C/C++ source code analyzing with CDT

Luong Thanh Binh, Hoang Anh Viet, Pham Xuan Toan, Takenobu Aoshima

***Abstract*—this paper will introduce a model used in reversing engineering. The target is used for building a C/C++ code analyzer. However, the article only describes static code analyzer. We use open source (CDT) for some components, so that this article also contains some dept description about CDT, especially, CDT parser and indexer.**

***Index Terms*—Reverse engineering, CDT, AST**

I. INTRODUCTION

We know that nowadays software develops more quickly than ever. Methodologies and processes are applied. Reusing source code is most common. If we know the influence of modules, class… to others we can assessment, optimize code easily.

There are many ways to archive these objectives. In this article, however we will introduce a methodology – static source code analyzing. We also introduce a general model which is used for static code analyzing. Through this article you can have closer looking at static code analyzing method, how to implement it. This model can be broadly applied to many problems.

Static code analysis is the analysis of computer software that is performed without actually executing programs built from that software (analysis performed on executing programs is known as dynamic analysis). In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. The term is usually applied to the analysis performed by an automated tool, with human analysis being called program understanding or program comprehension.

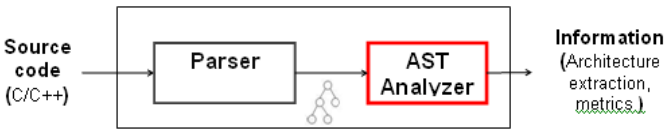
The sophistication of the analysis performed by tools varies from those that only consider the behavior of individual statements and declarations, to those that include the complete source code of a program in their analysis. Uses of the information obtained from the analysis vary from highlighting possible coding errors (e.g., the lint tool) to formal methods that mathematically prove properties about a given program (e.g., its behavior matches that of its specification).

Basically, we can build a code analyzer from scratch, but for quickly development we can use existing open source. For example, We can use GCC parser, ANTLR, ELSA … for parser component. However, in this article we will introduce another one – CDT. CDT is a C/C++ development toolkit developed by IBM, QNX…

In this article, we will use CDT as a demonstration for parser, indexing service.

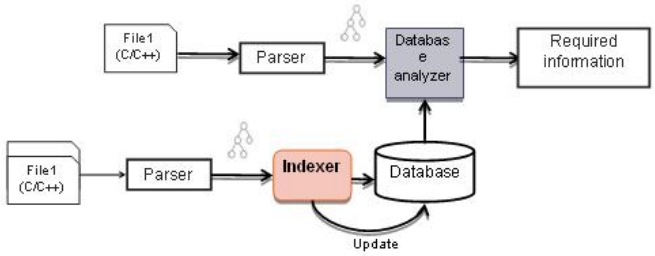
II. A MODEL FOR STATIC SOURCE CODE ANALYZING

A. General model:



As above figure, you can see that source code files are inputted a parser which performs parsing and generating a analyzing model called AST. If we want to extract, get any information about inputted source code, we need process the AST tree. Basically, this model satisfies almost situations. However, if we want to treat with a large source code (for example, number line of code more than 500,000), this model there are some disadvantages. One of them is performance. In order to overcome this problem, we use an automated indexing service whose role is as data base indexing management in Database management system software.

B. *Using indexing*



In this model, first of all, when we create new project or import a source code in order to analyzing, system will call an indexing service (this service call suitable parser to parse all source code files and stored important information into database). If system wants to extract/analyze a information in a source code file, it will corporate parsed AST of the file and information in database to get required information.

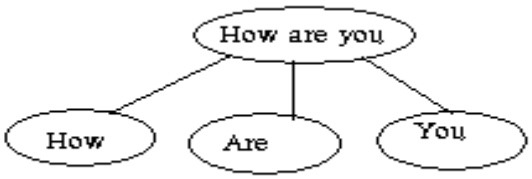
III. WHAT IS PARSER

In order to understand what a parser is, let us consider the input sentence, "Parser is a program." You might know the individual words, but if you aren't familiar with the subject-verb-object sentence structure, you won't understand its meaning. The sentence makes sense only if you can match the elements of the input to those in the abstract structure (“Parser" = subject, "is" = verb, "a program" = object). Of course, there are many other sentence structures and many other languages. The body of rules that specify how proper sentences are constructed in a given language is called the language's *grammar*.

With a programming language, the grammar contains an abstract model that all well-formed units of code (that is, source files) must match. The first step in performing this match is called *scanning*, or *lexical analysis*. A scanner reads in the individual characters of a buffer and returns symbols (called *tokens*) that the parser will understand -- such as keywords, operators, and language-specific punctuation. For example, when a `C` scanner reads in the characters *c*, *o*, *n*, *s*, and *t*, it returns a single token object representing the keyword *const*. The parser doesn't care about white space or comments, so the scanner disregards them.

The second step of the match process is parsing. Here, the parser tests combinations of tokens to see how they compare to the abstract elements in its grammar. Generally, this is much more complex than the subject-verb-object example above. For example, if a statement starts with a token corresponding to "enum" (declaring an enumeration), the parser will try to match the statement to its abstract model of an enum statement. That is, after the `ENUM` token, it will look for an optional *name*, then an *enumerated_list* between `L_BRACE` and `R_BRACE` tokens, and a final *enumerated_list*. Each italicized term refers to another abstract element in the grammar.

Many parsers, including the CDT parser, do more than simply check for well-formedness. After they've parsed the input, they store its structural information in a form suitable for analysis, indexing, and searching. This form is usually that of a tree, called an AST.



The elements in the tree are general at the top and become more particular with each downward node. If I had the space, I'd show how the nodes at the bottom contain sentences and finally specific words. For a programming language, these terminal nodes contain individual keywords, operators, and variables. It's much easier to find a function or variable name by searching an AST than it is to re-parse the entire document. Now, I show how the CDT parser generates the AST and how you can traverse it with your own code.

IV. WHAT IS INDEXING SERVICE

The indexing service has three main components as following:

A. Persistent Database:

Generally, Persistent database can be specific file in operating system. In CDT, The Persistent database is a flat file that stores binary information. This file is memory mapped into direct byte buffers in 4K chunks. A LRU cache is maintained to ensure that

only a manageable number of chunks are kept in memory.

The data stored in the PDOM database is organized by records. Records may contain any type of data, but generally store integers, strings, and offsets of other records in the database. To keep things fast and simple, records can not span chunks so must be less than 4K in size. Smaller records are also recommended to reduce wasted space.

A pretty standard memory management technique is used to managed the allocation and freeing of records. Memory in the database is allocated in multiples of 16 bytes. (4K/16 = 256 different sizes). A linked list is maintained of all blocks of the same size. Blocks are subdivided when no smaller block is available to fit the size of the request. When blocks are freed, they are placed at the front of the free list. Neighboring free blocks are not recombined, although this is an optimization we can do in a later release.
To help speed access to large lists of records, B-Tree indexes are available. The B-Tree features balanced inserts.
One database is maintained per project and is stored in the meta-data.

B. Persistent Indexer:

In the indexing business is that not all projects are created equal. With the Full indexer, a small project can be indexed in a matter of seconds, where a large project can take hours. Faster indexers may store less content and less accurate information, which may be good enough for some and not for others.

So the key is that we need to be able to plug in different indexers for different situations. To simplify the clients of indexing information, these indexing architectures must all be able to write their information to a Persistent database.

C. Indexing client:

Indexing clients are essentially all features that require information from the index.

V. BINDING

A binding is an abstracted language element that encapsulates all the ways a particular identifier is used in a program. For example, if a program has a function called foo() then there will be a single binding object that represents foo(), this binding object will store all the information about how the foo identifier is used in the program, including the location of the declaration, the location of the definition and the locations of all the places where the function is called.

There are many different kinds of bindings representing various language constructs that are “bound” to identifiers. For example there are class, struct, variable and macro bindings just to name a few. Bindings are the basis of many of CDT’s features and form the basis of how indexing works.
Binding Resolution:

Binding Resolution is the name of the algorithm used to compute bindings. Binding resolution is performed on the AST after the code has been parsed. IASTName nodes, which represent identifiers in the AST, are the interface for retrieving resolved bindings. The binding resolution algorithm is quite complex, but in essence it searches the AST for uses of an identifier and generates an IBinding object which stores information about those uses.

An example of how bindings are used is the “open declaration” action (triggered by selecting an identifier in the editor and pressing F3 or Ctrl + Left Click mouse in Eclipse IDE). When this action is invoked an AST is generated for the code in the editor. Then offsets are used to determine which name node in the AST corresponds to the identifier selected in the editor. The resolveBinding() method is then called on that name node to retrieve its binding. The binding stores information on where the identifier is declared, this information is used to open the location of the declaration in the editor.

If there are errors in the source code that prevent binding resolution from being successful, for example an identifier exists without having been declared, then a “problem binding” object is returned.

Parsing and binding resolution is a slow process, this is a problem because the user expects code editing features such as “content assist” to be fast. For this reason CDT stores binding information in an on-disk cache called “the index” or “the PDOM” (Persisted Document Object Model) in order to be able to provide features that respond quickly to user requests.

Building the index involves parsing all the code in a project, resolving all the bindings and writing those bindings to the index. The index is then incrementally updated every time the user edits a file.

CDT supports three different indexing modes, fast indexing, full indexing and no indexing. The default setting is the fast indexer because indexing a large project can be a time consuming process. The difference between the fast and full indexers is that the fast indexer will skip header files that have already been added to the index, while the full indexer will always write a copy of a header file to the index every time it is included.

The fast indexer is the preferred option; however it may not be fully accurate under certain circumstances. When a header file is included in a source file it is subject to any macros that have been defined at that point. Some library headers use macros in conjunction with preprocessor conditionals (#ifdefs) to partially include a header file. Sometimes such a header file is included more than once in a project, if the macros that the header depends on are different each time the header is included then different parts of the header may be included in different places. The fast indexer will not be accurate in this scenario because it will only index the header the first time it is encountered. The full indexer will properly index the header for each time it is included.

Each project has a single PDOM associated with it. The PDOM is stored on disk as a flat binary file. The indexer will only index headers that are included by source files, so if there is a .h file in the project that is not being included by any .c or .cpp file, then normally it won’t get indexed. However we can index all files in the project.

VI. AST MANIPULATION

In computer science, an **abstract syntax tree** (AST) is a finite, labeled, directed tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the operators. Thus, the leaves are NULL operators and only represent variables or constants. In computing, it is used in a parser as an intermediate between a parse tree and a data structure, the latter of which is often used as a compiler or interpreter's internal representation of a computer program while it is being optimized and from which code generation is performed. The range of all possible such structures is described by the abstract syntax. An AST differs from a parse tree by omitting nodes and edges for syntax rules that do not affect the semantics of the program. The classic example of such an omission is grouping parentheses, since in an AST the grouping of operands is implicit in the tree structure. Creating an AST in a parser for a language described by a context free grammar, as nearly all programming languages are, is straightforward. Most rules in the grammar create a new node with the node's edges being the symbols in the rule. Rules that do not contribute to the AST, such as grouping rules, merely pass through the node for one of their symbols. Alternatively, a parser can create a full parse tree, and a post-pass over the parse tree can convert it to an AST by removing the nodes and edges not used in the abstract syntax.

The AST is generated while parsing a file source code. In CDT, after finish parsing, and search the input, parsers of CDT will store its structural information in a form suitable for analysis, indexing. This form is usually that of a tree, called an AST.

CDT does not construct separate grammar file (like ANTLR) so it attach them into the code. You can find them in the org.eclipse.cdt.internal.core.parser package.

The first parsing method is `translationUnit()` because the `TranslationUnit` element represents the entire

source file, just as the `Article` element in my example represents an entire article.

A cryptic statement in the comment precedes the `translationUnit()` method:

```
translationUnit : (declaration)*
```

If you're familiar with the Extended Backus Naur Form (EBNF), you'll know that this rule means that a `TranslationUnit` element consists of any number of `Declaration` elements. These comments don't provide complete CDT grammar, but if you're trying to use or modify the `Parser` class, you'll them helpful.

Because the `Declaration` element is just beneath `TranslationUnit`, the `translationUnit()` method calls the `declaration()` method. The EBNF rule tells us that a declaration can take one of six forms:

Assembly statement
: The `asm` token followed by an `ASMDefinition` element

Namespace statement
: The `namespace` token followed by an `NamespaceDefinition` element

Using statement
: The `using` token followed by an `UsingDeclaration` element

Template statement
: The `template` or `export` token followed by an `TemplateDeclaration` element

Linkage statement
: The `extern` token followed by an `LinkageSpecification` element

Simple statement
: A `SimpleDeclaration` element

To determine the type of a given declaration, the `declaration()` method uses a `switch` statement whose execution depends on the type of the scanner's next token. Each token implements `IToken` and stores its type (an `int` between 1 and 141), the file being scanned, and its offset and length in the scanned `Document`.

If the declaration doesn't fall into one of the first five categories, the `simpleDeclaration()` method is called, along with a guess as to the nature of the declaration. The first guess is that the declaration is a constructor, but if the method throws a `BackTrackException`, the next guess is that it's a function declaration. If that fails, the method is called a third time, this time guessing it's a variable declaration. If this throws another exception, the method returns `null`.

The parser continues to call methods to match code with model elements, but the depth of analysis depends on its *mode*. The five parsing modes are:

QUICK_PARSE
: Does not parse inside functions or included files

STRUCTURAL_PARSE
: Does not parse inside functions but parses included files

COMPLETE_PARSE
: Parses inside functions and included files

COMPLETION_PARSE
: Parses inside functions and included files, stops at offsets, and optimizes symbol query lookups

SELECTION_PARSE
: Parses inside functions and included files, stops at offsets, and provides semantic information about a selected range

The mode also determines what callback object the `Parser` uses to store information during the parse. In `QUICK_PARSE` mode, a `QuickParseCallback` keeps track of macros, functions, declarations, and any errors that occur. In any other mode, a `StructuralParseCallback` stores such additional information as the `Parser`'s current scope, variables, `namespace` statements, `enum` declarations, and more. But because additional analysis requires extra time, `QUICK_PARSE` is the `Parser`'s default mode.

AST is a composite data structure so in order to traverse AST tree we implement Visitor pattern. The purpose of the Visitor Pattern is to encapsulate an operation that you want to perform on the elements of a data structure. In this way, you can change the operation being performed on a structure without the need of changing the classes of the elements that

you are operating on. Using a Visitor pattern allows you to decouple the classes for the data structure and the algorithms used upon them (GoF).

VII. Information Extraction

Our objective is to extract information from inputted source code. This component will do everything you want. If you want to build system that analyses source code and shows required information, nothing you have to do except implement this component.

VIII. Put All Together

First of all, code analyzer system will index all inputted source code files. The goal of indexing is to record all the named elements in the project's source code and store them with their related data. This is a complicated process and can run in the background, so it's performed as an Eclipse `Job`. Specifically, the indexer creates an indexing `Job` with a priority of `Job.LONG` and adds it to the platform's queue.

If the project has just been created, the `Job` doesn't accomplish much because there's no code to index. Nevertheless, it does build a very important object for the project called a `PDOM`. This provides the routines that store the indexer's data within the PDOM. The indexer doesn't begin immediately or automatically. Instead, the `PDOMManager` waits for changes to the `CProject`, such as a new, modified, or deleted source file. When it detects a change, it tells the indexer to start analyzing, and this time, the indexer `Job` works in earnest. In particular, it finds the altered `TranslationUnit` and the `ILanguage` that represents the language of the unit's source code. Then it starts the parsing process by calling `ILanguage.getASTTranslationUnit()`.

Secondly, parsing the TranslationUnit - The Parser from makes no distinction between programming languages. It looks for C and C++ elements and uses them to create a generic AST.

Thirdly, updating PDOM database After the indexer receives the IASTTranslationUnit, it gets a write lock on the PDOM and begins persisting the TranslationUnit's data. First, it calls IASTTranslationUnit.getIncludeDirectives() and IASTTranslationUnit.getMacroDefinitions(). Then, it stores a reference to the unit's file in the PDOM, along with references to the included directives and macro definitions. These references take the form of PDOMFiles that can be stored with PDOM.addFile().

Finally, do any thing you want! For example, if you want to search all references of a function. At this step we need:

- Parse containing file: AST tree of this file so we can get all information about this function (IASTName, starting offset, ending offset…).
- Check references of the IASTName in PDOM database. You will get IASTNames that reference to the function.
- For each reference will need parse its containing file to get all required information about it.
- Show result.

IX. Conclusion

The development of code analyzer has introduced many interesting innovations. From this direction, we can depth research and build a IDE, metric tool. Keep in mind that, however to treat with large project, you must carefully assess all algorithms before them are used in program in order to enhance performance of system.

X. References:

[1]. IBM open source development http://www.ibm.com/developerworks/views/opensource/libraryview.jsp?search_by=CDT+based+editor
[2]. Eclipse Rich client platform development: http://wiki.eclipse.org/index.php/Rich_Client_Platform
[3]. ECLIPSE CDT development http://www.eclipse.org/cdt/