STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIX**, Special Issue 1, 2014
10th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, May 21-25, 2014

VISUALIZATION TECHNIQUES OF COMPONENTS FOR LARGE LEGACY C/C++ SOFTWARE

MÁTÉ CSERÉP AND DÁNIEL KRUPP

Abstract. C and C++ languages are widely used for software development in various industries including Information Technology, Telecommunication and Transportation since the 80-ies. Over this four decade, companies have built up a huge software legacy. In many cases these programs become inherently complicated by implementing complex features (such as OS kernels or databases), and consisting several millions lines of code. During the extended development time, not only the size of the software increases, but a large number (i.e. hundreds) of programmers get involved in the project. Mainly due to these two factors, the maintenance of these software products becomes more and more time consuming and costly.

To handle the above mentioned complexity issue, companies apply software comprehension tools to help in the navigation and visualization of the legacy code. In our article we present a visualization methodology that assists programmers in the process of comprehending the functional dependencies of artifacts in a C++ source. Our novel graph representation not only reveals the connections between C/C++ implementation files, headers and binaries, but also visualizes the relationships between larger software components – e.g directories –, and provides a method for architecture compliance checking. The applied technique does not require any modification or documentation of the source code, hence it solely relies on the compiler generated *Abstract Syntax Tree* and the build information to analyze the legacy software.

1. Introduction

One of the main task of code comprehension software tools is to provide navigation and visualization views for the reusable elements of the source code, because humans are better at deducing information from graphical images [3, 12]. We can identify reusable software elements in C/C++ language on

Received by the editors: May 1, 2014.

2010 *Mathematics Subject Classification.* 68N99.

1998 *CR Categories and Descriptors.* I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling – *Object hierarchies*.

Key words and phrases. code comprehension, software maintenance, static analysis, component visualization, graph representation, functional dependency.

many levels (see Figure 1). On a smaller scale, functions provide reusable implementation of a specific behavior. The next level of modularity in C++ are classes, where a programmer can collect related functions and data that belong to the same subject-matter. Header files compose the next level, where related functions, variables, type declarations and classes (in C++ only) can be grouped into a semantic unit. Finally, related header files (possibly contained in a directory) can form the interface of a reusable binary component, a library.

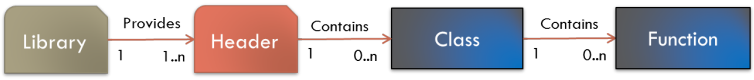


FIGURE 1. Modularity in the C++ language

State of the art software comprehension and documentation tools implemented visualization methods for many of these modularization layers. On the function level call graph diagrams can show the relations between the caller and the called functions [14], while on the class level, one can visualize the containment, inheritance and usage relations by e.g. UML diagrams. On the file level, header inclusion diagrams help the developers in the code comprehension process. [13]

We argue that the state of the art in file and module level diagrams are not expressive enough to reveal some important dependency relationships of the implementation among header files and directories containing source code.

In this paper, we describe a new visualization methodology that exposes the relations between implemented and used header files and the source file dependency chains of C/C++ software. In this article we also present that our approach identifies the usage and implementation relationships between directory level components.

This paper is structured as follows. In Section 2 the literature of the state of the art is reviewed with special focus on static software analysis. Section 3 describes which important views are missing from the current software comprehension tools, and in Section 4 we present novel views that can help C and C++ programmers to better understand legacy source code. Then in Section 5 we demonstrate our result by showing examples on real open-source projects. Finally in Section 6 we conclude the paper and set the directions for future work.

2. RELATED WORK

Researchers have proposed many software visualization techniques and various taxonomies have been published over the past years. These tools address one or more of three main aspects (static, dynamic, and evolutional) of

a software. The visualization of the static attributes focuses on displaying the software at a snapshot state, dealing only with the information that is valid for all possible executions of the software, assisting the comprehension of the architecture of the program. Conversely, the visualization of the dynamic aspects shows information about a particular execution of the software, therefore helps to understand the behavior of the program. Finally, the visualization of the evolution – of the static aspects – of a software handles the notion of time, visualizing the alternations of these attributes through the lifetime of the software development. For a comprehensive summary of the current state of the art see the work of Caserta et al.[4].

The static analysis of a software can be executed on different levels of granularity based on the level of abstraction. Above a basic source code level, a middle – package, class or method – level, and an even higher architecture level exists. In each category a concrete visualization technique can focus on various different aspects. Some of the most popular and interesting new visualization techniques are described below and a summary of categorization is shown on Table 1 for them.

Kind	Level	Focus	Techniques
Time T Visualization	Line	Line properties	Seesoft
	Class	Functioning, Metrics	Class BluePrint
	Architecture	Organization	Treemap
		Relationship	Dependency Structure Matrix
			UML Diagrams
			Node-link Diagrams
			3D Clustered Graphs
Visualizing Evolution			

TABLE 1. Categorization of visualization tools

The most simple, code line centered visualization has an abstraction level similar than the source code itself. An example for such a technique is *SeeSoft* [6, 1]. This visualization method is a miniaturized representation of the source code lines of the software, where a line of code is represented by a colored line with a length proportional to the length of the code line and colors can stand

for variety of mapping – e.g. the corresponding control structures. Indentation is preserved, hence the structure of the source code stays visible even when a large volume of code is displayed.

The class centered visualization techniques focus on the behavior or the metrics (e.g. about the cohesion) of a class, which helps in understanding the inner functioning of a class alone, or in the context of its direct inheritance hierarchy. As an example the *class blueprint* visualization technique [9, 5] displays the overall structure of a class, the purpose of methods within it and the relationship between methods and attributes. The information conveyed by it is otherwise often hard to notice because it would require the line by line understanding of the entire class.

Visualizing the architecture consists of depicting the hierarchy and the relationships between software components. It is one of the most frequently addressed topics in the software visualization field, as object-oriented programs are usually structured hierarchically both by the recursive package/module containment and by the fact that classes are structured by methods and attributes. Therefore most of the visualization techniques tackles the architectural level. One aspect is to focus on the organization like the well-known *treemap* method [8, 11], which recursively slices a box into smaller boxes (both horizontally and vertically) for each level of the hierarchy.

Our article focuses on assisting the code comprehension through visualizing the relationships between architectural components of a software. This category not only contains various prevalent and continuously improved visualizing techniques like the *UML diagrams* [7], but also recently researched, experimental diagrams like the three dimensional clustered graphs [2]. This technique aims to visualize large software in an integral unit, by generating graphs in a 3D space and grouping remote vertices and classes into clusters. The visibility of the inner content of a cluster depends dynamically on the viewpoint and focus of the user who can traverse the whole graph.

Our novel solution uses the classical node-link diagram in two dimensional space for visualization, which was formerly used at lower abstraction levels primarily.

3. Problems of visualization of C/C++ modularity

Modularity on the file level of a software implementation in C/C++ is expressed by separating interfaces and definition to header and implementation files. This separation allows the programmers to define reusable components in the form of – static or dynamic – libraries. Using this technique, the user of the library does not need to have knowledge about the implementation of it in order to use its provided services. Interfaces typically contain macro and

type definitions, function and member declarations, or constant definitions. Implementation files contain the definition of the functions declared in the headers.

Separation of these concerns is enforced by the C/C++ compiler, preprocessor and linker infrastructure. When a library is to be used, its header file should be *included* (with using the `#include` preprocessor directive) by the client implementation or the header files. Source files should almost never[1] be included in a project where the specification and implementation layers are properly separated. Unfortunately C/C++ does not enforce naming conventions to the header and implementation files (like e.g. Java does). Thus, based on a name of a file, it is not possible to find out where the methods of a class are declared or implemented. Furthermore, the implementation of the functions declared in a header file can be scattered through many implementation files that makes the analysis even more difficult.

When a programmer would like to comprehend the architecture of a software, the used and provided (implemented) interface of a library component or the implementers of a specific interface should quickly able to be fetched.

Problem (File dependencies)**.** Let us analyze the commonly presented header inclusion graph of a fileset in Figure 2. We assume that `lib.h` is an interface of a software library and that there are many user of this component, thus many files includes this header. If the programmer would like to understand where the functions declared in the header are implemented, the header inclusion graph is not helpful, since it does not show which C/C++ files only use, and which implement the `lib.h` interface.

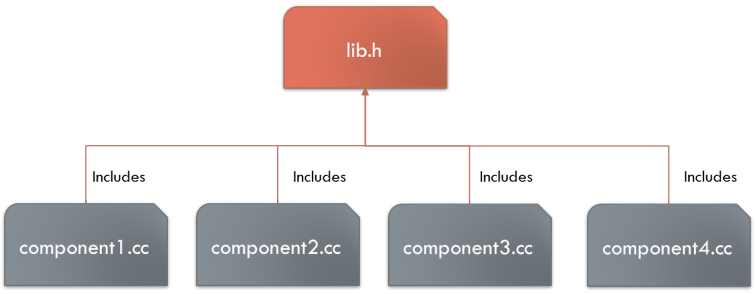


FIGURE 2. Implementation decision problem between component(s) and an interface.

[1]A few exceptions may exist, i.e. in some rare cases of template usage.

We propose an *Interface diagram* that is similar to the well-known header inclusion graph, but refines the include relation into uses and provides relationships. For this purpose we defined that a C/C++ file provides a header file, if it contains its implementation, while it only uses it if the mentioned file refers to at least one symbol in the header, but does not implement any of them. A proper and precisely defined description of this view is given in Section 4.1.

Problem (Directory-based dependencies)**.** Similar views are of great interest also on a higher, module level. For example we can examine the directory containment graph in Figure 3. It is a very basic question from a software developer to identify the dependency relations between the modules of a project, e.g. whether `module1` is used by `module2` and `module3` or not. To extract this information it is necessary to generalize the file level dependency relationships discussed beforehand.

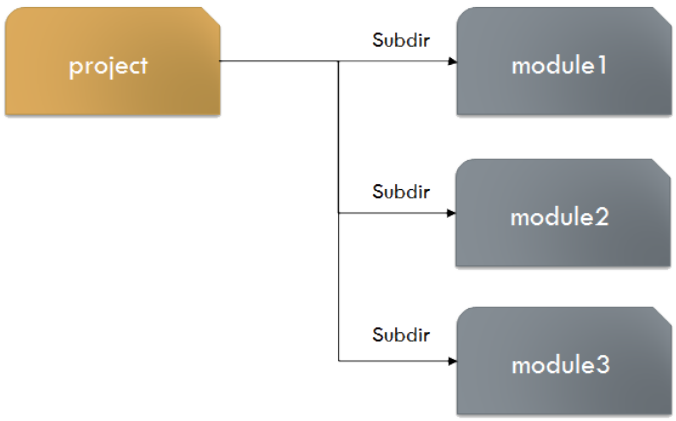


FIGURE 3. Dependency decision problem between modules.

In this paper we set forward views that code comprehension software tools can provide to unveil this hidden information.

4. Definitions of relationships between compilation artifacts

In this section first we introduce the commonly used basic terms of relationships defined between the C/C++ source files and the binary objects (see Figure 4), then present our more complex relationship definitions to describe the connections between the components of a software at a higher abstraction level.

Definition (Relations between source files)**.** At the level of the *abstract syntax tree* the main artifacts of a C/C++ source code are the user defined symbols[2], which can be declared, defined or referred/used by either the source files (`.c/.cc`) or the header files (`.h/.hh`). A C/C++ symbol might have multiple declarations and references, but can be defined only once in a semantically correct source code. To enforce the separation of the specification and implementation layer, header files should mainly consist of declarations, whose definitions are in the appropriate source files.[3] From our perspective only those C/C++ symbols are important, which are declared in a header file and are defined or referred by a source file.

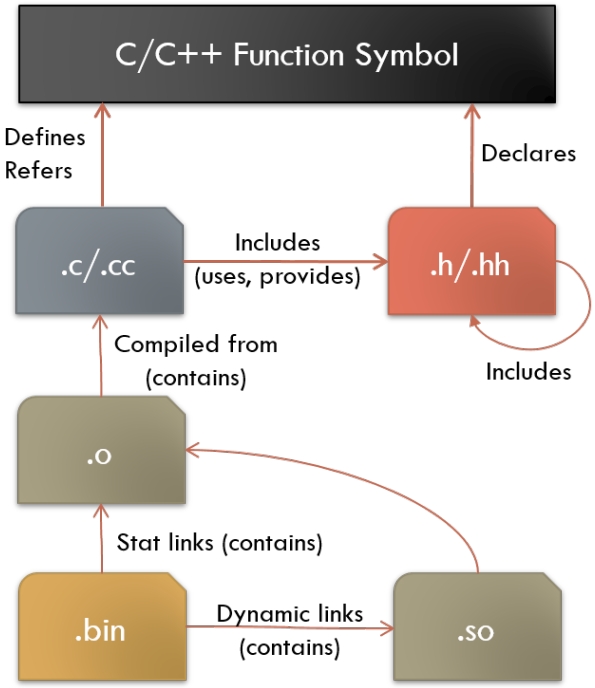


FIGURE 4. Relations between compilation artifacts.

Definition (Relations between binaries)**.** The source files of a project are *compiled* into object files, which are then *statically linked* into shared objects

[2]From our viewpoint only the function (and macro) symbols and their declaration, definition and usage are significant.

[3]In some cases, headers may contain definition and source file may also contain forward declarations as an exception.

or executable binaries. Shared objects are *linked dynamically* into the executables at runtime. To extract this information and visualize the relationship of binaries together with the relations declared between the C/C++ files, the analysis of the compilation procedure of the project is required beside the static analysis of the source code.

For the purpose of the presented visualization views in this paper the different kind of binary relationships is irrelevant, therefore they will be collectively referred as the *contains* relation henceforward (see Figure 4).

4.1. **File-based classification.** The basic include relationship among the source and header C/C++ files have already been introduced, however in order to solve the first problem raised in Section 3, the definitions of the proposed uses and provides relations have to be separated.

Definition (Provides relationship from source c to header h)**.** We say that in a fileset a source file c *provides* the interface specified by the header file h, when c includes h and a common symbol s exists, for which h contains the declaration, while c consists the definition of it.

Definition (Uses relationship from source c to header h)**.** Similarly to the previous provides relationship, we state that in a fileset a source file c *uses* the interface specified by the header file h, when c includes, but does not provide h and a common symbol s exists, which c refers and h contains the declaration of it.

Figure 4 shows the illustration for the above mentioned definitions. Based on the idea of the already introduced *Interface diagram*, which shows the immediate *provides*, *uses* and *contains* relations of the examined file, we defined the following more complex file-based views.

The nodes of these diagrams are the files itself and the edges represent the relationships between them. A labeled, directed edge is drawn between two nodes only if the corresponding files are in either *provides*, *uses* or in *contains* relationship. The label of the edges are the type of their relationship and they have the same direction as the relation they represent.

Definition (Used components graph of source c)**.** Let us define the graph with the set of nodes N and set of edges E as follows. Let S be the set of source files which *provides* an interface directly or indirectly *used by* c. N consists of c, the elements of S and the files along the path from c to the elements of S. Binaries containing any source file in S are also included in N. E consists the corresponding edges to represent the relationships between the nodes in N.

Intuitively we can say if source t is a used component of c, then c is using some functionality defined in t.

Definition (User components graph of source c)**.** Let us define the graph with the set of nodes N and set of edges E as follows. Similarly to the previous definition, let S be the set of source files which directly or indirectly *uses* the interface(s) *provided by* c. N consists of c, the elements of S and the files along the path from c to the elements of S. Binaries containing any source file in S are also included in N. E consists the corresponding edges to represent the relationships between the nodes in N.

Intuitively we can say if source t is a user component of c, then c is providing some functionality used by t.

4.2. **Directory-based classification.** Directory hierarchy in a C/C++ project carries very important architectural information. This is the most commonly used technique for modularizing the source code (think of the Linux kernel modules for example). Many times the build process also follows this modularity, since in a recursive Makefile based build system, the Makefiles are written per directory, many times linking the compiled binaries into a single executable or reusable library. Thus the directory boundaries are frequently the same as the library boundaries.

Files within a module are usually strongly coupled, they implement features that are closely related. Hence for understanding a large legacy software, it is vital to know how the modules are interconnected. To define these connections we use the relationships we defined for files.

Definition (Module)**.** A *module* is a directory tree which contains at least one C/C++ source code file. Please note, that from this definition it follows that a module can contain other modules. We say that module A and B are separate modules if A is not a sub-module of B and B is not a sub-module of A.

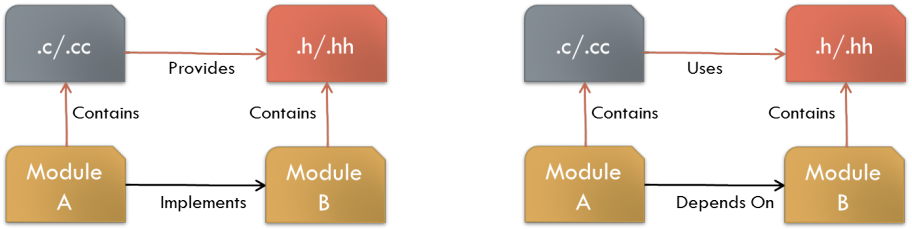
Definition (Implements relationship from module A to B)**.** We say that a module A *implements* another module B if module A and B are separate and A contains at least one implementation file c which which *provides* a header h which is in B.

Intuitively we can say if module A implements module B, then module A implements at least one of the services of module B.

Definition (Depends on relationship from module A to B)**.** We can say that module A *depends on* module B if module A and B are separate , A does not implement B and A contains at least one implementation file c which *uses* a header h contained by B.

Intuitively we can say if module A depends on module B, then module A is using the services of module B.

The illustration of the above definitions are depicted in Figure 5. Using these relationships we are able to visualize the internal architecture of a module and its external relationships by defining the following graph diagrams.



(A) The *implements* relationship. (B) The *depends on* relationship.

FIGURE 5. Relationships between modules.

The nodes of a directory level graph diagram are the modules according to the definitions above, while the edges represent the relationships between the modules. A labeled, directed edge is drawn between two nodes only if the corresponding modules are in either *subdirectory*, *depends on* or *implements* relationship. The labeling and directing of the edges has the same definition as for file level views.

Definition (Internal architecture graph of module M on level n)**.** Let us define the graph with the set of nodes N and set of edges E as follows. N consists of M and its n level sub-modules S and all the sub-modules of M that the elements of S *depends on* or *implements*. E consists of *subdirectory*, *depends on*, *implements* relationships between modules that correspond to nodes in N.

By examining the *Internal architecture diagram* it is easy to reveal which sub-modules are central, or in other words most commonly used in the analyzed module. This diagram also shows at which rate the implementation of a specific module containing interfaces is scattered through among other internal modules, hence a programmer can start by investigating the directories that contain the most commonly used interfaces when the understanding of a whole subsystem is required.

Definition (Implemented external modules graph of module M)**.** Let us define the graph with the set of nodes N and set of edges E as follows. A module o is in N if o is M or M *implements* o so that o is not a sub-module of M. E consists of the corresponding *implements* edges.

This diagram shows whether the examined module is not self contained, that is, it consist implementation of functions that are declared somewhere

in a directory that is not contained by the module. In many well structured projects, if a module provides some interfaces to external users, these header files are stored in a separate directory from the implementation files. These – interface – modules will then be implemented by the examined module.

Definition (External user modules graph of module M)**.** Let us define the graph with the set of nodes N and set of edges E as follows. A module o is in N if o is M or o *depends on* or *implements* M so that o is not a sub-module of M. E consists of *depends on* and *implements* relationships between modules that correspond to nodes in N.

This diagram answers the question which external modules are using the services of module M. In the diagram we also indicate the exact headers and functions that are used. This way a programmer can distinguish between header files that are only used within the examined module internally and which are provided for external use. In poorly structured C/C++ projects internal and external headers are frequently mixed in a directory and thus this separation becomes difficult without tool support.

5. Experimental results

In order to implement the proposed views in Section 3 by the definitions introduced in Section 4.1 and 4.2, a diagram visualizing tool was created as part of a larger code comprehension assisting project – named *CodeCompass* – developed in cooperation at Eötvös Loránd University and Ericsson Hungary. The created software provides an interactive graph layout interface, where the users are not only capable of requesting more information about the nodes representing files or directories, but they can also easily navigate between them, switching the perspective of the view they are analyzing.

For demonstration in this paper, the open-source *Xerces-C++* XML parser project[15] was selected, since it is a well-known, large (but not too broad for the purpose of presentation) legacy C++ software. In this section altogether five (including three file-based and two module-based) examples for the use of our tool is shown and information retrievable from them is examined.

Example. Figure 6 displays an *Interface diagram*, showing the immediate relations of a selected file with other files in the software. As the image shows, the C++ source file in the middle (`XMLReader.cpp`) uses/includes several header files, and the special connection for implementation (providing) is distinguished from the other ones. This diagram not only present the connections between C++ source and header files, but also displays that the mentioned source file was compiled into two object files through the compilation process of the project.

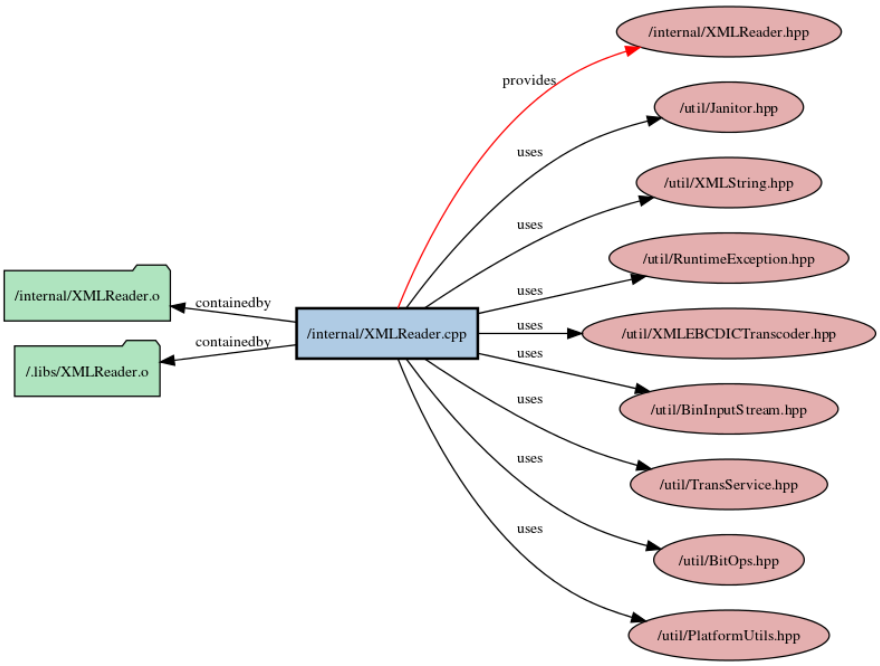


FIGURE 6. Interface diagram of `XMLReader.cpp`.

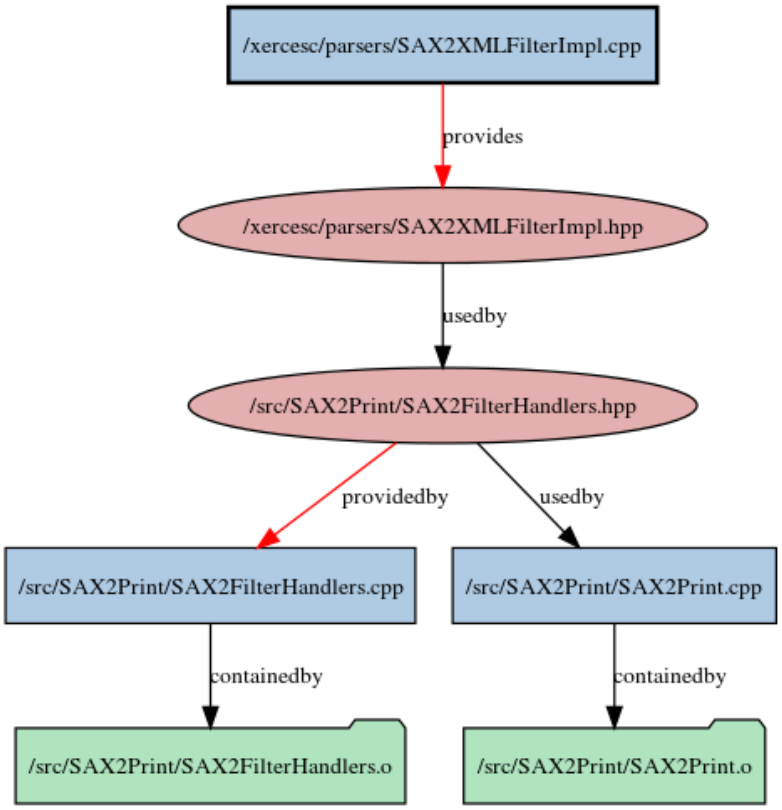


FIGURE 7. User components of `SAX2XMLFilterImpl.cpp`.

Example. Figure 7 presents the *User components diagram* of the source file `SAX2XMLFilterImpl.cpp` at the top. The goal of this visualization is to determine which other files and compilation units depends on the selected file. As the figure shows, the previous source file implements an interface contained by the `SAX2XMLFilterImpl.hpp` header. From here with multiple relations of usage and interface implementation we are able to specify two sources files which are the users of `SAX2XMLFilterImpl.cpp`.

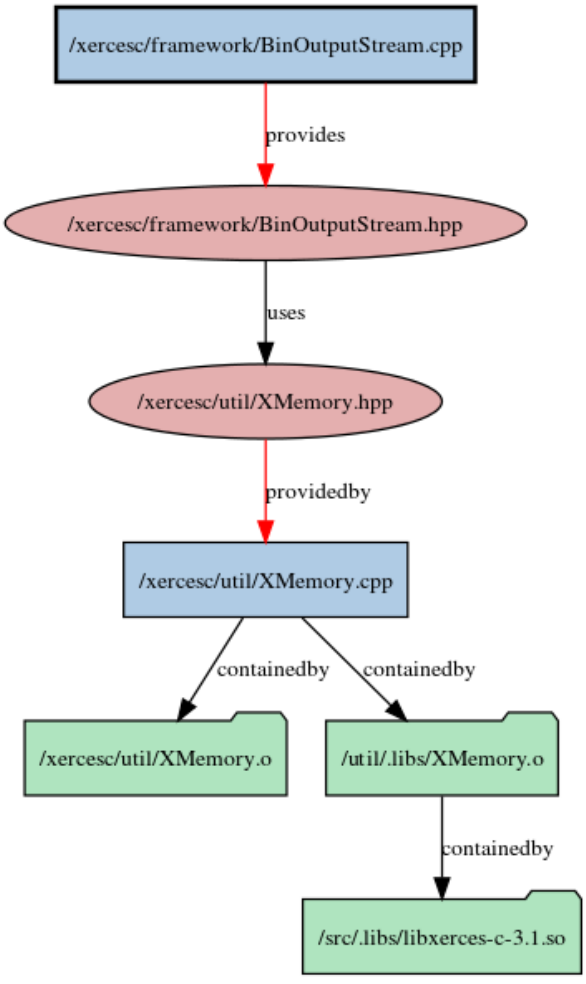


FIGURE 8. Used components by `BinOutputStream.cpp`.

Example. Parallel to Figure 7, the following example deduces the compilation artifacts the selected source `BinOutputStream.cpp` depends on. The *Used components diagram* displays (see Figure 8) that the interface specification for the `BinOutputStream.cpp` source file is located in the `BinOutputStream.hpp` header. This header file uses the `XMemory.hpp`, which is provided by the `XMemory.cpp` source. Hence the implication can be stated that the original `BinOutputStream.cpp` indirectly uses and depends on the `XMemory.cpp` file.

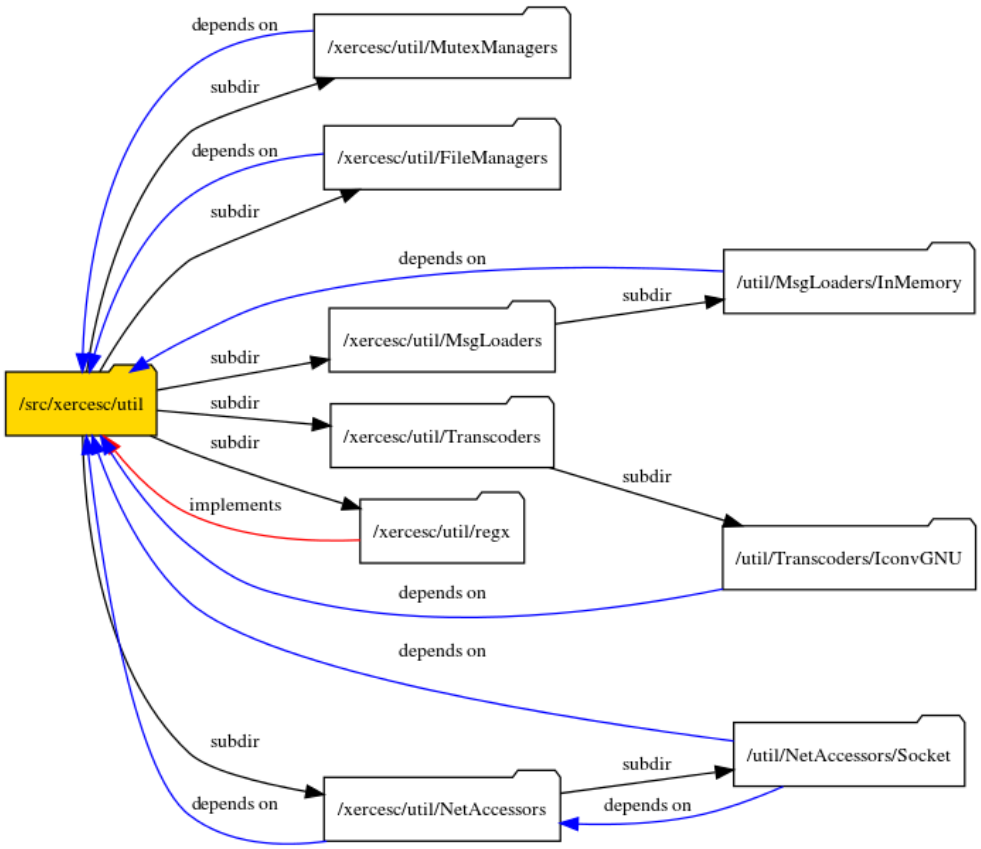


FIGURE 9. The internal dependencies in the `util` directory.

Example. The last two examples present views on a higher abstraction level, revealing the relationships between directory modules. Figure 9 shows the *Internal architecture graph* of the `util` module in *Xerces*. Beyond the dependency relations between the subdirectories, the view emphasizes implementation relationships. Finally, Figure 10 displays the *External user modules*

graph of `sax2` directory, unveiling information about the external usage and implementation of this module.

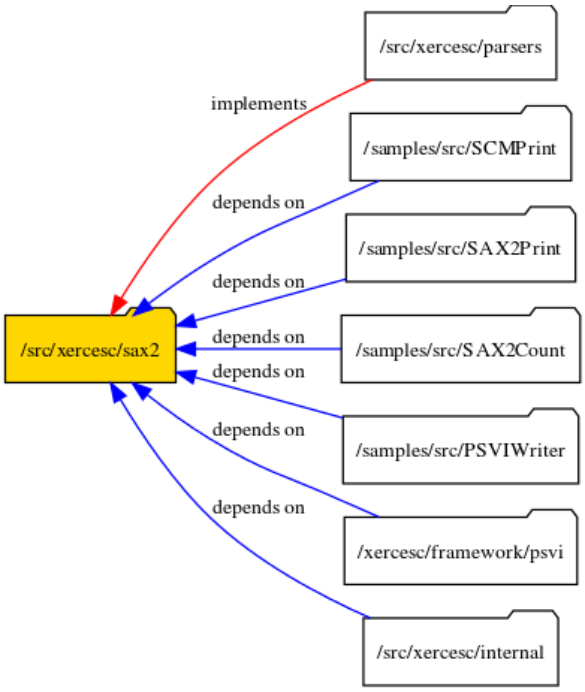


FIGURE 10. The external users of the `sax2` directory.

6. CONCLUSIONS AND FUTURE WORK

Assisting code comprehension in large legacy software projects is an important task nowadays, because through the extended development time often a huge codebase is built up and the fluctuation among programmers also becomes significant. Since humans are better at deducing information from graphical images than numerical data, it is a recognized method to support the (newcomer) developers with visualization views to understand the source code. In this paper we discussed what kind of architectural views are missing from current code comprehension tools, regarding the relationships between different type of compilation artifacts. We defined our novel graph view as a solution addressing this problem, and demonstrated the practical use of our technique through examples on a large legacy software. The new visualization techniques were found helpful and applicable for legacy software in supporting code comprehension.

Future work will include the examination of how the information retrieved by our definition rules can be used in the field of architecture compliance checking. Software systems often impose constraints upon the architectural design and implementation of a system, for example on how components are logically grouped, layered and upon how they may interact with each other. In order to keep the maintainability of a software system through a long development time with a large programmer team, it bears extreme importance that the design and implementation are compliant to its intended software architecture. Due to the complexity of large software systems nowadays, guaranteeing the compliance by manual checking is impossible, hence automation is required, which is a not completely solved issue until today [10].

References

[1] T. Ball, S. Eick, *Software visualization in the large*, Computer, 29 (1996), pp. 33-43.
[2] M. Balzer, O. Deussen, *Level-of-detail visualization of clustered graph layouts*, Proceedings of the 6th International Asia-Pacific Symposium on Visualization, (2007), pp. 33–140.
[3] I. Biederman, *Recognition-by-components: a theory of human image understanding*, Psychological review, 94 (1987), pp. 115–147
[4] P. Caserta, O. Zendra, *Visualization of the static aspects of software: a survey*, IEEE Transactions on Visualization and Computer Graphics, 17 (2011), pp. 913–933.
[5] S. Ducasse, M. Lanza, *The Class Blueprint: Visually Supporting the Understanding of Classes*, IEEE Transactions on Software Engineering, 31 (2005), pp. 75–90.
[6] S. Eick, J. Steffen, E. Sumner, *Seesoft-A Tool for Visualizing Line Oriented Software Statistics* IEEE Transactions on Software Engineering, 18 (1992), pp. 957–968.
[7] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, P. Mutzel, *A new approach for visualizing UML class diagrams*, Proceedings of the 2003 ACM Symposium on Software Visualization, (2003), pp. 179–188.
[8] B. Johnson, B. Shneiderman, *Tree-maps: A space-filling approach to the visualization of hierarchical information structures* Proceedings of the 2nd Conference on Visualization, (1991).
[9] M. Lanza, S. Ducasse, *A categorization of classes based on the visualization of their internal structure: the class blueprint.* OOPSLA, 1 (2001), pp. 300-311.
[10] L. Pruijt, C. Koppe, S. Brinkkemper, *On the accuracy of architecture compliance checking support: Accuracy of dependency analysis and violation reporting*, IEEE 21st International Conference on Program Comprehension, (2013), pp. 172–181.
[11] B. Shneiderman, *Tree Visualization with Tree-maps: 2D Space-filling Approach*, ACM Transactions on Graphics, 11 (1992), pp. 92–99.
[12] I. Spence, *Visual psychophysics of simple graphical elements*, Journal of Experimental Psychology: Human Perception and Performance, 16 (1990), pp. 683–692
[13] Doxygen Tool: `http://www.stack.nl/~dimitri/doxygen/`
[14] Understand Source Code Analytics & Metrics, `http://www.scitools.com/`
[15] Xerces-C++: `http://xerces.apache.org/xerces-c/`

Eötvös Loránd University, Ericsson Hungary
E-mail address: `mcserep@caesar.elte.hu, daniel.krupp@ericsson.com`