CodeCompass: An Open Software Comprehension Framework for Industrial Usage

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

Dániel Krupp, Márton Csordás
Ericsson Hungary Ltd.
Budapest, Hungary
[daniel.krupp,marton.csordas]@ericsson.com

ABSTRACT

CodeCompass is an open source LLVM/Clang-based tool developed by Ericsson Ltd. and Eötvös Loránd University, Budapest to help the understanding of 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 virtual functions - features that various existing tools support only partially. Steensgaard's and Andersen's pointer analysis algorithms 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 also utilizes build information to explore the system architecture as well as version control information.

CodeCompass is regularly used by hundreds of designers and developers. 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. The source code and a tutorial is publicly available on GitHub, and a live demo is also available online.

KEYWORDS

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

ACM Reference Format:
Zoltán Porkoláb, Tibor Brunner and Dániel Krupp, Márton Csordás. 2018. CodeCompass: An Open Software Comprehension Framework for Industrial Usage. In *ICPC '18: 26th IEEE/ACM International Conference on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3196321.3197546

1 INTRODUCTION

The maintenance of large, long-existing legacy systems is troublesome. During the extended lifetime of a system the code quality is continuously eroding, the original intentions are lost due to the fluctuation among the developers, and the documentation is getting unreliable. Especially in the telecom industry, high reliability software products, such as IMS signaling servers [1] have typically been in use for 20–30 years [2, 3]. This development landscape has the following peculiar characteristics: i) the software needs to comply to large, complex and evolving standards; ii) has a multiple-decade long development and maintenance life-cycle; iii) is developed in large (100+ heads) development organization; iv) which is distributed in multiple countries and; v) transfers of development responsibility occur from one site to the other occasionally. However, this software development landscape is not unique to the telecom industry and our observations can be applied at other industries, such as finance, IT platforms, or large-scale internet applications; all areas where complex software is developed and maintained for long time.

It is well-known, that in such a design environment, development and maintenance becomes more and more expensive. Prior to any maintenance activity – new feature development, bug fixing, etc. – programmers first have to locate the place *where* the change applies, have to understand the actual code to see *what* should be extended or modified, and have to explore the connections to other parts of the software to decide *how* to interact in order to avoid regression. All these activities require an adequate understanding of the code in question and its certain environment. Although, ideally the executor of the activity has full knowledge about the system, in practice this is rarely the case. In fact, programmers many times have only a vague understanding of the program they're going to modify. A major cost factor of legacy systems is the extra effort of comprehension. Fixing new bugs introduced due to incomplete knowledge about the system is also very expensive, both in terms of development cost *and* time.

As the documentation is unreliable, and the original design intentions are lost during the years and due to the fluctuation among the developers, the only reliable source of the comprehension is the existing code base.

Development tools are not performing well in the code comprehension process as they are optimized for writing new code, not for effectively browsing existing one. When creating new code, the programmer spends longer time working on the same abstraction level: e.g. defining class interfaces, and later implementing these classes with relationships to other classes. When one is going to understand existing code it is necessary to jump between abstraction levels frequently: e.g. starting from a method call into a different class we have to understand the role of that class with its complete interface, where and how that class is used, then we must drill down into the implementation details of an other specific method. Accordingly, when writing new code a few files are open in parallel

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

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3197546

in the development tool, while understanding requires a precise navigation through a large number of files.

In this paper we introduce **CodeCompass** [39] – an LLVM/Clang based open source tool developed by Ericsson and Eötvös Loránd University, Budapest, to help the code comprehension process of large legacy systems. The tool has been designed to be extremely scalable, seamlessly working with many million lines of code. Fast search options help locating the requested feature by text search, definition or even by emitted log message. Once the feature has been located, precise information on language elements like read and write access for variables, inheritance and aggregation relations of types, and call points of functions (including possible calls on function pointers and virtual functions) are provided by the LLVM/Clang infrastructure. Easy navigation possibilities and a wide range of visualizations extend far more than the usual class and function call diagrams help the user the more complete comprehension.

Code comprehension may not be restricted to existing code bases. Important architectural information can be gained from the build system, like relations between libraries, binaries and source files [7, 8]. To make the comprehension more extensive, CodeCompass also utilizes version control information, if available; Git commit and branching history, blame view are also visualized. Static analysis results produced by the Clang Tidy and Clang Static Analyzer to find possible erroneous constructions in the source are also integrated into the tool [30, 31]. Moreover some source based quality analytics software metrics are also measured.

For the sake of easy access for hundreds of developers, CodeCompass has a web-based architecture. The users can use it in a standard browser, however, an experimental Eclipse plugin has been also created using the open Apache Thrift [18] interface. CodeCompass currently supports systems written in C, C++, Java, and Python, but its pluginable infrastructure makes it extensible for new languages, visualizations or interfaces. Having this web-based, pluginable, extensible architecture, the CodeCompass framework is intended to be an open platform for further code comprehension, static analysis and software metrics efforts.

Our paper is organized as follows. In Section 2 we describe the specific problems which arise as a consequence of long-term, large-scale software development. We present CodeCompass in details in Section 3. Typical design and maintenance workflows are discussed along with their CodeCompass support in Section 4. Our experiences regarding the introduction and the generic acceptance of the framework is explained in Section 5, supported by usage statistics. Our paper concludes in Section 6 including some of our future plans.

2 NATURE OF THE ADDRESSED PROBLEMS

In this section we overview the challenges of large-software maintenance we experienced at Ericsson and collect the main requirements for a good comprehension tool.

2.1 Growing complexity

Telecom standards, such as the 3rd Generation Partnership Project (3GPP) IP Multimedia Subsystem (IMS) [22] are large (more than 1000 pages), complex and continuously evolving, so as the software that implements them. During a twenty years long development life-span, the size of the code base easily grows above ten million lines. With the influx of new features, software bugs are inevitably introduced, that needs continuous maintenance. As the size and complexity of the software increases, the amortized cost of bug fixing or adding new features also raises. This cost factor is due to the increased number of software components and dependencies. When a patch is introduced, the programmer needs to be cautious to avoid regressions which is much harder when implicit and invisible dependencies flood the system.

The business environment often requires an in-service, non-disruptive upgrade, therefore in many cases software components need to preserve backward compatibility, which also requires extra effort and has a negative impact on the system size.

The prime requirement towards a comprehension tool based on the above is that it should be scalable with regards to parsing and seamlessly work even more than 10 million lines of source code. It should be responsive, i.e. answering within 1 sec even on such a large code base, since any longer interruption diverts the attention of the programmer.

2.2 Knowledge erosion

The extended time of development causes serious fluctuation among the development teams' members. Knowledgeable developers who understand product use-cases, dataflow between components, and component dependencies are replaced by newcomers who need long learning curve to catch up and be near as efficient. In a multi-national environment transfer of the development activity from one site to another happens multiple times. At these occasions, the knowledge loss could be dramatic.

A program comprehension tool shall bolster novices in their learning process. The top-down method of knowledge transfer and information catch up is supported by high level architectural views such as graphical representation of source code structure, packaging structure and organized representation of documentation.

2.3 Multiple views of the software based on various information sources

Different design tasks, such as bug fixing, feature planning, or quality analysis require different views of the same software system. These views could not be created from the source code alone, they are synthesized from several other sources as well, such as *build process*, *revision control information* and *documentation*, *bug reports*.

While source code view is excellent for searching in the code and navigating the implementation, diagrams are more suitable for analyzing different types of static dependencies between language elements (such as call hierarchies, inheritance, interface usage patterns). Visualizing dependencies between source files, object files and executable libraries can help planning an upgrade procedure of a change.

Also, the history of the project can tell a lot about evolution of the system. Files regularly coming up together in commits may imply deeper connections. Recent changes in the code may point to the source of freshly reported bugs. If static analysis results are available on the system, they can give hints about the possible issues related to the source under investigation.

The tool should support the comprehension from the micro to the macro level. On the micro level, navigation between symbols, on the macro level, component dependency analysis or visualization of metrics is to be integrated in a single workflow.

2.4 Communication barriers

The possibility of communication inefficiency between teams located at different offices is quite high in a distributed development environment. When an incoming bug report arrives, a slow negotiation process starts between component development teams, sometimes blaming each other for the reason behind the fault. This inefficient process is partly due to the fact that they do not have a precise understanding about the actual, and the intended behavior of each others' components, and they cannot reason about that behavior efficiently via email or other communication channels.

The comprehension tool shall support knowledge sharing and team work, for example designers should be able to share program comprehension flow (e.g. currently examined file, position or diagram) with each other.

2.5 Multiple programming languages

Large software systems are rarely implemented in a single programming language. Core functionality, where efficiency is at utmost importance, may be implemented in C or C++. User interface may be written in Java (or even in JavaScript). There are usually service scripts around written in Perl, Python, Bash or other script languages. Third party open source components are also implemented, increasing the amount of different programming languages being used.

Naturally, the comprehension tool should support various languages within the same framework. The interface should be similar, but in the same time language specific features should also be reflected. This alloys an increased level of usability with the user-friendly approach of an established tool in the teams' workflow. It is even better, if the tool supports the *connections* between modules written in different languages. For these reasons the software comprehension tool should support multiple programming languages and should be extensible with new languages with limited effort.

2.6 Hard to deploy new tools

According to our experience it is difficult to convince developers to use new tools. Especially if the tool requires clumsy configuration or does not have an intuitive and responsive interface engineers tend to see it as a barrier to their work and give up its usage very soon.

The comprehension tool should have an intuitive and responsive user interface and should be easy to install and use.

2.7 Requirement of open extensibility

When a software product is planned for long term development, domain specific languages are considered to describe the domain knowledge in a simple and compact manner [4]. DELOS language of the TelORB real-time operating system [32] is one of the DSLs widely used in the telecom industry. The comprehension tool should be easily extensible to parse such DSLs or proprietary languages in a pluginable manner even by third party developers.

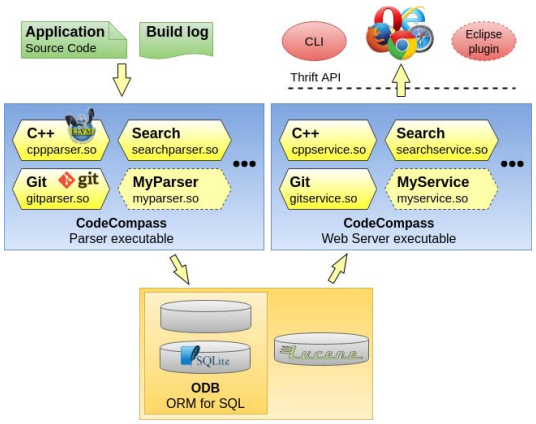

Figure 1: CodeCompass architecture – Service interfaces are remotely accessible; new parsers and services can be added as plugins

Moreover, it should reveal how artifacts in one language are mapped into other languages. CORBA interface definition language (IDL) [5], for example, is mapped into a client and a server function in the generated (C/C++/Java) code. We were looking for a comprehension tool that can be seamlessly follow these mappings from DSL to host or generated code.

3 CODECOMPASS

3.1 Architecture

CodeCompass [39] provides a read-only, searchable and navigable snapshot of the source code, rendered in both textual and graphical formats. CodeCompass is built with a traditional server-client architecture as depicted in Figure 1. The server application provides a Thrift [18] interface to clients over HTTP transport. The primary client that comes pre-packaged with the tool is a web browser based single-page HTML application written in HTML and JavaScript.

Since the interface is specified in the Thrift interface definition language, additional client applications (such as a command line client or an IDE plugin) can be easily written in more than 15 other languages supported by Thrift (including C/C++, Java, Python etc.). An experimental Eclipse plugin is already implemented.

A parsed snapshot of the source code is called a *workspace*. A workspace is physically stored as a relational database instance and additional files created during the *parsing process*. The parsing process consists of running different *parser plugins* on the source code. The most important parser plugins are:

A **search parser** iterates recursively over all files in the source folder and uses Apache Lucene [19] to collect all words from the source code. These words are stored in a search index, with their exact location (file and position).

The **C/C++ parser** iterates over a JSON compilation database [20] containing build actions, using the LLVM/Clang parser [21] and stores the position and type information of specific AST nodes in the database. This database will be used by the *C/C++ language service* to answer Thrift calls regarding C/C++ source code.

The **Java parser** iterates over the same JSON compilation database containing build actions, using the Eclipse JDT parser [33] and stores the position and type information of specific AST nodes in the database. This database will be used by the *Java language service* to answer Thrift calls regarding Java source code.

CodeCompass has an extensible architecture, so new parser plugins can be written easily in C/C++ language. Parser plugins can be added to the system as shared objects.

On the webserver, Thrift calls are served by so-called *service plugins*. A service plugin implements one or more Thrift services and serves client requests based on information stored in a workspace. A Thrift service is a (remotely callable) collection of method and type definitions. All Thrift services have one implementation with the exception of the *language service*, which is implemented for C/C++, Java and Python. The *language service* is distinct in the sense that it provides the basic code navigation functionality for the languages it is implemented for. To put it simply, if this interface is implemented for a language, the user will be able to click and query information about symbols in the source code view of a file written in the given language.

The most important services are the following:

Language service: This service provides query methods for symbols, for source files and globally for the workspace. It can return the symbol for a given source position, can return the corresponding node type and the corresponding source code fragment and documentation, the references to the symbol, diagrams – in GraphViz Dot format – or all information about the node in a tree format.

For files, the service can return file level diagrams (in GraphViz Dot format), references to the file from other files, extended information about the file in a tree format, syntax highlighting data. This extended information includes, among others, all type and function definition within the file, list of files this file is referred to and referred from.

The service can also return all defined symbols in a workspace (such as namespaces, types, functions), so symbol catalogs can be implemented.

Search Service: This service provides 4 different type of queries: search in *text* for words, search among *symbol definitions* only, search among *file names* and suggest search phrases, based on a search *phrase fragment*. Text, definition and file search can be filtered by file name and containing directory. Definition search can be filtered for a certain language (C/C++, Java, JavaScript, Python, etc.) and symbol type (function definition, type definition, variable declaration, etc.)

Developers can add additional service plugins delivered as shared objects in run-time.

3.2 Web User Interface

The web-based UI is organized into a static *top area*, extensible *accordion modules* on the left and also extensible *center modules* on middle-right – see Figure 4.

The source code and different visualizations are shown in the center, while navigation trees and lists, such as file tree, search results, list of static analysis (CodeChecker) bugs [31], browsing history, code metrics and version control navigation is shown on the left. New center modules and accordion panels can be added by developers.

The top area shows the search toolbar, the currently opened file, the workspace selector, simple navigation history (breadcrumbs) and a generic menu for user guides.

3.3 Functionality

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.

3.3.1 Search. The tool provides 4 different types of search possibilities: *full text search*, *definition search*, *filename search* and *log search*.

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.

Definition search has the same syntax as full text search, but it only queries among symbol definitions. Symbol definitions are recognized for the following languages: Java, C/C++, Perl, JavaScript, Python. Symbol definitions can be further restricted to functions, constants, types, fields, labels or macros.

It is possible to find locations in the source code where specific output are emitted using *log search*. One can simple use the output message as it is as the input of the search, like the following:
`DEBUG INFO: TSTHan: offset=-0.019, drift=-90.49, poll=5`
The search will identify the possible source locations ranking by the likelihood regarding to emit the message. In the example above, the search could identify the following line as the possible emission site: `debuginfo("offset=%s, drift=%s, poll=%d", lfptoa(&offset, 6), fptoa(drift, 4), sys_poll();`
Note that there is no need to rewrite the log entry into a regular expression.

3.3.2 Information about language symbols. In the source code view, the user can click on any symbol and get additional information about it or generate a diagram about its usage. The *Info tree* gives the most concise information about a symbol. For an example see the info tree of the `void DeekTimerHandler::tick()` function in Figure 2. You can read that the function is called in the `DeekTimer.cc` file from the `dispatcher()` function. The function `DeekTimerHandler::tick()` is also assigned to a function pointer in the source file `CoecuBoardSupport.cc`, in line 304.

For C/C++ variables, the tool lists location of reads, writes, and aliases of the variable (references and pointers pointing to the same memory location). For classes, one can query the definition, base classes, derived classes, methods, members, and how the class is used: in a declaration of a local, global, member variable, function parameter, or as return type. Regarding macros, all expansion locations and values can be listed.

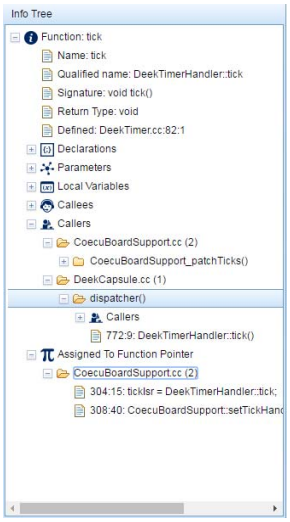

Figure 2: Info tree of C++ function `DeekTimer::tick()`

3.3.3 Symbol level diagrams. Sometimes a diagram representation of a symbol and its environment can be very helpful for comprehension. CodeCompass provides the interactive *CodeBites* diagram for understanding large call chains (for an example see Figure 4) and type hierarchies. *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 baseclass and recursively for all derived classes. *Class collaboration* diagram shows base inheritance chain and recursively shows all member classes as nodes. *Pointer aliases* diagram shows all references and pointers (as nodes) that refers to the same memory location as the queried variable.

3.3.4 File and directory level diagrams. It is possible to generate diagrams for files and diagrams. These graphs give a higher level architectural view of the system. *UML class overview* diagram of a file or a whole directory shows the inheritance chain and direct member classes of all classes declared within that file or directory. An *Interface diagram* called for a C/C++ source file shows which headers are "used only" or "implemented" by the given file.

CodeCompass directory diagrams introduce two relationships between directories containing C/C++ code: *depends on*, *implements*. A directory `A` *"depends on"* directory `B`, if `B` contains a file that declares a symbol which is used by a file in `A`. A directory `A` *"implements"* directory `B`, if `B` contains a file that declares a symbol which is defined by a file in `A`. *Users of a module* can be called for a directory. This diagram shows all directories that are in *"depends on"* relationship with the queried directory. *Internal Architecture* diagram can be called for a directory and shows the "depends on" and "implements" relationships between the subdirectories of the queried directory. You can find the details in [23].

3.3.5 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.

3.3.6 Code Metrics. CodeCompass can show the McCabe Cyclomatic Complexity [24], the lines of code and the number of bugs found by Clang Static Analyzer as quantities either for individual files or summarized over directory hierarchies. These metrics can be visualized on a treemap, where directories are indicated by boxes. The box size and its color shade is proportional to the chosen metric [26].

3.3.7 Browsing history. De Alwis and Murphy studied why programmers experience disorientation when using the Eclipse Java integrated development environment (IDE) [27]. They use visual momentum [28] 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 [29].

CodeCompass implements a *"browsing history"* view which records (in a tree form) the 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.

3.3.8 CodeChecker - C/C++ Bug Reporting. CodeCompass can visualize the bugs identified the Clang Static Analyzer and Clang Tidy by connecting it to a CodeChecker server [31].

Clang Static Analyzer implements an advanced symbolic execution engine to report programming faults. Among others, it can report null pointer dereference, resource leak, division by zero faults. CodeCompass shows the bug position, and the symbolic execution path that lead to a fault.

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

Table 1: Performance of CodeCompass *v4* release

	TinyXML 2.6.2	Xerces 3.1.3	CodeCompass v4	Internal Ericsson product
Source code size [MiB]	1.16	67.28	182	3 344
Search database size [MiB]	0.88	37.93	139	7 168
PostgreSQL db size [MiB]	15	190	2 144	7 729
Original build time [s]	2.73	361.77	2 024	—
Parse time [s]	21.98	517.23	6 409	—
Text/definition search [s]	0.4	0.3	0.43	2
C++ *Get usage of a type* [s]	1.4	2	2.3	3.1

3.4 Performance

CodeCompass scales well regarding the size of the analyzed code in parsing time, size of the data stored and response times of the webserver. We summarized the results on four different C/C++ applications in Table 1.

The parsing time is proportional to the compilation time – one can expect the parsing time to be approximately 160% of the original build time. We decided not to store the whole AST generated by the parsers as that could easily grow by thousand times by the original source size. Instead, we mostly store only *named* entities: class, function and variable definitions and appearances. Thus, the disk space needed to store a parsed snapshot is proportional to the number of symbol declarations and references. According to our measurements, the size of a *workspace* (including search database and the relational database) is approximately $5 - 10$ times the size of source code.

It can be read from Table 1 that the response times of search and C++ symbol usage query remains low even in case of large (3.2 GiB) source code size.

4 IMPORTANT DESIGN WORKFLOWS

In this section we show the most common design tasks which emerge during the maintenance and evolution of large programs, and highlight how static program comprehension techniques and specifically CodeCompass help to solve them.

In [6] the authors collected and categorized typical questions programmers ask during a change task. They identify 4 categories: i) *Finding focus points*: finding locations in the program code implementing a behavior; ii) *Expanding focus points*: exploring relations of interesting types, functions, variables; iii) *Understanding a subgraph*: "How a set of types or functions collaborate in run-time?"; iv) *Questions over groups of subgraphs*: "How parts of the program relate to each other?" We will use these categories to identify the intent of the programmer during the analyzed tasks.

4.1 Bug investigation

The ultimate goal of the developer is to understand *the minimum amount of code to change that is necessary to solve the bug, but enough not to introduce additional bugs.* This task typically starts by identifying a program location where the problematic behavior was observed [37], then gradually restoring the actual program state and call path leading to the fault. Finally planning code changes, considering how the rewritten program text affects other, non-faulty use-cases. This means that the designer needs to verify how the changed function was called, how a changed type or variable is used in other parts of the system.

A bug investigation process starts based on a trouble report, which is a written description of the unwanted behavior of the program. This textual description can be accompanied by the following additional artifacts: i) In case of a program crash, a full core dump. A core dump contains the full content of the stack and the heap. Thus the full call chain and the values of the variables are known at the point of the crash. ii) In live systems, full core dumps are usually disabled, as it eats up disc space and takes extensive amounts of time to create. A log of the stack state may still be available, which contains the function call chain up to the crash, but not the values of the variables. iii) If none of the above is present, a log may be available which is a sequence of arbitrary printouts from the time period around when the problem occurred.

The developer first identifies the program point when the problematic behavior was observed. If the exact function is known (from the stack trace of core dump), *function definition search* (Section 3.3.1) can be used to locate the function. If only logs are available *log search* can be used to look up the program point where the logs were created (Section 3.3.1).

After locating "focus points", the developer identifies the actual execution flow and values of the variables that lead to the error. To understand complex call chains, *CodeBites* visualization of CodeCompass can be very helpful (see Figure 4). Function callers (and callees) can be recursively listed, using the *call chain explorer* in the *info tree* (see Section 3.3.2). Analyzing call chains, especially identifying callers through function pointers (in C or C++), or polymorphic calls through virtual functions (in C++) is really difficult using a traditional IDE. Similarly, when one would like to discover the write locations of a variable, and the variable is written through a pointer or a reference, usual IDEs are of little help. It can be really time consuming for a programmer to discover these non-trivial connections manually, so a comprehension tool can save a considerable amount of time. CodeCompass can detect variable aliasing and also caller identification through virtual functions and function pointers (see Section 3.3.2). Pointer aliasing diagram shows variable aliasing in a graph form (Section 3.3.3). To understand how the involved types are used, CodeCompass can show the types are referred to at various program locations (Section 3.3.2). When navigating among function calls, disorientation is often a problem [27]. CodeCompass organizes browsing track record in a browsing history tree view (Section 3.3.7).

When investigating the reason behind a bug, it is often useful to check version control history, since according to [9] it is likely that the investigated issue is introduced by another, earlier bug fix. Using the *blame view* (described in Section 3.3.5), the programmer can visualize the recent code modifications, that affected the file where the bug occurred.

CodeCompass can show faulty programming constructs (memory leaks, null pointer dereferences, etc.) that Clang Static Analyzer and Clang Tidy detected (see Section 3.3.8) in the current file or in the whole analyzed source code. These are worth checking as the investigated fault may be related to well known faulty programming patterns.

When planning changes, it is vital to understand the wider context of the change, how the altered parts interact in different usage scenarios. To understand mapping of domain concepts to implementation, the Doxygen [13] or Javadoc documentation can be of great help (Section 3.3.9). Interactions of classes can be explored on the collaboration diagrams (Section 3.3.3). This visualization is available for a single class, a single header file containing multiple classes, or a directory containing header files. To get wider usage context, directory-level dependency diagrams visualize how C/C++ header files in a directory are implemented or used by files in other directories (Section 3.3.4).

There are some additional static analysis techniques that are useful for a program change task, but currently lack support in CodeCompass. A (backward) slicing feature could help to understand which statements have effect on the examined program value. A dataflow analysis could show, how a given variable gets its value assigned.

4.2 Feature development planning and estimation

When planning new features, the designer first locates those files, where related features are implemented. CodeCompass provides text search (see Section 3.3.1) for this purpose. In the next step, files are identified, which may be affected by the change. Clustering techniques, such as the mapping metaphore, implemented by CodeSurveyor [10] helps to identify group of files that are closely related. CodeCompass does not support software maps, but it shows relationship between files and directories (based on C/C++ symbol usage information) in the internal architecture diagram (3.3.4) and interface diagrams (3.3.4).

Binary dependency views can be used to estimate which binaries will be affected by the changes and thus can help in planning an upgrade procedure.

4.3 Knowledge transfer and newcomers' catch-up

In modern software development environment fluctuation among the team members is increasing. When a new engineer joins the team or when the whole maintenance responsibility is moving to an other team, someone who is completely new to the code and the domain needs to acquire knowledge as fast as possible. Domain concepts are best understood from text books or articles. However, it is crucial to understand how these domain concepts map down to a particular software's implementation. Design documentation is a useful apparatus for describing this mapping. According to our experience there are three important rules need to be kept to have an up-to-date design documentation for large systems:

- store design documents in the same version control system as the source code
- split design documents in the same modularization as the source code
- use text based documentation system instead of binary format

Design documents that are not stored in the same version control system as the source code tend to get out of sync of the real implementation. Symbol names and file names referred to in these documents change as the code evolves and programmers tend to neglect patching these in the documentation. Binary formats, such as Microsoft Word documents should be avoided, as it is more convenient for a programmer to apply the same text editors and diff tools for documentation they use for coding.

According to our experience Doxygen [13] is a versatile tool for documenting source code on function, type, and file level. If the source code is organized into source code level components such as suggested in [11] and in [12] component level documentation can also be written. In CodeCompass we implemented the rendering of symbol, file and component level documentation for Ericsson internal products (see 3.3.9).

5 USER ACCEPTANCE IN PRODUCTION

CodeCompass was introduced in Ericsson as an optional tool presented on internal conferences and short trainings. There were no mandatory policy to use it and the earlier available tool alteratives are still accesible for the users. A year after CodeCompass has been deployed for use for a dozen products, we conducted a questionnaire poll regarding its usage. We asked more than 400 possible users on their experiences on voluntary base, and received about 50% answer rate.

We observed that at that moment above 2 million lines of code was parsed by CodeCompass, and from those who answered 40% of the developers use CodeCompass at least two times a month and about 15% use it on a day-to-day basis. It was an interesting result, that OpenGrok, the earlier de-facto comprehension tool in the company was still used by a few teams for some lower level tasks while for some actions required higher abstraction tool usage was rare.

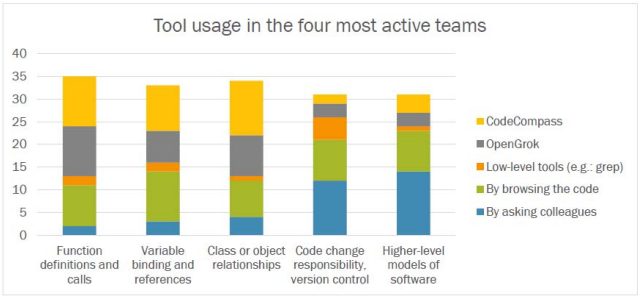

Figure 3: CodeCompass usage distribution per task

Figure 3 shows the distribution of the tools used by the respondents to solve specific tasks. In the questionnaire, we conducted the investigation of:

- *Function definitions and calls*: this basically means reference-based navigation in the code
- *Variable binding and references*: like the previous, but in the case of variables
- *Class or object relationships*: information clearly visible in higher-level models
- *Code change, responsibility*: version control information
- *Higher-level models of software*: component diagrams, domain specific architecture views, etc.

As the diagram shows, CodeCompass is mostly used to uncover and follow function and variable references, as well as to inspect class relationships; most probably users apply CodeCompass to these tasks because of its thorough static analysis that other tools (e.g. low-level search and OpenGrok) cannot perform. The proportions of "browsing the code" and "asking colleagues" are also worth noting; many employees try to solve complex problems by struggling on the code or by bothering others, rather than taking the right comprehension tool.

We are planning a controlled user study where we can analyse the effect of using CodeCompass on the software maintenance efficiency, and also we can compare workflows using the various alternative tools.

6 CONCLUSION & FUTURE WORK

We presented CodeCompass [39], 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. 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.

The most obvious shortage of the current version is that it presents a *snapshot* view of the software system. This is useful when a certain version of the system is maintained, but can be problematic when a large portion of the code is changed or newly written. We have plans for implementing *incremental parsing* of the changed code to better support the comprehension during active development.

The Language Server Protocol (LSP) is an initiative by Microsoft corp. and others to connect an editor or IDE to a *language server* that provides complex navigation and comprehension features [38]. Modifying our open Thrift interface according to LSP requirements could connect CodeCompass to various existing editors and IDEs. In fact, this way CodeCompass could act as a language server itself.

A live demo of CodeCompass exploring the Xerces XML parser library is available via the CodeCompass home page [39].

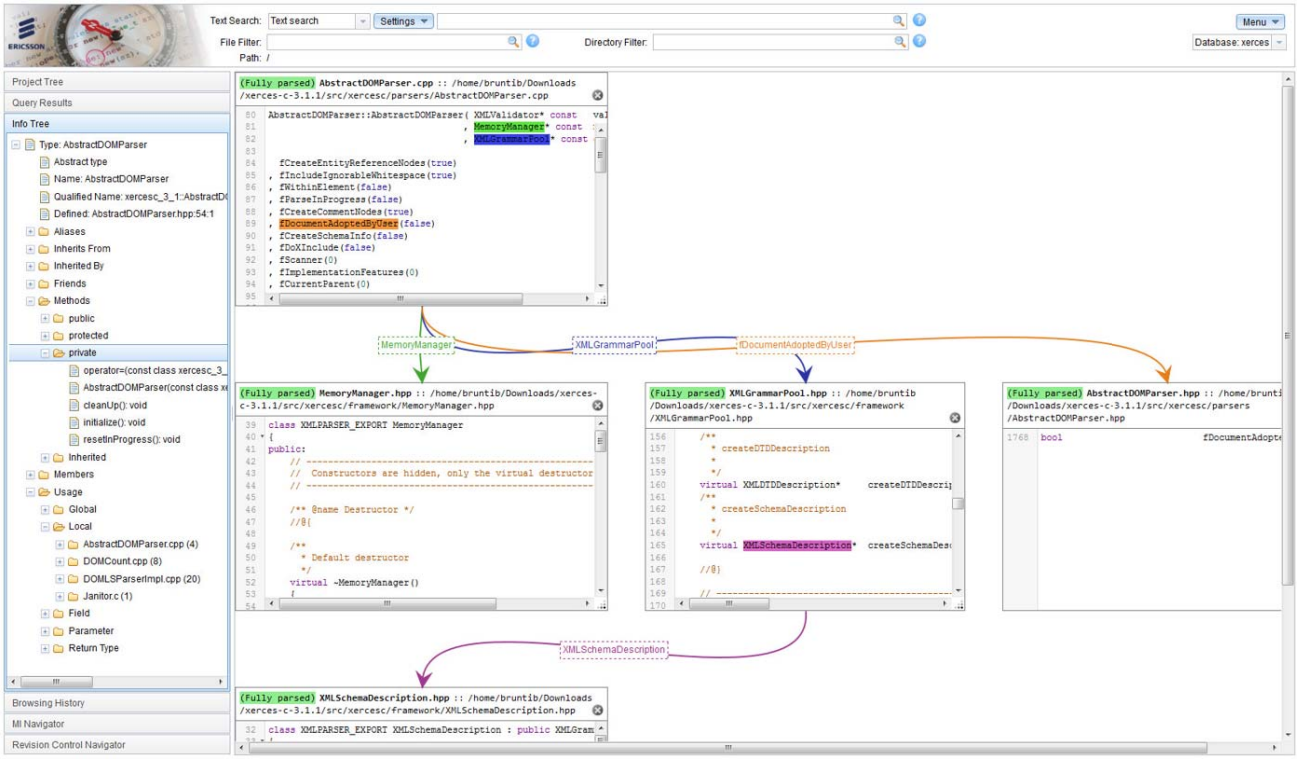

Figure 4: User interface running in a browser. Clicking an element (either type, function call or variable) opens its definition in a new box.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Ferraro-Esparza, Victor, Michael Gudmandsen, and Kristofer Olsson, *Ericsson telecom server platform 4*, Ericsson Review 3 (2002): 104-113.
[2] Enderin, Magnus, LeCorney, D., Lindberg, M., and Lundqvist, T., *AXE 810-The evolution continues*, Ericsson Review 4 (2001): 10-23.
[3] Karlsson, Even-André, and Lars Taxen, *Incremental development for AXE 10*. Software Engineering – ESEC/FSE'97. Springer Berlin Heidelberg, 1997. 519-520.
[4] Krzysztof Czarnecki and Ulrich W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, ACM Press/Addison-Wesley Publ. Co., New York, NY, USA, 2000.
[5] Object Management Group, OMG, *The Common Object Request Broker, architecture and specification*, Ver. 2.4.2, OMG Doc. formal/01-02-33. Framingham, Mass., Feb. 2001.
[6] Jonathan Sillito, Gail C. Murphy, Kris De Volder, *Asking and Answering Questions during a Programming Change Task*, IEEE Transactions on Software Engineering, VOL. 34, NO. 4, July/August 2008
[7] Szalay, R., Porkoláb, Z., Krupp, D, *Towards Better Symbol Resolution for C/C++ Programs: A Cluster-Based Solution*, In IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), Shanghai, China, 2017, pp. 101-110. doi:10.1109/SCAM.2017.15, 2017
[8] Szalay, R., Porkoláb, Z., Krupp, D, *Measuring Mangled Name Ambiguity in Large C/C++ Projects*, In: Zoran Budimac (Ed), Proceedings of the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications. Belgade, Serbia, 2017. CEUR-WS.org, Vol. 1938. Paper 17. 9 p. http://ceur-ws.org/Vol-1938/paper-sza.pdf
[9] Sunghun Kim, Thomas Zimmermann, Kai Pan, E. James Whitehead, Jr., *Automatic Identification of Bug-Introducing Changes*, 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06),18-22 Sept. 2006
[10] Nathan Hawes, Stuart Marshall, Craig Anslow, *CodeSurveyor: Mapping Large-Scale Software to Aid in Code Comprehension*, 2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT) , 27-28 Sept. 2015
[11] Adam Darvas, Raimund Konnerth, *System Architecture Recovery based on Software Structure Model*, 13th Working IEEE/IFIP Conference on Software Architecture, 2016, pp. 109-114
[12] Merijn de Jonge, *Build-Level Components*, IEEE Transactions on Software Engineering, VOL. 31, NO. 7, JULY 2005
[13] Doxygen, http://www.stack.nl/ dimitri/doxygen/, 18.03.2018
[14] Woboq, https://woboq.com/codebrowser.html, 18.03.2018
[15] OpenGrok, https://opengrok.github.io/OpenGrok, 18.03.2018
[16] CTAGS, http://ctags.sourceforge.net, 18.03.2018
[17] Understand, https://scitools.com, 18.03.2018
[18] Apache Thrift, https://Thrift.apache.org/, 18.03.2018
[19] Apache Lucene, https://lucene.apache.org/core/, 18.03.2018
[20] JSON compilation database format specification, http://clang.llvm.org/docs/JSONCompilationDatabase.html, 18.03.2018
[21] `clang`: a C language family frontend for LLVM, http://clang.llvm.org/, 18.03.2018
[22] 3GPP technical specifications, http://www.3gpp.org/specifications/79-specification-numbering, 18.03.2018
[23] Máté Cserép, Dániel Krupp, *Visualization techniques of components for large legacy C/C++ software*, Studia Univ. Babes Bolyai, Volume LIX, Special Issue 1, 2014 10th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, May 21-25, 2014
[24] Thomas J. McCabe, *A Complexity Measure*, IEEE Transactions on Software Engineering: 308âĂŞ320, December 1976
[25] Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity* Prentice-Hall, 1996, Upper Saddle River, NJ, ISBN-13: 978-0132398725
[26] Brunner, T., Porkoláb, Z., *Two Dimensional Visualization of Software Metrics*, In: Zoran Budimac (Ed), Proceedings of the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications. Belgrade, Serbia, 2017. CEUR-WS.org, Vol. 1938. Paper 2. 6 p. http://ceur-ws.org/Vol-1938/paper-bru.pdf
[27] 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. E. Baniassad and G. Murphy, "Conceptual Module Querying for Software Engineering," Proc. IntâĂŹl Conf. Software Eng., pp. 64-73, 1998.
[28] D. D. Woods., *Visual momentum*, A concept to improve the cognitive coupling of person and computer. Int. J. Man-Mach. St., 21:229-244, 1984.
[29] 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 377-407. Wiley, 1999.
[30] Clang Static Analyzer, http://clang-analyzer.llvm.org/, 18.03.2018
[31] Daniel Krupp, Gyorgy Orban, Gabor Horvath and Bence Babati, *Industrial Experiences with the Clang Static Analysis Toolset*, EuroLLVM 2015 Confernece, April 2015
[32] Lars Hennert and Alexander Larruy, *TelORB - The distributed communications operating system*, Ericsson Review No. 3, 1999
[33] Eclipse Java development tools (JDT), http://www.eclipse.org/jdt/, 18.03.2018
[34] Apache Xerces 3.1.3, http://xerces.apache.org/xerces-c/, 18.03.2018
[35] TinyXML, https://github.com/leethomason/tinyxml2, 18.03.2018
[36] CodeSurfer, https://www.grammatech.com/products/codesurfer, 18.03.2018
[37] J. Sillito, K. De Voider ; B. Fisher ; G. Murphy, *Managing software change tasks: an exploratory study*, 2005 International Symposium on Empirical Software Engineering, 2005.
[38] The official page for Language Server Protocol, https://microsoft.github.io/language-server-protocol, 18.03.2018
[39] CodeCompass, https://github.com/Ericsson/CodeCompass, 18.03.2018