The CodeCompass Comprehension Framework

Zoltán Porkoláb, Tibor Brunner
Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary
[gsd,bruntib]@caesar.elte.hu

ABSTRACT

CodeCompass is an open source LLVM/Clang based tool developed by Ericsson Ltd. and the Eötvös Loránd University, Budapest to help understanding large legacy software systems. Based on the LLVM/Clang compiler infrastructure, CodeCompass gives exact information on complex C/C++ language elements like overloading, inheritance, the usage of variables and types, possible uses of function pointers and the virtual functions – features that various existing tools support only partially. Steensgaard's and Andersen's pointer analysis algorithm are used to compute and visualize the use of pointers/references. The wide range of interactive visualizations extends further than the usual class and function call diagrams; architectural, component and interface diagrams are a few of the implemented graphs. To make comprehension more extensive, CodeCompass is not restricted to the source code. It also utilizes build information to explore the system architecture as well as version control information e.g. git commit history and blame view. Clang based static analysis results are also integrated to CodeCompass. Although the tool focuses mainly on C and C++, it also supports Java and Python languages.

In this demo we will simulate a typical bug-fixing work flow in a C++ system. First, we show, how to use the combined text and definition based search for a fast feature location. Here we also demonstrate our log search, which can be used to locate the code source of an emitted message. When we have an approximate location of the issue, we can start a detailed investigation understanding the class relationships, function call chains (including virtual calls, and calls via function pointers), and the read/write events on individual variables. We also visualize the pointer relationships. To make the comprehension complete, we check the version control information who committed the code, when and why.

This Tool demo submission is complementing our Industry track submission with the similar title. A live demo is also available at the homepage of the tool https://github.com/ericsson/codecompass.

KEYWORDS

code comprehension, C/C++ programming language, software visualization

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05.
https://doi.org/10.1145/3196321.3196352

ACM Reference Format:
Zoltán Porkoláb, Tibor Brunner. 2018. The CodeCompass Comprehension Framework. In *ICPC '18: 26th IEEE/ACM International Conference on Program Comprehension, May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3196321.3196352

1 INTRODUCTION

CodeCompass is a pluginable, extensible, open source code comprehension framework. Its main purpose is to help understanding large code bases by gathering static analysis information from the whole knowledge portfolio and to present it to the user through various navigation and visualization techniques.

The comprehension setup has two phases. First the source code has to be parsed to collect information and store them into a database. This phase optimally is integrated into the build process, so we can access not only to source but also many meta information of the system: macro settings, build instructions, version control database, etc. In the second phase a server presents the collected information to the clients on a public API. An example web-based GUI client is a standard web browser which queries the server from JavaScript, however it is also possible to create other clients for the users' own purposes, e.g. editor plug-ins. The main supported languages are C and C++. There is also a limited support for Java and Python source code. In this demo we will use C++ as the demonstration language.

The limitation of the tool is that we capture snapshot information on the system, but there exists many use such cases, like bug fixing of a certain version of the system or design a new feature, when all (or most of) the system can be considered stable. There is an ongoing effort to implement fast incremental reparsing for code modification.

In this demo session we present the main features of the tool presuming that the parsing phase is already done. We will use a standard web browser as a client. The source code and a tutorial of CodeCompass is publicly available on GitHub, from where a live demo is also available at https://github.com/ericsson/codecompass.

2 THE SEARCH TOOL

The web interface layout is divided into three main parts: the search panel on the top, the info tools on the left side and the main area in the middle of the browser panel.

Most code comprehension work starts with a fast feature location: we want to narrow down the whole system to the area where the features we are interested in are implemented. Many cases this is done by some text-based search arction. On the top panel of the web interface there is a free-text search bar. It consists of the following items: the search type selector, the search field and the file/directory filters. There are four search types:

Text search Choosing this option a simple traditional text-search functionality is available. This gives results in the source code, string literals, macro names, comments; everywhere the search pattern matches. We can use the usual wildcard notations and the **AND**, **OR** and **NOT** logical operators to combine search tags.

Definition search By this option only named entities (functions, variables, classes, etc.) of a specific programming language can be searched. As opposed the previous Text search, no matching will be detected in comments, string literals, etc. Note that this is still a text-based search, where a third-party tool (Lucene) is used for collecting these names and their locations.

File name search Many cases we know which files are interesting for us. With File name search we are looking for file and directory names anywhere in the software system by regular expressions.

Log search A fuzzy search which doesn't have to be an exact match. This is suitable e.g. for searching log messages which are copied from the terminal output. The advantage of this option is that it finds the output commands in the source code even if the printed message contains dynamically generated fragments, such as timestamps, line and column numbers or file names.

```
printf("[%s] There is an error in line %d in file %s",
     timestamp, lineno, filename);
std::cout << "[" << timestamp << "]" <<
     "There is an error in line " << lineno <<
     " in file " << filename;
```

Both of these statements will be found if searching for the string `[12:34] There is an error in line 42 in file hello.cpp`.

3 INFO TREE

The *InfoTree* is the main information area which is filled by the specific language parsers. When opening a source file and selecting a named entity all the known information are presented here in a tree structure as seen on Figure 1. The content of this panel depends on the type of the entity.

In case of functions one can check its (qualified) name, signature, place of definition and declarations. The parameters, local variables callers/callees and assignments to function pointers can also be listed. All these items in the tree can be clicked to ensure quick navigation on the given detail. The content of the callers' node is built recursively. Opening the subtree lists the caller functions of the selected one on the next level. All these nodes can be opened to get their callers too. This way the user can inspect the whole call chain of a function. This call chain also contains the possible calls via a function pointer. Although this is dynamic information, CodeCompass enables to list all possible places where there is an invocation on a pointer which is assigned this function earlier.

In case of a class the aliases (typedefs), inheritance relationships, friends, methods and members can be listed. The place of usages is also enumerated grouped by the nature of the usage, i.e. where the class was used as a global/local variable type, as a field of another class or as parameter/return type of a function.

For an enumerated type the enum constants can be listed with their numeric values. For variables one can check its type and the locations where the variable is written and read. Passing the variable by value is considered a read action, and passing by reference is considered a write. For macro constants one can fetch the place of its definition and the locations of its expansions and undefinitions. Not only named entities but source files can also be queried for further information. In the "File outline" panel the included (imported in Java) files, defined macros/functions/types can be listed for easier navigation.

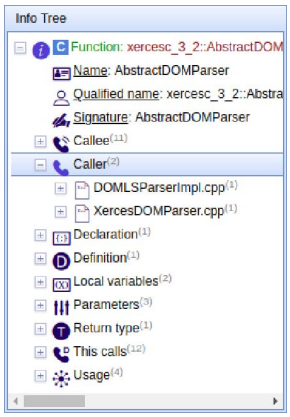

Figure 1: Info tree of a C++ function

4 SYSTEM ARCHITECTURE

The most basic module view is the class diagram. But besides this one CodeCompass can present even higher level information about the system architecture, as it uses all the build information available at compile time.

There are different relationships between the source, header and binary files. In the simplest case a .cpp file *uses* a header file if it uses a symbol declared in the header, but it can also *provide* it, when defines the names declaed in the hearder. There can be containment relation between binaries and source files: a binary may contain several source files if those are linked together in the build phase. Such a diagram can be seen on Figure 2.

Figure 2: System architecture diagram.

Using these relations we can define diagrams in order to review the *Software architecture*. For example one may want to check which module use a specific implementation. Querying the "component users" on a .cpp source file a graph based diagram shows up. In the first level we can see the source file, then the provided interfaces. On the next level there are the interface user .cpp files and the binaries in the end, which contain these user modules.

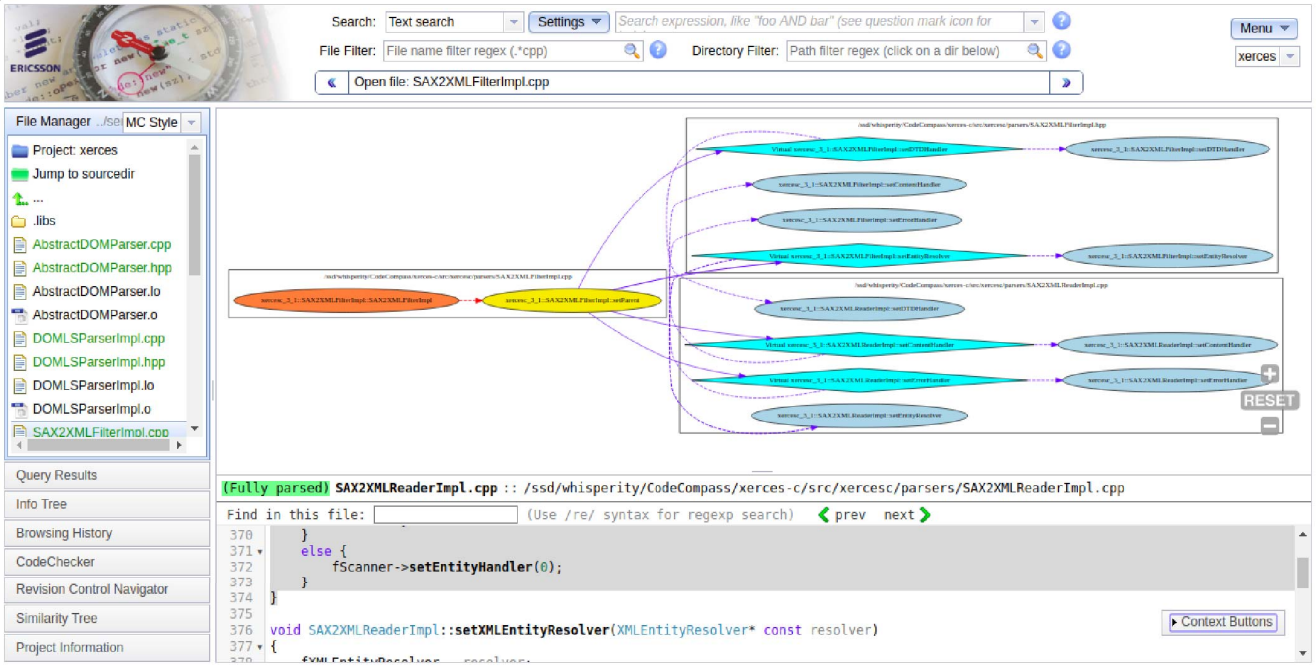

Figure 3: Function call chain diagram. The center yellow ellipse is the inspected function, on the left is the caller, on right the callees. Black rectangels represent files. All functions are clickable. At the bottom is the code of the inspected function.

5 FUNCTION CALL CHAIN

The program behavior coded in functions are the most basic means for decomposing a complex operation. When understanding a specific function, we inspect the bodies of other called functions as well. On a certain level of jumps through functions, one may easily loose the path on the call chain. This process is supported in CodeCompass by different visualizations which provide the whole path in a single view. We have already seen such a view in the "info tree", where function callers and callees could be listed recursively.

CodeCompass also provides a *Function call diagram* as seen on Figure 3. This is a graph based diagram, of which the nodes are functions, and there is an edge between two nodes if a function calls the other. In this diagram the virtual functions are also visualized. Even though this is a dynamic information, CodeCompass can present all overriding versions of that function so the programmer can be aware all of them.

6 AD-HOC ROAMING IN THE CODE

During code comprehension, ad-hoc browsing the code should be straightforward and cheap. As seen on Figure 4, our *CodeBites* interactive graph-based view encourages the user to enterprise new paths and connections in the code. The nodes of this graph are definitions – either a function, class, macro, variable. Clicking on such entity, its definition appears in a new box whether it is in the same file or not. Useless boxes can be closed at any time. With this try-and-error way the user can see all the related item definitions on a single picture and ignore the irrelevant information.

7 BROWSING HISTORY

When a programmer is intended to understand a source code, it is a common issue to browse the different parts of the code base. For reading through a function definition it is necessary to follow some function calls, to check the definition of classes or to inspect the behavior of certain objects. After a long navigating session the programmer may loose the path on which he got to the current code part.

To resolve this issue the *Browsing history* has been introduced. This view shows the direction of navigation in a tree structure. The nodes of the tree are file open actions and the children are "jump to definition" actions. This way one can always find the path back to the origin of the code browsing actions, since the "from" and "to" locations of the jumps are recorded. The browsing history tree is connected with the browser's own history, so the navigation back and forth in the browser is synchronized with the tree structure.

8 REVISION CONTROL NAVIGATOR

One of the main resources of knowledge about the source code is hidden in commit messages. That is the reason why CodeCompass is also equipped with a Git parser which gathers the *Git history* of the project. This means that not only the committed changes can be seen for a file, but for each line the author of that line is also visible. The color of the line number indicates how old the specific change is (see Figure 5). Clicking on the author's name one can inspect all changes of the commit with a short description in the commit message. Besides these information the traditional *Git branch view* is also available.

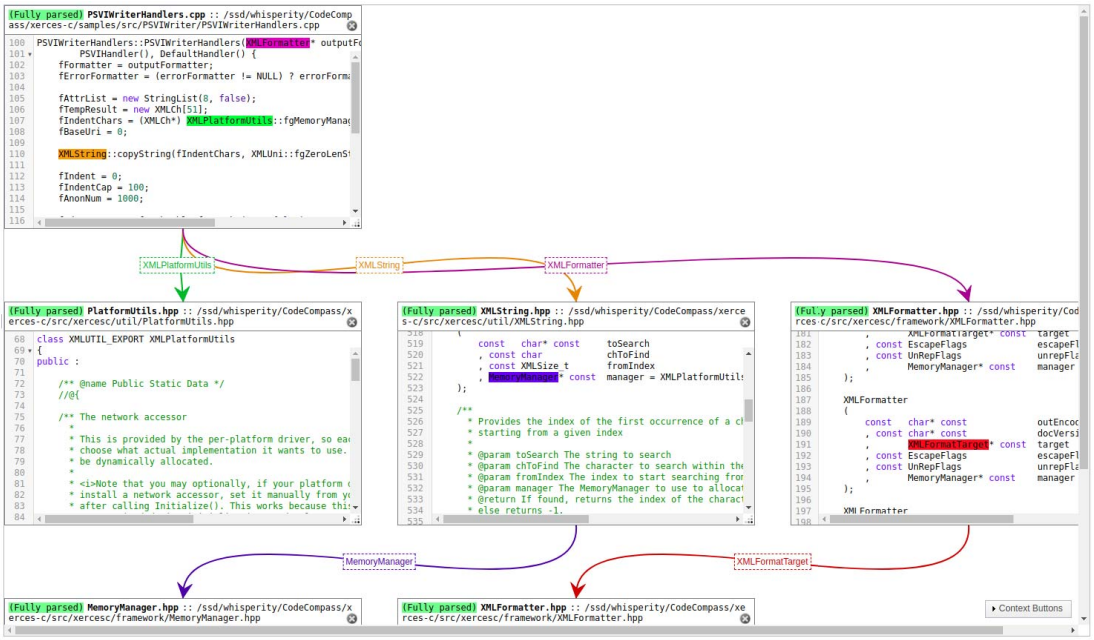

Figure 4: Ad-hoc browsing of the code with CodeBites view. Clicking on an element (either type, function call or variable) opens its definition in a new box. User can open and close boxes at any time.

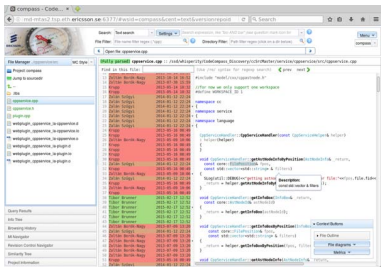

Figure 5: Git blame view

9 CODECHECKER INTEGRATION

CodeChecker is a good example how suitable CodeCompass is for integrating other tools. CodeChecker is an infrastructure around Clang Static Analyzer and Clang Tidy which aim to find bugs in the source code. These tools are developed as the part of the open source LLVM/Clang project. CodeChecker runs them and stores their results in a database. Typical programming issues like division by zero, null-pointer dereference, possible memory leaks and misuse of the "rule of three", etc. are found and reported for the users.

Clang Static Analyzer uses the powerful technique of symbolic execution. When the analyzer finds a potential bug the place of error is reported. In CodeCompass beside this point, the full execution path of the control flow is also visualized that shows, which assumptions were made to get the system there.

10 SOFTWARE METRICS

The metrics view can present measurement results about the code. The most classical metrics are the "lines of code" and the McCabe cyclomatic complexity. Furthermore any other number can be used which describes a source file, for example the number of bugs in a file found by CodeChecker.

This view gives a rectangular representation of the directory hierarchy. A rectangle belongs to each file and folder to show the selected metrics. There are two dimensions: size and color. Both of these can be assigned a selected metric. E.g. if "lines of code" is assigned to size dimension then the bigger rectangles represent larger files and folders.

11 CONCLUSION

In this demo session we demonstrate CodeCompass, a static analysis tool for comprehension of large-scale software. Having a web-based, pluginable, extensible architecture, the framework can be an open platform to further code comprehension, static analysis and software metrics efforts. The feedback from the hundreds of industrial users suggests that the tool is useful for developers in comprehension activities and it is used besides traditional IDEs and other cross-reference tools.

The source code and a tutorial of CodeCompass is publicly available on GitHub, from where a live demo is also available at https://github.com/ericsson/codecompass.

ACKNOWLEDGMENTS

This work is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).