point, we are not interested in the usage of system headers, so to minimize clutter we hide the `usr` subdirectory. Double-clicking a few more times expands the root of the Firefox code base which enables us to see the directories of the subsystems that reside there (right hand image of figure 34).

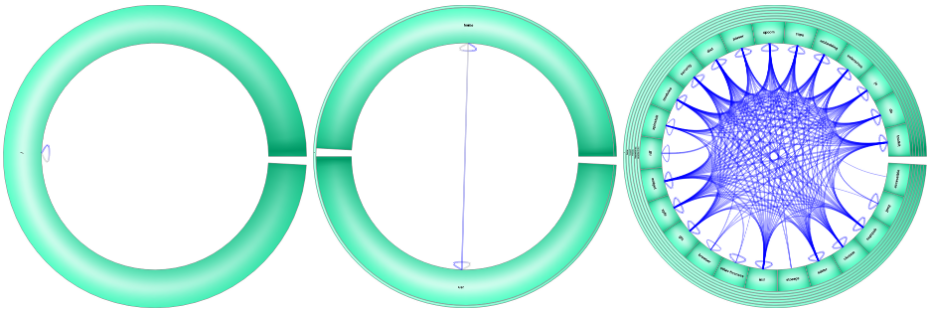

Figure 34: Left: The initial presentation of the call graph of Mozilla Firefox. Center: The first subdirectories of the root directory. Right: The contents of the root directory of the Firefox code base.

The right hand image of figure 34 shows us that there are quite a number of subsystems, so let us choose one to focus on. Let us focus on the subsystem that appears most connected to the other subsystems, i.e., the subsystem that shares an edge with the most other subsystems. Since the most connected subsystem is likely to be an important, much used component it would be interesting to find out more about what it is and how it is used.

To determine what subsystem is most connected, we brush the mouse cursor over the different subsystem directories. This will show the edges that the brushed subsystem shares with the other subsystems and will allow us to determine the most connected subsystem by simply counting the number of edges.

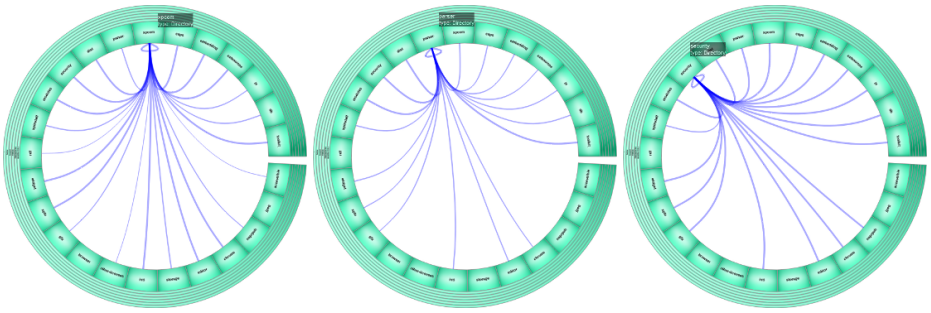

Figure 35: The connectedness of the `XPCOM` (left), `parser` (center) and `security` (right) subsystems.

It turns out that `XPCOM` is the most connected subsystem, sharing edges with 22 of the 24 other subsystems. Figure 35 shows the edges of the `XPCOM` subsystem and the less connected `parser` and `security` subsystems. So, we will focus on the `XPCOM` subsystem.

7.2.1 *Exploring the `XPCOM` subsystem*

A little research on the internet quickly reveals that `XPCOM` is a technology similar to Microsoft's COM [11]:

> "XPCOM is a cross platform component object model, similar to Microsoft COM. It has multiple language bindings, letting the XPCOM components be used and implemented in JavaScript, Java, and Python in addition to C++." [19]

From our own experience with Microsoft's COM and the above quotation we can conclude that the `XPCOM` subsystem allows the developers of Firefox to create and use well-defined, platform independent subsystems that can be written in, and used from, a multitude of programming languages and which are easily reusable between applications. Now, to be able to assess the degree in which Firefox leverages the potential advantages of `XPCOM` we would like to know more about how exactly `XPCOM` is used thoughout Firefox. Specifically, we would like to know:

1. How functionality is being exposed via `XPCOM`:
   a) How many subsystems are exposing functionality via `XPCOM`?
   b) How many `XPCOM` objects are exposed in total?
   c) What subsystems are exposing the majority of the functionality?
2. How functionality exposed via `XPCOM` is being used:
   a) How many subsystems are using functionality exposed via `XPCOM`?
   b) How many of the exposed objects are being used in total?
   c) What subsystems are using the highest number of exposed objects?

First, we will attempt to answer these questions using SolidSX. Then, we will use the answers to conclude on the impact that the use of `XPCOM` has on Firefox and on its development.

How functionality is being exposed via `XPCOM`

A little more research shows us ([21]) that for an object to expose functionality via `XPCOM` it must implement the `QueryInterface` method. This method is to provide information about what functionality (i.e., what interface) is implemented by that object. Other methods can then call this method to determine, at run-time, what functionality the object exposes.

Translating this observation to our problem reveals that finding an occurence of the `QueryInterface` method indicates that an object is being exposed via `XPCOM`. So, to answer the first three questions we use SolidSX to find and highlight all functions named `QueryInterface`. The result is depicted in figure 36.

SolidSX indicates that 264 nodes exist which are named `QueryInterface`. Now, since counting the number of selected nodes in each subsystem is somewhat difficult using figure 36, we simply expand the individual subsystems one at a time and count the selected nodes in each of them. After that, we have the answers to the first three questions:

How functionality is being exposed via `XPCOM`:

1. 17 of the 25 subsystems expose functionality via `XPCOM`,
2. In total 264 objects are exposed via `XPCOM`, and
3. Approximately 65% of the exposed functionality resides in the `security` (55), `toolkit` (35), `modules` (34), `widget` (20) and `extensions` (20) subsystems.

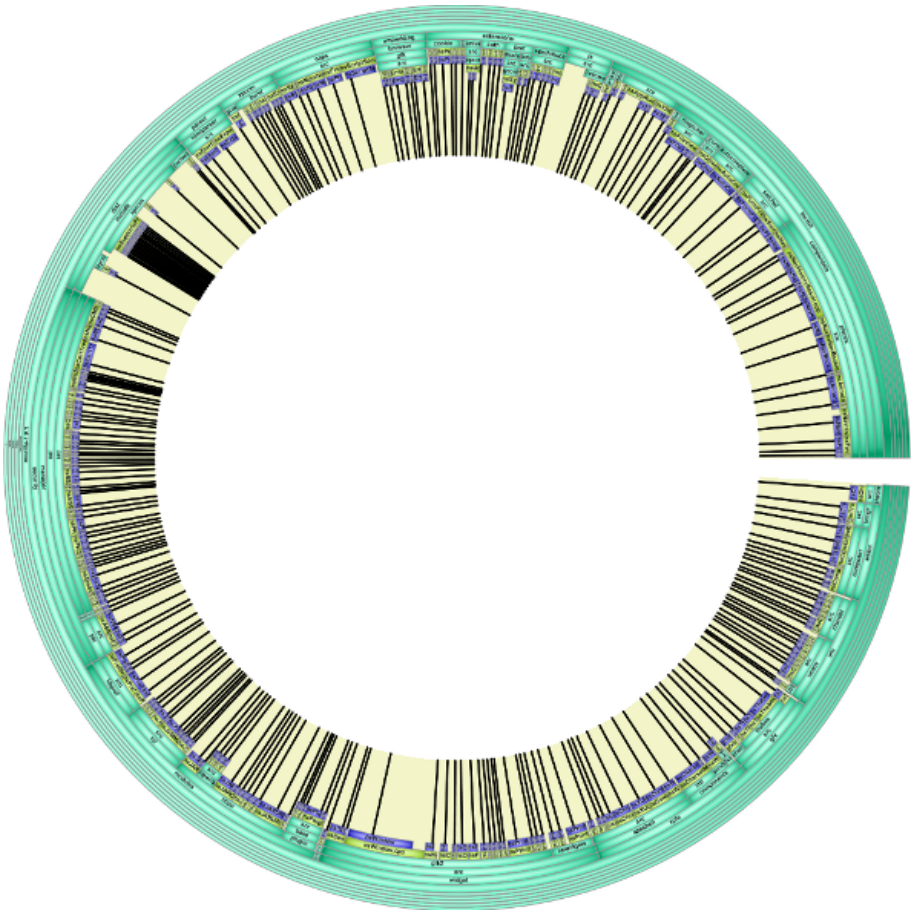

Figure 36: All implementations of `QueryInterface`.

How functionality exposed via `XPCOM` is being used

Doing once more a small amount of research on how to use `XPCOM` quickly introduces ([22]) us to the `nsCOMPtr` template class. The `nsCOMPtr` class is a variation of the Counted Pointer software pattern from [26] and provides developers with a convenient and safe way to handle pointers to `XPCOM` interfaces; it encapsulates the use of the `QueryInterface`, `AddRef` and `Release` methods and makes sure that all requested references are also released again. It is regarded as good practice to use this template class instead of directly accessing the `QueryInterface`, `AddRef` and `Release` `XPCOM` methods, since it eases the use of `XPCOM` objects and prevents memory leaks. Since this is regarded to be good practice, we will make the assumption that (at least the majority) of `XPCOM` object usage is done using this method.

So, using the above assumption it seems that we have found a way to translate our original problem of finding out how functionality exposed via `XPCOM` is being used, into a much more concrete problem: Finding all calls to the `nsCOMPtr` class.

The first thing to do is to find the nodes representing the instances of the `nsCOMPtr` template class. Unfortunately, at the time of writing SolidSX does not support partial matching of search queries. That is, only nodes that exactly match the search query will be highlighted. As a consequence, to find where the instances of the `nsCOMPtr` template class reside we need to know the exact name of one of the instance, or, alternatively, the name of the file containing them. Again, a quick search on the internet reveals ([20]) the, rather obvious, name of the file that contains the template class instances: `nsCOMPtr.h`. Searching for this exact file

name yields the results depicted in 37, which show that two files with the name `nsCOMPtr.h` were found.

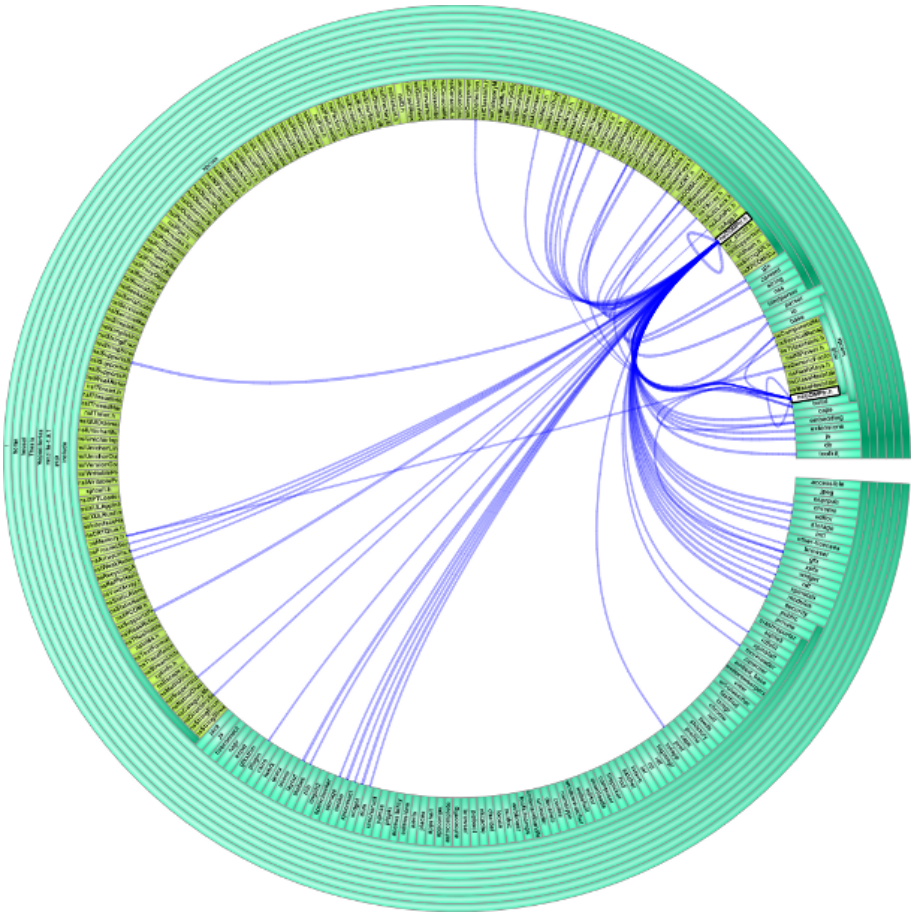

Figure 37: All edges coming from the subsystems and going to the various instances of the `nsCOMPtr` template class.

Some investigation of the code base reveals that one of these nodes represents a symbolic link to the file represented by the other node. So, even though we have two nodes called `nsCOMPtr.h` and they do represent two different file system entities (a file and a symbolic link), ultimately, they represent the same source code entity.

Regardless, it turns out that the template class instances of `nsCOMPtr` reside in only one of the two nodes that we found. When we show only the nodes representing the `nsCOMPtr` template class instances, together with the collapsed nodes of the remaining subsystems, we end up with the image on the left hand side of figure 38.

Now that we found where the different `nsCOMPtr` instances reside, we can use them to answer our last three questions.

When we brush an edge with our mouse cursor, SolidSX diplays summary information about that edge. Doing so reveals that both edges *coming* from the subsystems and edges *going* to the subsystems are being displayed. The edges coming from the subsystems are what we are looking for: The calls to the `nsCOMPtr` classes, made in the subsystems. The edges going to the subsystems, are, however, not what we are looking for: Since the `nsCOMPtr` classes are wrapper classes for the functionality implemented (and then exposed via `XPCOM`) in the subsystems, it is no surprise that the `nsCOMPtr` classes are making calls to the `QueryInterface`,

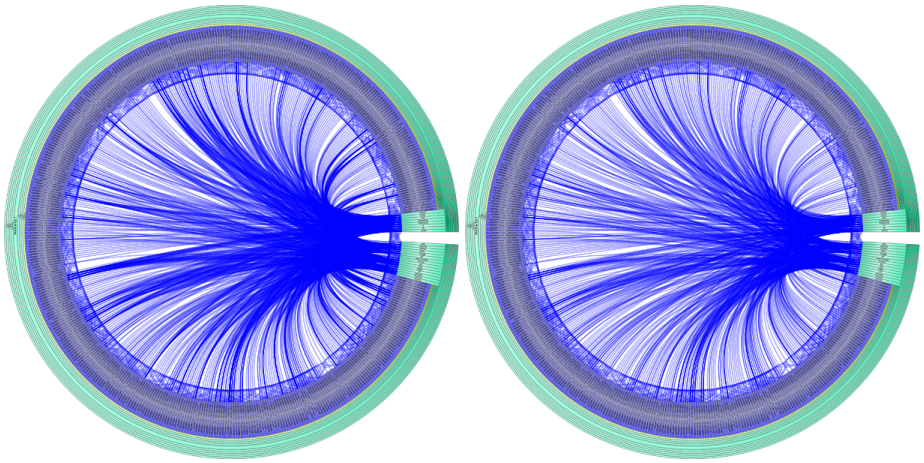

Figure 38: Left: All edges between the subsystems and the various instances of the `nsCOMPtr` template class. Right: Only the edges coming from the subsystems and going to the various instances of the `nsCOMPtr` template class.

`AddRef` and `Release` methods implemented by the subsystems. Although expected, these latter edges are not what we are looking for and we would like to hide them, since they clutter our display. Luckily, there is an easy way to achieve this: The `QueryInterface`, `AddRef` and `Release` methods are all *virtual* methods. This makes that calls (edges) to these methods are also marked as *virtual*, through their `call_is_virtual` attribute. So, when we instruct SolidSX to only display edges that have their `call_is_virtual` attribute set to false, all of the unwanted edges are effectively filtered out and we are left only with calls *to* the `nsCOMPtr` classes. The right hand side of figure 38 illustrates this.

At this point we have the ability to determine what subsystems are using objects exposed via `XPCOM` and what subsystems are not by simply brushing the subsystem nodes. The effect is depicted in figures 39 and 40. It turns out that 5 of the 25 subsystem are not using `XPCOM` objects through one of the `nsCOMPtr` classes.

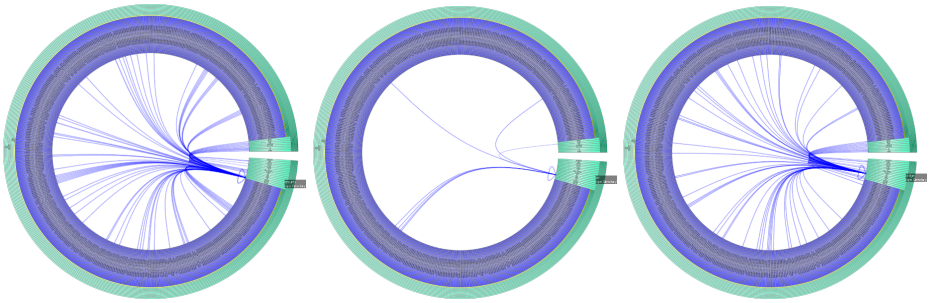

Figure 39: The usage of `XPCOM` objects by the `security` (left), `rdf` (center) and `widget` (right) subsystems.

Then, to find out how many of the exposed `XPCOM` objects are actually used by the other subsystems, we simply select the subsystem nodes in SolidSX. As a result, all functions that are called by the selected subsystems are highlighted as well, as can be seen from figure 41. This gives us a really nice overview of what `XPCOM` exposed objects are used and which ones are unused and it allows us to conclude that 36 of the 443 `XPCOM` exposed objects are unused.

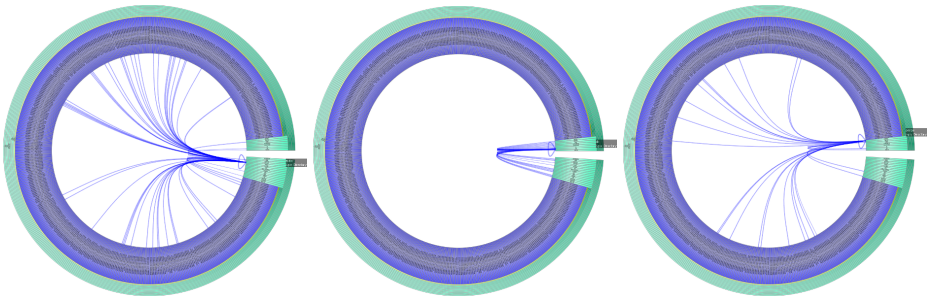

Figure 40: The usage of `XPCOM` objects by the `editor` (left), `db` (center) and `parser` (right) subsystems.

Finally, the question what subsystems are using the highest number of exposed objects is also easily answered using the current view: We select the subsystems one by one and then simply count the number of distinct `XPCOM` objects they call. This approach tells us that the `toolkit` subsystem relies most heavily on the exposed `XPCOM` objects by calling 149 different objects. A distant second is the `extensions` subsystem with 73 different objects, followed by the `modules` (71), `embedding` (67) and `security` (66) subsystems.

To summarize the answers that we have just found:

How functionality exposed via `XPCOM` is being used:

1. 20 of the 25 subsystems are using functionality that is exposed via `XPCOM`.
2. 407 of the 443 exposed objects are actually used.
3. The subsystems that are using the highest number of `XPCOM` exposed objects are: `toolkit` (149), `extensions` (73), `modules` (71), `embedding` (67) and `security` (66).

Some missing pieces

Before we move on to analyze the impact of `XPCOM` on Firefox, we first discuss something that caught our attention while we were attempting to answer the questions we posed in the previous section: We found a total of 264 *implementations* (i.e., implementation of the *QueryInterface* method) of `XPCOM` exposed objects, whereas we found a total of 443 *available* `XPCOM` objects. Obviously, we would like to know where this difference comes from and we find the answer in the quotation from [19] that we made earlier:

> "XPCOM is a cross platform component object model, similar to Microsoft COM. It has multiple language bindings, letting the XPCOM components be used and implemented in JavaScript, Java, and Python in addition to C++."

This suggests that the implementations that we are missing may still exist, but we were unable to detect them since they were not written in C/C++! A quick text search on the code base reveals that many `XPCOM` objects are indeed implemented in JavaScript, which explains the gap between the number of implementations and the number of available objects that we found. Furthermore, this potentially also explains why 36 of the 443 available objects appear unused: Although they are not used from any C/C++ source code, they may still be used from source code written in JavaScript.

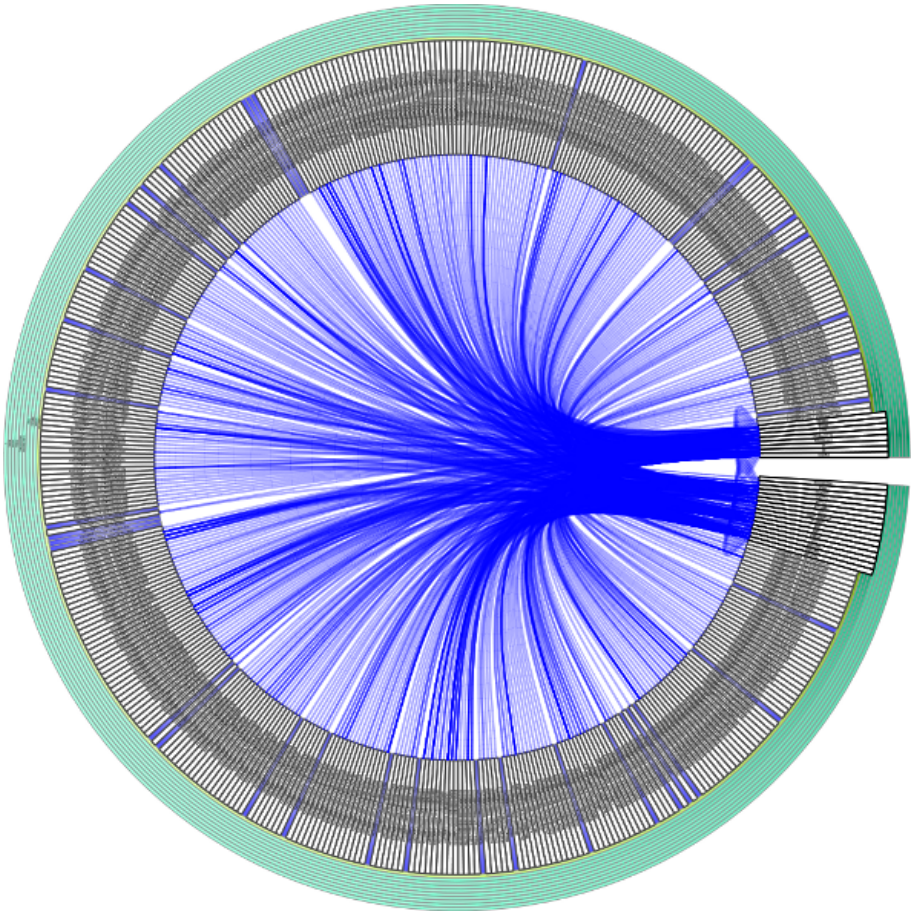

Figure 41: All nsCOMPtr instances that are used by the subsystems are selected.

The impact of `XPCOM` on Firefox

After the analyses that we did in the previous sections it is time for us to conclude on what insights our investigation has provided us with.

First, since many core components of Firefox are implemented as XPCOM objects, much of the functionality they provide can easily be reused to create new applications. This is exactly what has happened: Mozilla provides an integrated code base containing the source code for multiple applications, like:

- Firefox ([6]),
- Thunderbird ([16]),
- Sunbird ([15]),
- SeaMonkey ([13]).

Second, since many core components are implemented as XPCOM objects, porting issues are reduced to local (i.e., component level) problems. Hence, the dependencies between subsystems pose no difficulties during porting and staying portable. Since Firefox is available for a range of platforms, this is a really non-trivial property.

Next, supporting implementation and use of XPCOM objects from multiple languages allows the utilization of needed language features (performance, ease-of-

implementation, security) where appropriate. For instance, it allows performance critical subsystems to be implemented in a high-performance/native language, like C++ and non-performance critical subsystem to be implemented in a more high level language, like JavaScript, to support maintainability, robustness (no memory leaks), and security (no buffer overflows).

Lastly, since XPCOM has language bindings for JavaScript, it provides a mechanism for websites to use all of the functionality exposed via XPCOM. This makes Firefox a potentially very powerful application platform. Do note, however, that the use of XPCOM objects from untrusted webpages will obviously be restricted to some extend, for security reasons.

Part III

CONCLUSION

8 CONCLUSIONS AND FUTURE WORK

In this chapter we will briefly reflect on what has been done. We will talk about which of the requirements that we formulated (in 2.1 and 6.1) have been met and which requirements have not been met and what is future work that can be done to further improve the results.

8.1 EVALUATION

Initially, what we set out to do was to create a set of tools which would allow us to create and explore the call graph of a large C/C++ code base. We aimed for the tools to be scalable, efficient, robust, user-friendly and open-source. Additionally, we stated that the call graph construction tool should produce complete and correct call graphs, require only source code as input, be generic enough to handle a wide range of C/C++ dialects and preferably be open source (see 2.1). Similarly, the exploration tool should produce clear overviews of very large graphs and be capable of visual navigation to support many different views of the graph (see 6.1).

8.1.1 *What has been accomplished*

When we look back at what has been done throughout this thesis, we can say that we have presented a complete toolchain that constructs containment-and-call compound directed graphs from C/C++ code bases. When we look to see to what extend we have been able to satisfied the requirements, we can say the following.

The toolchain that we presented *scales well* to very large C/C++ code bases, consisting of hundreds of thousands up to millions of lines of code, as was required by graph construction requirement 1. At the same time, the toolchain performs *efficiently* when applied to a large, real-life code base such as Mozilla Firefox ([5]), taking roughly three times as long as an efficient C/C++ compiler (GCC, [8]) would take to process the entire code base (graph construction requirement 2).

The call graphs that it delivers are *complete* in that they contain all of the explicit and implicit function calls that can exist according to the C/C++ language specification ([29]), which was required by graph construction requirement 3. Also, the delivered call graphs are *correct* in that they are guaranteed to be conservative whenever possible and, whenever it is not possible to guarantee this, an effort is made to deliver call graphs that are conservative nonetheless (graph construction requirement 4).

As required by graph construction requirement 6, the toolchain is not dependent on the presence of executables, object files or debug information files in order to be able to operate correctly. Also, as preferred by graph construction requirement 9, the graph construction component of the toolchain is provided as open source software, to allow and encourage its future development.

The user friendliness requirement for the graph construction component is well satisfied, since it requires no more effort to construct a call graph from the code base using the toolchain than it does to build the code base using its

own build system (graph construction requirement 8). The same requirement for the graph visualization component is also well satisfied by the easy, intuitive and configurationless interface that SolidSX ([14]) provides (graph visualization requirement 4). Since SolidSX is a ready-to-use, actively developed and well maintained software system that allows developers to visually navigate and explore our call graphs, while providing them with a clear overview of the containment and adjacency relations at any level of the containment hierarchy, we conclude that graph visualization requirements 5, 3 and 2 are also satisfied.

8.1.2 *What has not or only partially been accomplished*

Though we can be pleased with the capabilities and properties of the presented toolchain, there are still some requirements that go, perhaps partially, unsatisfied.

First, even though we made a significant effort to make the resulting call graphs as accurate as possible, we could further improve on the correctness requirement (graph construction req. 4) in the context of calls via pointers-to-function and pointers-to-members and calls to virtual functions on object references and object pointers (see section 8.2 on how this can be done). Hence, we conclude that this requirement has been largely, but not completely, satisfied.

Second, we should note that the C/C++ Call Information Extractor indeed accepts, to some extend, incomplete or incorrect source code. Also, the system has been setup in such a way that a missing or incorrect translation unit merely causes a gap in the call graph, instead of terminating the entire process. However, the latter is only true if the code base's original build system also accepts this faulty code and does not terminate. If it does terminate on this faulty code, which is not uncommon, then our toolchain will terminate as well. This behaviour is due to the nature of the compiler wrapping approach and is, in our case, by design, since it is very likely that forcing the build system to continue the build is only going to propagate the error, and will probably make debugging the errors more difficult. As such, we must conclude that the robustness requirement (graph construction req. 5) has only been satisfied to a limited extend.

The last thing that we should note is on the range of supported C/C++ dialects. Even though we set out to support a wide range of C/C++ dialects, we have only implemented the compiler wrapping approach for the GCC compiler, which means that, at current, it is not possible to analyze code bases that cannot be build using GCC. For this reason we conclude that the genericity requirement (graph construction req. 7) has only partially been satisfied. We should note, on the other hand, that the design of the toolchain allows the range of supported dialects to be expanded relatively easily (again, see section 8.2 on how this can be done).

8.2 FUTURE WORK

Although a large portion of the initial requirements have been satisfied, the effectiveness of the toolchain can still be improved on a number of points. This section will first discuss what these points are and how the toolchain can be improved on these points. Then, we will briefly discuss what room for future improvement exists on points that we did not initially set out to accomplish during this thesis.

8.2.1 *Improvements to the current toolchain*

The improvements that can be made to the current toolchain, with respect to the requirements that we initially set out to satisfy, can be summed up as follows:

- The accuracy of the toolchain in the context of calls via pointers-to-function and pointers-to-member and calls to virtual functions on object references and object pointers can be increased by utilizing data-flow analysis (see [38] for a comprehensive discussion of data-flow analysis). This would allow us to significantly reduce the size of the set of call candidates in many common scenarios within this context.

- By introducing namespace nodes and their corresponding containment edges it would be possible to provide developers with insight into how code bases are structured in terms of namespaces. On would, however, need to find a solution to the problem that namespaces cannot be modelled together with files using a tree-like datastructure without making some sort of compromise as to who-contains-who: The C/C++ language specification allows a namespace to span multiple files, while at the same time a single file is allowed to contain multiple namespaces.

- Currently, the call graph only displays adjacency relations in the form of call edges. It is possible though, to extend the call graph with edges that represent inheritance and overrides relations between classes and functions, respectively. This would allow developers to, for instance, reason about the 'abstractness' of a code base.

- By implementing compiler wrapping support for other compiler platforms, such as the Intel C++ compiler [9], the range of C/C++ dialects to which the toolset can be applied can be extended without having to modify any components of the toolchain other than the compiler wrapping scripts.

- At current, when analyzing a library target in isolation, it is not possible to inspect the initializing and finalizing function calls (see 3.2.5) that the library would cause in the context of an executable application. However, it is not hard to understand that this would be a desirable feature when, for instance, developing a library. A possible solution to this would be to insert a Root node for library targets, similar to the Root node that is currently inserted for executable targets (see 4.3.2). The only difference would be that the Root node for library targets does not make a function call to an entry point, as is the case with the Root node for executable targets (which makes a call to the program's `main` function).

  Implementing this feature would also be rather straightforward, since one would one need to slightly modify the graph construction algorithm to include the Root node for library targets, along with the corresponding initializing and finalizing function calls. The implementation of this would not be much different from that of the Root node for executable targets, which already exists.

8.2.2 *Future improvements*

Here we will briefly discuss points of improvement for both the graph construction component as well as the visualization component of the toolchain that are beyond the scope of this thesis, but which could significantly improve the usefulness of the toolchain.

Add support for multiple programming languages

As we have seen during the analysis of the Mozilla Firefox code base in section 7, it is not uncommon for large code bases to be written in more than one programming language. Also, having some form of binding between the different languages, enabling their interoperation, is a very conceivable situation: One could think, for example, of an application written in C++ that exposes the core of its functionality via a scripting language like Python. If we would extend the toolchain to include call information extraction support for other programming languages, then we would be able to analyze such code bases in their entirety and see, for instance, via what interfaces the core functionality is exposed to the scripting language. Obviously, it goes without saying that we would then also be able to analyze code bases that are completely written in the newly supported languages.

However, considering the amount of effort it takes to implement support for a single language, as we did, we must note that adding support for new languages can be a difficult and time-consuming task. On the other hand, only part of the toolchain would require to be extended, while the call graph constructor (ccc) and SolidSX need no modification or extension in any way to support these new languages. Also, adding support for other programming languages is *the* way to increase the number of code bases to which the toolchain can be applied, indicating that the potential pay-off for such effort is, like the effort required to realize the support, large.

Extend SolidSX to allow visualization of the exploration path by branching

At current, SolidSX provides the developer with a single view of the graph, at any point in time. That is, if the developer wants to get a different view of the graph, he needs to navigate away from the current view to the view that he wishes to obtain. Although this is a very natural way of exploring, it would be of great value if the developer would be able to *branch* the current view, allowing the original view to remain intact, while the developer keeps on exploring using the branched view. By presenting the developer with several of such branches side-by-side, he is effectively able to construct and view his *exploration path*, instead of only the *currently explored site*.

An example of an application that would greatly benefit from this, is *iterative hypothesis validation*, whereby one would continuely pose, verify and reject new hypotheses by digging deeper into the presented data. In such a case, the developer could branch the current view when he wishes to verify a new hypothesis he formed based on the current view. Then, when the hypothesis is verified, he could continue to branch it further during a next iteration, or, when the hypothesis is rejected, he can simply throw away the branched view and continue his research using the original view.

8.3 FINAL WORDS

Even though relevant and useful work is being done, there is still a long way to go towards significantly easing the task of program understanding for the purpose of reengineering or maintenance tasks. The tools currently available, including the one presented here, though powerful and useful, are in most cases not yet able to *alleviate* a developer of his program understanding related tasks. Rather, they are only able to *support* a developer by presenting him with information that is relevant to the task at hand. Because of this, program understanding remains a

difficult and time-consuming activity that continues to require the knowledge and expertise of experienced developers.

Part IV

APPENDIX

APPENDIX A

A.1 THE G++ WRAPPER SCRIPT

```
#! /bin/bash
###############################################################################
#   Copyright (C) 2009 by Hessel Hoogendorp                                   #
#   bugs.ccc@gmail.com                                                        #
#                                                                             #
#   This program is free software; you can redistribute it and/or modify      #
#   it under the terms of the GNU General Public License as published by      #
#   the Free Software Foundation; either version 2 of the License, or         #
#   (at your option) any later version.                                       #
#                                                                             #
#   This program is distributed in the hope that it will be useful,           #
#   but WITHOUT ANY WARRANTY; without even the implied warranty of            #
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the             #
#   GNU General Public License for more details.                              #
#                                                                             #
#   You should have received a copy of the GNU General Public License         #
#   along with this program; if not, write to the                             #
#   Free Software Foundation, Inc.,                                           #
#   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.                 #
###############################################################################

# ------------------------------------------------------------------------------
# This script wraps g++.
#
# First, it makes the original call to g++, and, if that succeeds, then calls
# the call information extractor with the same parameters.
# ------------------------------------------------------------------------------


# ------------------------------------------------------------------------------
# Call the original g++ compiler.
# ------------------------------------------------------------------------------

# Find the installation path of g++.
CXX=$(whereis g++)
CXX=${CXX#*: }         # Strip off 'g++: '.
CXX=${CXX%% *}         # Strip off any secondary paths.

# Test whether we found an existing, executable program.
if [ ! -x $CXX ]; then
    echo "ERROR: The g++ wrapper script could not find a usable C++ compiler."
    exit 1
fi

# Reconstruct the arguments passed to this script; sometimes a -D option is
# passed to the compiler, with escaped quotes in the argument. The original
```

```
# command line may look like this:
# -DSOMENAME="\"SomeText to be quoted in the source file.\""
# To make sure this gets passed properly to the actual compiler, we need to
# reconstruct such arguments. That is, we need to manually put back the outer
# pair of quotes, since they have been interpreted by the shell (to form a
# single argument). The next step will be to write out the reconstructed command
# line to a temporary file and source that file. This will then properly call
# the compiler.
NEWARGS=
EXECBYSOURCING=0
for ARG in "$@"; do
    case $ARG in
        -D*=\"*\")
            EXECBYSOURCING=1
            ARG="${ARG/=\"/=}"
            ARG="${ARG/\"/\\\"\"}"
            ARG="${ARG/=/=\"\\\"}"
            ;;
    esac
    NEWARGS="$NEWARGS ""${ARG}"
done

# Supply the compiler with all arguments passed to this script, but add the
# -save-temps switch. This will cause the compiler to save the preprocessed
# source code. The extractor will then use this preprocessed source code to
# extract the call information.

if [ $EXECBYSOURCING -eq 0 ]; then

    CMD="$CXX $@ -save-temps"
    $CMD
    exitCode=$?

else

    CMD="$CXX"" ""$NEWARGS"" -save-temps"

    # Generate and store the name of a temporary file.
    TMPFILE="ccc-$RANDOM.$RANDOM.$RANDOM-ccc"

    # Output the command line to the temporary file.
    echo "$CMD" > $TMPFILE

    # Source the temporary file.
    . $TMPFILE

    exitCode=$?

    # Remove the temporary file.
    rm $TMPFILE

fi

# Check whether the call to g++ was successful. If not, return with the
```

```
# compiler's exit code.
if [ $exitCode -ne 0 ]; then
    exit $exitCode
fi


# ------------------------------------------------------------------------
# Extract call information.
# ------------------------------------------------------------------------

# Call the call information extractor with the same arguments that were
# supplied to this script.
CMD="ccc-extract $@"
$CMD

# The compiler returned success, so return success regardless of the output of
# the extraction script.
exit 0
```

A.2 SOURCE CODE OF THE *fib* PROGRAM

A.2.1 *File* ***main.cc***

```
#include "fib.h"

int main(int argc, char** argv)
{
    Fib f;
    f.Calculate(20);
    return 0;
}
```

A.2.2 *File* ***fib.h***

```
#ifndef FIB_H
#define FIB_H

class Fib
{
public:
    int Calculate(unsigned int n);
};

#endif
```

A.2.3 *File* ***fib.cc***

```
#include "fib.h"

int Fib::Calculate(unsigned int n)
{
    switch(n)
    {
```

```
    case 0: return 0;
    case 1: return 1;
    default: return Calculate(n - 1) + Calculate(n - 2);
    }
}
```

A.3 SOURCE CODE OF THE *precalc* PROGRAM

A.3.1 *File* **pc_main.cc**

```
// ----------------------------------------------------------------------
// PreCalc - A simple prefixed-operator calculator
// ----------------------------------------------------------------------
// GRAMMAR:
//
// <EXPRESSION> ::= <OPERATOR>(<EXPRESSION>,<EXPRESSION>) | <NUMBER>
// <OPERATOR>   ::= + | - | * | /
// <NUMBER>     ::= <NUMBER><DIGIT> | <DIGIT>
// <DIGIT>      ::= 0 | 1 | 2 | 3 |4 | 5 | 6 | 7 | 8 | 9
// ----------------------------------------------------------------------
// EXAMPLE INPUT: *(+(5,3),2)
// ----------------------------------------------------------------------

#include "pc_printing.h"
#include "pc_lexer.h"
#include "pc_parser.h"
#include "pc_ast.h"
#include "pc_astvisitor.h"
#include <stdio.h>

int main(int argc, char** argv)
{
    if (argc != 2)
    {
        printError("Illegal number of arguments.");
        printUsage();
        return 1;
    }

    Lexer lexer(argv[1]);
    Parser parser(&lexer);
    ASTNode* pNode = parser.Parse();

    ASTVisitor visitor;
    int result = visitor.VisitAST(pNode);

    delete pNode;

    printf("Result: %i\n", result);

    return 0;
}
```

A.3.2 *File* ***pc_printing.h***

```
#ifndef PC_PRINTING_H
#define PC_PRINTING_H

void printError(const char* error);
void printUsage();

#endif
```

A.3.3 *File* ***pc_printing.cc***

```
#include "pc_printing.h"
#include <stdio.h>

void printLine(const char* line)
{
    printf("%s\n", line);
}

void printError(const char* error)
{
    printf("ERROR: ");
    printLine(error);
    printLine("");
}

void printUsage()
{
    printLine("Usage: precalc EXPRESSION");
    printLine("Calculates the values of simple prefixed-"
              "operator arithmatic expressions.");
    printLine("");
    printLine("Example: Invoking precalc with expression"
              " *(5,-(3,2)) calculates");
    printLine("the value of the infix expression 5*(3-2)");
}
```

A.3.4 *File* ***pc_lexer.h***

```
#ifndef PC_LEXER_H
#define PC_LEXER_H

enum TokenType
{
    OP_ADD,
    OP_SUBTRACT,
    OP_MULTIPLY,
    OP_DIVIDE,
    OL_BEGIN,
    OL_END,
    COMMA,
    NUMBER
};
```

```
class Lexer
{
public:
    Lexer(char* expr);

public:
    bool Next();

    TokenType GetTokenType();
    int GetNumber();


private:
    char* m_expr;
    int m_pos;

    TokenType m_tokenType;
    int m_number;
};

#endif
```

A.3.5 *File* ***pc_lexer.cc***

```
#include "pc_lexer.h"
#include "pc_printing.h"
#include <string.h>

Lexer::Lexer(char* expr)
{
    m_expr = expr;
    m_pos = 0;
}

bool Lexer::Next()
{
    if (m_pos >= (int)strlen(m_expr)) return false;

    switch(m_expr[m_pos])
    {
        case '+':
            m_tokenType = OP_ADD;
            break;
        case '-':
            m_tokenType = OP_SUBTRACT;
            break;
        case '*':
            m_tokenType = OP_MULTIPLY;
            break;
        case '/':
            m_tokenType = OP_DIVIDE;
            break;
        case '(':
```

```
            m_tokenType = OL_BEGIN;
            break;
        case ')':
            m_tokenType = OL_END;
            break;
        case ',':
            m_tokenType = COMMA;
            break;
        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
            m_tokenType = NUMBER;
            m_number = 0;
            while (m_expr[m_pos] >= '0' && m_expr[m_pos] <= '9')
            {
                m_number = (m_number * 10) + (m_expr[m_pos] - 48);
                m_pos++;
            }
            return true;
        default:
            printError("Unknown token.");
            return false;
    }
    m_pos++;

    return true;
}

TokenType Lexer::GetTokenType()
{
    return m_tokenType;
}

int Lexer::GetNumber()
{
    return m_number;
}
```

A.3.6 *File* `pc_parser.h`

```
#ifndef PC_PARSER_H
#define PC_PARSER_H

#include "pc_lexer.h"

class ASTNode;
class OperatorNode;
```

```
class Parser
{
public:
    Parser(Lexer* pLexer);
    ASTNode* Parse();

private:
    ASTNode* ParseExpression();
    bool ParseOperandList(ASTNode** ppOperand1, ASTNode** ppOperand2);
    bool ParseOperandListBegin();
    bool ParseOperandListEnd();
    bool ParseComma();

    bool TryNext();
    bool TestToken(TokenType expected);

private:
    Lexer* m_pLexer;
};

#endif
```

A.3.7 *File* `pc_parser.cc`

```
#include "pc_parser.h"
#include "pc_lexer.h"
#include "pc_ast.h"
#include "pc_printing.h"

Parser::Parser(Lexer* pLexer)
{
    m_pLexer = pLexer;
}

ASTNode* Parser::Parse()
{
    return ParseExpression();
}

ASTNode* Parser::ParseExpression()
{
    if (!TryNext()) return 0;

    ASTNode* pNode     = 0;
    ASTNode* pOperand1 = 0;
    ASTNode* pOperand2 = 0;
    switch(m_pLexer->GetTokenType())
    {
        case OP_ADD:
            if (!ParseOperandList(&pOperand1, &pOperand2)) return 0;
            pNode = new AddNode(pOperand1, pOperand2);
            break;
```

```
        case OP_SUBTRACT:
            if (!ParseOperandList(&pOperand1, &pOperand2)) return 0;
            pNode = new SubtractNode(pOperand1, pOperand2);
            break;
        case OP_MULTIPLY:
            if (!ParseOperandList(&pOperand1, &pOperand2)) return 0;
            pNode = new MultiplyNode(pOperand1, pOperand2);
            break;
        case OP_DIVIDE:
            if (!ParseOperandList(&pOperand1, &pOperand2)) return 0;
            pNode = new DivideNode(pOperand1, pOperand2);
            break;
        case NUMBER:
            pNode = new NumberNode(m_pLexer->GetNumber());
            break;
        default:
            printError("Unexpected token in input. Expected operator or number.");
            return 0;
    }

    return pNode;
}

bool Parser::ParseOperandList(ASTNode** ppOperand1, ASTNode** ppOperand2)
{
    if (!ParseOperandListBegin()) return false;
    if ((*ppOperand1 = ParseExpression()) == 0) return false;
    if (!ParseComma()) return false;
    if ((*ppOperand2 = ParseExpression()) == 0) return false;
    if (!ParseOperandListEnd()) return false;
    return true;
}

bool Parser::ParseOperandListBegin()
{
    return TryNext() && TestToken(OL_BEGIN);
}

bool Parser::ParseOperandListEnd()
{
    return TryNext() && TestToken(OL_END);
}

bool Parser::ParseComma()
{
    return TryNext() && TestToken(COMMA);
}

bool Parser::TryNext()
{
    if (!m_pLexer->Next())
    {
        printError("Unexpected end of expression.");
        return false;
```

```
    }

    return true;
}

bool Parser::TestToken(TokenType expected)
{
    if (m_pLexer->GetTokenType() != expected)
    {
        printError("Unexpected token found");
        return false;
    }

    return true;
}
```

A.3.8 *File* **pc_ast.h**

```
#ifndef PC_AST_H
#define PC_AST_H

enum NodeType
{
    TP_ADD,
    TP_SUBTRACT,
    TP_MULTIPLY,
    TP_DIVIDE,
    TP_NUMBER
};


class ASTNode
{
public:

    virtual ~ASTNode();
    virtual NodeType GetType() = 0;
};


class OperatorNode : public ASTNode
{
public:
    OperatorNode(ASTNode* pOperand1, ASTNode* pOperand2);
    virtual ~OperatorNode();

    ASTNode* GetOperand1();
    ASTNode* GetOperand2();

protected:
    ASTNode* m_pOperand1;
    ASTNode* m_pOperand2;
};
```

```
class AddNode : public OperatorNode
{
public:
    AddNode(ASTNode* pOperand1, ASTNode* pOperand2);
    virtual NodeType GetType();
};


class SubtractNode : public OperatorNode
{
public:
    SubtractNode(ASTNode* pOperand1, ASTNode* pOperand2);
    virtual NodeType GetType();
};


class MultiplyNode : public OperatorNode
{
public:
    MultiplyNode(ASTNode* pOperand1, ASTNode* pOperand2);
    virtual NodeType GetType();
};


class DivideNode : public OperatorNode
{
public:
    DivideNode(ASTNode* pOperand1, ASTNode* pOperand2);
    virtual NodeType GetType();
};


class NumberNode : public ASTNode
{
public:
    NumberNode(int number);
    virtual NodeType GetType();
    int GetNumber();

private:
    int m_number;
};

#endif
```

A.3.9 *File* **pc_ast.cc**

```
#include "pc_ast.h"

ASTNode::~ASTNode()
{
}
```

```
OperatorNode::OperatorNode(ASTNode* pOperand1, ASTNode* pOperand2)
{
    m_pOperand1 = pOperand1;
    m_pOperand2 = pOperand2;
}

OperatorNode::~OperatorNode()
{
    if (m_pOperand1 != 0)
    {
        delete m_pOperand1;
        m_pOperand1 = 0;
    }

    if (m_pOperand2 != 0)
    {
        delete m_pOperand2;
        m_pOperand2 = 0;
    }
}

ASTNode* OperatorNode::GetOperand1()
{
    return m_pOperand1;
}

ASTNode* OperatorNode::GetOperand2()
{
    return m_pOperand2;
}


AddNode::AddNode(ASTNode* pOperand1, ASTNode* pOperand2)
    : OperatorNode(pOperand1, pOperand2)
{
}

NodeType AddNode::GetType()
{
    return TP_ADD;
}


SubtractNode::SubtractNode(ASTNode* pOperand1, ASTNode* pOperand2)
    : OperatorNode(pOperand1, pOperand2)
{
}

NodeType SubtractNode::GetType()
{
    return TP_SUBTRACT;
}
```

```
MultiplyNode::MultiplyNode(ASTNode* pOperand1, ASTNode* pOperand2)
    : OperatorNode(pOperand1, pOperand2)
{
}

NodeType MultiplyNode::GetType()
{
    return TP_MULTIPLY;
}


DivideNode::DivideNode(ASTNode* pOperand1, ASTNode* pOperand2)
    : OperatorNode(pOperand1, pOperand2)
{
}

NodeType DivideNode::GetType()
{
    return TP_DIVIDE;
}


NumberNode::NumberNode(int number)
{
    m_number = number;
}

NodeType NumberNode::GetType()
{
    return TP_NUMBER;
}

int NumberNode::GetNumber()
{
    return m_number;
}
```

A.3.10 *File* ***pc_astvisitor.h***

```
#ifndef PC_ASTVISITOR_H
#define PC_ASTVISITOR_H

class ASTNode;
class NumberNode;
class OperatorNode;


class ASTVisitor
{
public:

    int VisitAST(ASTNode* pAST);

private:
```

```
    int VisitNode(ASTNode* pNode);
    int VisitNumber(NumberNode* pNumber);
    int VisitOperator(OperatorNode* pOperator);

};

#endif
```

A.3.11 *File* ***pc_astvisitor.cc***

```
#include "pc_astvisitor.h"
#include "pc_ast.h"
#include "pc_printing.h"

int ASTVisitor::VisitAST(ASTNode* pAST)
{
    NodeType nt = pAST->GetType();
    if (nt == TP_ADD      || nt == TP_SUBTRACT ||
        nt == TP_MULTIPLY || nt == TP_DIVIDE)
    {
        return VisitNode(pAST);
    }
    else
    {
        printError("The supplied AST node is not an Operator node.");
        return 0;
    }
}

int ASTVisitor::VisitNode(ASTNode* pNode)
{
    if (pNode->GetType() == TP_NUMBER)
        return VisitNumber((NumberNode*)pNode);

    return VisitOperator((OperatorNode*)pNode);
}

int ASTVisitor::VisitNumber(NumberNode* pNumber)
{
    return pNumber->GetNumber();
}

int ASTVisitor::VisitOperator(OperatorNode* pOperator)
{
    int operand1 = VisitNode(pOperator->GetOperand1());
    int operand2 = VisitNode(pOperator->GetOperand2());

    switch(pOperator->GetType())
    {
        case TP_ADD:
            return operand1 + operand2;
        case TP_SUBTRACT:
            return operand1 - operand2;
        case TP_MULTIPLY:
```

```
            return operand1 * operand2;
        case TP_DIVIDE:
            return operand1 / operand2;
    }

    return 0;
}
```

BIBLIOGRAPHY

[1] Bison - gnu parser generator. URL http://www.gnu.org/software/bison/. (Cited on page 82.)

[2] Doxygen source code documentation generator tool. URL http://www.stack.nl/~dimitri/doxygen/. (Cited on page 10.)

[3] Eclipse.org home. . URL http://eclipse.org. (Cited on page 9.)

[4] Eclipse c/c++ development tooling. . URL http://www.eclipse.org/cdt. (Cited on page 9.)

[5] Mozilla firefox source code. . URL https://developer.mozilla.org/en/download_mozilla_source_code. (Cited on pages 82, 89, and 101.)

[6] Firefox web browser. . URL http://www.mozilla.com/en-US/firefox/firefox.html. (Cited on page 96.)

[7] GNU Binutils. . URL http://www.gnu.org/software/binutils/. (Cited on page 74.)

[8] GCC, the GNU Compiler Collection. . URL http://gcc.gnu.org/. (Cited on pages 74 and 101.)

[9] IntelÂ® C++ Compiler Professional Edition for Linux. URL http://software.intel.com/en-us/articles/intel-c-compiler-professional-edition-for-linux-documentation/. (Cited on page 103.)

[10] KDevelop - KDE Development Environment. URL http://www.kdevelop.org/. (Cited on page 10.)

[11] COM: Component Object Model Technologies. URL http://www.microsoft.com/COM/. (Cited on page 90.)

[12] Rigi: A visual tool for understanding legacy systems. URL http://www.rigi.csc.uvic.ca/index.html. (Cited on page 10.)

[13] The Seamonkey Project. URL http://www.seamonkey-project.org/. (Cited on page 96.)

[14] SolidSX - Solid Software Xplorer. URL http://www.solidsourceit.com/products/SolidSX-source-code-dependency-analysis.html. (Cited on pages 82, 83, and 102.)

[15] Mozilla Sunbird. URL http://www.mozilla.org/projects/calendar/sunbird/. (Cited on page 96.)

[16] Mozilla Messaging. URL http://www.mozillamessaging.com/. (Cited on page 96.)

[17] Ibm visualage c++. URL http://www-01.ibm.com/software/awdtools/vacpp/. (Cited on page 10.)

[18] Microsoft visual studio. URL http://www.microsoft.com/visualstudio/en-us/default.mspx. (Cited on page 9.)

[19] XPCOM: An Introduction to XPCOM. . URL https://developer.mozilla.org/en/XPCOM. (Cited on pages 91 and 95.)

[20] Embedding/NewApi/XpcomAccess. . URL https://wiki.mozilla.org/Embedding/NewApi/XpcomAccess#Include_Files. (Cited on page 92.)

[21] An Overview of XPCOM. . URL https://developer.mozilla.org/en/Creating_XPCOM_Components/An_Overview_of_XPCOM. (Cited on page 91.)

[22] Education/Learning/UsingXpcom. . URL https://wiki.mozilla.org/Education/Learning/UsingXpcom. (Cited on page 92.)

[23] David Auber. Tulip Software. 2009. URL http://tulip.labri.fr/. (Cited on pages 2, 69, 81, and 82.)

[24] Árpád Beszédes, Rudolf Ferenc, and Tibor Gyimóthy. Columbus: A reverse engineering approach. In *Pre-Proceedings of IEEE Workshop on Software Technology and Engineering Practice (STEP 2005)*, pages 93–96, sep 2005. (Cited on page 10.)

[25] F. Boerboom and A Janssen. Fact extraction, querying and visualization of large c++ code bases. 2006. (Cited on pages 8, 11, and 15.)

[26] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., 1996. URL http://delivery.acm.org/10.1145/780000/776917/p694-schmidt.pdf?key1=776917&key2=5813349011&coll=GUIDE&dl=ACM&CFID=38883359&CFTOKEN=61561711. (Cited on page 92.)

[27] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990. (Cited on pages 27, 31, 33, 46, 47, and 59.)

[28] Stephan Diehl. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2008. (Cited on page 1.)

[29] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. (Cited on pages 7, 17, and 101.)

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Inc., 1st edition, 1995. (Cited on page 26.)

[31] Emden Gansner, Yifan Hu, Yehuda Koren, Stephen North, John Ellson, and Arif Bilgin. Graphviz - Graph Visualization Software. URL http://www.graphviz.org/. (Cited on pages 69, 81, and 82.)

[32] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006. ISSN 1077-2626. doi: http://doi.ieeecomputersociety.org/10.1109/TVCG.2006.147. (Cited on page 83.)

[33] Hessel Hoogendorp, Ozan Ersoy, Dennie Reniers, and Alexandru Telea. Extraction and Visualization of Call Dependencies for Large C/C++ Code Bases: A Comparative Study. IEEE VISSOFT, 2009. (Cited on pages 1, 69, and 82.)

[34] Tim Littlefair. CCCC: C and C++ Code Counter. URL http://cccc.sourceforge.net/. (Cited on page 82.)

[35] Scott McPeak. Elsa: The Elkhound-based C/C++ Parser. 2006. URL http://www.eecs.berkeley.edu/~smcpeak/elkhound/sources/elsa/. (Cited on pages 11 and 26.)

[36] Scott McPeak, Daniel S. Wilkerson, and Karl Chen. Oink: a Collaboration of C++ Static Analysis Tools. URL http://www.cubewano.org/oink/. (Cited on page 11.)

[37] Petru Florin Mihancea, George Ganea, Ioana Verebi, Cristina Marinescu, and Radu Marinescu. Mcc and mc#: Unified c++ and c# design facts extractors tools. In *SYNASC*, pages 101–104, 2007. (Cited on page 10.)

[38] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, Boston, MA, USA, 1999. (Cited on page 103.)

[39] István Siket and Rudolf Ferenc. Calculating metrics from large c++ programs. 2008. URL http://citeseerx.ist.psu.edu/viewdoc/summary10.1.1.121.3534. (Cited on page 75.)

[40] A. Telea and L. Voinea. An Interactive Reverse-Engineering Environment for Large-Scale C++ Code. ACM SoftVis, 2008. (Cited on page 11.)

[41] Alexandru Telea, Ozan Ersoy, Hessel Hoogendorp, and Dennie Reniers. Comparison of node-link and hierarchical edge bundling layouts: A user study. In Daniel A. Keim, Aiko Pras, Jürgen Schönwälder, and Pak Chung Wong, editors, *Visualization and Monitoring of Network Traffic*, number 09211 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. URL http://drops.dagstuhl.de/opus/volltexte/2009/2154. (Cited on page 85.)