CodeCompass
an Extensible Comprehension Framework

Tibor Brunner

Faculty of informatics, Eötvös Loránd University,
Budapest, Hungary
bruntib@caesar.elte.hu

Abstract. CodeCompass is an open source tool to help understanding large legacy software systems. Based on the LLVM/Clang compiler infrastructure, CodeCompass gives exact information on complex C/C++ language elements. The wide range of interactive visualizations includes class and function call diagrams; architectural, component and interface diagrams and "points to" diagrams and many others. CodeCompass also utilizes build information to explore the system architecture as well as version control information when available. Clang based static analysis results are also integrated. Although the tool focuses mainly on C and C++, it also supports Java and Python languages. Having a web-based, pluginable, extensible architecture, the CodeCompass framework can be an open platform to further code comprehension, static analysis and software metrics efforts.

1 Introduction

Bug fixing or new feature development requires a confident understanding of all details and consequences of the planned changes. Code comprehension tools can help to reveal the original intentions and implementation details by building a model from the source code and other available information. Although a number of such tools are available either as proprietary or free software, their feature set is limited.

CodeCompass was developed to eliminate these restrictions. The CodeCompass project is a joint open source effort of Ericsson Ltd. and the Eötvös Loránd University, Budapest to help understanding large software systems. To provide 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 – CodeCompass is based on a real compiler, the LLVM/Clang infrastructure. Thus, it eliminates the weaknesses of the usual "light-weight" comprehension tools, like OpenGrok.

CodeCompass, however, is not restricted to the source code. It uses the build information of the system to reveal architectural connections. It also employs the version control information if available, so one can identify connections between different source files "accidentally" modified in the same commit. To help

fast and precise perception CodeCompass uses both textual and graphical representation of the software system to comprehend. A number of (interactive) diagrams are accessible from the usual function call graphs to the unique architectural diagrams. To provide easy access for the users, CodeCompass has a web-based architecture. The client can be a standard web browser, an editor plug-in or any 3rd party application. The communication is based on a REST API and scales well for parallel client requests.

In this paper we will compare CodeCompass to existing comprehension tools and describe its feature set. In Section 2 we overview the main archetypes of existing tools for code comprehension. We introduce the extendible architecture of CodeCompass in 3. The main features of the tool are discussed in Section 4. We summarize the paper in Section 5.

2 Related work

On the software market there are several tools which aim some kind of source code comprehension. Some of them uses static analysis, others examine also the dynamic behavior of the parsed program. These tools can be divided into different archetypes based on their architectures and their main principles. On the one hand tools are having server-client architecture. Generally these tools parse the project and store all necessary information in a database. The (usually web-based) clients are served from the database. These tools can be integrated into the workflow as nightly CI runs. This way the developers can always browse and analyze the whole, large, legacy codebase. Also there are client-heavy applications where smaller part of the code base is parsed. This is the use case for IDE editors where the frequent modification of the source requires quick update of the database about analyzed results. In this section we present some tools used in industrial environment from each categories.

Woboq [3] is a web-based code browser for C and C++. This tool has extensive features which aim for fast browsing of a software project. The user can quickly find the files and named entities by a search field which provides code completion for easy usability. The navigation in the code base is enabled through a web page consisting of static HTML files. These files are generated during a parsing process. The advantage of this approach is that the web client will be fast since no on-the-fly computation is needed on the server side while browsing.

Hovering the mouse on a specific function, class, variable, macro, etc. can show the properties of that element. For example, in case of functions one can see its signature, place of its definition and place of usages. For classes one can check the size of its objects, the class layout and offset of its members and the inheritance diagram. For variables one can inspect their type and locations where they are written or read.

In C and C++ macros form a sublanguage which is evaluated in a precompilation step. This evaluation is a textual substitution of macro tokens which

means that the compilation phase works with another code than the original one. In Woboq, the final value of macro expansions can also be inspected.

A very handy feature of the tool is the semantic highlighting. By this feature the different language elements can easily be distinguished: the formatting of local, global or member variables, virtual functions, types, typedefs, classes, macros, etc. are all different.

Woboq can provide the aforementioned features because the information needed is collected in a real compilation phase. The examined project first has to be compiled and parsed by Woboq. The parsing is done by LLVM/Clang infrastructure which makes the whole abstract syntax tree available. This way all pieces of semantic information can be extracted with the same semantics the final program is to have. This also gives a disadvantage of the tool, namely Woboq can only be used for browsing C and C++ projects.

OpenGrok [4] is a fast source code search and cross reference engine. Opposed to Woboq, this tool doesn't perform deep language analysis, therefore it is not able to provide semantic information about the particular entities. Instead, it uses *Ctags* [5] for parsing the source code only textually, and to determine the type of the specific elements. Simple syntactic analysis enables the distinguishing of function, variable or class names, etc. The search among these is highly optimized, and therefore very fast even on large code bases. The search can be accomplished via compound expressions (e.g. `defs:target`), containing even wild cards, furthermore, results can be restricted to subdirectories. In addition to text search there is opportunity to find symbols or definitions separately. The lack of semantic analysis allows Ctags to support several (41) programming languages. Also an advantage of this approach is that it is possible to incrementally update the index database. OpenGrok also gives opportunity to gather information from version control systems like Mercurial, SVN, CSV, etc.

Understand [6] is not only a code browsing tool, but a a complete IDE. Its great advantage is that the source code can be edited and the changes of the analysis can be seen immediately.

Besides code browsing functions already mentioned for previous tools, Understand provides a lot of metrics and reports. Some of these are the lines of code (total/average/maximum globally or per class), number of coupled/base/derived classes, lack of cohesion [2], McCabe complexity [1] and many others. *Treemap* is a common representation method for all metrics. It is a nested rectangular view where nesting represents the hierarchy of elements, and the color and size dimensions represent the metric chosen by the user.

For large code bases, the inspection of the architecture is necessary. Understand can show dependency diagrams based on various relations such as function call hierarchy, class inheritance, file dependency, file inclusion/import. The users can also create their custom diagram type via the API provided by the tool.

In programming, the core concepts are common across languages, but there are some concepts which are interpreted differently in a particular language. Understand can handle $\sim$ 15 languages and can provide language specific in-

formation about the code e.g. function pointer analysis in C/C++ or package hierarchy diagrams in Ada.

Understand builds a database from the code base. All information can be gathered via a programmable API. This way the user can query all the necessary information which are not included in the user interface.

CodeSurfer [7] is similar to Understand in the sense that it is also a thick client, static analysis application. Its target is understanding C/C++ or x86 machine code projects. CodeSurfer accomplishes deep language analysis which provides detailed information about the software behavior. For example, it implements pointer analysis to check which pointers may point to a given variable, lists the statements which depend on a selected statement by impact analysis, and uses data flow analysis to pinpoint where a variable was assigned its value, etc.

3 The CodeCompass Architecture

In the previous section we have listed some aspects concerning the goals and architectures of code comprehension tools. Now we present where CodeCompass stands among these tools.

CodeCompass has a client-server architecture in which it presents the information gathered in a preceding parsing phase. The reason why this architecture was chosen comes from the goal of the tool. As opposed to code editors, CodeCompass has been planned to be a code comprehension tool. There are fundamental differences between these two use-cases. During code writing, programmers are manipulating only a few files at the same time. In code comprehension, however, it is needed to consider the sources of multiple modules through the code base. In editors code completion is one of the most useful features: the programmer doesn't want to remember all methods and fields of a class, but requires the editor to list these. In code comprehension the wide range of visualizations is needed in order to overview the relations of code parts. While editing the source, the programmer focuses only to a relatively small fragment of the code, like a function or a class. In code comprehension it is not only the low-level behavior of the functions, but their dependencies and effects are considered in the context of high-level module system.

The main user interface of CodeCompass is web-based. All the aforementioned visualizations and functionalities can be queried via a public API which is assigned to a server application. The web interface handles the use-cases that aim fast and handy browsing, inspection and comprehension tasks. However, CodeCompass is more than just a code browsing tool. It is also a framework, i.e. an extensible collector and presenter of static analysis processes. That is why the intention was not to create a client-heavy application which stores the analysis results on the client side, but being able to serve the various needs of users. This way it is possible to implement a script for example which collects the set of functions that form a closure by function call relation, thus specifying a coherent slice of the software.

Another design requirement of CodeCompass was to handle large-scale code bases and still answering user requests very fast, i.e. in terms of seconds at most. This is accomplished by storing all the least amount of information in a database which are sufficient to answer the requests. Since we intended to give precise results for the queries, a preceding parsing process is required. In the first we stored the whole abstract syntax tree of the source, but this resulted a 1:1000 ratio between the source code and the database size. However, it turned out that most cases the users are interested in named entities only (function, variables, classes, macros, etc.), so it was unnecessary to store anything else, such as control structures or other statements. Nonetheless, there are some tasks which require more than the stored information, like a slicing algorithm. If the user wants to see the effects of changing the value of a variable then state modifying statements have to be taken into account too. This requires the reparsing of the code on the fly.

4 CodeCompass features

In this section we will give an overview about the features available through the standard GUI. When describing language specific features, such as listing callers of a method, we will always assume the project's language to be C++ as that has the most advanced support in CodeCompass, but similar features are available for Java and Python.

4.1 Search

Probably the most fundamental use-case of a code comprehension tool is searching. One may search either for a file or source code. For finding source code elements the tool provides 3 different search possibilities:

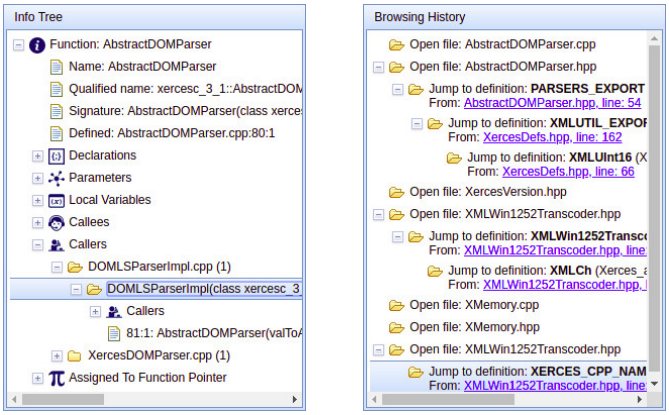
In *full text search* mode the search phrase is a group of words such as "returns an astnode*". A query phrase matches a text block, if the searched words are next to each other in the source code in that particular order. Wildcards, such as *, or ? can be used, matching any multiple or single character. Logical operators such as AND, OR, NOT can be used to join multiple query phrases at the same time.

On a higher level it is possible to find symbols in source codes by *definition search*. Here we are using CTags for indexing the code base thus being able to find variables, functions, classes, macros, etc. It is important to know that this language entity search has nothing to do with deep language parsing.

While debugging a program, sometimes the only information to start with is an output message in the console log emitted by our software. This is the only trace where one may start, e.g. `"DEBUG INFO: TSTHan: sys_offset=-0.019821, drift_comp=-90.4996, sys_poll=5"`. Note that such a message can contains timestamps or other dynamically generated fragments, so it is impossible to find this message as a direct string. However, in CodeCompass a fuzzy search can be done by *log search*.

4.2 Information about language symbols

When the element has been found, the next step is gathering information about it. The user can choose “Info tree” from the pop-up menu after selecting a named entity. This tree contains all information that is provided by a language parser. In case of C/C++ we are using the LLVM/Clang compiler in order to fetch information about the symbols.



(a) Info tree of a function (b) Browsing history

Fig. 1: Information collector panels

For functions we can check their parameters, local variables, callers and callees. An interesting feature of the tree is that the callers are presented recursively i.e. the children of a node are the callers of a function. Their children nodes are the callers of these functions, and this goes on recursively, theoretically back to the main function. However, function calls are not always direct, but can happen via function pointers. Even though this is a dynamic behavior, CodeCompass summons all the occurrences where a function was assigned to a function pointer and the invocation happens through this pointer.

In case of classes the collected information are the aliases (by `typedef` the class can have a synonym), inheritance relations (grouped by visibility), friends, methods/fields (direct or inherited) and usages (as local/global variable, function parameter/return type or field of another class).

For variables it is useful to know the places in the code where it was written and read. For enumeration types the enumeration constants are listed with their integer values.

4.3 Diagrams

Visualizations are one of the most helpful representations for humans to overview a system. CodeCompass presents several symbol and file based diagrams. These

diagrams are graph-based, i.e. they represent entities and their connections. These are also interactive diagrams: hovering the mouse over the nodes the represented entity is displayed in the text view, and by clicking them the selected entity becomes the center node showing its relations according to the diagram type.

Function call diagram shows all callers and callees of a function in a graph. *UML class inheritance* diagram shows the full inheritance chain up until the root base class and recursively for all derived classes. We have also implemented a *pointer analysis* diagram which shows the allocated objects and the pointers which possibly point to them. Of course this is a dynamic information which can only partly be collected in a static analysis.

An *interface diagram* called for a C/C++ source file shows which headers are "used only" or "implemented" by the given file. Usage means that a source file uses another file if there is a symbol usage in it which is declared in the other file. Implementation relationship means that a symbol is declared in a file (thus forming an interface) and defined in an other. These relations are also applicable for directories considering the contained files. In case of a compiled language there are also the output files like objects and executables. Based on linkage information we can present which sources make a binary file up.

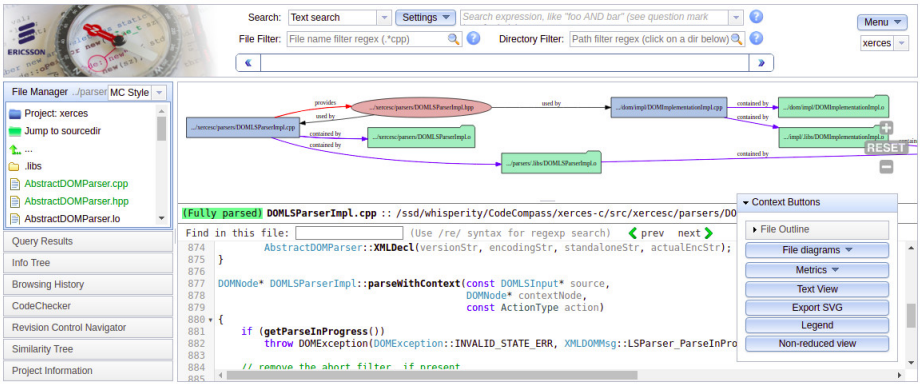


Fig. 2: Interface diagram

CodeBites provides a different visualization of the inspected source code. In this view the nodes of the graph are the definitions of specific named symbols, like classes, functions, etc. The idea is that a programmer would like to discover this entity by understanding its behavior but without loosing the focus. So the parts of the code text in a node are clickable which triggers the addition of the selected element's definition.

4.4 Version control visualizations

Visualization of version control information is an important aid to understand software evolution. Git *blame view* shows line-by-line the changes (commits) to a given file. Changes that happened recently are colored lighter green, while older changes are darker red. This view is excellent to review why certain lines were added to a source file. CodeCompass can also show Git commits in a filterable list ordered by the time of commit. This search facility can be used to list changes made by a person or to filter commits by relevant words in the commit message.

4.5 Metrics

CodeCompass can show the McCabe Cyclomatic Complexity [1], the lines of code and the number of bugs found by Clang Static Analyzer metrics for individual files and summarized over directory hierarchies. These metrics can be visualized on a tree map, where directories are indicated by boxes. The box size and its color shade is proportional to the chosen metric.

4.6 Browsing history

De Alwis and Murphy studied why programmers experience disorientation when using the Eclipse Java integrated development environment (IDE) [8]. They use visual momentum [9] technique to identify three factors that may lead to disorientation: i the absence of connecting navigation context during program exploration, ii thrashing between displays to view necessary pieces of code, and iii the pursuit of sometimes unrelated subtasks.

The first factor means that the programmer, during investigating a problem visits several files as follows a call chain, or explores usage of a variable. At the end of a long exploration session, it is hard to remember why the investigation ended up in a specific file. The second reason for disorientation is the frequent change of different views in Eclipse. The third contributor to the problem is that a developer, when solving a program change task, evaluates several hypotheses, which are all individual comprehension subtasks. Programmers tend to suspend a subtask (before finishing it) and switch to another. For example, the programmer investigates how a return value of a function is used, but then changes to a subtask understanding the implementation of the function itself. It was observed that, for a developer, it is hard to remind themselves about a suspended subtask [10].

CodeCompass implements a *browsing history* view which records (in a tree form) the path of navigation in the source code. A new subtask is represented by a new branch of the tree, while the nodes are navigation jumps in the code labeled by the connecting context (such as “jump to the definition of init”). So problem i) and ii) is addressed, by the labeled nodes in the browsing history, while problem iii) is handled by the branches assigned to subtasks.

4.7 CodeChecker - C/C++ Bug Reporting

Clang Static Analyzer implements an advanced symbolic execution engine to report programming faults. CodeCompass can visualize the bugs identified the Clang Static Analyzer and Clang Tidy by connecting it to a CodeChecker server [11]. CodeCompass shows the bug position, and the symbolic execution path that lead to a fault.

4.8 Namespace and type catalog

CodeCompass processes Doxygen documentation and stores them for the function, type, variable definitions. It also provides a *type catalog* view that lists types declared in the workspace organized by a hierarchical tree view of namespaces.

5 Summary

We presented CodeCompass, a static analysis tool for comprehension of large-scale software. It was designed to avoid the various shortages of the existing comprehension tools which are either lightweight, easy to use but without the deep knowledge of a real compiler; or heavyweight, non-scalable installed on the client machine. Having a web-based, pluginable, extensible architecture, the framework can be an open platform to further code comprehension, static analysis and software metrics efforts. Initial user feedback and usage statistics suggests that the tool is useful for developers in comprehension activities and it is used besides traditional IDEs and other cross-reference tools.

Acknowledgement

This paper is part of the Intellectual Output O2 of the Erasmus+ Key Action 2 (Strategic partnership for higher education) project No. 2017-1-SK01-KA203-035402: “Focusing Education on Composability, Comprehensibility and Correctness of Working Software”. The information and views set out in this paper are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

References

1. Thomas J. McCabe, *A Complexity Measure*, IEEE Transactions on Software Engineering: 308–320, December 1976
2. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity* Prentice-Hall, 1996, Upper Saddle River, NJ, ISBN-13: 978-0132398725
3. Woboq, https://woboq.com/codebrowser.html, 18. 03. 2018
4. OpenGrok, https://opengrok.github.io/OpenGrok, 18. 03. 2018

5. CTAGS, http://ctags.sourceforge.net, 18. 03. 2018
6. Understand, https://scitools.com, 18. 03. 2018
7. CodeSurfer, https://www.grammatech.com/products/codesurfer, 18. 03. 2018
8. B. De Alwis and G.C. Murphy, *Using Visual Momentum to Explain Disorientation in the Eclipse IDE*, Proc. IEEE Symp. Visual Languages and Human Centric Computing, pp. 51-54, 2006. [2] E. Baniassad and G. Murphy, Conceptual Module Querying for Software Engineering, Proc. Intl Conf. Software Eng., pp. 64-73, 1998.
9. D. D. Woods., *Visual momentum*, A concept to improve the cognitive coupling of person and computer. Int. J. Man-Mach. St., 21:229244, 1984.
10. D. Herrmann, B. Brubaker, C. Yoder, V. Sheets, and A. Tio. *Devices that remind*, In F. T. Durso et al., editors, Handbook of Applied Cognition, pages 377407. Wiley, 1999.
11. Daniel Krupp, Gyorgy Orban, Gabor Horvath and Bence Babati, *Industrial Experiences with the Clang Static Analysis Toolset*, EuroLLVM 2015 Confernece, April 2015