Generating Accurate Dependencies for Large Software

by

Pei Wang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2013

Abstract

Dependencies between program elements can reflect the architecture, design, and implementation of a software project. According a industry report, intra- and inter-module dependencies can be a significant source of latent threats to software maintainability in long-term software development, especially when the software has millions of lines of code.

This thesis introduces the design and implementation of an accurate and scalable analysis tool that extracts code dependencies from large C/C++ software projects. The tool analyzes both symbol-level and module-level dependencies of a software system and provides an utilization-based dependency model. The accurate dependencies generated by the tool can be provided as the input to other software analysis suits; the results along can help developers identify potential underutilized and inconsistent dependencies in the software. Such information points to potential refactoring opportunities and assists developers with large-scale refactoring tasks.

Acknowledgements

I would like to thank my supervisor, Dr. Lin Tan, who first leads me into this interesting research and help me proceed. One of my readers, Dr. Derek Rayside, provided me with precious advice and directions which tremendously improve this thesis. I also greatly appreciate all the help and feedback from my colleagues and industry partners. They help me make this possible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The size and complexity of software systems have been increasing. Many modern software projects contain millions or tens of millions of lines of code, e.g., Chromium, Firefox, and the Linux kernel. They typically consist of hundreds of modules. Such software systems are often under active development with daily or more frequent commits over years or decades by hundreds or thousands of developers. To keep adding new features into these large software projects and maintain the code bases in a healthy state, developers need to understand the dependencies between different components of the software. In addition to revealing the status of the software, dependencies can also be used as input to other advanced program analyses, e.g., code smell detection [52, 49, 43], managing technical debt in software development [39], software clustering and visualization [27, 46], and performance profiling [33, 45].

1.1 Motivation

For many large software projects, developers constantly join and depart from the development. Due to the complexity and fast evolution of software, the coupling between modules can deviate from the original design, which hurts software maintainability. In addition, developers who are in charge of different modules also tend to be grouped into different teams. Due to limited across-team communication, individual developers are often unaware of the changes of modules that they are not in charge of. This can lead to useless code blocks in the repository because there are cases in which developers are not informed when some pieces of their code are no longer needed by other modules. A recent study shows that inter-module dependencies can be a significant source of threats to software maintainability in long-term software development, especially if an organization is running a large and shared code base [39].

To manage such challenges, a first and key step is to provide developers with an accurate view of the code dependencies among and inside modules. Benefits of such dependency information include:

1. Identifying bad dependencies that hurt software maintainability, and

2. Helping developers make informed decisions when they perform system-wide or large-scale refactoring.

This work focuses on two main types of bad dependencies: *underutilized dependencies* and *inconsistent dependencies*. A dependency is called underutilized if only a small portion of the target module is utilized by the client module. Underutilized dependencies often slow down the building process and blow up the code size [39]. In addition, underutilized dependencies can indicate poor cohesion of the target because low utilization shows that a small portion of the target may be loosely coupled with the rest of the module. As a result, it may be better to separate the target into two parts so that the client can depend on only one of them.

Inconsistent dependencies are dependencies that violate software design. For example, a project may want to build its core modules without using any third-party libraries. If a core module depends on a third-party library, such a dependency is an inconsistent dependency. Programmers may break the design rules and introduce inconsistent dependencies for short-term gains (e.g., meeting the release deadline), with the cost of hurting long-term maintainability of the project.

In addition to helping detect unhealthy cohesion and coupling in the code base, accurate dependencies can also assist the developers in large-scale refactoring. As software grows and evolves, some large-scale refactoring tasks become mandatory. In 2010 and 2011, the profile-guided optimization version of Firefox failed to be built on 32-bit Windows because the code base was too large and the linker ran out of virtual memory address space. Figure 1.1 shows the growth of Firefox code base in recent years. As a temporary fix, the developers turned off and reverted a few pieces of new code. Noticing that the monolithic design became a critical blocker for the evolution of the project, the Firefox team finally decided to break a giant module (libxul, the core part of Firefox) into several smaller standalone libraries and build them separately.

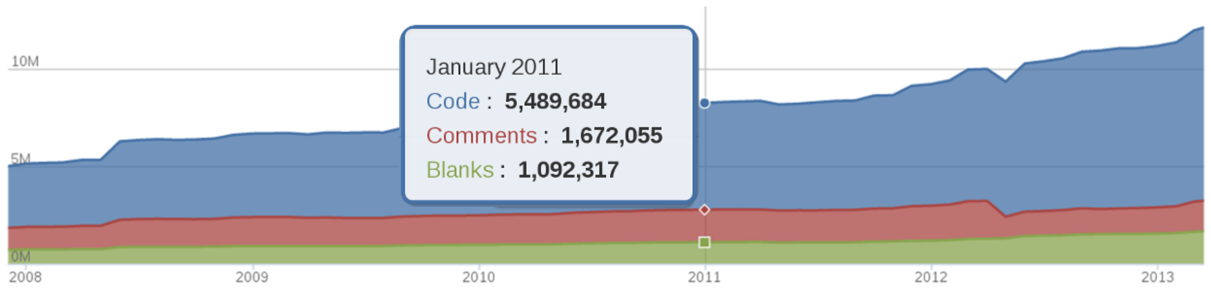


Figure 1.1: Firfox code base statistics[2]

While the detailed plan of this fix is still under investigation, comments in several Bugzilla[1] tickets (709721, 711386, 753056) show that the dependencies are confusing the developers when they work on such large-scale software refactoring tasks. Thus, a better understanding of the inter- and intra-module dependencies can help the developers choose the right points to refactor and make the task easier.

Although software maintainance can significantly benefit from dependency analysis, extracting accurate dependencies from projects consisting of millions of lines of code is challenging, especially for C/C++ programs. As a widely used programming language, C++ comes with advanced features that make fine-grained dependency analysis difficult. The goal of this work is to design and implement a practical tool to ease the analysis work and make it scalable.

1.2 Objectives

Program dependencies can be classified into two main categories: *structural dependencies* and *behavioral dependencies* [21].

Structural dependencies are statically decided dependencies. From the view of the toolchain, structural dependencies are explicit interactions between different program elements (types, variables, and subroutines). For the C++ programming language, the interactions are mainly *symbol references*. In the scope of this thesis, a symbol reference is defined as a pair of global symbols (sym_a, sym_b) where sym_b is referred to inside the

[1]Bugzilla (`https://bugzilla.mozilla.org/`) is an open bug tracker for developers and users to report bugs spotted in open source software projects.

[2]Data from `http://www.ohloh.net/p/firefox/analyses/latest/languages_summary`. Numbers indicate lines of text.

definition of sym_a. The most common symbol reference pairs are two functions of which one is the caller and the other is the callee. References to symbols in program literals must be resolved by the toolchain during program compilation and linking. This is the type of dependencies that the programmers directly manage in development.

Behavioral dependencies, in contrast to structural dependencies, often involve logical abstractions like public interfaces, signal handling, and some broadcast mechanisms. These abstractions are mostly implemented by indirect references to memory addresses during run-time, thus making accurate behavioral dependencies analysis quite difficult.

There are mainly two reasons why structural and behavioral dependencies are considered separately. Firstly, structural dependencies are closely related to software build. Since there are a variety of issues with respect to building large software [39], programmers are interested in dependencies *directly* related to the build process. Secondly, in a large software development team, developers who define the interface of dynamic dependencies and those who actually inject dynamic dependencies can be different. For example, developers of the IPC facility in a multi-process browser will define the interfaces and handlers for incoming IPC requests; however, how these interfaces will be implemented by other modules can be out of the concern of IPC facility developers. When inspecting the design and implementation of the IPC module, developers may want to exclude the involvement of other modules if these modules do not explicitly impact the IPC mechanism. Based on these two reasons, it is reasonable to differentiate structural (static) from behavioral (dynamic) dependencies.

The dependency analysis tool introduced in this thesis focuses on scalable analyses for structural dependencies. Especially, the tool will stick to symbol references when performing the analysis. After obtaining symbol-level dependencies, the tool will further group symbols into modules and calculate module utilization. The utilization-based metrics can provide the developers with a quantitative view of the cohesion and coupling of their software modules. Details of the metrics will be introduced in Chapter 4.2. Also, we want the tool to be as accurate as possible. In the scope of this thesis, the term "accurate dependencies" refers to analysis results that are close to actual dependencies in the software. That means we want to cover actual dependencies as much as possible (suppress false negatives) and exclude nonexistent dependencies from the result to the greatest extent (suppress false positives).

However, considering the nature of the C++ programming language and the demand of the developers in practice, the tool does attempt to partially support behavioral dependency analysis. For C++ programs, a major source of uncertain runtime dependencies are virtual function invocations. Chapter 3.4 and Chapter 5 will explain the details about virtual function call analysis.

Chapter 2

Related Work

Dependency analysis is not a new topic. During the past two decades, many different approaches have been proposed to solve different dependency analysis problems. However, for C++ programs, most existing dependency analysis techniques are light weight. Typically, they are performed on source code level or abstract syntax tree level. There are also techniques trying to reveal software structures from a much lower level — program binaries. Tools of this kind are usually better at supporting reverse engineering than providing information to software developers.

Some other C/C++ oriented techniques are based on software development history rather than source code. Generally code fact based techniques are more accurate and more suitable for structural dependency analysis. History based techniques, on the contrary, do not strictly detect the connections between program elements which are recognized by programming toolchain (e.g., function calls and global variable usages in static initializers). Instead, their results are implied by empirical patterns that showing which modules are more likely to change simultaneously.

2.1 Textual Analysis

The most representative light-weight dependency analysis techniques are those working directly on source code text with Textural analysis can be performed on two different levels – lexical level and syntactical level. Lexical techniques are mostly based on pattern matching, while syntactical ones exploit the power of formal language grammars.

Murphy et al. proposed a lexical dependency extraction technique [41, 40]. Their work defined a specification language and users need to provide patterns for dependency recognition. Their specification language, like other lexical systems, is based on regular expressions. Although several features are added in the specification to ease pattern generation, the onus of writing correct patterns is still on the users. Besides the inherent inaccuracy of lexical analysis, the quality of the analysis results also heavily depends on the quality of the patterns devised by the users.

Syntactical approaches are more flexible and can tackle a wider range of dependency-related problems than lexical ones. DSketch [23] utilizes the island grammar technique [38] to excavate dependencies from polylingual systems. The main focus of DSketch is interactions between different programming languages (e.g., calling SQL from Java). This technique further alleviates the burden of users for the system is more tolerant to pattern glitches because it allows users to provide fuzzy patterns. However, the fact that DSketch relies on patterns still makes the system not fully automatic. Also, there remains an accuracy problem because of user provided patterns.

2.2 Abstract Syntax Tree (AST) Analysis

Modern IDEs such as Eclipse CDT [5] can provide rich dependency information to assist basic code refactoring by traversing the abstract syntax trees. Since the integrated development environments are expected to provide programmers with interactive assistance, they usually have to keep the ASTs of the entire project in the memory, making it difficult for them to scale to millions of lines of code. Moreover, dependency analysis is often not the main focus of such IDEs, so they can only provide limited dependency-related metrics.

Doxygen [3] is a widely used tool which can provide basic dependency analyses for C++ software. The main objective of Doxygen is to automatically generate documentation from annotated source code. However, Doxygen can generate dependency graphs, inheritance diagrams, and collaboration diagrams as the byproducts of its source code parsing. Basically, Doxygen is also an AST based analyzer. It consists of multiple parsers to process different contents of the source code. Due to limitations of light-weight analyses, dependencies extracted by Doxygen is not complete and even less accurate than that provided by IDEs like Eclipse CDT with respect to some complicated language features, e.g., templates in C++.

Program Database Toolkit (PDT) [33] is a framework for analyzing source code and making rich program knowledge accessible to developers of static and dynamic analysis tools. Potentially, the output of PDT can be used as the material to build accurate dependencies for C++ programs. PDT extracts code facts, e.g., function definitions, function calls, and type hierarchies, from source files and dump them in an uniform format named as PDB (program database). By merging all PDBs of a project, it is possible to generate the complete structural dependency graph for the entire project. PDT takes advantage of the EDG C++ front end [6] and performs the extractions over the EDG intermediate language, which is essentially the abstract syntax tree generated by the front end.

However, PDT has some significant weaknesses. First of all, PDT does not extract global variable related dependency facts, which makes analysis based on PDB incomplete. Another problem of PDT is that it is not compatible with legacy code. We applyed PDT to the Chromium open source project and only 992 out of 9337 C/C++ source files in Chromium can be successfully processed by PDT. Finally, when trying to merge these 992 PDB objects PDT raised a segmentation fault, suggesting that there is a scalability problem with its implementation, if not a design issue.

CPPX [26] is another source code fact extractor designed for C++. It translates GCC's intermediate representation to a new format. This new format is a standardized textual form designed for a variety of software analysis and visualization tools. Similar to PDT, CPPX does not directly extracts dependencies from the software but extract code facts from individual source files. It is non-trivial to merge CPPX outputs and transform them to software dependencies.

2.3 Binary Code Analysis

BFX/LDX is a tool suit which extracts code facts from machine code [53]. In the suit BFX processes individual object files and LDX merge BFX outputs together. Technically, BFX works like the binary code dump tool *objdump* and LDX works like a linker. They analyze programs at link-time and output program facts in a standardize format. Same as CPPX and PDT mentioned earlier, BFX and LDX do not directly extract program dependencies but only information which can be used to infer the dependencies, namely that they fail to provide a complete solution to the dependency analysis problem. In addition, because some source code information (especially type information) is lost in binary code, BFX and LDX have difficulties in extracting complete dependencies.

Although binary code analyses may not be suitable for reporting complete and human-readable dependencies, it can still be used to improve software quality. Davis et al. proposed a hybrid technique which combines both source code and binary (assembly) code analysis to eliminate dead functions [24]. Their work exploits a loopback approach. The static analysis tool first analyzes program binaries and comment out possible unreachable functions in the source code. The program is then rebuilt and runs against test cases to verify the dead code removal before another round of elimination is dispatched. Dead code analysis techniques exploited in this work can be borrowed to increase dependency analysis accuracy.

2.4 History Based Analysis

In addition to code based facts, researchers exploit software release histories to further explore software dependencies and potential design violations [52, 54]. This class of techniques keeps track of software changes and mines change patterns from the development or release histories. The basic idea is to find out which parts of the project are likely to be changed simultaneously and then draw dependencies between these parts. Strictly, history based analysis provides neither structural dependencies nor behavioral dependencies; however, the results of such techniques can be useful hints to further explorations on structures and behaviors of the software.

2.5 Commercial Tools

Several commercial tools [2, 15, 7, 9] are available in the market for fine-grained dependency analysis on C and C++ programs. Some of these tools claim that they can scale to millions of lines of code. With limited technical details about these commercial tools, it is unclear that how they perform dependency analysis and achieve the claimed accuracy and scalability. Also, none of these commercial tools provide utilization-driven dependency analysis according to publicly available information.

Chapter 3

Dependency Extraction

This chapter introduces how the analysis tool extracts symbol-level dependencies from C/C++ programs. The primary goal of the design is to make the extractor *scale to millions of lines of code*. In addition to scalability, the extractor is supposed to achieve the following objectives:

- Support salient C++ language features, and
- Maximize automation of the analysis procedure.

The rest of this chapter will emphasize the design decisions which make these objectives possible.

3.1 Challenges

It has, historically, been difficult to write dependency extraction tools for C/C++ due to the different dialects and the lack of a program representation that is well suited to dependency extraction. Programs written with Microsoft's C/C++ compiler might not compile with GCC, and vice versa. Source code is missing important dependency information, as is assembly, as is GCC's intermediate form — all of which have been used as the basis of previous dependency extraction tools. In this chapter we survey some of the challenges that researchers have faced trying to build dependency extraction tools with older compiler infrastructure for C/C++.

```
// Define localtime_override() function with asm name "localtime", so that all
// references to localtime() will resolve to this function. Notice that we need
// to set visibility attribute to "default" to export the symbol, as it is set
// to "hidden" by default in chrome per build/common.gypi.
__attribute__ ((__visibility__("default")))
struct tm* localtime_override(const time_t* timep) __asm__ ("localtime");
```

Figure 3.1: Example: affection of non-standard syntax on software dependencies

3.1.1 Main Challenges

Function overloading and default parameters. The C++ programming language allows the programmers to define functions with the same name that are different from each other by the type of parameters. Moreover, C++ allows the functions to define default parameters, which programmers do not need to provide for function invocation. As a result, functions called by the same name and provided with the same number of arguments can still have different definitions, thus mapped to different symbols. Without type checking, it is extremely difficult to match overloaded functions correctly in the source code.

Non-standard language syntax. Production compilers usually support non-standard language syntaxes. Some of these syntaxes allow the programmers to alter symbol linkage or set link-time alias at source code level, thus having impacts on software dependencies. The code snippet shown in Figure 3.1, which is picked from the Chromium project, gives a real-world example. In this example, a function `localtime_override` in the "*content_browser*" module is assigned a link-time alias, overriding the `localtime` function in the standard library and making every module calling `localtime` actually depend on "*content_browser*". To include such dependencies in the results, the analysis tool should support these non-standard language syntax as far as possible.

Implicit call sites. Unlike normal function calls, some special functions in C++ (e.g., copy constructors and overloaded operators) are not invoked by the conventional "()" annotation but triggered by other operators such as "+" and "=". Several other kinds of call sites, like default object constructions and destructions, can only be inferred by the lifetimes of the corresponding objects. Clearly, these implicit call sites cannot be identified by simple lexical matching. In fact, these call sites are not included even in many high-level intermediate representation (most ASTs). In order to determine the calling locations, dependency analysis tools working on the AST level need to additionally resolve this problem by processing all the contexts in which object lifetimes are handled

Templates. Template is an important C++ feature which provides parametric polymophism. With templates, programmers can define different instances (functions or classes) with similar behaviors given different data types. For dependency analysis, the problem is that template instances are not directly defined in the source code but are instantiated separately at compile time, and the instantiation information is usually not available in the AST [33, 13].

3.1.2 Other Challenges

Although this research focuses on structural dependency extraction and analysis, there are indeed a demand of inspecting behavioral dependencies. At present, fulfilling such demands is not the task of this work, but enumerating potential challenges can be helpful to consider the future work.

Virtual functions in C++ are designed to achieve subtype polymorphism which is dynamically handled. This is a special case when the classification of structural and behavioral dependencies is considered. On one hand, the compiler will do compile-time check on virtual function call sites. On the other hand, the actually called functions are decided at runtime. In the scope of this thesis, we consider that virtual functions lie on the boundary of structural and behavioral dependencies. Due to this consideration, out tool has partial support for virtual function calls, using some simple heuristics.

For C and C++ programs, one important type of behavioral dependencies is introduced by function calls via arbitrary function pointers. Since C and C++ have very few limitations on the target a pointer can point to, accurately extracting such dependencies can by extremely difficult, especially for software with millions of lines of code.

There are some even more challenging problems when considering program behaviors. In some computing environments, functions can be invoked across process boundaries, e.g., the binder IPC mechanism used by Android operating system [1]. In these cases, dependencies are established via low-level facilities like specific libraries or even system calls. Undoubtedly, automatically digging out these dependencies requires non-trivial effort.

In addition to the problems directly related to dependency extraction, C and C++ dependency extraction and analysis are often bothered by *software variants*. With the existence of pre-processor directives such as `ifdef` and `ifndef`, same C/C++ source files can be built into different binaries when given different configurations. Sometimes users may want the dependency analysis tool to cover multiple (even all possible) configurations in a single report. Some C/C++ software projects are well known for their diversity in terms of possible build configurations, e.g., the Linux kernel. For these projects, it is difficult for users to describe all the configurations of interest in detail, and it becomes the analysis tools onus to assist the users. Solving the problems that how to deal with preprocessor directives and to what extent the tool should support software variant analysis is a notable challenge in C/C++ dependency analysis researches.

Finally, depending on the nature of different projects, inline assembly code in C/C++ programs can also be a concern in terms of dependency extraction. Since assembly is actually another programming language, it can be difficult to process with only C/C++ oriented techniques.

3.2 Program Abstraction Selection

Based on the reasons listed above and the investigation on the related work mentioned in Chapter 2, neither source code nor abstract syntax tree is the appropriate level for the analysis tool to work on. To achieve the expected accuracy (identify symbol references correctly), the tool will need to work on some program abstraction other than AST.

In recent years, LLVM and Clang have emerged as new compiler infrastructure for C/C++ that address many of these concerns: the LLVM Intermediate Representation (IR) is well suited to dependency extraction, and Clang can parse a wide variety of C/C++ dialects and transform them into the LLVM IR. We leverage these developments to produce a new dependency extraction tool that is robust, accurate, and scalable.

The LLVM Compiler Infrastructure is a collection of modular and reusable compiler and toolchain technologies [10]. The LLVM core defines a language-independent and low-level intermediate representation for all front-ends in the infrastructure. LLVM IR is quite similar to assembly languages in form, but is expressive and typed at the same time. The most powerful feature of LLVM IR is that it keeps a large amount of high-level information about the program while being close to native code in form. The LLVM infrastructure is supported by a powerful and production-quality C family compiler front end – Clang, which is now the default C/C++ compiler on Mac, MINIX, and FreeBSD.

In summary, there are two significant reasons to choose LLVM IR as the input of a scalable and accurate dependency analyzer:

- Low-level representation with rich source code information. Compared to abstract syntax trees, LLVM IR is much closer to native code and contains information not available at AST stage, e.g., template instantiation information and call sites of default constructors and destructors. In LLVM IR, all the symbol names are mangled, so it is quite straightforward to distinguish overloaded functions and identify overloaded operators. Unlike assembly languages, LLVM IR is well typed, thus allowing high-level program information extraction. With the widely used and standardized DWARF debugging data [4] embedded in the IR, generating human-readable analysis results is possible.

- Production-quality front end support. As a fast-growing project, Clang is now fully capable of supporting C++11 standard. In addition, Clang supports most GNU language extensions (non-standard syntax) and can process legacy code originally written for GCC. Recent versions of Clang are able to compile real-world software projects including the Boost C++ libraries, a working Linux kernel with X window system, a working Android Linux kernel for Nexus 7, and 2 popular web browsers Chromium and Firefox. All of these projects contain several million lines of C/C++ code, proving the scalability of Clang.

When compared to LLVM IR, textual or AST-based analyses [41, 40, 23, 38] lack some low-level program information which is vital for generating accurate dependencies. On the other hand, binary analyses [53, 24] cannot exploit some high-level information to interpret the extracted facts. To the best of our knowledge, LLVM IR is the most appropriate abstraction for a dependency analysis tool to work on. By working on LLVM IR, the analyzer will be able to address the previously mentioned *main challenges.* in an efficient manner. As for the *other challenges* related to software variants and behavioral dependency analysis, LLVM IR also has the potential to address them except the inline assembly problem.

3.3 Pre-Processing

In the first step of the work flow, source code compilation is based on *one particular configuration*. As mentioned in Chapter 3.1, C and C++ programmers usually exploit pre-processor directives (e.g., `ifdef` and `ifndef`) to compile different part of the source code according to different build configurations. With these directives, the compiler can disable some pieces of the source code and exclude them in the intermediate representation. Since all subsequent analyses are based on IR objects generated by the compiler, code disabled by the directives will not be covered, which is a limitation of the current design. Chapter 7.2 has more discussions about this issue.

3.4 Virtual Function Analysis

Concerning virtual function calls, two different results are provided by the analysis tool: a naive analysis result and a Class Hierarchy Analysis (CHA) [25] result. In the naive result no subtype polymorphism is considered. It means that every virtual function call is considered as a normal function call, so only virtual functions in base classes can be invoked (as long as they are not pure virtual functions). This result corresponds to the virtual function dependencies required for compilation, and as mentioned in 1.2 this information is useful to software developers.

On the other hand, in the Class Hierarchy Analysis result every possible target virtual function in the class hierarchy is considered to be invoked. The class hierarchy analysis provides an upper bound for virtual function dependencies. Bacon and Sweeney evaluated CHA on 7 C++ programs (5,000 to 20,000 lines of code each), and reported that CHA can resolve on average 51% of the virtual function calls to a single target [18]. There is an improvement to CHA which is called Rapid Type Analysis (RTA) [18]. Based on the call graph generated by CHA, RTA uses information about instantiated classes to narrow down the range of possible virtual function calls. RTA is not implemented in this tool so it can be considered as future work.

In addition to RTA, pointer analysis is another potential solution to analyzing virtual function calls (and even the arbitrary function pointer challenge mentioned in Chapter 3.1.2). There are many pointer analysis algorithms available, fulfilling different application demands. For dependency analysis, pointer analysis algorithms do not need to be context- or flow-sensitive. Two representative algorithms of this type were proposed by Anderson [17] and Steensgaard [48]. When choosing the appropriate algorithm for a particular application, the question is how to make the trade-off between scalability and accuracy. In general, Anderson-style algorithms are more accurate but less scalable than Steensgaard-style algorithms. However, recent researches claimed that it is feasible to make Anderson-style algorithms scale to one million lines of C code [29, 28].

Although there is a possibility that pointer analyses can benefit extracting dependencies introduced by virtual function calls, no pointer analysis is applied in our tool. There are several reasons for this decision:

- As stated in Chapter 1.2, the primary design objectives of our tool is scalability and accuracy for structural dependencies. Behavioral dependencies decided at run-time is of less a concern. It is unclear that whether the gain of applying pointer analyses is worthy of the cost and risk that affects the scalability of the tool.

- There are technical challenges when implementing existing pointer analysis algorithms for LLVM IR. As for dependency extraction, pointer analyses usually require the whole program as inputs, but the LLVM linker is not able to link IR objects compiled from millions of lines of C++ code (see Chapter 5.2.3).

In addition to static pointer analyses, there are also techniques solving the problem dynamically, usually by instrumenting the source code or executable binaries [37, 36, 47, 44]. Resolving pointer aliases at run-time fits some on-demand applications well, because all the results obtained via dynamic approaches are accurate. However, pointer analyses give no extra benefits when dependencies are extracted dynamically. A dynamic analysis tool can obtain the dependencies by instrumenting the entries of functions and identify the callers by examining stack frames. Although this approach is simple, the limitation is also evident. Dynamically extracted results depend on program input and are very likely to be incomplete given limited computing resources.

Chapter 4

Dependency Analysis

After obtaining symbol-level dependencies, the tool will further provide high-level dependency information for the analyzed software. This high-level analysis will allow users to inspect software dependencies on the module level. This part of work is performed by the post-analyzer. Combined with the dependency extraction step introduced in Chapter 3, the complete work flow of the analysis tool is shown in Figure 4.1.

4.1 Modularization Strategy

With symbol-level dependencies extracted, the post-analyzer needs users to decide which module a symbol belongs to. In the scope of this thesis, a module is a set of arbitrary symbols regardless of where these symbols are defined. Technically, a modularization strategy is an oracle mapping a symbol to the module which it should be assigned to. How to design this oracle is decided by the users. At present, our tool provides users with a convenient interface to apply file-level modularization strategies. Users can exploit this interface by providing a plain text file which labels every source file in the software with a module name. According to the plain text, all symbols defined in a source file are grouped into the module indicated by the label of that file.

Please note the design of the tool is not limited to the current implementation. For instance, it is possible to implement another interface which takes a set of regular expressions from users and modularizes symbols based on their names. The only constraint on a modularization strategy is that the strategy must be scalable, so generally a modularization strategy can be any information reflecting system structures. For file-level strategies,

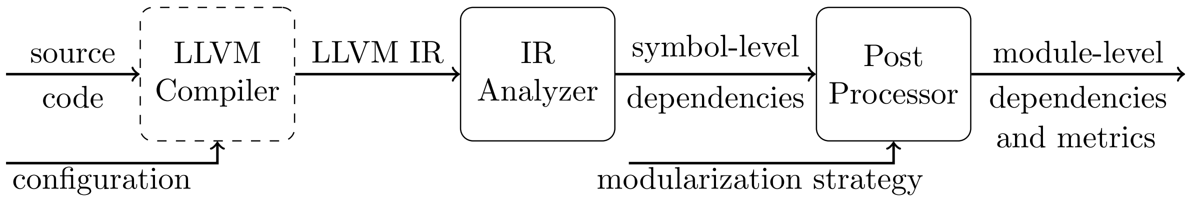


Figure 4.1: Complete work flow

the information can be as simple as the directory layout of the source tree. In this case, module names are defined as directory names. Source files under the same directory are labeled by the same module name. In addition to source tree layout, software build configurations can further improve the quality of the strategy. For example, if objects compiled from source files under directories `A`, `B`, and `C` are linked together into a library called *libM* during the build process, then users can choose to group all files (including the headers) under directories `A`, `B`, and `C` in a module called `M` in the file-level modularization strategy.

Both source tree directory layouts and software build configurations are shipped with source code by default and can be processed by simple text processing scripts. This means that if users want to group symbols at the scale of source files, they usually do not need to construct modularization strategies from scratch, Since the supposed users of this tool are software developers themselves, obtaining a modularization strategy for their own software should be easier than composing regular expressions or grammars, which are required by light-weight techniques used by some other analysis tools (see Chapter 2).

4.2 Proposed Analysis Metrics

In our two-level dependency design, the symbol-level dependency of a software is the set of all symbol references in the program. By grouping symbols into modules, the analysis tool can derive module-level dependencies from symbol references and abstract the dependencies as a directed weighted graph. Although the symbol-level dependencies extracted by our tool can be fed to some existing relational calculators such as Grok [30] and CrocoPat [19] to get a variety of software metrics, we focus on module utilization in this work. There have been utilization-based metrics proposed on the level of class to describe program coupling and cohesion, e.g., Lack of Cohesion of Methods (LOCM) and Coupling between Object Classes (CBO) from Chidamber and Kemerer's metric suit for object oriented design [22], Class-Attribute (CA) and Class-Method (CM) interaction metrics by Briand et al. [20]. Since utilization-based metrics are intuitive and easy to understand, we choose to extend the utilization-based metrics to the module level. Similar to existing class-level metrics, module utilization describes the complexity of the dependencies from the developer's view by showing how many program elements are involved with respect to a particular dependency inspection scope. Especially, during the collaboration with Chromium developers we found that utilization metrics can well help developers decide where to start and estimate refactor cost when they want to compact the code base [50]. It is also possible to exploit our module-utilization metrics in a design flaw detection strategy which is not limited to particular metrics [34].

4.3 Metric Definitions

For each node in the dependency graph which represents a particular module of the software, our tool provides the size of the module. The size is defined as the total number of symbols created by the module, The analyzer then calculates the **overall utilization** of the module which is the percentage of symbols utilized by other modules. For each edge in graph representing a dependency from one module to another, the tool calculates the **pairwise utilization** between the two modules.

To honestly reflect module utilization, both overall and pairwise utilization metrics are calculated transitively *inside the target module*. For instance, if module A directly depends on symbol foo in module B and foo directly depends on symbol bar which is also in the module B, then bar is considered utilized by module A as well. Furthermore, if bar directly depends on another symbol baz in module B, then baz is also depended on by A.

Formally, given module A and B, denote by $\mathcal{U}$ the set of symbols in B utilized by A. $\mathcal{U}$ is the maximum subset of B satisfying that, for every symbol s in $\mathcal{U}$, (1) there exists a reference pair (k, s) such that k is a symbol in A; or (2) there exists a sequence of reference pairs $(k, t_1), (t_1, t_2), \cdots, (t_{n-1}, t_n), (t_n, s)$ such that k is a symbol in A and t_i $(1 \leq i \leq n)$ are symbols in B. The pairwise utilization from A to B is calculated as the size of $\mathcal{U}$ divided by the size of B. Note that no third module is involved in this calculation.

With this definition, the utilization not only reflects the coupling but also provides some cohesion information of the modules. If the pairwise utilization from a client module to a target module is low, it can suggest that the utilized part of the target module is loosely connected to the rest of it.

There may be concerns that module-level transitive dependencies should also be considered. For example, if symbols in A depends on symbols B and those symbols in B further depends on symbols in C, then A should be considered to depend on C as well (A,

B, and C are all modules). Rationales of only considering transitive dependencies *inside the target modules* include:

- In large software development, every developer has different expertise. It is very unlikely that a developer is involved in the development of many modules, even though these modules are related. If the dependency analysis chooses to take transitive dependencies across modules into consideration when calculating pairwise utilization, the developers may get confused by information which they may not want to pay attention to.

- There is also the option to provide both transitive and non-transitive dependencies at the module level. However, transitive dependencies can be extremely complicated. For example, there are 238 modules in Chromium and 86 of them are strongly connected (any module can reach another, including itself, via transitive dependencies). Determining how to process dependencies among these strongly connected modules is non-trivial and requires extensive research effort, which is beyond the scope of this thesis.

Similar considerations also exist in previous researches. Reddy et al. introduced a similar metric *Dcoupling* (degree of coupling) for software artifacts in the dependency graph [42]. In their research, an artifact can be classes, sets of classes, and subsystems etc., and the Dcoupling metrics indicates the *direct* utilization between adjacent artifacts. However, since they focus on design defects rather than internal implementation details, their utilization is only about the *interfaces* of an artifact and no transitivity inside an artifact is considered. Our utilization metrics, on the other hand, try to cover both design and implementation, so transitive dependencies inside modules are taken into consideration.

Calculating utilization at the symbol level rather than class level is because that C++ is not a fully object-oriented programming language. There is a chance that a considerable portion of the symbols in a project does not belong to any class. Also, numbers of class members can vary significantly (from fewer than 10 to over 200 in Chromium) among different classes. Chapter 6.2 provides quantitative evidence which shows the benefits of symbol-level utilization over class-level utilization in terms of accuracy. Another consideration of choosing symbol-level utilization is that it can be converted to other kinds of utilization smoothly, e.g., file-level utilization and lines-of-code utilization, under certain modularization strategies.

Chapter 5

Implementation

This chapter describes the implementation details of the dependency analyzer, including an overview of the major parts of the tool and why they can achieve the proposed scalability.

5.1 IR Analyzer

As explained in chapter 3.2, the tool consumes the LLVM intermediate representation of the programs as the analysis input. The tool utilizes Clang to generate the binary object of IR instructions for each source file. The binary object is named as LLVM *bitcode*. The IR analyzer is implemented as an LLVM pass, which scans each bitcode object separately. After the analyzer completes the scanning, it outputs the following information for each source file:

- All symbol definitions and the corresponding debug information,
- All symbol reference pairs (sym_a, sym_b) for every sym_a defined in the object,
- Class hierarchies declared in the source code,
- Virtual function table layout of each class, and
- All virtual function call tuples $(sym, class, vf\_idx)$, where sym is the caller, $class$ is the base class to which the called virtual function belongs, and vf_idx is the index of the callee in the virtual function table.

```
%vtable9 = load void (%class.SkCanvas*)*** %4; // virtual function table
%vfn10 = getelementptr inbounds void (%class.SkCanvas*)** %vtable9, i64 6;
                                               // index function pointer
%5 = load void (%class.SkCanvas*)** %vfn10;    // load function
call void %5(%class.SkCanvas* %3);             // call function
```

Figure 5.1: Virtual function call in LLVM IR[1]

The idea for extracting symbol reference pairs is straightforward. Every LLVM IR instruction consists of an operation and several operands. The IR analyzer traverses the instructions in every function definition and static variable initializer. Then it inspects if any of the operands is a global symbol.

The rest of the items in the output are for virtual function related analysis. The DWARF debug data embedded in the IR can be used for class hierarchy extraction. As for virtual function tables, in C++ they are implemented as arrays of function pointers, so the layout can be obtained by regarding the table definition as a static initializer. Finally, since Clang compiles virtual function calls into a certain IR sequence, the $(sym, class, vf\_idx)$ tuples can be extracted via peephole recognition, i.e., searching for all the occurrences of that particular sequence. An example of the virtual function call sequence in the IR is shown in Figure 5.1.

5.2 Post-Analyzer

After processing all the bitcode objects of a software project, the output will be fed to the post-analyzer. The post-analyzer need to integrate the symbol-level dependency facts and then construct module-level dependencies.

5.2.1 Generating Virtual Symbol References

The post-analyzer will first construct symbol reference pairs for base class virtual function calls according to the virtual function call tuples and the layouts of virtual function tables. After that the post-analyzer will construct reference pairs for derived class virtual functions

[1]Detailed semantics of the IR code snippet can be interpreted with the aid of LLVM IR reference documentation: http://llvm.org/docs/LangRef.html

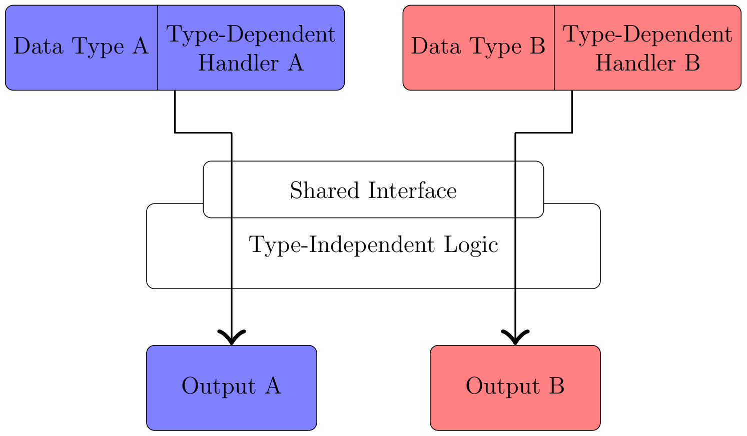


Figure 5.2: Semantic abstraction of templates and higher-order functions

using Class Hierarchy Analysis (CHA) which is discussed in Chapter 3.4. Pure virtual functions are excluded in this step because there are no corresponding symbol definitions.

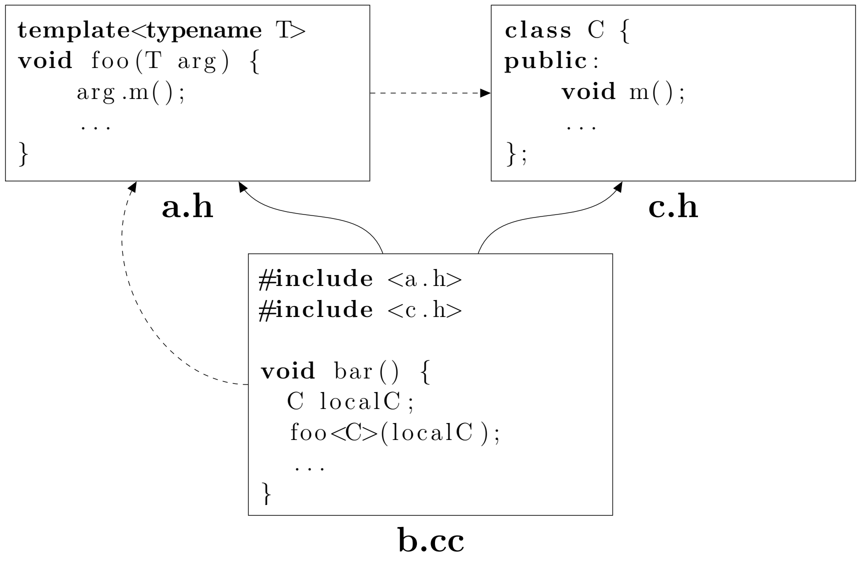
5.2.2 Handling Templates

Template-related symbol reference pairs require extra manipulation by the post-analyzer. Given a template, a programmer needs to fill in it with one or more type arguments and obtain an instance of the template before using it as a normal function or class. From the programmer's view, the usage of template has no significant semantic difference from another polymorphic programming paradigm – the higher-order functions which take other functions as inputs, such as the `qsort` function[2] offered by the standard C library. *The logic behind template and higher-order function programming paradigms can be generalized in Figure 5.2,* which will be explained later in this section.

The `qsort` function is a polymorphic implementation of the quick sort algorithm [35]. It is a high-order function which takes another routine as input. That routine, when the programmer invokes `qsort`, is provided as the `compar` parameter, which is essentially the type-dependent handler in Figure 5.2. The `qsort` function itself is mapped to the shared interface and type-independent logic.

Correspondingly, a template case is illustrated in Figure 5.3. In this example, the template `foo` can be regarded as the shared interface and type-independent logic part

[2] **void** qsort (**void**∗ base, size_t num, size_t size, **int** (∗compar)(**const void**∗,**const void**∗));



```
template<typename T>
void foo(T arg) {
    arg.m();
    ...
}
```

```
class C {
public:
    void m();
    ...
};
```

```
#include <a.h>
#include <c.h>

void bar() {
  C localC;
  foo<C>(localC);
   ...
}
```

Figure 5.3: A template-related dependency example

in Figure 5.2, while class `C` and the interface `C.m` can be viewed as the type-dependent handlers. Semantics of the arrows in Figure 5.3 will be explained later.

Although there is such a semantic match from the template case to the `qsort` case, the compiler has different views for these two programming paradigms. In the `qsort` case, suppose that a function `my_sort` calls `qsort` and uses the subroutine `my_compare` as the last argument for `qsort`. Among these program elements, `qsort` can be mapped to the shared interface and type-independent part in Figure 5.2, while `my_compare` can be viewed as the type-dependent handler. There are two symbol reference pairs involved: `(my_sort, qsort)` and `(my_sort, my_compare)`, because symbol `my_compare` appears in the definition of `my_sort` rather than `qsort`. To sum up, in the `qsort` case *the user routine depends on both the shared interface and the type-dependent handler.*

In the template case, however, the situation is different. Still take Figure 5.3 as the example. In `b.cc`, function `bar` instantiates `foo<C>` from template `foo` with type `C` as the argument. The symbol references indicate two dependencies (denoted by dashed lines): (1) `bar` depends on template instance `foo<C>`; (2) `foo<C>`, whose template definition is in `a.h`, depends on `C.m`. Different from the `qsort` case, in the template example *the user routine only depends on the shared interface and the shared interface depends on the type-dependent handler provided by the user.* This result is against the programmers' intuition. When the prototype of `C.m` is changed, it is usually the function `bar` in `b.cc` that should be changed accordingly. The author of `bar` may subclass `C`, add a new interface compatible with `foo`, or just re-implement `bar` to make the program buildable and functional. But he or she should not touch the template definition in `a.h` since it is possibly used by other modules out of his or her control.

To make the analysis results for these two semantically alike programming patterns consistent, the tool will adjust the template-related dependencies to the form denoted by the solid lines in Figure 5.3, namely the symbol reference pairs extracted will be `(bar, foo<C>)` and `(bar, C.m)`.

5.2.3 Resolving Symbol Linkage

After all the symbol reference pairs are extracted and generated, the final step for symbol-level dependency analysis will be resolving symbol linkage. For any symbol reference pair (sym_a, sym_b) extracted from a bitcode object, sym_a and sym_b are just symbol names. To obtain the complete dependency information, the analyzer needs to know in which source files sym_a and sym_b are defined. Looking for symbol definitions according to symbol names is not a trivial task, because C++ allows plural definitions for the same symbol name. To avoid conflicts, symbols can be defined with different visibility and linkage types. For example, a function denoted by keyword **static** is only visible in the file where it is defined, and this allows different function definitions with the same name outside that file. The C++ application binary interface (ABI) defines a set of rules for resolving symbol visibility and linkage, which linkers need to implement correctly.

The LLVM tool chain supports bitcode linking, but it does not scale to millions of lines of C/C++ code. Our experiment of linking all bitcode objects of the Chromium browser finally failed after taking over 120GB memory and 24 hours on a 32-core server. This procedure is resource consuming because the linker has to resolve address relocation when merging separate objects into a single binary. The linker also need to fulfill function inlining requests, which requires some heuristics and thus costs both time and memory.

As a consequence, the post-analyzer needs to implement a light-weight linker. For dependency analysis, the only task of the linker is to find the correct definitions of symbols in reference pairs according to symbol names. Other operations performed by the original LLVM bitcode linker (address relocation, function inlining, etc.), which are unnecessary in dependency analysis, can be safely omitted. LLVM provides data structures to represent linkage types for every symbol in the bitcode. By following the Itantium C++ ABI standard [8], a light-weight linker can finish the linking based on the original linking configuration of a software project and saves time and memory significantly.

5.2.4 Calculating Module Size

With the linking process is over, all symbol-level dependencies are constructed. To obtain the module-level dependencies, the post-analyzer will first group symbols into modules and compute the utilization metrics introduced in Chapter 4.2. In this phase an adjustment for module size calculation is implemented. As defined in Chapter 4.2, the size of a module is the number of symbols defined in the module. For templates, multiple symbols can be instantiated by providing different type parameters. In large C++ projects, it is common that tens of instances are derived from a single template. Due to this fact, the raw count of symbols in the program is against the intuition of the programmers, because they do not directly write definitions for these different symbols. From the developer's view, the only program element they manage is the template definition. To make the metrics in accordance with the developers' intuition, symbols instantiated from the same template are counted only once in the implementation of this tool.

Chapter 6

Evaluation

This chapter evaluates the dependency analysis tool by applying it to two real-world C++ projects. The experiments show that the tool is efficient enough for practical deployment. In addition to the performance data, this chapter also gives some preliminary findings about the maintainability state of Chromium, one of the projects used in the evaluation. The findings are based on the dependency analysis results and the information from Chromium developers.

6.1 Performance

Two popular C++ projects are used for performance evaluation: the Boost library and the Chromium browser. The experiments are performed on a 3.1GHz Core i5 machine with 8GB memory. Table 6.1 shows the details of the experiments. One point to note is that some of the reference pairs are inferred by the tool based on Class Hierarchy Analysis (see Chapter 3.4). Although this is a flexible part affected by the inherent nature of individual software projects, we believe that the Chromium browser is typical enough to represent projects with complicated virtual function design and usage. The timing results proved that the tool is able to scale to millions of lines of C++ code, even on a desktop machine.

Moreover, since both Boost and Chromium are projects with years of active development, it proves that the tool has good compatibility with legacy C++ source code. This benefits from the fact that the analyzer is designed to work with the production-quality compiler Clang and the LLVM infrastructure. As stated in Chapter 3.2, Clang is a fast-evolving open source project which keeps providing support for latest C++ standard.

	Boost	Chromium
Lines of Source	384,000+	9,000,000+
# of Symbols	22,872	408,406
# of Reference Pairs	29,235	1,487,540
# of CHA References	990	187,803
# of Modules	48	238
Time (minute)	13	123
Memory (MB)	696	5,734

Table 6.1: Performance evaluation details

Convinced by that, the dependency analysis tool built upon Clang and LLVM is expected to be compatible with future C++ software projects.

6.2 Symbol-Level Utilization Accuracy

Some existing techniques analyze dependency for Java programs at the granularity of class [39, 51], namely they regard class as the smallest program element in the analysis. For C++ programs, however, class-level utilization does not accurately reflect module coupling. The reason is that C++ is not a fully object-oriented programming language. In Java programs, every symbol must be defined in a class. On the contrary, C++ programmers can and often do define standalone symbols that do not belong to any class. As a result, class-level utilization analysis for C++ programs has a risk of omitting important information about program structures.

Among the two projects evaluated for tool performance, Boost is a pure C++ project without C code. Table 6.2 displays the statistics of standalone symbols (symbols not belonging to any class) defined in Boost. There are two rows in this table which represent two different methods of calculating the number of symbols. In the first row (the original result) symbols are counted directly, while in the second row (the compact result) symbols instantiated from the same template are counted once, as described in Chapter 5.2.4. No matter which method is considered, symbols defined outside of classes occupy a significant portion in the project, suggesting that omitting standalone symbols may introduce inaccuracy in dependency analysis.

To compare the symbol- and class-level module utilization for Boost, we derive the class reference pairs from symbol reference pairs. For every symbol reference pair extracted from Boost, symbols in the tuple are replaced with classes to which the symbols belongs. If either

	# of Standalone Symbols	# of All Symbols	Standalone Ratio (%)
Original Result	14,955	22,872	65.4
Compact Result	1,711	3,855	44.4

Table 6.2: Standalone symbols in Boost

of the symbols in the tuple is a standalone symbol, the reference pair is discarded. After obtaining the class reference pairs, we calculate the class-level utilization in a similar way to that described in Chapter 4.2. When grouping classes into modules, a file-level modularization strategy based on source tree layout is exploited. Under this strategy, classes *declared* in files under the same directory are grouped into one module. We choose this strategy because C++ programmers can *define* class members in different files. However, since these files are usually kept in the same directory with the header file where the class is declared, a directory-based modularization strategy can avoid the situation where a class is grouped into different modules.

With the class-level utilization is ready, the symbol-level utilization is calculated also using the source tree layout modularization strategy. Utilization is represented in the form of directed weighted graphs, where nodes represent modules and edges represent pairwise utilization between modules. Because both dependency graphs use the same modularization strategy, they have the same nodes. Furthermore, since the class reference pairs are directly derived from symbol reference pairs, the two graphs will likely have the same edges (with different weights) when using the directory-based modularization strategy. For Boost this is true.

After getting the dependency graphs with symbol- and class-level utilization respectively, we calculate the absolute difference between the weights of corresponding edges. Given all the absolute differences, we further calculate their average and standard deviation. Recall that the analysis tool provides two different views in the results (Chapter 3.4). For the naive analysis results, the average and standard deviation are 0.098 and 0.121; for utilization based on Class Hierarchy Analysis, we get the average of 0.104 and standard deviation of 0.124. Since the range of pairwise utilization is from 0 to 1 inclusive, these statistics suggest a significant difference between symbol-level and class-level utilization. Since class-level utilization analysis omits the standalone symbols, it can be concluded that symbol-level analysis achieves far better accuracy in terms of describing the structure of Boost.

6.3 Preliminary Findings on Chromium

Based on the analysis results, several dependencies suspicious of hurting software maintainability are identified in Chromium. This section reveals the two kinds of bad dependencies introduced in Chapter 1, i.e., underutilized dependencies and inconsistent dependencies.

6.3.1 Underutilized Dependencies

The analysis shows that some modules in Chromium have low overall utilization. Although most of them are third-party modules included by Chromium, some others are Chromium's internal modules. This indicates that a considerable part of these internal modules are dead code in the worst case. This may not be true because Chromium is an advanced web browser supporting plugins. Some of the unused symbols can be provided for plugins, thus should not be treated as dead code. However, these underutilized dependencies should still draw attention from the developers, because they may suggest better module partition as explained in Chapter 1. Table 6.3 lists some of the non-third-party modules with less than 20% overall utilization in Chromium. The threshold is set to be 20% based on the suggestions from Chromium developers.

Module	**# of Symbols**	**Overall Util**[†]
notifier	181	4.4∼17.1%
dbus	334	18.9∼18.9%
remoting_jingle_glue	97	12.4∼19.6%

[†] *The range shows the impact of virtual function calls.*

Table 6.3: Partial list of underutilized modules in Chromium

6.3.2 Potential Inconsistent Dependencies

With the help of Chromium developers, it is also possible to manually identify inconsistent dependencies according to analysis data. An investigation on module "*base*" in Chromium shows that there are some dependencies potentially violating the design principle. Module "*base*" contains common code shared by all sub-projects of Chromium. According to the design, "*base*" should not depend on any other modules; however, it turns out that "*base*" depends on eight modules even without considering potential virtual function calls. Among the eight modules, two are marked acceptable exceptions to the design after the result is reviewed by the developers. The remaining six dependencies point to third-party libraries, which are potentially inconsistent with the design of “*base*”.

Chapter 7

Conclusion

With the size and complexity of the software both increasing, managing large-scale software projects becomes more and more challenging. As a basic and important methodology for revealing code base status, dependency analysis can help developers get a better understanding of the potential threats to software maintainability. This chapter concludes the thesis, which introduced the design and implementation of a scalable and accurate dependency analysis tool for large C++ software with millions of lines of code.

7.1 Contribution

The design and implementation of the analysis tool makes the following contributions:

- **Scalable and accurate dependency extraction and analysis:** The analyzer provides fine-grained and accurate results. It completes the dependency extraction and analysis for the Chromium project with over 9 million lines of code in 123 minutes on a 3.1GHz Core i5 machine, showing that the tool is scalable and efficient.
- **Full C++ Support:** The tool can process almost all salient C++ features, e.g., template and operator overloading. In addition, the analysis can handle some non-standard features supported by the compiler, which are widely used in large-scale C++ software development.
- **Potential bad dependencies detected in :** By applying the dependency analysis to the Chromium, a popular open source web browser, the analysis tool found some

underutilized modules, of which only less than 20% of the symbols are utilized by the other modules, meaning a considerable portion of the modules may be dead code. In addition to the underutilized dependencies, the analysis result also revealed several dependencies potentially violating software design.

7.2 Limitations and Future Work

Despite the satisfactory scalability and accuracy, the tool has the following limitations:

- Currently the tool can only cover one particular configuration of the analyzed software in the result. Source code irrelevant to that configuration will be excluded.
- Virtual function dependencies extracted are not as accurate as the rest of the results.
- The tool relies on an external toolset, i.e., the Clang compiler, limiting the environments in which the tool can be deployed. For instance, software projects created by Microsoft Visual Studio are usually not fully parsable by Clang.
- The tool cannot analyze inline assembly code, because it is difficult to translate a lower level representation (assembly code) to a higher level one (LLVM IR), and Clang does not support such translations. Although inline assembly support is not mandatory for C/C++ compilers according to the ISO standard, it is a widely used feature in large C/C++ projects, especially operating systems, browsers, and virtual machines.
- Many forms of behavioral dependencies are out of consideration, e.g., call sites with arbitrary function pointers and dependencies introduced by IPC mechanisms.

To address these problems, a plan for future improvements of the tool is proposed:

- Combine analysis results of multiple configurations together. To achieve this, the tool needs to resolve possible conflicts introduced by different configurations. Build optimization strategies are also necessary, because different configurations can overlap a lot. Since compilation time occupies a considerable portion of the overall analysis time, repetitive compilation can be highly expensive. Some researchers have already implemented scalable parsers to generate variability-aware abstract syntax trees for C/C++ programs with `ifdef` pre-processor directives [31, 32]. It may be possible to extend these techniques and apply them in our tool.

- Evaluate the accuracy of the current virtual function call analysis and improve it with other techniques if necessary. This can be challenging because it is difficult to obtain the ground truth of virtual function calls for software projects with millions of lines of code, making the evaluation work non-trivial. If it is shown that the virtual function dependency results indeed require more accuracy, more research can be performed on how to achieve this.

- Add support for more C++ front ends. At present Clang developers have already started working on supporting Microsoft Visual Studio syntax and linker. Some other compiler infrastructures are also adding support to translate their own intermediate representations to LLVM IR [13]. By supporting more front ends, it is even feasible to support programming languages other than C++. There have already been projects which try to compile Java [16], Python [11, 12], or Scala [14] programs into LLVM IR.

- Add a special component for the analysis tool to process assembly code. Because the only information necessary for dependency analysis is symbol references, it is possible to segregate assembly code from the other program elements and extract symbol references via light-weight techniques such as pattern matching.

- Explore other techniques to include behavioral dependencies in the results. Potentially, arbitrary function pointers can be analyzed by points-to analysis algorithms. However, extracting IPC-related dependencies may need more innovative research.

References

[1] Android Binder. http://elinux.org/Android_Binder.

[2] CodeSurfer. http://www.grammatech.com/research/technologies/codesurfer.

[3] Doxygen. http://www.stack.nl/~dimitri/doxygen/.

[4] The DWARF debugging standard. http://www.dwarfstd.org/.

[5] Eclipse CDT. http://www.eclipse.org/cdt/.

[6] Edison design group. https://www.edg.com/index.php?location=c_frontend.

[7] Frontendart source code analyzer toolset. http://www.frontendart.com/product/source-code-analyzer-toolset.

[8] Itanium C++ ABI. http://refspecs.linux-foundation.org/cxxabi-1.83.html.

[9] Lattix for C/C++. http://lattix.com/g_cpp.

[10] The LLVM compiler infrastructure. http://llvm.org.

[11] NUMBA. http://numba.pydata.org/.

[12] py2llvm. https://code.google.com/p/py2llvm/.

[13] ROSE user manual - ROSE compiler infrastructure. http://rosecompiler.org/ROSE_UserManual/ROSE-UserManual.pdf.

[14] Scala on LLVM. http://greedy.github.io/scala-llvm/.

[15] Understand your code. http://www.scitools.com/.

[16] VMKit: A substrate for virtual machines. http://vmkit.llvm.org/.

[17] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.

[18] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM.

[19] Dirk Beyer. Relational programming with CrocoPat. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 807–810, New York, NY, USA, 2006. ACM.

[20] Lionel Briand, Prem Devanbu, and Walcelio Melo. An investigation into coupling measures for C++. In *Proceedings of the 19th international conference on Software engineering*, ICSE '97, pages 412–421, New York, NY, USA, 1997. ACM.

[21] Trosky B. Callo Arias, Pieter Spek, and Paris Avgeriou. A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering*, 16(5):544–586, October 2011.

[22] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[23] Brad Cossette and Robert J. Walker. DSketch: Lightweight, adaptable dependency analysis. In *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 297–306, New York, NY, USA, 2010. ACM.

[24] Ian J. Davis, Michael W. Godfrey, Richard C. Holt, Serge Mankovskii, and Nick Minchenko. Analyzing assembler to eliminate dead functions: An industrial experience. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 467–470, Washington, DC, USA, 2012. IEEE Computer Society.

[25] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag.

[26] Thomas R. Dean, Andrew J. Malton, and Ric Holt. Union schemas as a basis for a C++ extractor. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 59–, Washington, DC, USA, 2001. IEEE Computer Society.

[27] Jens Dietrich, Vyacheslav Yakovlev, Catherine McCartin, Graham Jenson, and Manfred Duchrow. Cluster analysis of Java dependency graphs. In *Proceedings of the 4th ACM symposium on Software visualization*, SoftVis '08, pages 91–94, New York, NY, USA, 2008. ACM.

[28] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.

[29] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 254–263, New York, NY, USA, 2001. ACM.

[30] Ric Holt. Introduction to the Grok language. http://plg.uwaterloo.ca/~holt/papers/grok-intro.html, 2002.

[31] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 805–824, New York, NY, October 2011.

[32] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE-13, New York, NY, August 2013.

[33] Kathleen A. Lindlan, Janice Cuny, Allen D. Malony, Sameer Shende, Forschungszentrum Juelich, Reid Rivenburgh, Craig Rasmussen, and Bernd Mohr. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[34] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society.

[35] M. D. McIlroy. A killer adversary for quicksort. *Softw. Pract. Exper.*, 29(4):341–344, April 1999.

[36] Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. Improving program slicing with dynamic points-to data. *SIGSOFT Softw. Eng. Notes*, 27(6):71–80, November 2002.

[37] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 66–72, New York, NY, USA, 2001. ACM.

[38] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of 8th Working Conference on Reverse Engineering*, WCRE'01, pages 13–22, 2001.

[39] J. David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali. Searching for build debt: Experiences managing technical debt at google. In *Proceedings of the Third International Workshop on Managing Technical Debt*, MTD '12, pages 1–6, 2012.

[40] Gail C. Murphy. *Lightweight structural summarization as an aid to software evolution.* PhD thesis, University of Washington, 1996. AAI9704521.

[41] Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, July 1996.

[42] K. Reddy Reddy and A. Ananda Rao. Dependency oriented complexity metrics to detect rippling related design defects. *SIGSOFT Softw. Eng. Notes*, 34(4):1–7, July 2009.

[43] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 167–176, New York, NY, USA, 2005. ACM.

[44] Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 264–274, New York, NY, USA, 2012. ACM.

[45] Sameer Shende, Allen D. Malony, Craig Rasmussen, and Matthew Sottile. A performance interface for component-based applications. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 278.2–, Washington, DC, USA, 2003. IEEE Computer Society.

[46] Mark Shtern and Vassilios Tzerpos. Clustering methodologies for software engineering. *Advances in Software Engineering*, 2012:1:1–1:1, January 2012.

[47] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM.

[48] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[49] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 99–108, New York, NY, USA, 2001. ACM.

[50] Pei Wang, Jinqiu Yang, Lin Tan, Robert Kroeger, and J. David Morgenthaler. Generating precise dependencies for large software. In *Proceedings of the Forth International Workshop on Managing Technical Debt*, 2013.

[51] S. Wong and Yuanfang Cai. Predicting change impact from logical models. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, pages 467–470, ICSM '09.

[52] Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 411–420, New York, NY, USA, 2011. ACM.

[53] Jingwei Wu. *Open source software evolution and its dynamics.* PhD thesis, University of Waterloo, 2006.

[54] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.