Appeared in the 52^{nd} Annual ACM Southeast Conference, March 28–29, 2014

Exploiting the Clang AST for Analysis of C++ Applications

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

ABSTRACT

In this paper, we describe our methodology for leveraging the Clang parser front-end to address the problem of tool development. We describe our approach and demonstrate its utility by computing some common metrics for three version releases of a popular, open source video game, *Aleph One*, a continuation of *Marathon*, a game produced by Bungie Software. We describe our results for applying two commonly used metrics to the three version releases and we describe an interesting contradiction in the two metrics when applied to successive version releases of Aleph One.

1. INTRODUCTION

The development of tools to facilitate comprehension, evaluation, testing, and debugging of software applications is contingent upon parsing and analysis tools for the language or languages used to implement the particular application under study. There is an extensive array of tools to facilitate parsing and analysis of applications written in Java, Python, C, and for most of the languages used to develop current and legacy systems. For example, both Java and Python support parsing of the respective languages using the native parser, and both include APIs to facilitate this parsing [26, 29]. In addition, there are a plethora of parsers and APIs currently available for both of these languages, including a Java parser included in Eclipse, with parallel development of analysis tools [32, 30]. In fact, the Java grammar is so amenable to parsing that many researchers build their own parsers and accompanying tools [2, 23, 3].

However, there is a serious shortage of parsing tools and concomitant analysis tools for applications written in the C++ language, even though the C++ language has been shown to be a clear winner in the Google language tests [8, 15]. This lack of tool development for the C++ language derives from the complexity of the grammar that defines the language and the corresponding difficulty in building parser and front-end analysis tools for C++ [16, 21, 27, 28, 31]. There have been considerable research efforts expended in attempts to develop a full parser for the C++ language, including the use of the GNU compiler collection, but these efforts have been daunted by the difficulty in parsing C++, or the difficulty in accessing the *gcc* parser internals [1, 10, 11, 18, 19, 28]. The main difficulty with *gcc* is that the parser front-end is not amenable to extension or modification, and its integration into new or existing projects has been described by one *gcc* developer as "Trying to make the hippo dance [14]."

In this paper, we describe an approach to exploiting the Clang parser and front-end for developing tools to facilitate comprehension, evaluation, testing, and debugging of software applications written in C++. The goal of the Clang project is to offer an alternative to the *gcc* compiler suite [1], including an abundance of analysis tools and, more importantly, access to the Clang abstract syntax tree (AST), and other parser internals using consumers and visitors [6, 9]. To demonstrate the utility of our approach, we describe our use of the Clang AST to compute two commonly used metrics, *measuring lines of code* (LOC), and McCabe's cyclomatic complexity measure, and we apply these metrics to three version releases of a popular open source video game, *Aleph One*, a continuation of *Marathon*, a game produced by Bungie Software [5]. Aleph One is an open source video game written in C++, and available under the GNU General Public License (GPL). Our study details some interesting results for the two metrics that we compute and we describe an important contradiction in the results for these metrics when applied to the three version releases of the Aleph One legacy video game. We provide these results to illustrate that our approach for using Clang is viable.

In the next section we provide background information about Clang, and the two metrics that we use in this paper. In Section 3 we provide an overview of the Clang parser front-end, and a high level view of our use of the Clang visitor and consumer as we use it to compute the two metrics. In Section 4 we provide details about our use of the Clang internals, including a C++ pseudo-code description of our algorithm for computing cyclomatic complexity. In Section 5 we list some of the results that we obtained for the two metrics, and in Section 6 we review some of the research that relates to our work. Finally, in Section 7 we draw some conclusions and describe future work.

2. BACKGROUND

In this section we provide background information about the tools, terms and concepts that we use in this paper. In the next section we describe *Clang*, the parser front-end that we use to process our C++ test cases. In Section 2.2 we describe LibTooling, which provides access to an AST representation of the test cases we study. Finally, in Section 2.3, we describe *lines of code* (LOC) and McCabe's cyclomatic complexity, the two metrics that we use to evaluate the history of complexity of our C++ test cases.

2.1 Clang

To compute metrics for C++ applications, we exploit *Clang*, an open source compiler front-end for the C, C++, Objective-C, and Objective-C++ programming languages [6]. The goal of the *Clang* project is to offer an alternative to the *gcc* compiler suite [1], including a plethora of analysis tools and, more importantly, access to the *Clang* internals using consumers and visitors [9]. Clang uses LLVM as its back end, and LLVM has been part of *Clang* since LLVM version 2.6 [20, 34]. Clang is used by several software developers including Google and Apple, and Apple has made extensive use of LLVM in a number of their commercial systems, including the iPhone development kit, and Xcode 3.1 [33].

In our previous research in reverse engineering and metrics computation, we attempted to augment an abstract syntax tree constructed with the *gcc* compiler suite [1, 10, 11, 18, 19, 28]; however, the *gcc* parser front-end is not amenable to extension or modification and its integration into new or existing projects has been been virtually impossible. On the other hand, the results that we report in this paper substantiate the decisions by Google, Apple, and other developers, to utilize the *Clang* and LLVM front- and back-ends to facilitate tool development. Moreover, Zadeck et al. have proposed the integration of LLVM and *gcc* [14], and the Ubuntu *Clang* implementation uses the *Clang* front-end together with the *gcc* back-end.

2.2 Clang LibTooling

LibTooling is a *Clang* C++ library that supports the construction of tools that leverage the *Clang* parsing front-end [7]. LibTooling can be used to extract both syntactic and semantic information about a program by permitting access to the Clang abstract syntax tree (AST), which contains both syntactic and semantic information about the program under analysis. Some of the tools included in LibTooling are *ast-print*, which builds an AST for the program being compiled and then prints the AST in readable format; *ast-dump*, which builds an AST and then dumps all of its contents; and *ast-dump-filter*, which uses ast-dump and ast-print to print only the selected AST declaration nodes.

2.3 Metrics

2.3.1 LOC

One frequently disparaged yet universally utilized metric for evaluating software complexity entails counting the number of lines of code (LOC) in the program [13]. An early detractor of LOC was McCabe, who observed that a program containing fifty assignment statements would be evaluated by LOC as equivalently complex as a program containing fifty decision statements [24]. Nevertheless, LOC remains the most frequently used measure of software complexity and even researchers who malign LOC frequently use it as a basis of comparison to their newly developed metric [12].

2.3.2 Cyclomatic Complexity

In an effort to improve on LOC for evaluating the complexity of software systems, McCabe developed a new metric called *Cyclomatic complexity*, which measures the number of linearly independent paths through a flow graph [24]. A *flow graph* is a set of vertexes and edges, where the vertexes are basic blocks, and the edges illustrate the flow of control from one basic block to the next basic block. A *basic block* is a sequence of statements such that the only entrance into the block is through the first statement, and the only exit from the block is through the last statement.

3. SYSTEM OVERVIEW

In Section 2.1 we described the *Clang* parser and front-end, as well as the LLVM back-end, used to compile many systems in use today. In the following section, we provide further details about *Clang* to provide a context for our extensions to *Clang* to permit our computation of complexity for the case study in Section 5. In Section 3.2, we provide an overview of the extensions that we incorporate into the *Clang* front-end.

3.1 The Clang/LLVM Framework

Figure 1 illustrates some of the classes and modules in the Clang parser front-end. The extensions that we show in the figure permit access to the Clang abstract syntax tree (AST), which is actually an extended AST that is augmented to include semantic information about names and language constructs used in the program under study. The torn paper icon on the left side of Figure 1 illustrates the C++ Source Code used as input to Clang. The dotted box in the middle of the figure illustrates the Clang front-end, and the dotted box to the right illustrates the LLVM back-end. Finally, the torn paper icon on the right side of the figure illustrates the Object Code generated by the Clang/LLVM compiler.

The classes that we show in the Clang front-end capture both syntactic and semantic information about the C++ source code. In particular, both ASTFrontendAction and ASTConsumer, shown as base classes in Figure 1, facilitate code generation, as illustrated by derived classes CodeGenAction, BackendConsumer, and CodeGenerator. Clang generates intermediate code that is then optimized by LLVM. The two base classes provide access to any developer interested in extracting syntactic and semantic information about the source program. In the next section, we show our approach for subclassing ASTFrontendAction and ASTConsumer to extract cyclomatic complexity.

For completeness, Figure 1 also includes information about the LLVM back-end, which generates object code for 21 different platforms that utilize the Clang/LLVM compilation suite. The right side of the figure illustrates an LLVM base class, LLVMTargetMachine, which is subclassed for each of the 21 target machines; we only show one such target in the figure, x86TargetMachine.

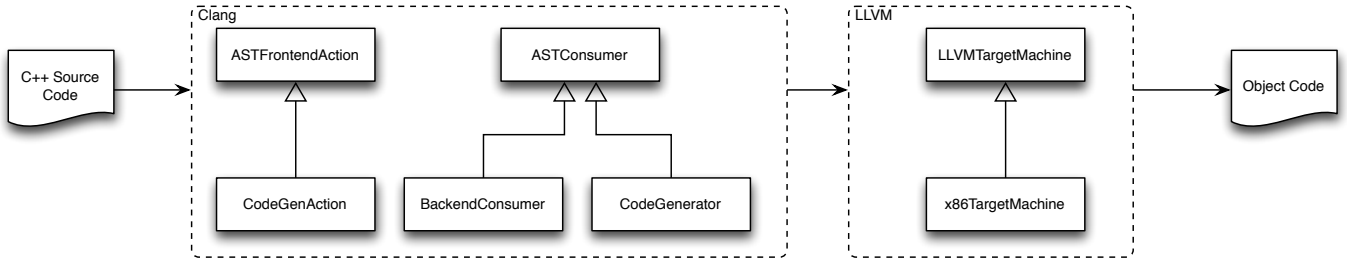


Figure 1: Clang/LLVM. *This figure summarizes important classes and modules in the Clang/LLVM compiler framework. The compiler accepts C++ source code as input, and generates object code for more than 21 different target platforms.*

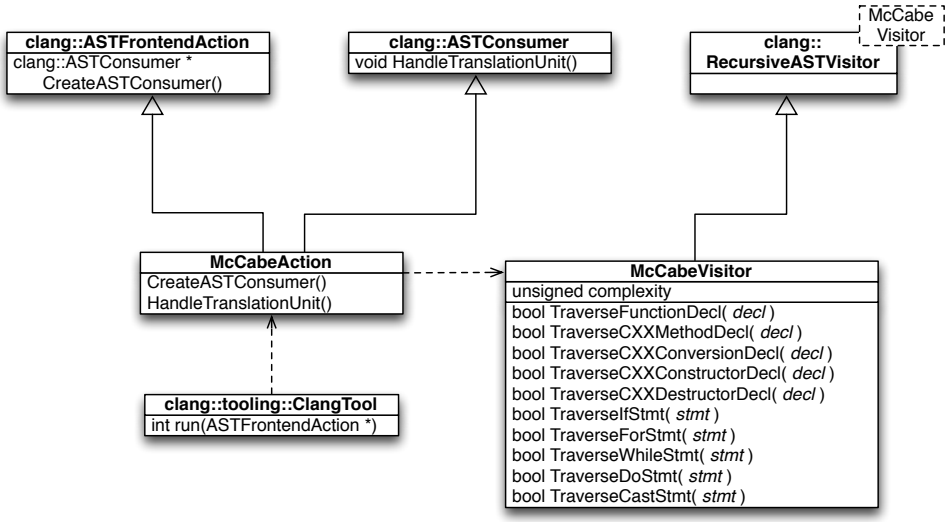


Figure 2: Extensions to Clang. *This figure illustrates the classes,* McCabeAction *and* McCabeVisitor*, that we added to the Clang framework for computing cyclomatic complexity.*

3.2 Our Extensions to Clang

Figure 2 summarizes our extension to the Clang framework. Recall that the three base classes in the figure are supplied by Clang, including clang::ASTFrontendAction, clang::ASTConsumer, and clang::RecursiveASTVisitor. The clang::RecursiveASTVisitor is a template class that we instantiate with the McCabeVisitor class that we developed to compute cyclomatic complexity. The McCabeVisitor class includes ten functions that we wrote to traverse the Clang AST to compute cyclomatic complexity. The first five classes, TraverseFunctionDecl, TraverseCXXMethodDecl, TraverseCXXConversionDecl, TraverseCXXConstructorDecl, and TraverseCXXDestructorDecl, are used to compute metrics for the five types of functions in a C++ application. The remaining five functions, TraverseIfStmt, TraverseForStmt, TraverseWhileStmt, TraverseDoStmt, and TraverseCaseStmt, are used to handle the five types of language constructs that induce an increase in cyclomatic complexity for the respective function.

We also build an additional class, McCabeAction, which we use to instantiate our McCabeVisitor class, and to pass McCabeVisitor to the template class clang::RecursiveASTVisitor.

4. METRICS COMPUTATION

In this section, we provide details about our methodology for using the Clang front end to compute cyclomatic complexity. Figure 3 illustrates the classes, and some of the actions, that we wrote to perform the computation. Lines 1–10 summarize class McCabeAction, which inherits from ASTFrontendAction and ASTConsumer, as described in Section 3, and further illustrated on lines 1 and 2 of Figure 3. The main purpose of McCabeAction is to instantiate and choreograph the McCabeVisitor class, which performs the actual computation to compute the complexity metric.

Lines 11–38 summarize class McCabeVisitor, and lines 39–42 summarize the main function in our approach. McCabeVisitor inherits from template class RecursiveASTVisitor, which we described in Section 3. Line 13 in the figure summarizes the constructor for McCabeVisitor, which initializes a pointer, context, to the translation unit under investigation. Lines 14–22 list nine methods, which illustrate the nine types of language constructs that we must investigate to compute cyclomatic complexity. Lines 26–30 summarize the actions of one of these methods in McCabeVisitor, TraverseWhileStmt, which computes cyclomatic complexity for

```
1  struct McCabeAction : public clang::ASTFrontendAction,
2                        public clang::ASTConsumber, {
3    McCabeAction() : clang::ASTFrontendAction(), clang::ASTConsumer
4                     visitor(new McCabeVisitor() { }
5    ~McCabeAction() { delete visitor; }
6    virtual void HandleTranslationUnit(clang::ASTContext &context) {
7          visitor->TraverseDecl(context.getTranslationUnitDecl);
8    }
9    McCabeVisitor* visitor;
10 }

11 class McCabeVisitor : public clang::RecursiveASTVisitor<McCabeVisitor> {
12 public:
13   McCabeVisitor(clang::ASTContext *context) { }
14   bool TraverseFunctionDecl(clang::FunctionDecl* decl);
15   bool TraverseCXXMethodDecl(clang::CXXMethodDecl* decl);
16   bool TraverseConstructorDecl(clang::CXXConstructor* decl);
17   bool TraverseDestructorDecl(clang::CXXDestructor* decl);
18   bool TraverseIfStmtDecl(clang::IfStmt* stmt);
19   bool TraverseCaseDecl(clang::CaseStmt* stmt);
20   bool TraverseWhileDecl(clang::WhileStmt* stmt);
21   bool TraverseDoDecl(clang::DoStmt* stmt);
22   bool TraverseForDecl(clang::ForStmt* stmt);
23 private:
24   unsigned complexity;
25 }

26 bool McCabeVisitor::TraverseWhileStmt(clang::WhileStmt *stmt) {
27   ++complexity;
28   if (stmt->getBody()) TraverseStmt(stmt->getBody();
29   return true;
30 }


31 bool McCabeVisitor::TraverseFunctionDecl(clang::FunctionDecl *decl) {
32   complexity = 1;
33   if (decl->getBody()) TraverseStmt(decl->getBody();
34   string qualifiedName = getQualifiedName(decl);
35   Report complexity for qualifiedName;
36   complexity = 0;
37   return true;
38 }

39 int main {
40   ClangTool tool;
41   return tool.run(newFrontendActionFactory<McCabeAction>());
42 }
```

Figure 3: McCabe Algorithm. *This figure illustrates the important steps, written in C++ pseudo-code, for using Clang to compute McCabe's complexity metric for a C++ application.*

while loops, which entails incrementing the current complexity by one, shown on line 27, and then traversing the body of the **while** loop, line 28, in search of other language constructs that might impact cyclomatic complexity, for example another **while** statement, or an **if** statement.

When a function or method terminates, lines 14–17, we report the cyclomatic complexity for that function and continue traversing the Clang AST in search of additional language constructs that might impact cyclomatic complexity.

5. RESULTS

In this section we report results that we obtained using the Clang parser front-end to compute LOC and cyclomatic complexity. Our test suite is composed of three versions of a popular, open-source, legacy video game, *Aleph One* [5], a continuation of *Marathon*, a game produced by Bungie Software. Aleph One is an open source video game written in C++, and available under the GNU General Public License (GPL). Aleph One is available for Mac OS X, Windows, and Linux, and also supports the other two games in the Marathon Trilogy: Marathon 2 and Marathon Infinity. Our experiments were executed on a Dell workstation using the Clang parser front-end running on Ubuntu 12.04.

Release	LOC	Number of Files
May 2003	42,733	387
Nov 2005	41,682	355
Apr 2007	44,019	375

Table 1: LOC metrics. *This table lists lines of user code (LOC) for each of three versions of Aleph One.*

In the next section, we describe results for computing *LOC* for three versions of Aleph One released on May of 2003, November of 2005, and April of 2007. In section 5.2 we describe results for computing *cyclomatic complexity* for the same three version releases of Aleph One.

5.1 Lines of Code, LOC

Table 1 lists the lines of code (LOC) for each of the three versions of Aleph One in our test suite. The first column of the table lists the date of the version release, the second column of the table lists the LOC in user code, and the final column lists the *Number of Files* in each of the three versions of Aleph One user code.

A cursory examination of the LOC and *Number of Files* columns might indicate that the code base for Aleph One decreased from the May 2003 release to the November 2005 release, and then increased again from the November 2005 release to the April 2007 release. This conclusion might follow from the fact that the LOC decreased from 42,733 to 41,682, and the *Number of Files* decreased from 387 to 355. However, we examined the code base for all three releases where we found that the May 2003 release of Aleph One contained the complete source code for the expat XML parser. However, the source code for expat was obtained from the system, and thus removed from the November 2005 release, which accounts for the reduction in LOC for the two releases from 2003 to 2005.

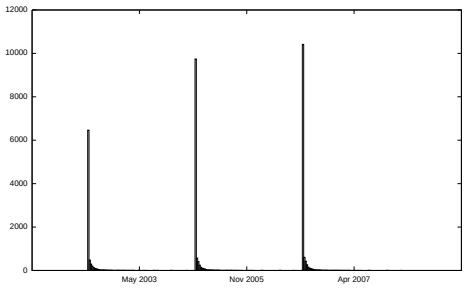
More importantly, the increase in LOC from the November 2005 release to the April 2007 release increased from 41,682 to 44,019, and the *Number of Files* increased from 355 files to 375 files during the same period. This increase in the size of the code base for the last two releases might indicate to a prospective user of the application that Aleph One increased in size and complexity between November 2005 and April 2007. However, the McCabe metrics, presented in the next section, indicate otherwise.

5.2 Cyclomatic Complexity

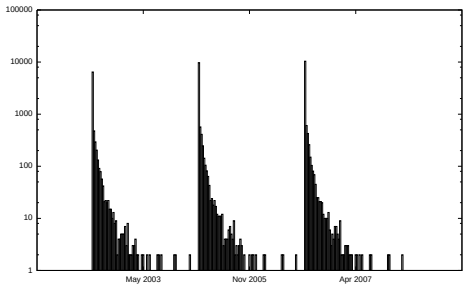
Figure 4 lists results for the cyclomatic complexity metric computed for all of the code, including both user code and system code, in the three version releases of Aleph One. The table in Figure 4(a) lists the version release in the first column, the *Mean*, *Median*, and *Mode* values for cyclomatic complexity computed for each function, and the final column lists the largest cyclomatic complexity value for the functions in the release. Table 4(a) would seem to corrobo-

Release	Mean	Median	Mode	Range
May 2003	1.99	1	1	75
Nov 2005	1.76	1	1	75
Apr 2007	1.93	1	1	75

(a) Summary of metrics for all functions



(b) Linear plot for all functions

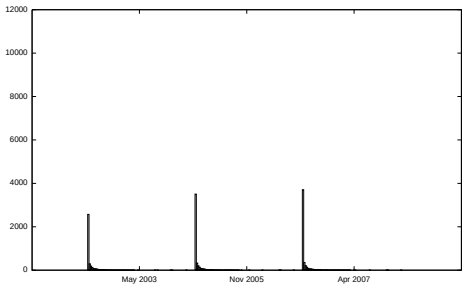


(c) Logarithmic plot for all functions

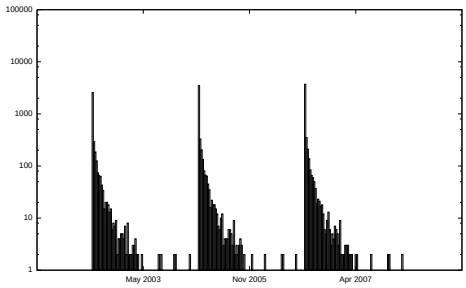
Figure 4: Cyclomatic complexity metrics for all functions in the application.

Release	Mean	Median	Mode	Range
May 2003	2.67	1	1	75
Nov 2005	2.38	1	1	75
Apr 2007	2.37	1	1	75

(a) Summary of metrics for user-defined functions



(b) Linear plot for user-defined functions



(c) Logarithmic plot for user-defined functions

Figure 5: Cyclomatic complexity metrics for user-defined functions in the application.

rate the results for LOC described in the previous section, where the *Mean* cyclomatic complexity was reduced from 2003 to 2005, and then increased from 2005 to 2007.

However, the linear and logarithmic histogram plots shown in Figures 4(b) and 4(c) indicates that the *Mean* cyclomatic complexity for the functions with value of one increased, but the complexity for all functions is more dispersed and, perhaps smaller [1]. Recall that the inlined code for the expat XML parser was elided from the November 2005 release.

A more illustrative presentation of the complexity of the Aleph One releases is shown in Figure 5, where we present cyclomatic complexity values only for user code, with values for system code removed from consideration. Table 5(a) shows that the *Mean* values for user written functions in Aleph One steadily decreased over the three version releases, reducing from 2.67 for the 2003 release to 2.38 for the 2005 release, and then reducing further to 2.37 in the 2007 release. This reduction is further corroborated by both the linear and logarithmic plots shown in Figures 5(b) and 5(c), where the number of functions with complexity of one increases, and the number of functions with lower values increases over the three plots.

[1]Note that the largest spikes in the histogram for these values represent functions with cyclomatic complexity value of one, i.e., straight line code

6. RELATED WORK

The difficulty of parsing C++ is well-studied [16, 21, 27, 28, 31], and this difficulty has lead to alternative approaches to parsing such as *fuzzy* parsing [17], and *island grammars* [25], 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 McCabe's metric

for the *Aleph One* video game that we study.

As a basis of comparison with our approach, we analyzed two open-source projects that compute McCabe's metric for C++ programs: (1) the Cyclomatic Complexity Analyzer (ccm) [4], which analyzes programs written in C/C++, C Sharp, JavaScript, and TypeScript; and (2) CCCC, which analyzes C and C++ programs [22]. However, neither of these projects performs a full parse of C++ programs, but instead use a fuzzy approach, scanning the source code for keywords and other tokens to identify branches in the code. While a fuzzy approach has some advantages for code analysis, such as tolerating minor syntax errors and handling dialects of C++ that are not accepted by Clang, it is inherently less accurate than one based on a full parse of the C++ program under investigation. For example, because Clang invokes the preprocessor, our analyzer easily detects branches introduced by preprocessor-defined keywords that would be omitted by tools using fuzzy parsing technology.

7. CONLUDING REMARKS

In this paper, we described our approach to exploiting the Clang parser and front-end to compute metrics for C++ applications. We provided both an overview of the Clang system, and some details about our use of the Clang internals, including a detailed description of a *visitor* that we wrote to traverse the Clang AST. We provided some results for computing two commonly used software metrics, LOC and cyclomatic complexity, and we described an interesting contradiction for the results of these metrics when applied to successive version releases of the *Aleph One* video game. Our future work entails further exploitation of Clang to provide additional analysis of C++ applications.

8. REFERENCES

[1] GCC. The GNU Project. `http://gcc.gnu.org/`.

[2] B. Bassett and N. A. Kraft. Structural information based term weighting in text retrieval for feature location. In *ICPC*, pages 133–141, 2013.

[3] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft. Configuring latent dirichlet allocation based feature location. In *Empirical Software Engineering*, pages 1–36, 2012.

[4] J. Blunck. ccm, [accessed 26-December-2013]. `"http://www.blunck.info/ccm.html"`.

[5] Bungie. Aleph one: Marathon open source, Aug 2013.

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

[7] Clang. LibTooling, [accessed 27-August-2013]. `"http://clang.llvm.org/docs/LibTooling.html"`.

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

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, January 1995.

[10] M. Hennessy, B. A. Malloy, and J. F. Power. gccXfront: Exploiting gcc as a front end for program comprehension tools via XML/XSLT. In *Proceedings of IWPC*, pages 298–299. IEEE, May 2003.

[11] A. C. Jamieson, N. A. Kraft, J. O. Hallstrom, and B. A. Malloy. A metric evaluation of game application software. *The Future Play Game Development Conference*, pages 13–15, October 2005.

[12] C. Jones. Function points as a universal software metric. *SIGSOFT Softw. Eng. Notes*, 38(4):1–27, 2013.

[13] C. F. Kemerer and B. S. Porter. Improving the reliability of function point measurement: An empirical study. *IEEE TSE*, 18(11):1011–1024, 1992.

[14] Kenneth Zadek. LLVM/gcc integration proposal, November 2005.

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

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

[17] R. Koppler. A systematic approach to fuzzy parsing. *Software Practice and Experience*, 27:649, 1996.

[18] N. A. Kraft, E. B. Duffy, and B. A. Malloy. Grammar recovery from parse trees and metrics-guided grammar refactoring. pages 780–794, 2009.

[19] N. A. Kraft, B. A. Malloy, and J. F. Power. A tool chain for reverse engineering C++ applications. *SCP*, 69(1–3):3–13, December 2007.

[20] C. A. Lattner. Llvm: An infrastructure for multi-stage optimization. Technical report, 2002.

[21] J. Lilley. PCCTS-based LL(1) C++ parser: Design and theory of operation. Version 1.5, February 1997.

[22] T. Littlefair. CCCC - C and C++ Code Counter, [accessed 26-December-2013]. `"http://cccc.sourceforge.net/"`.

[23] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. In *Proceedings of the 2008 15th WCRE*, pages 155–164, 2008.

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

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

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

[27] J. F. Power and B. A. Malloy. Metric-based analysis of context-free grammars. In *Proceedings of the 8th IWPC*, Limerick, Ireland, June 2000.

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

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

[30] Python. Python language parsing. Small Discussion and Evaluation of Different Parsers, 2014.

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

[32] R. Tairas and J. Gray. An information retrieval process to aid in the analysis of code clones. In *Empirical Software Engineering*, volume 14, pages 33–56, February 2009.

[33] A. Treat. "mkspecs and patches for LLVM compile of Qt4. *Qt4-preview-feedback mailing list*, February 2005.

[34] Wikipedia. LLVM, [accessed 27-August-2013]. `"http://en.wikipedia.org/wiki/LLVM"`.