How to Calculate Software Metrics for Multiple Languages Using Open Source Parsers

Andrea Janes, Danila Piatov, Alberto Sillitti, Giancarlo Succi

How to calculate software metrics for multiple languages using Open Source parsers

Andrea Janes, Danila Piatov, Alberto Sillitti, and Giancarlo Succi

CASE, Free University of Bolzano, Piazza Domenicani 3, Italy
{danila.piatov, ajanes, asillitti, gsucci}@unibz.it

Abstract. Source code metrics help to evaluate the quality of the code, for example, to detect the most complex parts of the program. When writing a system which calculates metrics, especially when it has to support multiple source code languages, the biggest problem which arises is the creation of parsers for each supported language. In this paper we suggest an unusual Open Source solution, that avoids creating such parsers from scratch. We suggest and explain how to use parsers contained in the Eclipse IDE as parsers that support contemporary language features, are actively maintained, can recover from errors, and provide not just the abstract syntax tree, but the whole type information of the source program. The findings described in this paper provide to practitioners a way to use Open Source parsers without the need to deal with parser generators, or to write a parser from scratch.

1 Introduction

Measurement in software production is essential for understanding, controlling, and improving the result of the development process. Since software is highly complex, practitioners need tools to measure and analyze the quality of the source code (see e.g., [1, 2, 3, 4, 5]).

To calculate various source code metrics for different programming languages we developed a extensible metrics calculation system based on two component types:

- **parsers**: for every supported language, a parser produces an intermediate representation of the source code, used for further analysis;
- **analyzers**: analyzers take the intermediate representation of the source code and calculate the desired source code metric independently from the originating language.

The main problem we encountered was writing the parsers themselves. This paper describes our findings in the creation of parsers based on Open Source components. Moreover, it gives a brief overview of the architecture the metric calculation system we developed.

Parsing C/C++ is **difficult** [6, 7, 8] and therefore robust (can recover from errors), fully functional, actively maintained, and Open Source C/C++ parsers

are hard to find. To cite a Senior Engineer at Amazon.com: "After lots of investigation, I decided that writing a parser/analysis-tool for C++ is sufficiently difficult that it's beyond what I want to do as a hobby." [7].

Due to the inherited syntactic issues from C, for example that "declaration syntax mimics expression syntax [9]" the C++ grammar is context-dependent and ambiguous. This makes the creation of a C++ parser a complex hence difficult task (not to mention the complex template mechanism present in C++). An example for the statement above is that a construct like `a*b` in C/C++ can either mean a multiplication of `a` and `b` or, if `a` is a type, a declaration of the variable `b` with type `a*`, i.e., a pointer to `a`.

Strictly speaking, a "parser" performs a syntactic analysis of the source code. However, to correctly interpret the parsing results, we needed more than just a syntactic analysis, namely:

1. Particularly for C/C++, preprocessing to expand includes and macros.
2. Semantic analysis, i.e., obtaining type information and bindings is also part of the parsing process. To completely parse C/C++ code, the parser has to resolve the types of all symbols. Resolving bindings means to understand which declaration (e.g., a function, type, namespace) some reference is pointing to.

Moreover, in the particular case which we describe in this article, we had the following additional requirements:

3. The parser has to be able to ignore syntax errors and to continue parsing the remaining source code correctly.
4. C/C++ is continuously developing, and new language features are introduced from time to time, therefore a parser needs to be well-maintained.

Open source compilers like the GNU C compiler (GCC) or LCC are available that fulfill the criteria described above, but—to the best of our knowledge—they do not provide an API to obtain the (intermediate) parsing results. Moreover, LCC supports only C, and not C++.

The availability of a functioning C/C++ parser is important for developers of the Open Source community to evaluate and improve their software [10]. This applies particularly to larger projects which need a quantitative approach to quality. Unfortunately such a parser is so hard to find. To overcome this issue, we evaluated writing a C/C++ parser from scratch (using a parser generator) or to identify or adapt an Open Source alternative. The next section describes how we decided.

2 Related works

We found the following 7 Open Source parsers for C++: clang [11], cppripper [12], Elsa [13], GCC [14] using the `-fdump-translation-unit` option,

GCC_XML [15], the Eclipse CDT C/C++ parser [16], and Doxyparse used within the Analizo metric system [17], a parser based on Doxygen [18].

The time it takes for existing Open Source communities to develop such parsers as well as comments on various blogs (e.g., [19]) induced us to think that to write a parser from scratch would take us months, if not years:

- it took clang **8 years** from the 1st commit in July 2001 [20] to the release 1.0 in October 2009 [21];
- GCC_XML has a first commit more than **12 years** ago (August 2000) [22] and its authors still do not consider it production-ready, since the current release version 0.9;
- the first release of the Eclipse CDT C/C++ parser was more than **10 years** ago (August 2003 [23]) and it is still constantly updated;
- several projects are **abandoned**, such as cpp-ripper (first and last commit during September 2009 [24]) or Elsa (last release August 2005, after 3 years of development [25]).

Moreover, writing such a parser from scratch, we would need to keep it up-to-date with the new language features introduced for C++, which would require a continuous effort.

Based on such information, we decided not to write the parser from scratch but to select an Open Source C/C++ parser that can be instrumented to correspond to our requirements. Moreover, we added the following requirement for such a parser:

5. To avoid learning new parsing technologies for different languages, we prefer Open Source parser solutions that are able to parse **several** languages, not only C++.

3 Results and implementation of a proof of concept

We evaluated the candidates using the 5 requirements stated above. Three parsers fulfilled the requirements 1–4: clang, the Eclipse CDT parser, and Doxyparse. Because of requirement 4, we excluded parsers that are not maintained anymore:

- GCC_XML does not parse function bodies, a feature we need to calculate software metrics for functions [26].
- GCC, using the option `-fdump-translation-unit`, is able to "dump a representation of the tree structure for the entire translation unit to a file" [14], but this option is only designed for use in debugging the compiler and is not designed for use by end-users [27].

Doxygen, on which Doxyparse is based, does not parse the source code completely. To calculate metrics, the authors of Doxyparse had to "hack" the code of Doxygen and, for instance, modify the Lex [28] source file for C to support

the calculation of McCabe's cyclomatic complexity on the lexical analysis stage. Since the information provided by Doxyparse is not complete, we ruled it out.

Both clang and Eclipse CDT fulfill our requirements 1–4, but only the parsers provided by Eclipse fulfill requirement 5, i.e., it is an environment not only for C/C++, but also for Java, JavaScript, and other languages.

We successfully developed a proof of concept that instruments the Eclipse C/C++ parser. The Eclipse C/C++ parser is installed together with the Eclipse CDT development environment [16]. However, as we found out, it is possible to use it as a Java library, without initializing the whole Eclipse platform. Since **the instrumentation of the Eclipse CDT parser is undocumented**, we briefly describe here how we accomplished this.

To use the parser in our application, we added the CDT core library org.eclipse.cdt.core_*.jar, as well as the other libraries in the CDT installation folder on which the plugin depends, to the classpath of our application.

As shown in listing 1, we pass to the parser a list of preprocessor definitions and a list of include search paths (lines 3 and 4). By extending `InternalFileContentProvider` and using this class instead of the empty files provider we are able to instruct the parser to load and parse all included files (line 6).

`InternalFileContentProvider` allows the use of the interface `IIncludeFileResolutionHeuristics`, which can be implemented to heuristically find include files for those cases in which include search paths are misconfigured. To use the C parser instead of C++, the GCCLanguage class should be used instead of the GPPLanguage class (line 10).

```
1  private static IASTTranslationUnit parse(char[] code) throws Exception {
2    FileContent fc = FileContent.create("/Path/ToResolveIncludePaths.cpp", code);
3    Map<String, String> macroDefinitions = new HashMap<String, String>();
4    String[] includeSearchPaths = new String[0];
5    IScannerInfo si = new ScannerInfo(macroDefinitions, includeSearchPaths);
6    IncludeFileContentProvider ifcp = IncludeFileContentProvider.getEmptyFilesProvider();
7    IIndex idx = null;
8    int options = ILanguage.OPTION_IS_SOURCE_UNIT;
9    IParserLogService log = new DefaultLogService();
10   return GPPLanguage.getDefault().getASTTranslationUnit(fc, si, ifcp, idx, options, log);
11 }
```

Listing 1. Calling the CDT parser

Listing 2 shows a C++ parsing example. It outputs "`C C f f `", i.e., each encountered name in the abstract syntax tree of the parsed code.

The parser returns an abstract syntax tree (AST) as the result of parsing the code. The AST is the representation of the structure of the program as a tree of nodes. Each node corresponds to a syntactic construct of the code, e.g. a function definition, an `if`-statement, or a variable reference. This AST is used to calculate metrics, e.g., one could count all conditional statements to estimate McCabe's cyclomatic complexity.

We briefly evaluated the functioning of the Eclipse C/C++ parser by parsing the Linux kernel version 2.6.27.62. It worked without problems out of the box. The parser counted 19,744 files, 8.4M lines of code, and 2.6M logical lines of

code. It took 11 minutes to parse and calculate metrics on 2 CPUs and 3 minutes on 12 CPUs.

```
1  public static void main(String[] args) throws Exception {
2    String code = "class C { private : C f(); }; int f();";
3    IASTTranslationUnit translationUnit = parse(code.toCharArray());
4    ASTVisitor visitor = new ASTVisitor() {
5      @Override public int visit(IASTName name) {
6        System.out.print(name.toString() + " ");
7        return ASTVisitor.PROCESS_CONTINUE;
8      }
9    };
10   visitor.shouldVisitNames = true;
11   translationUnit.accept(visitor);
12 }
```

Listing 2. Parsing "`class C { private : C f(); }; int f();`"

We successfully applied the same method to instrument the parsers for Java and JavaScript that are part of the Eclipse JDT[1] and JSDT[2] projects, but due to space constraints, we can only provide the UML component diagram of our implementation in figure 1. In this implementation, parser wrappers execute the parsers and convert the output to the intermediate representation used in the system. The analyzer then uses this information to calculate the metrics independently from the source language.

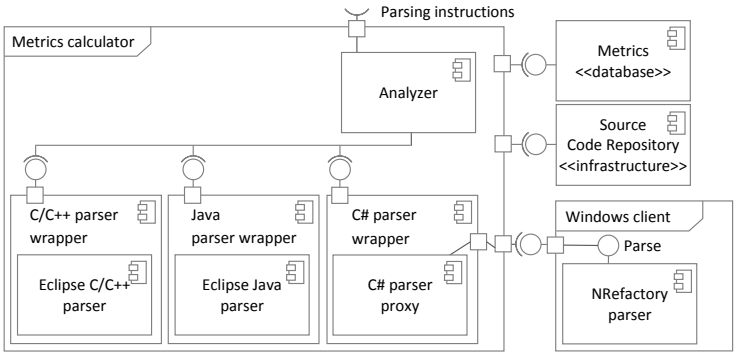

Fig. 1. UML component diagram of the whole metrics calculation architecture.

The C# parser (NRefactory [29]) depicted also in figure 1 is written in C# and has to be executed on a Windows machine. This is why we execute it through a proxy on a remote machine.

The component "metrics calculator" wraps all parser wrappers and the analyzer into one component that is able to parse and extract metrics of a variety of languages. It requires the presence of two additional components: a database to store the extracted code structure and metrics, as well as a component to interface the source repository.

[1] `http://www.eclipse.org/jdt`
[2] `http://www.eclipse.org/webtools/jsdt`

4 Conclusion and future works

This article deals with an (apparently) simple problem: to "create or find a working C/C++ parser". We defined four requirements for such a parser which are relevant for us and we think also for the research community: we ask for a parser that supports contemporary language features, is actively maintained, can recover from errors, and provides not just the abstract syntax tree, but the whole type information of the source program.

Due to the complexity of writing a C/C++ parser ourselves, we decided to evaluate existing Open Source parsers adding a fifth requirement: that we look for parser solutions that support multiple languages.

The conclusions of our research are that Eclipse fulfills all five requirements, contains a C/C++ parser, as well as other parsers like Java and JavaScript. Unfortunately there is no official documentation about using the Eclipse parsers as a library outside of Eclipse. In the last section we provide an example of how to achieve this and give a birds-eye view of the architecture of our measurement system that is based on the parsers contained in Eclipse.

In the future we intend to adopt a systematic approach to evaluate the maturity of parsers, e. g., using the Open Maturity Model [30] or automatic measurement techniques [31, 32].

References

1. Sillitti, A., Janes, A., Succi, G., and Vernazza, T. (2004) Measures for mobile users: an architecture. *J. Syst. Archit.*, **50**, 393–405.
2. Jermakovics, A., Scotto, M., Sillitti, A., and Succi, G. (June) Lagrein: Visualizing user requirements and development effort. *15th IEEE International Conference on Program Comprehension*, pp. 293–296.
3. Jermakovics, A., Moser, R., Sillitti, A., and Succi, G. (2008) Visualizing software evolution with lagrein. *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, New York, NY, USA, pp. 749–750, OOPSLA Companion '08, ACM.
4. di Bella, E., Sillitti, A., and Succi, G. (2013) A multivariate classification of open source developers. *Information Sciences*, **221**, 72 – 83.
5. Janes, A. and Succi, G. (2012) The dark side of agile software development. *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, New York, NY, USA, pp. 215–228, Onward! '12, ACM.
6. Werther, B. and Conway, D. (1996) A modest proposal: C++ resyntaxed. *ACM SIGPLAN Notices*, **31**, 74–82.
7. Birkett, A. (2001), Parsing C++. `http://www.nobugs.org/developer/parsingcpp/index.html`, accessed 2012.04.14.
8. Piatov, D., Janes, A., Sillitti, A., and Succi, G. (2012) Using the Eclipse C/C++ development tooling as a robust, fully functional, actively maintained, open source C++ parser. Hammouda, I., Lundell, B., Mikkonen, T., and Scacchi, W. (eds.), *OSS*, vol. 378 of *IFIP Advances in Information and Communication Technology*, p. 399, Springer.

9. Ritchie, D. M. (1993) The development of the C language. pp. 201–208.
10. Russo, B., Scotto, M., Sillitti, A., and Succi, G. (2009) *Agile Technologies in Open Source Development*. Information Science Reference - Imprint of: IGI Publishing.
11. clang: a C language family frontend for LLVM. `http://clang.llvm.org`, accessed 2013.02.01.
12. Diggins, C. (2012), cpp-ripper, An open-source C++ parser written in C#. `http://code.google.com/p/cpp-ripper/`.
13. McPeak, S., Elsa: The Elkhound-based C/C++ parser. `http://scottmcpeak.com/elkhound/sources/elsa/`, accessed 2012.04.14.
14. GCC Development Team, GCC, the GNU Compiler Collection. `http://gcc.gnu.org`, accessed 2012.04.14.
15. King, B., GCC-XML, the XML output extension to GCC. `http://www.gccxml.org`, accessed 2012.04.14.
16. (2012), Eclipse CDT (C/C++ Development Tooling). `http://www.eclipse.org/cdt`.
17. Terceiro, A., Costa, J., Miranda, J., Meirelles, P., Rios, L. R., Almeida, L., Chavez, C., and Kon, F. (2010) Analizo: an extensible multi-language source code analysis and visualization toolkit. *Brazilian Conference on Software: Theory and Practice (CBSoft) – Tools*, Salvador-Brazil.
18. Doxygen. `http://www.stack.nl/~dimitri/doxygen/`, accessed 2013.02.01.
19. stackoverflow.com, How much time would it take to write a C++ compiler using flex/yacc? `http://stackoverflow.com/questions/1961604`, accessed 2013.02.01.
20. First revision of clang. `http://llvm.org/viewvc/llvm-project?view=rev&revision=1`, accessed 2013.02.01.
21. Lattner, C. (2009), Llvm 2.6 release! `http://lists.cs.uiuc.edu/pipermail/llvm-announce/2009-October/000033.html`, accessed 2013.02.01.
22. GCC-XML, the XML output extension to GCC repository. `https://github.com/gccxml/gccxml`, accessed 2013.02.01.
23. Eclipse CDT (C/C++ Development Tooling) repository. `http://git.eclipse.org/c/cdt/org.eclipse.cdt.git/refs/tags`.
24. cpp-ripper list of repository changes. `https://code.google.com/p/cpp-ripper/source/list`, accessed 2013.02.01.
25. Elkhound: A glr parser generator and elsa: An elkhound-based c++ parser. `http://scottmcpeak.com/elkhound/`, accessed 2013.02.01.
26. GCC-XML, Frequently Asked Questions. `http://www.gccxml.org/HTML/FAQ.html`, accessed 2013.02.01.
27. Mitchell, M., GCC Bugzilla Bug 18279. `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=18279`, accessed 2013.02.01.
28. Lex – A Lexical Analyzer Generator. `http://dinosaur.compilertools.net/lex/`, accessed 2013.02.01.
29. The NRefactory library. `https://github.com/icsharpcode/SharpDevelop/wiki/NRefactory`, accessed 2013.02.01.
30. Petrinja, E., Nambakam, R., and Sillitti, A. (2009) Introducing the opensource maturity model. *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, Washington, DC, USA, pp. 37–41, IEEE Computer Society.
31. Sillitti, A., Janes, A., Succi, G., and Vernazza, T. (April) Monitoring the development process with eclipse. *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, vol. 2, pp. 133–134 Vol.2.
32. Scotto, M., Sillitti, A., Succi, G., and Vernazza, T. (2006) A non-invasive approach to product metrics collection. *J. Syst. Archit.*, **52**, 668–675.