Generating Precise Dependencies for Large Software

Pei Wang, Jinqiu Yang, Lin Tan
University of Waterloo
Waterloo, ON, Canada
{p56wang,j233yang,lintan}@uwaterloo.ca

Robert Kroeger
Google Inc.
Kitchener, ON, Canada
rjkroege@google.com

J. David Morgenthaler
Google Inc.
Mountain View, CA, USA
jdm@google.com

Abstract—**Intra- and inter-module dependencies can be a significant source of technical debt in the long-term software development, especially for large software with millions of lines of code. This paper designs and implements a precise and scalable tool that extracts code dependencies and their utilization for large C/C++ software projects. The tool extracts both symbol-level and module-level dependencies of a software system and identifies potential underutilized and inconsistent dependencies. Such information points to potential refactoring opportunities and help developers perform large-scale refactoring tasks.**

Index Terms—**dependency, large scale, technical debt**

I. 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 tens or 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. Developers constantly join and depart from the software development. Due to the complexity and fast evolution of software, the coupling between modules can deviate from the original design, which hurts software maintainability. A recent study shows that inter-module dependencies can be a significant source of technical debt in long-term software development [7].

To manage such technical debt, a first and key step is to provide developers with a precise view of the code dependencies among modules. The benefits 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. We elaborate on these two benefits below.

Identifying Bad Dependencies: Two main types of bad dependencies are *underutilized* dependencies and *inconsistent* dependencies. If only a small portion of a target module is utilized by a client module, we call such a dependency from the client module to the target module an *underutilized dependency*. Underutilized dependencies often slow down the building process and blow up the code size [7]. 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, thus should be separated to be a standalone module.

Inconsistent dependencies are dependencies that violate the design of the software. 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), which hurt the long-term maintainability of the project, thus causing technical debt.

Facilitating Large-Scale Refactoring: As software grows and evolves, some large-scale refactoring becomes 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. 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. While the detailed plan of this fix is still under investigation, comments in several Bugzilla 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.

We design and build a technique to generate code dependencies precisely and automatically. For each dependency, we also provide the pairwise utilization and overall utilization of each target to help developers decide whether to adjust the dependency or refactor the target. To the best of our knowledge, our tool is the *first* one that scales to generate precise dependencies and identify bad dependencies for millions of lines of *C/C*++ code bases.

We make the following contributions:

- **Scalable and precise dependency extraction and analysis:** Our tool provides fine-grained and precise results. It completes the dependency extraction and analysis for the Chromium project with 6 million lines of code (6 MLOC) in 123 minutes on a 3.1GHz Core i5 machine, showing that the tool is scalable and efficient (Section V).

- **Full C++ Support:** Our 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 (Section II).
- **Potential bad dependencies detected in Chromium:** By applying our analysis to the Chromium browser, we 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. We also identified several dependencies potentially inconsistent with the software design (Section V).

II. Design Overview

Informally we define *dependency* as that, when the interface of module A is modified, if the content of module B should be modified accordingly to keep the whole project buildable, then there is a dependency from B to A. This definition makes dependencies discussed in this paper fall into the category of structural dependencies [3].

To obtain precise module-level dependencies, we build module-level dependencies from symbol-level dependencies. For example, symbol `func_foo` depends on symbol `func_bar` if `func_foo` calls `func_bar`. If `func_foo` is part of module A and `func_bar` is part of module B, then A depends on B. Similar to the previous work [7], we define the *pairwise utilization* from module A to B as the proportion of symbols in B on which A depends. The *overall utilization* of a module B is the proportion of symbols in B on which all other modules in the project depends. We rank the dependencies by pairwise utilization and rank modules by overall utilization when reporting them.

Figure 1 visualizes part of our analysis result for the Chromium browser, showing the dependencies between some key components of the browser. In the graph, each node is a module and each edge denotes a pairwise dependency.

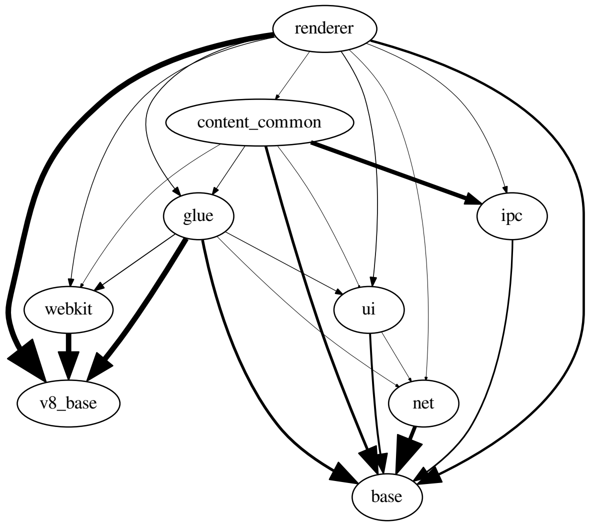


Fig. 1. Dependencies between key components of Chromium. The thickness of the edges indicates the pairwise utilization.

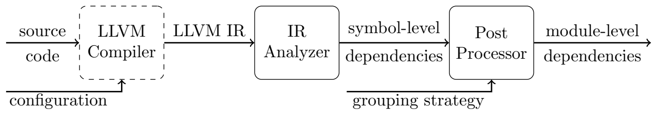


Fig. 2. Design overview. The component with dotted lines is existing facility; and the components with solid lines are implemented by us.

Figure 2 shows the three main steps of our technique:

1) **Compilation.** Compile the source files into LLVM Intermediate Representation (IR) with debug information.
2) **IR Analyzer.** Extract the symbol-level dependencies from the LLVM IR instructions.
3) **Post Processor.** Group the symbol-level dependencies to construct the module-level dependencies. The grouping strategy, i.e., which symbols belong to which module, can be based on any information reflecting the system's structure (e.g., code directory layout and build files).

III. Symbol-level Dependency Extraction

A. Basic Approach

In this section, we summarize the difficulties in performing dependency analysis on C++ programs and explain why we choose LLVM IR rather than source code or abstract syntax tree (AST) as the abstraction level for our tool to work on.

1) Function Overloading and Default Parameters: Without type checking, it is nearly impossible for tools based on pure pattern-matching to match overloaded functions correctly in the source code, and the default parameters of overloaded functions further aggravate the difficulty.

2) Non-Standard Language Syntax: Production compilers usually support non-standard language syntaxes, some of which can alter symbol linkage or set link-time alias, and it is hard for pattern-matching techniques to recognize all of them. We confirmed such a real-world case that shows the impact of non-standard syntaxes on dependency analysis. In the Chromium project, a function in the *content_browser* module is assigned a link-time alias, overriding a function in the standard library and making every module calling that standard function depends on *content_browser*.

3) Implicit Call Sites: In C++, many functions (e.g., copy-constructors and overloaded operators) are called without using the conventional "`()`" operator. Several other kinds of call sites, e.g., default object construction and destruction, do not even show up in the source code.

4) Templates: Template instances are not directly defined in the source code but are instantiated separately at compile time. The instantiation information is usually not available in the AST [5].

For the challenges above, we decide to go further than AST-based tools and choose the LLVM IR as the abstraction level to work on. LLVM IR is close to native code but still keeps rich source code information. It is a language-independent format, which gives tools built upon them the potential to support languages other than C and C++.

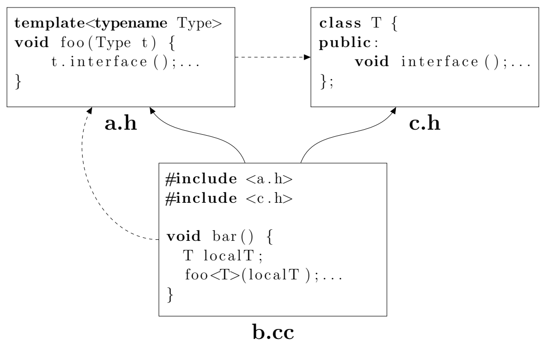
B. Extraction Details

We obtain the symbol references by traversing LLVM IR instructions. Since LLVM IR is a close-to-native format, the instructions include implicit call sites, all template instances, and linkage aliases. In IR, symbol names are mangled, saving us the effort to distinguish overloaded function calls.

However, even with well formatted abstraction, there are still challenges to address:

1) Template-Related Dependencies: Template is a special feature in C++ which provides compile-time polymorphism for different types sharing common interfaces. A programmer needs to fill in the template with one or more types to obtain an instance of the template before using it as a normal function or class. This flexibility, however, causes dependencies that do not match our dependency definition in Section II.

We illustrate the problem in Figure 3. In `b.cc`, function `bar` instantiates `foo<T>` from the function template `foo` with type `T`. The symbol references indicate two dependencies (denoted by dashed lines): (1) `bar` depends on the template instance `foo<T>`; (2) `foo<T>`, whose template definition is in `a.h`, depends on `T.interface`. The first dependency matches our dependency definition, but the second one does not, because if the prototype of `T.interface` is changed, it is usually the function `bar` in `b.cc` that should be changed accordingly. The developer of `bar` may subclass `T`, add a new interface compatible with `foo`, or just reimplement `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. Based on this scenario, we adjust the template-related dependencies to the form denoted by the solid lines in Figure 3.



```
template<typename Type>
void foo(Type t) {
    t.interface();...
}
```

```
class T {
public:
    void interface();...
};
```

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

void bar() {
  T localT;
  foo<T>(localT);...
}
```

Fig. 3. Template-related dependency. The dashed dependencies are indicated by symbol references. The solid ones are expected dependencies from the view of software design (`bar` depends on `foo<T>` and `T.interface`).

2) Virtual Function Calls: Typically, an analysis tool, that focuses on structural dependencies rather than behavioral dependencies, should not be bothered by virtual function calls. However, unlike dependency extraction tools for Java, most of which perform analysis at the class level, our analysis handles symbol-level dependencies. For that reason, if we ignore virtual function calls, many classes will get close-to-zero overall utilization, which may be too far from the truth. Currently, we circumvent this problem by reporting two extreme results. Firstly we get the conservative result by assuming only virtual functions in the base classes are called. Then we get the aggressive result by propagating virtual function calls in the conservative result to every derived class.

IV. Module-Level Dependency Analysis

With the symbol-level dependency information extracted, we further construct the module-level dependency model by linking and grouping the symbols. Our tool connects separate pieces of symbol-level dependency data together through a mock linking process. We resolve the symbol linkages according to the Itanium C++ Application Binary Interface. After the linking, we calculate the pairwise utilization of every module-level dependency and the overall utilization of every module to identify potential bad dependencies.

The pairwise utilization is used to measure the dependency between two modules quantitatively. For example, module A directly uses some symbols defined in module B. We first get the dependency transitive closure of the directly dependent symbols in module B (If `f` calls `g` and `g` calls `h`, then `h` is in `f`'s dependency transitive closure). Then we calculate the total number of symbols in module B by merging the symbols derived from the same templates. The pairwise utilization will be the size of the transitive closure divided by the size of the target module B.

The overall utilization represents the overall usage of a module within the project. For example, a module with 100% overall utilization means that every symbol in this module is utilized at least once by other modules in the project directly or indirectly.

V. Evaluation

We evaluate our tool on a popular open source web browser—the Chromium project (svn revision 171054, 6 million lines of C/C++ code). The scale of the analysis is presented in Table I. Our tool can finish the analysis of this scale in 123 minutes (88 minutes' compilation time and 35 minutes' analysis time) on a 3.1GHz Core i5 machine. All tasks performed in the experiment are single-threaded. The peak memory usage during the analysis phase is 5.6 GB. Since there is no other tool, to the best of our knowledge, handling the same or similar analysis on large-scale C/C++ projects, we are not able to set any opponent for our results. However, we believe that our tool is eligible for practical deployment encouraged by the absolute numbers reported above.

TABLE I
Analysis details of Chromium

# of Symbols	470,797
# of Symbol References	13,912,651
# of Modules	238

Underutilized dependencies: Our analysis shows that some modules in Chromium have low utilization (<20%). Although most of them are third-party modules included by Chromium, some others are Chromium's internal modules. This indicates that 80% or more of these internal modules are dead code in the worst case (for Chromium, some unused symbols can be provided for plugins, thus should not be treated as dead code). Table II lists some of the non-third-party modules with less than 20% overall utilization in Chromium.

TABLE II
PARTIAL LIST OF UNDERUTILIZED MODULES IN CHROMIUM

Module	# of Symbols	Overall Util[†]
notifier	181	4.4∼17.1%
ppapi_cpp_objects	1195	17.5∼17.6%
dbus	334	18.9∼18.9%
ppapi_ipc	3228	19.4∼19.4%
remoting_jingle_glue	97	12.4∼19.6%

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

Potential Inconsistent dependencies: We investigate whether there is any dependency violation to the design in Chromium. We started with the "*base*" module, which contains common code shared by all sub-projects of Chromium. According to the design, "*base*" should not depend on any other modules; however, we found that "*base*" depends on eight modules even without considering potential virtual function calls.

Among the eight modules, there are two marked by the developers as acceptable exceptions to the design after we showed them the results. In the future, we would like to add a whitelist to allow developers to specify such acceptable exceptions. Nonetheless, it is beneficial for developers in the team to be aware of such exceptions. The remaining six dependencies point to third-party libraries, which are potentially inconsistent with the design of "*base*".

VI. RELATED WORK

A recent study describes the build debt problem at Google and builds the Clipper tool to detect underutilized dependencies [7]. While Clipper analyzes dependencies at the class level, this paper analyzes dependencies at the more fine-grained symbol-level. Since a class can contain a large number of methods, e.g., the `RenderViewImpl` class in Chromium contains more than 200 methods, our symbol-level analysis is more precise in detecting underutilized dependencies. In addition, we analyze C/C++ projects instead of Java projects, which has its unique challenges such as implicit call sites, linkage resolution, and template analysis.

Murphy [8] proposed a lexical dependency extraction technique. Lexical techniques are typically lightweight and efficient, but inaccurate. The quality of such analysis depends heavily on patterns specified by the users of the tools.

Syntactical approaches are more flexible and can tackle a wider range of dependency-related problems than lexical ones. Modern IDEs such as Eclipse CDT [1] can provide rich dependency information to assist basic code refactoring by traversing the AST. DSketch [4] utilizes the island grammar technique [6] to excavate dependencies from polylingual systems based on fuzzy patterns, and focuses on interactions between different programming languages (e.g., calling SQL from Java). Neither CDT nor DSketch reveals the inter- or intra-module dependencies of large-scale software projects.

In addition to code-based facts, researchers also exploit software release histories to further explore software dependencies and potential design violations [11], [12].

Many papers assume dependencies are given and focus on dependency analysis. Bouwers et al. [2] build the dependency profiles for the systems and quantify the encapsulation and independency of the modules. Lattix Inc's dependency manager [9] helps manage the architecture of Java projects by leveraging design structure matrices (DSM) [10], a known metric about modularity.

Commercial tools *CodeSurfer*[1] and *CppDepend*[2] can extract structural and behavioral dependencies for C/C++ projects, but neither provides utilization-driven dependency analysis according to the publicly available information.

VII. CONCLUSIONS AND FUTURE WORK

This paper presents a new tool for extracting fine-grained dependencies from large-scale C/C++ software systems with millions of lines of code. Our tool performs utilization analysis at the module level to prioritize bad dependencies. We believe that our analysis can be instructive to various kinds of software enhancement and maintenance activities. We implemented the tool on top of LLVM and evaluated the performance on the Chromium project with 6 MLOC. The results show that our tool is efficient enough for practical deployment. In the future, we plan to apply the tool to real-world large software refactoring problems.

ACKNOWLEDGMENT

We thank Sanjay Bhansali, Robert Bowdidge, and Stephanie Van Dyk for their early discussion and investigation. The work is partially supported by the National Science and Engineering Research Council of Canada and a Google gift grant.

REFERENCES

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

[2] E. Bouwers, A. van Deursen, and J. Visser. Dependency profiles for software architecture evaluations. ICSM'11.

[3] T. B. Callo Arias, P. van der Spek, and P. Avgeriou. A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering*, 2011.

[4] B. Cossette and R. J. Walker. DSketch: Lightweight, adaptable dependency analysis. FSE'10.

[5] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, F. Juelich, R. Rivenburgh, C. Rasmussen, and B. Mohr. A tool framework for static and dynamic analysis of object-oriented software with templates. CDROM'2000.

[6] L. Moonen. Generating robust parsers using island grammars. WCRE'01.

[7] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali. Searching for build debt: Experiences managing technical debt at google. MTD'12.

[8] G. C. Murphy. *Lightweight structural summarization as an aid to software evolution*. PhD thesis, 1996. AAI9704521.

[9] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. OOPSLA'05.

[10] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. ESEC/FSE'01. ACM.

[11] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. ICSE'11.

[12] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *TSE*, June 2005.

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