Comprehensive Analysis of C++ Applications using the libClang API

Stephen Schaub and Brian A. Malloy
School of Computing
Clemson University
Clemson, SC 29634, USA
{sschaub,malloy}@clemson.edu

Abstract

The C++ language has lagged behind in tool development due to ambiguities inherent in the grammar that defines the language. This phenomenon has induced some developers to resort to a partial parse of C++, omitting language constructs deemed irrelevant to the particular tool under development. In this paper, we describe a design and implementation that exploits an API provided by the Clang compiler to build a tool that uses a full parse of the C++ language. We evaluate two tools built with partial parsers and compare their results to those obtained using our tool built with a complete parse of a C++ application. We validate our results and highlight some dangers of software evaluation using a partial parse approach.

1 Introduction

Identifying the quality of software systems is one of the most important and time consuming aspects of the software development life cycle [1, 4, 13]. In order to reduce the computational burden of the developer in localizing potentially faulty parts of a program, many developers utilize software metrics, which are easier and less time consuming to compute than other evaluative approaches such as testing and fault localization [1, 11]. However, software metric computation is contingent upon cooperating analysis tools and parsers to enable the developer to automatically evaluate the application. Fortunately, there is an ample array of tools to support analysis and parsing of applications written in some languages, such as Java and Python, and both Java and Python include APIs to facilitate parsing as part of the language [15, 17].

However, there is a deficiency of parsing and analysis tools for applications written in C++, even though the C++ language has been shown to be the best performing programming language on the market [5, 7]. This deficiency derives from the complexity of the grammar that defines the language, and the difficulty in developing parsers for C++ has induced many developers to resort to alternative techniques such as *island grammars* and ad-hoc parsing, which attempt to analyze certain relevant language constructs and omit all other constructs [2, 12]. Thus, most analysis tools available for C++ use a partial parse. However, there is a danger in partial analysis of a program and the accuracy and completeness of program evaluation may depend directly on the quality of the adopted parsing technique. Moreover, there are no studies in the literature that evaluate the effects of analyses derived from a partial parse as compared with a complete parse of a C++ application.

In this paper, we describe our use of an interface provided by the Clang compiler to perform a complete parse of a C++ program, thereby enabling comprehensive analysis of any C++ application acceptable to the Clang front-end [3]. Clang is an open source compiler for C, Objective-C, and C++, and is used by Google, Apple, and other software developers. The Clang compiler has become notorious for its efficient compilation and lucid error messages.

To evaluate the comprehensive analysis provided by a complete parse of a program, we apply a long-standing popular software metric, McCabe's Cyclomatic Complexity [13], to several open source applications. To validate the correctness of our analysis, we compare the results of our metric computation to two other tools that perform the same metric computation using an island grammar and an ad-hoc parsing approach. The results of our experiments illustrate the advantages of a comprehensive approach to parsing, and highlight some of the dangers in using a partial parse of a program to draw conclusions about the quality of a software system.

In the next section we provide background, and in Section 3 we review the important features of Clang. One of the drawbacks of using libClang is the dearth of available documentation to aid developers in using the Python bindings. A comprehensive description of the libClang interface is beyond the scope of this paper. Nevertheless, we provide an overview of the libClang interface in Section 4, and in Section 5 we detail our

use of the interface to compute cyclomatic complexity. In Section 6 we list the results of our experiments, and we review related research in Section 7. Finally, in Section 8 we draw conclusions.

2 Background

In this section we provide background information about the tools and terms used in this paper. We first review McCabe's Cyclomatic Complexity metric, and we then describe various approaches to parsing, highlighting two approaches used to deal with ambiguous grammars, and corresponding tools built with the partial parse approach that we use as a basis of comparison to our full parse approach.

2.1 Cyclomatic Complexity

McCabe's *Cyclomatic Complexity* metric was developed in an effort to improve on the ubiquitous *lines of code* metric (LOC) [13]. McCabe observed that a program containing fifty assignment statements would be evaluated by LOC as equivalently complex as a program containing fifty decision statements. Instead, cyclomatic complexity measures the number of linearly independent paths through a *flow graph* [13].

For structured programs, cyclomatic complexity is computed for each subroutine, and complexity is the number of branching structures plus 1, since all flow graphs in a subroutine have at least one path. For the binary search example in Figure 1, consisting of a **while** and two **if** statements, the *Cyclomatic Complexity* is 4. However, as we show in Section 6, various tools make different assumptions about switch statements, ternary operators, and short-circuit evaluation, producing different cyclomatic complexity.

2.2 Parsing Approaches

We reference three parsing approaches in this paper: *full parsing*, *parsing with island grammars*, and *ad-hoc parsing*. In this section, we survey each approach and present a tool that is based upon it.

Traditional parsers perform a full parse of the program source, accepting the input if the source text conforms to the language grammar, and rejecting the input otherwise. This approach is used by the Clang parser on which our tool is based.

In contrast to a full parse, an island grammar approach does not examine the complete source, but rather examines portions of the source ("islands") that are relevant to the problem under study, ignoring the rest of the code. A parser utilizing this approach can be significantly simpler than a full language parser because it can avoid the complexities of the full language. However, island grammar parsers can omit language

```
1  typedef unsigned int size_t;
2
3  template<typename T>
4  int search(T item, T items[], size_t size) {
5    int low, hi;
6    bool found = false;
7    low = 0;
8    hi = size - 1;
9
10   while (hi > low && !found) {
11     size_t mid = (hi + low) / 2;
12     if (items[mid] != item) {
13       if (item < items[mid])
14         hi = mid - 1;
15       else
16         low = mid + 1;
17     } else {
18       found = true;
19     }
20   }
21 }
```

Figure 1: Binary Search Test Case. This program provides a base line test case that should produce consistent results for the three tools that we evaluate.

constructs that contain code segments relevant to the analysis under study. Moreover, the complexity of the C++ language presents challenges even to island grammar parsers, as we show in our evaluation of the tools in Section 6.

The CCCC tool evaluated in Section 6 uses an island grammar approach. CCCC computes a number of metrics, including cyclomatic complexity, for programs written in C, C++, and Java [12]. Based on the PCCTS parsing tool, CCCC computes cyclomatic complexity by counting branch keywords and operators, including **if**, **while**, **for**, **break**, **return**, **switch**, `&&`, `||`, and the ternary operator. CCCC counts **break** and **return** keywords rather than **case** (counted by the CCM tool discussed below) because **switch** statements can have multiple consecutive cases terminated by a single break or return, and these cases represent a single independent path that should add only 1 to the cyclomatic complexity [11]. Both approaches to dealing with **switch** statements provide an estimate of cyclomatic complexity.

Finally, an ad-hoc parser, like an island grammar parser, is often designed to analyze only portions of a program, and has advantages and disadvantages similar to the island grammar approach. The distinction between these two techniques lies in the implementation strategy: an island grammar uses parser tools to generate portions of the parser; whereas an ad-hoc parser is hand-crafted using strategies appropriate to the designer. The CCM tool, evaluated in Section 6, is written in C# and uses an ad-hoc approach to compute cyclomatic complexity for programs written in

C#, C, C++, and JavaScript, and counts the following branching keywords and operators: **if**, **while**, **for**, **case**, **catch**, `&&`, and `||` [2].

3 Overview of Clang

Clang provides three sets of programming interfaces that developers can use to perform source code analysis and transformation: *libClang*, *libTooling*, and the *Clang AST*. In this section, we provide a brief overview of these interfaces.

The most mature and stable of the three interfaces is *libClang*. libClang is a C-language API that provides a cursor-like interface to the Abstract Syntax Tree (AST) built by the Clang parser, as well as support for building code completion mechanisms for IDE's. It exposes a subset of the information available in the AST that is sufficiently rich to compute the cyclomatic complexity metric. A Python binding for libClang is provided as part of the standard Clang distribution, and we provide more detail about this API in Section 4.

The *libTooling* API is a newer C++-language interface to Clang that focuses on convenience, reducing the amount of boilerplate code that developers must write to perform common analysis tasks. It is less stable than libClang, and prone to backwards-incompatibilty changes between Clang releases.

The *Clang AST* API, like libTooling, is a newer, less stable C++-language interface to Clang. It provides a visitor interface to the Clang AST that exposes more AST information than libClang [6].

4 The libClang Interface

The libClang API provides a C-language API for accessing the AST of a parsed source file. In this section, we introduce the Python bindings to libClang, which consist of the following classes:

- *Index* is the entry point to the Python API. It holds some global state and provides methods to parse source files and read pre-parsed AST files.
- *TranslationUnit* represents a parsed source file.
- *Cursor* represents a node in the AST.

Parsing a C++ source file involves invoking the static create() method in the Index class to obtain an instance of Index, and then invoking the parse() method on the Index instance to parse the desired source file. The parse() method also accepts arguments to specify options such as include paths and preprocessor symbol definitions. On successful completion, parse() returns an instance of TranslationUnit. parse() can be invoked repeatedly if a program consists of multiple source files.

Queries for information from a source file begin with a TranslationUnit instance, which represents a parsed source file, and is typically obtained from an Index instance using its parse() method. A TranslationUnit instance has a cursor property that yields a Cursor instance representing the root AST node for the parsed file. The cursor property provides the root for functions which traverse the AST to extract information or perform source transformations.

The Cursor class is the heart of the Python Clang API. Each node in a parsed AST has a separate Cursor instance associated with it which exposes information about the node through various methods and properties. Here are some of the more significant ones:

- The *kind* property returns an instance of CursorKind indicating the type of node. CursorKind is a class with a fixed set of enumerated instances indicating node types, such as CursorKind.DO_STMT for a do statement node and CursorKind.CXX_METHOD for a method.
- The *spelling* property gives the text of nodes representing entity declarations, e.g., a variable name for a variable definition node, or a function name for a function definition node.
- The *get_tokens()* method returns an iterator that provides access to the source tokens that comprise the node.
- The *is_definition()* method returns True if the node represents a definition of an entity, such as a variable or method.
- The *get_children()* method returns an iterator for traversing the children of the node.

5 Application: Metric Computation

To illustrate the Python libClang bindings, we now provide an overview of our application that computes cyclomatic complexity using a full parse of the application. One of the drawbacks of using libClang is the dearth of available documentation to aid developers in using the Python bindings. The previous section and our overview in this section will partially address this drawback, but a comprehensive description of the libClang interface is beyond the scope of this paper.

The cyclomatic complexity computation begins by creating an Index instance and using its parse() method to parse a file provided to the script as a command line argument, illustrated in Figure 2, lines 27-28. The resulting top-level cursor is provided to the function find_subs() as the starting point for traversing the AST, shown in Figure 2, line 29.

Function find_subs() recursively performs a breadth-first search of the AST, scanning for function

```
1  import sys
2  import clang.cindex
3  from clang.cindex import CursorKind
4
5  keywordOpList = ["if","while","for","case"];
6
7  def compute_branches(node):
8      branch_count = 0
9      for tok in node.get_tokens():
10         if tok.spelling in keywordOpList:
11             branch_count += 1
12     return branch_count
13
14 def find_subs(node):
15     if (node.is_definition() and
16         node.kind in [
17           CursorKind.FUNCTION_TEMPLATE,
18           CursorKind.FUNCTION_DECL]):
19         branch_count = compute_branches(node)
20         print '%s: %d' % (node.spelling,
21           branch_count + 1)
22     else:
23         for c in node.get_children():
24             find_subs(c)
25
26 if __name__ == '__main__':
27     index = clang.cindex.Index.create()
28     tu = index.parse(sys.argv[1])
29     find_subs(tu.cursor)
```

Figure 2: McCabe Summary. This figure summarizes our use of the libClang Python bindings to compute cyclomatic complexity using a full parse of the input.

definitions. Lines 15-18 check the current node to determine if it is either a template or regular function definition and, if it is, line 19 computes the cyclomatic complexity using compute_branches(). If the node is not a function, lines 23-24 search the node's children for function definitions.

Function compute_branches() computes the complexity for a function by returning the number of branches that it detects. It works by iterating over the tokens in the function, incrementing the branch count each time it encounters a token that signals a branch.

To simplify our presentation in Figure 2, we have elided processing of class methods. However, our cyclomatic complexity computation can handle any function in a syntactically correct C++ program accepted by Clang, and the class methods in the test cases (Section 6) were processed correctly by our tool.

6 Results

In this section, we provide the results of our experiments. In the next section, we describe our testing methodology for comparing our tool that uses a full parse to compute cyclomatic complexity, to the two tools that we reviewed in Section 2 that use a partial parse approach. In Section 6.2 we compare the results for the three tools, and in Section 6.3 we discuss the advantages and disadvantages of the three approaches.

	Subroutines	Common	Different
Clang	141		
CCCC	76	69	1
CCM	141	140	8

(a) Test case TinyJS

	Subroutines	Common	Different
Clang	3032		
CCCC	2283	2160	20
CCM	3020	2530	80

(b) Test case CLucene

Figure 3: Results for two test cases.

6.1 Methodology

All of the experiments that we report in this section were computed on a system using the Ubuntu Linux 12.04 operating system, and our tool is implemented using version 3.4 of the Clang compiler. We computed cyclomatic complexity for three C++ programs: a binary search function listed in Figure 1; TinyJS, an open source JavaScript interpreter composed of 700 lines of code (LOC); and CLucene, an open source text-based search engine consisting of 33,000 LOC. Since CCCC and CCM compute cyclomatic complexity by counting different sets of keywords (see Section 2.2), we were required to modify our tool to match each of the two comparison tools. This modification was easily achieved with our tool and allowed us to be fair and unbiased in our comparison. Our approach also allowed us to focus our analysis on the functions where the complexity metric reported by our tool differed from the two comparison tools, knowing that such differences would not be due to the set of keywords used in computing the metric.

When comparing the results, we found that overloaded subroutines were problematic because CCCC and CCM use different formats to report overloaded subroutines, which occur in C++ when two or more subroutines have the same name and are distinguished by their argument lists. Therefore, in our comparison results, we list only the uniquely named subroutines.

6.2 Tool Comparison

All three tools produced the same result for the binary search in Figure 1, which forms a baseline and starting point for our comparison. However, there were important differences in the results for the other two test cases, illustrated in Figure 3, where the *Subroutines* column lists the number of non-overloaded

subroutines reported by the respective tool for each test case; the *Common* column lists the number of subroutines identified by CCCC and CCM whose names matched those reported by our tool; and the *Different* column reports the number of *Common* subroutines where the respective tool reported a complexity number different from our tool. For example, the second row of Figure 3b shows that the CCCC tool reported a metric for 2,283 non-overloaded subroutines; of those, 2,160 matched subroutines identified by our tool, and of those 2,160 matches, all but 20, (Different), had identical complexity metric values.

The most significant difference relates to the number of non-overloaded subroutines identified by each tool. As shown in the first row of Figure 3a, our tool identified 141 non-overloaded subroutines for TinyJS, as did CCM, shown in the third row of Figure 3a; however, CCCC identified only 76 subroutines, shown in the second row of Figure 3a, due to difficulties in parsing the TinyJS source.

The results in Figure 3b show that, for CLucene, our tool reported more subroutines than both CCCC and CCM due to parsing difficulties those tools encountered during processing.

The second difference relates to the correct identification of the subroutines. Both CCM and CCCC had parsing difficulties that led to inaccurate reporting of the names of some of the subroutines. For example, CCM identified the same 141 TinyJS subroutines as our tool, but misreported the name of one due to a parsing problem. In the CLucene test, of the 3,020 non-overloaded subroutines CCM identified, only 2,530 were reported with names that matched those identified by our tool. Although we did not perform an exhaustive analysis of the 490 subroutines identified by CCM that failed to match those in our list, our survey of the results indicated that many of these CCM naming errors involved operator overloading and nested classes.

Finally, we report the number of subroutines where our approach yielded a complexity metric that differed from that yielded by the comparison tool. Since we had adjusted our tool to count the same branching tokens as the corresponding reference tool, the differences were minimal, as expected. In the case of TinyJS, we carefully analyzed each subroutine where the metric differed to determine the reason for the difference. For the single TinyJS subroutine where CCCC yielded a different metric, the CCCC logs indicated a parsing difficulty that explained the difference. Of the 8 subroutines where CCM gave a different result, we found the difference was due to factors related to conditional compilation and the handling of Boolean operators. Regarding the former, CCM excludes portions of subroutines that appear within **#ifdef** regions, while our implementation includes them. Regarding the latter, CCM counts `&&` and `||` operators as contributing to complexity only when they appear within a branching statement such as **if** or **while**, while our algorithm counts all occurrences of these operators regardless of the context. Thus, the differences with CCM are due solely to interpretational considerations of the cyclomatic complexity metric.

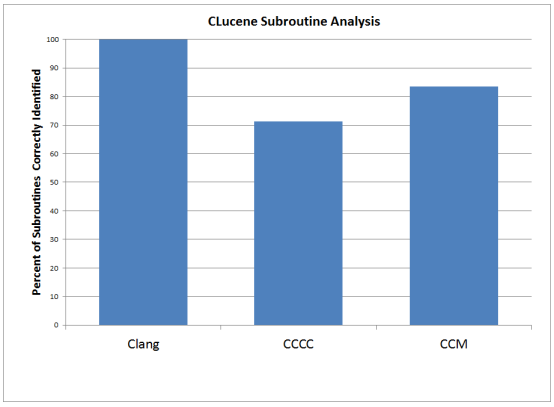

Figure 4: Percentage of subroutines correctly reported by Clang, CCCC, and CCM, in the CLucene test case.

6.3 Discussion

The results presented in the previous section show that our approach to computing McCabe's complexity metric yields accurate results, given our close agreement with two independent tools' computation of the metric on the small TinyJS code base, and our analysis of the few differences.

Our approach has the following advantages. First, it provides a more comprehensive analysis, successfully reporting on more subroutines than either of the competing approaches represented by CCCC (island grammar) and CCM (ad-hoc parser). Due to our use of libClang, our tool can successfully analyze virtually all subroutines in any code base that can be compiled with Clang; the alternative approaches missed or incorrectly identified up to 50% of the subroutines in a code base, or report inaccurate metric numbers due to parsing difficulties. An an illustration, Figure 4 shows the percentage of subroutines identified by our tool whose names were correctly reported by CCM and CCCC in the CLucene test case; both tools had significant parsing difficulties that had an impact on the scope and accuracy of their results.

Second, the implementation is reasonably straightforward. Our complete analysis program is less than 150 lines of Python code, much of which relates to logging results and comments. Since the heavy lifting of parsing C++ is done by Clang, our implementation can focus on the work of computing the cyclomatic complexity metric. Our algorithm, substantially similar to the streamlined version pre-

sented in Figure 2, is thus easy to understand and customize.

Finally, the implementation is flexible, easily tailored to count branch tokens used by CCCC and CCM.

7 Related Work

The difficulty of parsing C++ is well-studied [8, 10, 16, 18], and this difficulty has led to alternative approaches to parsing such as *fuzzy parsing* [9] and *island grammars* [14], which perform analysis on selected portions of the input rather than performing a detailed analysis of a complete source text. These approaches contrast with our use of Clang, which provides a full parse of the input along with the construction of an accompanying AST. We used this full parse to provide a precise computation of the cyclomatic complexity metric for the programs that we study.

In [6], we used the Clang AST API to perform static analysis of C++ programs using a Clang AST visitor. Our current paper describes an alternative approach, using the Python bindings for the libClang API instead of the Clang AST API, which yields a simpler implementation.

8 Concluding Remarks

In this paper, we have demonstrated the utility of the libClang API, provided by the Clang C++ compiler, for creating tools that produce accurate static analysis of C++ applications. We provided an overview of the Clang API that can facilitate its use by other developers, and we described our implementation of a tool that uses libClang to compute McCabe's cyclomatic complexity metric. We chose McCabe's metric because it can be computed without symbol table construction or lookup, since symbol table construction cannot be accomplished using a partial parse of the application under development.

We compared the results of our complete parse approach to those achieved by two open-source C++ analyzers that use a partial parse of the program, and we described some dangers of using a partial parse to build evaluation tools. For example, the tools built with a partial parse missed or incorrectly identified up to 50% of the subroutines in a code base, or report inaccurate metric numbers due to difficulties partially parsing the C++ language.

References

[1] B. Baudry, F. Fleurey, and Y. L. Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th ICSE*, pages 82–92, 2006.

[2] J. Blunck. Cyclomatic Complexity Analyzer, 2012. `http://www.blunck.info/ccm.html`.

[3] Clang. A C language family frontend for LLVM, [accessed 27-August-2013]. `"http://clang.llvm.org"`.

[4] B. C. Dean, W. B. Pressly, B. A. Malloy, and A. A. Whitley. A linear programming approach for automated localization of multiple faults. In *Proceedings of the 2009 IEEE/ACM International ASE*, pages 640–644, USA, 2009.

[5] D. du Preez. C++ clear winner in google language tests, 2011. `http://www.computing.co.uk/ctg/news/2076322/-winner-google-language-tests`.

[6] E. B. Duffy, B. A. Malloy, and S. Schaub. Exploiting the Clang AST for analysis of C++ applications. *Proceedings of the 52nd Annual ACM Southeast Conference*, March 2014.

[7] M. Klimek. The clang ast, August 2013. `http://clang.llvm.org/docs/IntroductionToTheClangAST.html`.

[8] G. Knapen, B. Lague, M. Dagenais, and E. Merlo. Parsing C++ despite missing declarations. In *7th IWPC*, Pittsburgh, PA, USA, May 5-7 1999.

[9] R. Koppler. A systematic approach to fuzzy parsing. *Software Practice and Experience*, 1996.

[10] J. Lilley. PCCTS-based LL(1) C++ parser: Design and theory of operation. February 1997.

[11] T. Littlefair. *An Investigation into the Use of Software Code Metrics in the Industrial Software Development Environment*. PhD thesis, Edith Cowan University, 2001.

[12] T. Littlefair. C and C++ Code Counter, 2005. `http://cccc.sourceforge.net/`.

[13] T. J. McCabe. A complexity measure. *IEEE TSE*, 2(4):308–320, December 1976.

[14] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th WCRE*, pages 13–22. IEEE Computer Society Press, 2001.

[15] Oracle. Annotation processing tool. Oracle Java Technology, 2014.

[16] J. F. Power and B. A. Malloy. A metrics suite for grammar-based software. *Journal of Software Maintenance*, 16(6):405–426, 2004.

[17] Python. parser – access python parse trees. The Python Standard Library, 2014.

[18] H. Sutter. C++ conformance roundup. *C/C++ User's Journal*, 19(4):3–17, April 2001.