# How Imagix 4D supports the understanding of software based on source code

J.D. Baltzer, M.Sc.
baltzer@verifysoft.com
Verifysoft Technology GmbH
In der Spöck 10-12
DE-77656 Offenburg

***Abstract*—This work will show how Imagix 4D improves the process of exploring and understanding unfamiliar source code. An introduction into why and where such a process is required will be followed by a description of the methods Imagix 4D introduces to support such source code analysis.**

## I. Introduction

The size and complexity of software projects typically increase over their lifetime. As a result, it becomes more and more expensive to sustain enough knowledge to support maintenance, enhancements and verification against existing and changed requirements. This is exacerbated by the realities of personnel turnover and use of 3rd party and legacy code. Shortages of resources lead to shortcuts in quality like violations of guidelines and poor documentation. This overall decline is termed „Technical Debt“ and „Software Decay“. At some point, the effort to continue such a project becomes too high; the remaining options for it are rebuilding or termination. But how to decide if that point has been reached? It’s difficult to appraise a project that has degenerated over decades. Especially if the documentation is useless, and if the people who were involved are no longer available. The only way to get detailed and up to date information about such a software project is to explore its source code. This is very hard to be done manually. For this reason, a tool is needed to allow insights on every level of abstraction of a software based on source code.

## II. Imagix 4D

Developed and sold by the Imagix Corporation, Imagix 4D is a tool whose purpose is software inspection based on source code [Ima18]. Imagix 4D creates graphical and tabular representations of projects written in C, C++ and Java. The granularity of the views range from an architectural perspective down to individual symbols. Imagix 4D also integrates visualizations of relationships and metrics into its diagrams. Which software properties are shown can be configured for any specific purpose. Imagix 4D has several basic, predefined diagrams that will be introduced in the following sections. The Git 2.17.1 project [Git19] configured for Ubuntu 18.04.2 LTS [Can18] is used as the example here.

*DSMs (Design Structure Matrices)* deliver an overview of all contained dependencies between subsystems of the system. This is shown in a matrix, hence its name. Rows and columns represent subsystems. The granularity of the subsystems start with the root directory „git-2.17.1“ and can be broken down to the level of individual functions. The fields of the cross sections show dependencies, or relationships, from the symbols listed in the column on the left to the symbols listed in the row along the top. The entries can be binary values or a numeric count of relationships. Figure 1 contains an excerpt of the DSM of the example project. From the detail can be determined that along with 8 sub-directories, there are a number of C files directly under root directory that comprises the main part of the project. Because of spatial restrictions, Figure 1 contains only an excerpt of the whole DSM. Relationships are shown as binary values. The principal diagonal emphasizes relationships from a subsystem to itself. This might be function calls within an individual file, for example. Several selection methods are available in the DSM. They enable focus on interesting sections in the matrix, such as relationships from file „commit.c“ to file „usage.c“. When an interesting cross section is found, its relationships can be expanded in a new subsystem architecture.

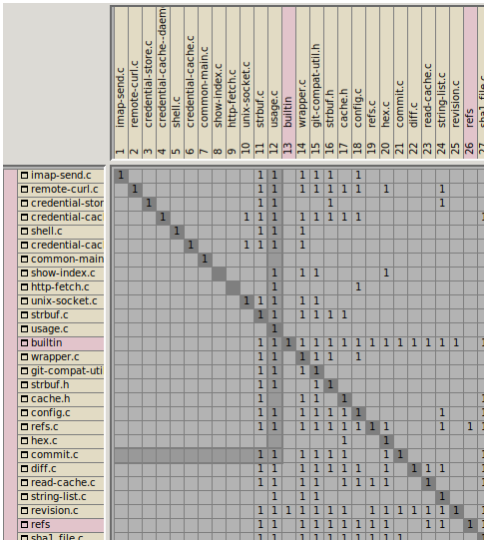


Figure 1. Excerpt of the DSM of the example project.

*Subsystem Architectures* enable focus on the architecture of a project. This could be either the physical distribution of files in a directory tree or the logical hierarchy of packages and classes. For the example project, because it is written in C, only the physical variant is available. In the project, the root directory is used for main development. As a result, it is full of files, which makes the diagram big and messy if no filters are applied to it. Filters are very useful for focusing on significant parts of software, avoiding a big and clumsy diagram that can occur when all available properties are enabled. For an example, the relationship from „commit.c“ to „usage.c“, referred to in the last section, is shown in Figure 2. Here, the functions involved in this relationship can be observed. To get more information about the symbol usage underlying other relationships, the diagram can be appropriately modified.

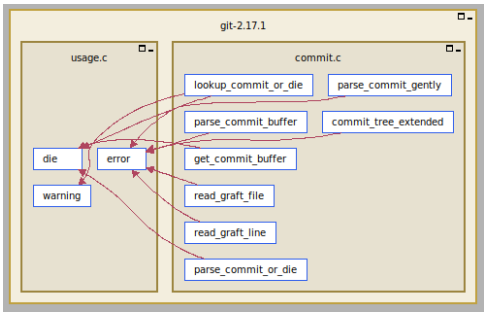


Figure 2. Excerpt of the architecture of the example project.

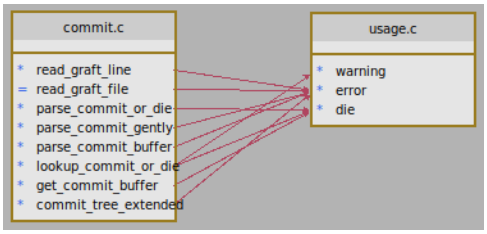


Figure 3. Function calls from „commit.c“ to „usage.c“ as UML diagram.

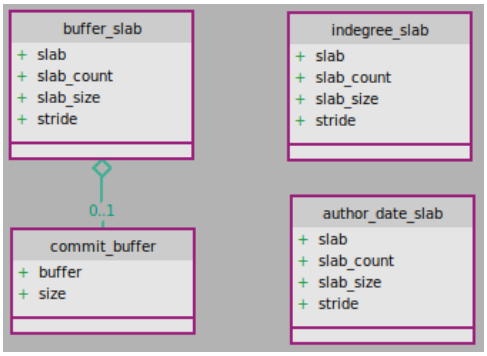


Figure 4. Structures contained in „commit.c“ and „usage.c“ as UML diagram.

*UML Diagrams* aid in understanding complex container symbols such as classes, structs and files, including their relationships. Imagix 4D offers 3 configurable UML diagrams. The diagram for classes and structs shows their members and relations in UML notation. File diagrams show properties of compilation units. Both variants address the question: „How are the logical units in my projects are assembled and how do they work together? “. The diagrams can show members of the class symbols with their data encapsulation and storage class. Relationships between the members can be superimposed. For the class symbols themselves, relationships like composition and inheritance are visible. The example in Figure 3 shows a UML file diagram for the files „commit.c“ and „usage.c“. Functions from „commit.c“ call 3 functions from „usage.c“. All of the functions expect the static function „read_graft_file“ are global. To inspect the structs defined in that files, the view can be changed to a UML class diagram. This will show the 4 structs of Figure 4. As can be seen „commit_buffer“ is part of „buffer_slab“ and the other structs are independent.

*File Diagrams* show the dependencies between files. If the C language is used, these are headers and compilation units. Relationships in this diagrams are inclusions or arbitrary accesses between symbols contained in the files. A look at the include hierarchy of the files „usage.c“ and „commit.c“ in Figure 5 clearly

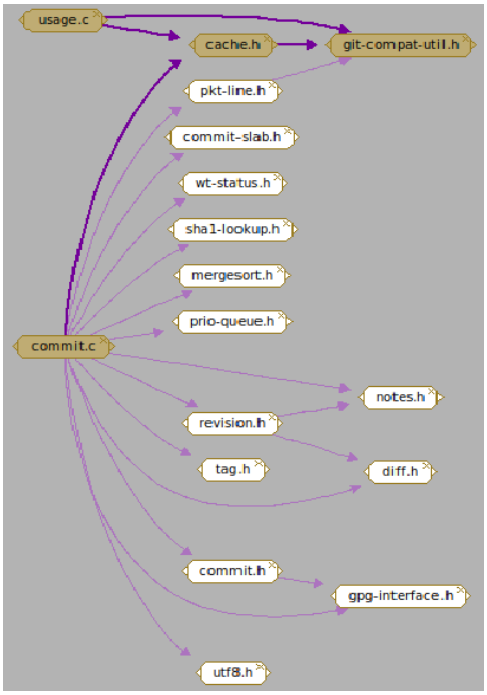


Figure 5. Direct inclusions of „commit.c“ and „usage.c“.


Figure 6. Files with functions called by functions from „usage.c“ and their common calls.

indicates what header files they include in common. Figure 6 shows only files having functions called by any function in „usage.c“. There are a lot of recurring relationships among the files indicating a tight coupling between them.

*Data Type Diagrams* are very valuable for understanding complex data types, which can be very frustrating and time-consuming otherwise. Imagix 4D provides the option to visualize relationships between types and variables. Types themselves are categorized into pointer, enums, classes, structs, typedefs and templates. These categories can also be distinguished optically in the diagram, based on

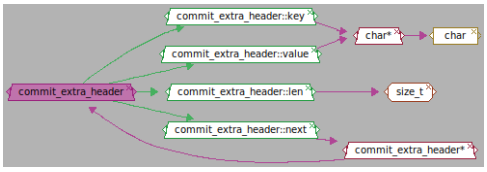


Figure 7. Relations of the struct „commit_extra_header“.

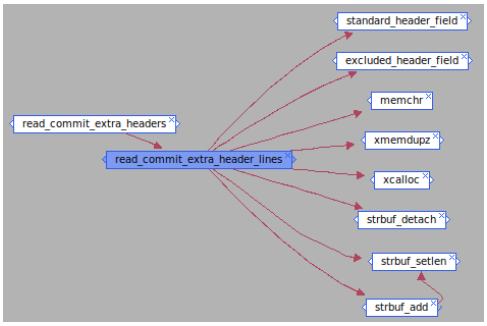


Figure 8. Incoming and outgoing calls of the function „read_commit_extra_header_lines“.


Figure 9. The control flow between „read_commit_extra_headers“ and „read_commit_extra_header_lines“.

different colors and shapes. From the example project, the return type of the function „read_commit_extra_header_lines“ in file „commit.c“ is shown in Figure 7. The type „struct commit_extra_header“ consists of 4 variables of scalar and pointer types. One of the variables is a reference to the same type. This indicates that this type is some kind of unidirectional list.

*Function Call Diagrams* provide insight into the calling hierarchy of a project. Calls to and from functions are shown. It is also possible to include function pointers. For example, Figure 8 shows all direct calls to and from „read_commit_extra_header_lines“. It is only called by „read_commit_extra_headers“, leading to the conclusion that „read_commit_extra_header_lines“ is just a helper function to read lines.

*Control Flow Graphs* focus on the control flow between functions, showing calls to functions along with the sequence and conditions of such calls. Knowledge about the control flow is critical to understand a program. For this reason, control flow constructs are also shown in this more interprocedural focused diagram. Viewing the functions „read_commit_extra_header_lines“ and „read_commit_extra_headers“ in a control flow graph provides further indication that „read_commit_extra_header_lines“ is a helper function from „read_commit_extra_headers“. It is only called once in the main control flow, as shown in Figure 9. On the left side, there is the line of source code calling the helper function. Other control elements are hidden because they are not related to calls of the current state of the diagram.

*Flowcharts* allow inspection of the control flow of a single function at a time. Imagix 4D implements 4 variants of flow charts, along with the possibility of extending and/or reducing additional information. This serves different requirements, from a rough understanding of

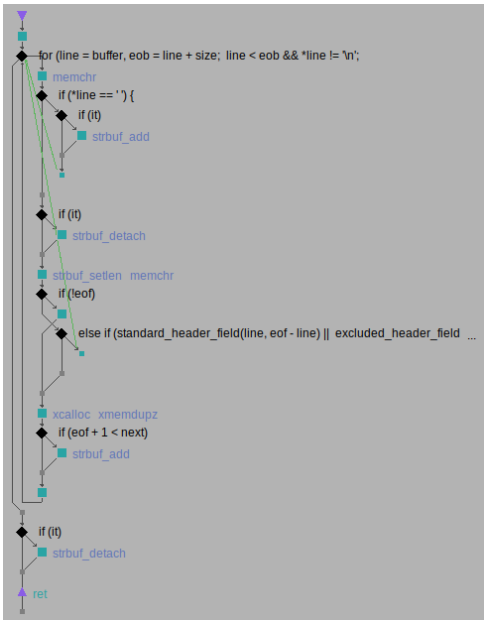


Figure 10. The flowchart of the function „read_commit_extra_header_lines“.

the control flow down to examination of implementation details and comments. Figure 10 gives an generalized view of the control flow of „read_commit_extra_header_lines“ including embedded function calls. What can be seen is that there is a main loop that processes an array.

*Source Code Inspections* can be fundamentally improved by Imagix 4D. Beyond mainly manual browsing, Imagix 4D offers methods to automatically detect anomalies in source code. This allows concentration on small, potentially problematic areas of code, and can save a lot of temporal effort. In addition, the generated reports can serve as a basis for further investigation of a project. The generated reports deliver metrics for various programming elements, information about anomalies in control and data flow, and information about potential redundancies.

## III. Retrospection

This work demonstrated through an real example how Imagix 4D supports exploring and understanding of unfamiliar source code. The main focus was on the graphical representations available, from an architectural down to individual symbol point of view. This proved that Imagix 4D is an appropriate tool for the problem defined in the introduction.

## References

[Can18] Canonical: *Ubuntu 18.04.2 LTS (Bionic Beaver)*. http://releases.ubuntu.com/18.04/. Version: 2018

[Git19] Git Community: *GitHub - git/git: Git Source Code Mirror - This is a publish-only repository and all pull requests are ignored. Please follow Documentation/SubmittingPatches procedure for any of your improvements.* https://github.com/git/git. Version: 2019

[Ima18] Imagix Corp.: *Reverse Engineering Tools - C, C++, Java - Imagix*. https://www.imagix.com/index.html. Version: 2018