Using the Eclipse C/C++ Development Tooling as a Robust, Fully Functional, Actively Maintained, Open Source C++ Parser

Conference Paper · September 2012

DOI: 10.1007/978-3-642-33442-9_45

CITATIONS
7

READS
1,553

4 authors:

Danila Piatov
Free University of Bozen-Bolzano
7 PUBLICATIONS **55** CITATIONS
SEE PROFILE

Andrea Janes
Free University of Bozen-Bolzano
74 PUBLICATIONS **835** CITATIONS
SEE PROFILE

Alberto Sillitti
Innopolis University
271 PUBLICATIONS **2,723** CITATIONS
SEE PROFILE

Giancarlo Succi
Innopolis University
540 PUBLICATIONS **5,647** CITATIONS
SEE PROFILE

Using the Eclipse C/C++ Development Tooling as a Robust, Fully Functional, Actively Maintained, Open Source C++ Parser

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

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

Abstract. Open Source C/C++ parsers that support contemporary C/C++, can recover from errors, include a preprocessor, and that are actively maintained, are rare. This paper describes how to use the parser contained in the Eclipse C/C++ development tooling (CDT) as a library. Such parser provides not only the abstract syntax tree of the parsed file but also the semantics, i.e., type information and bindings. The findings described in this paper provide to practitioners a C++ parser without the need to hassle with parser generators, or to write a parser themselves. The authors used the same approach also to obtain Java and JavaScript parsers with comparable attributes.

1 Introduction

Parsers are used to interpret text, i.e., to identify the meaning of text and to decide what to do with it. More formally, "parsers are used to analyze a string of characters in order to associate groups of characters with the syntactic units of the underlying grammar [1]".

Since the upcoming of static program analysis, parsers are also used in the evaluation of the quality of source code. For example, a parser can extract properties of the source code that are then used in a subsequent step to calculate so called source code metrics. Practitioners have used parsers in this field to develop tools to support them in achieving software quality goals (see e.g., [2, 3, 4, 5, 6, 7, 8].

Parsing C++ is particularly tricky [9, 10]. For example, a construct like `a * b` seems to be a multiplication of `a` and `b`. In C++ it can either mean a multiplication or, if `a` is a type, a declaration of the variable `b` with type `a*`, i.e., a pointer to `a`.

Due to the inherited syntactic issues from C, e.g., that "declaration syntax mimics expression syntax" [11] as in the example of `a * b`, the C++ grammar is context-dependent and ambiguous. This makes the creation of a C++ parser a complex hence difficult task.

This task is not alleviated by the availability of Open Source parser generators (see e.g., [12, 13, 14]) since the ambiguities cannot be resolved by a parser alone but require type information.

This article reports our experience in the identification and adaptation of an Open Source solution to parse C++ code, which we later used to calculate source code metrics.

This article is structured as follows: section 2 describes why chose the CDT parser, in section 3 we describe how to use it, and section 4 states our concluding remarks.

2 Choosing an Open Source C++ parser

We had the following requirements for an Open Source C++ parser:

- Preprocessing to expand C++ includes and macros is a must.
- Semantic analysis, i.e., obtaining type information and bindings is also part of the parsing process. To completely parse 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.
- The parser has to be able to ignore syntax errors and to continue parsing the remaining source code correctly.
- C++ is continuously developing, and new language features are introduced from time to time, therefore a parser needs to be well-maintained.

After understanding the amount of work needed to make such a parser from scratch or using a parser generator, we decided to search on the web for Open Source C++ parsers that fit to our requirements.

After an extensive search, we evaluated the following candidates: Clang [15], cpp-ripper [16], Elsa [17], GCC [18] using the `-fdump-translation-unit` option, GCC_XML [19], and the Eclipse CDT C++ parser [20].

Two parsers fulfilled the requirements stated above: Clang, and the CDT parser. The remaining parsers were either not maintained anymore or did not fulfill one ore more of the requirements:

- GCC_XML, which does not output function bodies, a feature we need to calculate software metrics for functions[1]. Therefore, and because it was last updated in 2004, we excluded GCC_XML.
- Elsa, which includes no preprocessor [17], does not keep track of macro expansions [21], and was last updated in 2006.
- GCC, using the option `-fdump-translation-unit`, to "dump a representation of the tree structure for the entire translation unit to a file" [18], which could be used for further processing. However, this option is only designed for use in debugging the compiler and is not designed for use by end-users[2].

[1] FAQ, GCC_XML website, `http://www.gccxml.org/HTML/FAQ.html`

[2] GCC bug tracking system post by Mark Mitchell, GCC's Release Engineer, `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=18279`

In summary, according to our initial requirements, Clang would have been a good choice since it is better documented then CDT. Nevertheless, since the main development language in our project was Java, we decided to explore the possibility to use the C++ parser contained in CDT since it is written in Java. Clang is written in C++, without bindings for Java[3].

Another reason for trying out the CDT parser was that Eclipse is an environment not only for C++, but also for C, Java, JavaScript, and other languages. It was our hope that we will be able to use also these parsers in a similar way.

3 The Eclipse C/C++ development tooling parser

After the installation of CDT using the "Install New Software" feature of any Eclipse installation, the Eclipse C++ parser is located in the file "org.eclipse.cdt.core_*X*.jar", in the Eclipse installation folder, where *X* stands for version of the file. The jar itself is an Eclipse plugin, however, it is possible to use it as a simple Java library, without initializing the whole Eclipse platform. To use the parser in our application, we added the CDT core jar to the classpath of our application[4] .

Using the imports of listing 1, we were able to call the parser contained within CDT using the code in listing 2.

```
1 import java.util.*;
2 import org.eclipse.cdt.core.dom.ast.*;
3 import org.eclipse.cdt.core.dom.ast.gnu.cpp.GPPLanguage;
4 import org.eclipse.cdt.core.index.IIndex;
5 import org.eclipse.cdt.core.model.ILanguage;
6 import org.eclipse.cdt.core.parser.*;
7 import org.eclipse.core.runtime.CoreException;
```

Listing 1. Imports needed to call the CDT parser

As shown in listing 2, 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).

[3] Internet forum entry by Eric Christopher, a Clang developer, `http://lists.cs.uiuc.edu/pipermail/cfe-dev/2011-November/018852.html`

[4] The parser also depends on the following jars: com.ibm.icu_*X*.jar, org.eclipse.core{contenttype_*X*.jar, resources_*X*.jar, jobs_*X*.jar, runtime_*X*.jar}, org.eclipse.equinox{common_*X*.jar, preferences_*X*.jar, registry_*X*.jar}, org.eclipse.osgi_*X*.jar, where *X* stands for version of the corresponding file.

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

Listing 2. Calling the CDT parser

Listing 3 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.

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

Listing 3. Parsing "`class C { private : C f(); }; int f();`"

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.

The CDT parser returns the root AST node of the type `IASTTranslationUnit`, which is derived from a basic AST node, `IASTNode`. There are many other derivatives from it, each for every C/C++ syntax element, e.g. `IASTStatement`, from which `IASTIfStatement` is derived. `IASTIfStatement` has references to the condition, *then*- and *else*-clauses.

There are two ways how to use the AST. The first is to traverse the tree using methods specific for each node type (e.g., calling `getThenClause()` of the `IASTIfStatement` node). The second method is to use instances of `ASTVisitor`, which can be passed to any node via `accept(ASTVisitor)` (see listing 3).

The power of the CDT parser is in `IASTName` nodes, which represent each name symbol in the AST. For each name we can get its binding by calling `resolveBinding()`. A binding in an implementation of `IBinding` or a derived interface like `IVariable`, `IFunction`, `ICPPClassType`. Bindings are used to get the place of the definition of a name in program namespace (by recursively calling `getOwner()`) and to get its type (e.g. `IVariable.getType()`). The type can be translated to human readable string using the `ASTTypeUtil` class [22].

The following listing demonstrates how the CDT parser is able to resolve bindings and type information:

```
public static void main(String[] args) throws Exception {
```

```
2   String code = "typedef float myType; void f(int i) {} void f(double d) {} " +
3       "void main() { myType var = 4; f(var); }";
4   IASTTranslationUnit translationUnit = parse(code.toCharArray());
5   ASTVisitor visitor = new ASTVisitor() {
6     @Override public int visit(IASTName name) {
7       // Looking only for references, not declarations
8       if (name.isReference()) {
9         IBinding b = name.resolveBinding();
10        IType type = (b instanceof IFunction) ? ((IFunction) b).getType() : null;
11        if (type != null)
12          System.out.print("Referencing " + name + ", type " + ASTTypeUtil.getType(type));
13      }
14      return ASTVisitor.PROCESS_CONTINUE;
15    }
16  };
17  visitor.shouldVisitNames = true;
18  translationUnit.accept(visitor);
19 }
```

Listing 4. Demonstration that the CDT parser is able to resolve bindings and type information.

This example outputs "`Referencing f, type void (double)`" since the parser not only resolves the types of variables, but associates `f(var)` (`var` being of type `myType`) to `f(double)` since `myType` is of type `float` and `float` in this specific example can be implicitly converted only to `double`.

4 Conclusion

This article deals with an (apparently) simple problem: to "find a working C++ parser". On the first sight, it seems strange that a robust (can recover from errors), fully functional, actively maintained, and Open Source C++ parser is so hard to find. Eclipse CDT contains such parser, but there is no official documentation about using it as a library outside of Eclipse. We hope that this article can fill this gap and be of help.

We briefly evaluated the performance of the CDT parser. For this purpose we created a file with one line containing `#include <iostream>` and used GCC (version 4.2.1) to generate the file after preprocessing. The resulting file had a length of about 25000 lines. We parsed this file with the CDT parser and with GCC with the syntax check only option.

The execution times were approximately 0.16 sec for GCC and 0.40 sec for the CDT parser using Java in 32-bit mode and the client VM. For the 64-bit mode and the server VM the results were about two times slower.

We successfully applied the same method to parse Java and JavaScript code using the corresponding parsers contained in the Eclipse [23] Java and JavaScript sub-projects. This is interesting for practitioners that are interested in parsing several languages since they can reuse part of their knowledge when using Eclipse based parsers.

References

1. Grune, D. and Jacobs, C. J. H. (2008) *Parsing Techniques: A Practical Guide*. Springer.
2. Pugh, B. and Loskutov, A. (2012), FindBugs™– find bugs in java programs. `http://findbugs.sourceforge.net`.
3. Scotto, M., Sillitti, A., Succi, G., and Vernazza, T. (2006) A non-invasive approach to product metrics collection. *J. Syst. Archit.*, **52**, 668–675.
4. Scotto, M., Sillitti, A., Succi, G., and Vernazza, T. (2004) Dealing with software metrics collection and analysis: a relational approach. *Stud. Inform. Univ.*, **3**, 343–366.
5. Scotto, M., Sillitti, A., Succi, G., and Vernazza, T. (2004) Non-invasive product metrics collection: an architecture. *Proceedings of the 2004 workshop on Quantitative techniques for software agile process*, New York, NY, USA, pp. 76–78, QUTE-SWAP '04, ACM.
6. Scotto, M., Sillitti, A., Vernazza, T., and Succi, G. (2004) Webmetrics: A tool for improving software development. Arabnia, H. R. and Reza, H. (eds.), *Software Engineering Research and Practice*, pp. 545–548, CSREA Press.
7. Scotto, M., Sillitti, A., Succi, G., and Vernazza, T. (2004) A relational approach to software metrics. *Proceedings of the 2004 ACM symposium on Applied computing*, New York, NY, USA, pp. 1536–1540, SAC '04, ACM.
8. Russo, B., Scotto, M., Sillitti, A., and Succi, G. (2009) *Agile Technologies in Open Source Development*. Information Science Reference - Imprint of: IGI Publishing.
9. Werther, B. and Conway, D. (1996) A modest proposal: C++ resyntaxed. *ACM SIGPLAN Notices*, **31**, 74–82.
10. Birkett, A. (2001), Parsing C++. `http://www.nobugs.org/developer/parsingcpp/index.html`, accessed April 14th, 2012.
11. Ritchie, D. M. (1993) The development of the C language. pp. 201–208.
12. JavaCC Development Team (2012), Java Compiler Compiler™(JavaCC™) – The Java parser generator. `http://javacc.java.net`.
13. Johnson, S. C. (2012), Yacc: Yet Another Compiler-Compiler. `http://dinosaur.compilertools.net`.
14. Parr, T. (2012), ANTLR, ANother Tool for Language Recognition. `http://www.antlr.org`.
15. Clang Development Team, Clang: a C language family frontend for LLVM. `http://clang.llvm.org`, accessed April 14th, 2012.
16. Diggins, C. (2012), cpp-ripper, An open-source C++ parser written in C#. `http://code.google.com/p/cpp-ripper/`.
17. McPeak, S., Elsa: The Elkhound-based C/C++ parser. `http://scottmcpeak.com/elkhound/sources/elsa/`, accessed April 14th, 2012.
18. GCC Development Team, GCC, the GNU Compiler Collection. `http://gcc.gnu.org`, accessed April 14th, 2012.
19. King, B., GCC-XML, the XML output extension to GCC. `http://www.gccxml.org`, accessed April 14th, 2012.
20. The Eclipse Foundation (2012), Eclipse CDT (C/C++ Development Tooling). `http://www.eclipse.org/cdt/`.
21. Clang Development Team, Clang vs Elsa. `http://clang.llvm.org/comparison.html`, accessed April 14th, 2012.

22. The Eclipse Foundation, Eclipse Indigo CDT Documentation: Class ASTTypeUtil. http://help.eclipse.org/indigo/topic/org.eclipse.cdt.doc.isv/reference/api/org/eclipse/cdt/core/dom/ast/ASTTypeUtil.html, accessed April 14th, 2012.
23. The Eclipse Foundation, Eclipse – The Eclipse Foundation open source community website. http://www.eclipse.org, accessed April 14th, 2012.