Wie Imagix 4D das Verständnis von Programmen auf Quelltextebene unterstützt

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

***Zusammenfassung*—Diese Arbeit zeigt, wie Imagix 4D dabei unterstützt unbekannten Quelltext zu erkunden und zu verstehen. Nach einer Einführung in den Kontext der Problemstellung folgen Erörterungen verschiedener Methoden von Imagix 4D und wie diese zur Lösung beitragen.**

I. Einleitung

Mit wachsender Laufzeit von Softwareprojekten steigen typischerweise deren Umfang und Komplexität. Damit wird es auch zusehends aufwendiger den Überblick zu behalten, sei es zur Wartung, Erweiterung oder Verifikation gegen bestehende und veränderte Anforderungen. Verstärkt wird dies zudem durch Fluktuationen in der Belegschaft und der Einbindung von zusätzlichen Programmteilen, sei es zugekauft, frei zugänglich oder aus bestehenden Projekten. Engpässe an Ressourcen führen dazu, bei der Umsetzung Abkürzungen zu nehmen, beispielsweise durch Verstöße gegen architektonische Richtlinien und mangelhafte Dokumentation. Diese Umstände prägten maßgeblich die Begriffe „Technical Debt“ und „Software Decay“. Der Aufwand zur Handhabung solcher Projekte rechtfertigt irgendwann eine Überarbeitung, Neuimplementierung oder sogar das Ende des Projektes. Die umgesetzte Software zu verstehen und zu bewerten gestaltet sich allerdings als schwierig. Insbesondere bei älteren Projekten, deren Kernentwickler nicht mehr verfügbar sind. Man kann nicht davon ausgehen, dass die Software einer gegebenen Dokumentation entspricht. Um den tatsächlichen Stand eines Software-Projektes zu ergründen, führt nichts am zugrunde liegenden Quelltext vorbei. Aufgrund der hohen Komplexität wird ein Werkzeug benötigt, welches Einblicke beliebiger Granularität anhand des vorliegenden Quelltextes ermöglicht.

II. Imagix 4D

Die Imagix Corporation entwickelt und vertreibt das Werkzeug Imagix 4D zur Inspektion von Programmen anhand ihres Quelltextes [Ima18]. Es ist in der Lage Projekte der Sprachen C, C++ und Java grafisch wie tabellarisch aufzubereiten. Die Darstellung erstreckt sich von der Architektur- bis zur Quelltext-Ebene herunter und integriert verschiedene Darstellungen von Abhängigkeiten und diversen Metriken. Welche Eigenschaften von Software wie dargestellt werden ist für jeden Anwendungsfall konfigurierbar. Im Folgenden werden die grundlegenden, eingebauten Darstellungen von Imagix 4D vorgestellt und erläutert. Das in der Programmiersprache C entwickelte Versions-Verwaltungs-Programm Git 2.17.1 [Git19], erstellt für die Plattform Ubuntu 18.04.2 LTS [Can18] dient hierbei als Beispielprojekt.

DSM (Design-Struktur-Matrizen) liefern eine Übersicht der Abhängigkeiten aller enthaltenen Subsysteme von Projekten. Aufbereitet werden die Informationen in Form einer Matrix. Die

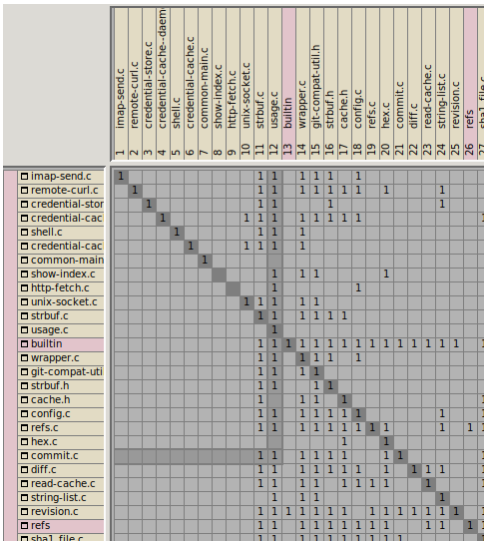

Abbildung 1. Ausschnitt der DSM des Beispielprojektes.

Zeilen und Spalten geben die Subsysteme des Projektes wieder. Die Granularität der Subsysteme kann vom Wurzelverzeichnis „git-2.17.1“ bis auf einzelne Funktionen heruntergebrochen werden. In den Feldern der Matrix werden die Relationen von den Subsystemen der Zeilen zu den Subsystemen der Spalten gezeigt, entweder als Wahrheitswert oder als Anzahl der Relationen. In Abbildung 1 wird ein Ausschnitt der DSM gezeigt. Zu erkennen ist, welche C-Dateien Relationen zu anderen Subsystemen haben und dass neben 8 Unterverzeichnissen der Hauptteil der Dateien direkt im Hauptverzeichnis liegt. Aus Platzgründen enthält Abbildung 1 nur einen Ausschnitt der DSM. Dargestellt werden die Relationen hier als Wahrheitswert. Die Diagonale zeigt die Relationen von Subsystemen auf sich selbst, was beispielsweise Funktionsaufrufe innerhalb einer Datei sein können. Mehrere Selektionsmöglichkeiten erlauben es, Relationen hervorzuheben, wie die der Datei „commit.c“ auf „usage.c“. Um mehr Details zu Relationen von Interesse zu erhalten, ist es möglich, über die Felder des DSM Teilabschnitte der Subsystem-Architektur zu öffnen.

Subsystem-Architekturen eröffnen die Möglichkeit, den Fokus auf die Architektur von Projekten zu richten. Die Architektur kann entweder die physikalische Verteilung von Dateien im Verzeichnisbaum oder die logische Aufteilung in Namensräume und Klassen sein. Das Beispielprojekt ist in C geschrieben, deshalb steht nur die physikalische Variante zur Verfügung. Die Subsystem-Architektur des gesamten Beispielprojektes ist etwas unübersichtlich, da die meisten Dateien direkt im Hauptverzeichnis sind und

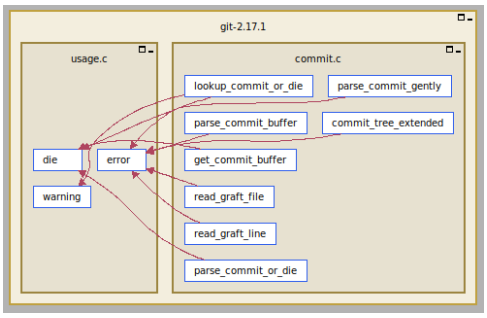

Abbildung 2. Ausschnitt der Architektur des Beispielprojektes.

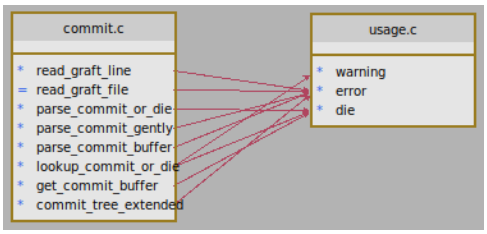

Abbildung 3. Funktionsaufrufe von „commit.c“ nach „usage.c“ als UML-Diagramm.

dementsprechend dargestellt werden. In diesem Fall ist es hilfreich sich mittels Filter gezielt auf Bereiche von Interesse zu fokussieren, um aussagekräftige Diagramme zu erhalten. Beispielsweise wurde die im letzten Abschnitt erwähnte Relation von „commit.c“ auf „usage.c“ in Abbildung 2 als Subsystem-Architektur geöffnet. Beobachtet werden kann nun, welche Funktionen der Dateien wie an der Relation beteiligt sind. Um weitere Kontextinformation zu erhalten, ist das Diagramm nun beliebig ergänzbar. Zudem können Subsysteme im Diagramm farblich hervorgehoben werden, um Auffälligkeiten im Bezug auf Metriken zu verdeutlichen.

UML-Diagramme erlauben es komplexere Komponenten wie Dateien, Klassen, Strukturen und deren Interaktion zu verstehen. Imagix 4D stellt 3 konfigurierbare UML-Diagramme zur Verfügung. Das Klassendiagramm gibt die Eigenschaften der Projektklassen in UML-Notation wieder. Hierzu zählen ebenfalls Datenstrukturen, wie sie im C-Sprachgebrauch gängig sind. Die Dateidiagramme geben Eigenschaften von Quelltextdateien wieder. Beide Varianten beantworten die Fragen: „Wie ist meine Datenstruktur aufgebaut und wie spielt sie mit anderen zusammen? “. Zu den Strukturen dargestellt werden können ihre Eigenschaften mit Speicherklasse und Sichtbarkeit als auch deren Interaktion mit Eigenschaften anderer Strukturen. Relationen zwischen den Strukturen selbst wie Komposition und Vererbung sind einsehbar. Ausgehend von den Dateien „commit.c“ und „usage.c“ verwenden Funktionen der Datei „commit.c“ 3 Funktionen aus „usage.c“. Davon sind alle bis auf die statische Funktion „read_graft_file“ global, wie aus Abbildung 3 hervorgeht. Wird aus diesem Diagramm nun das Klassendiagramm erzeugt, so werden die in den beiden Dateien definierten, soweit vorhandenen Strukturen und deren Komposition dargestellt. Wie in Abbildung 4 zu sehen besteht „buffer_slab“ aus „commit_buffer“, während die übrigen Strukturen unabhängig voneinander sind.

Datei-Diagramme geben das Zusammenspiel von Dateien und damit der physischen Komponenten wieder. Dateien sind im C-Kontext Header und kompilierbare Dateien. Relationen zwischen diesen können Inklusion oder diverse Zugriffe zwischen enthaltenen

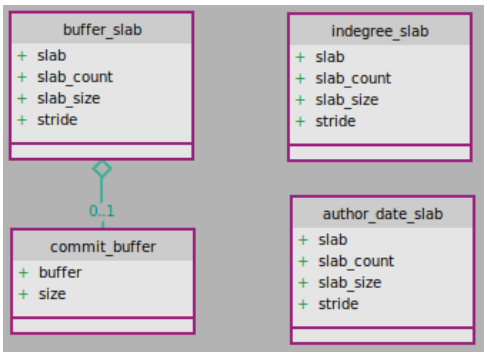

Abbildung 4. Die in „commit.c“ und „usage.c“ enthaltenen Strukturen als UML-Diagramm.

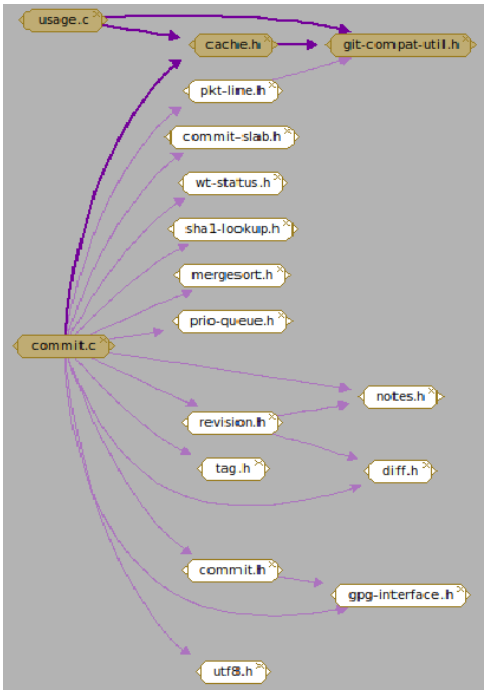

Abbildung 5. Direkte Inklusionen von „commit.c“ und „usage.c“.

Elementen sein. Ausgehend von den Dateien „usage.c“ und „commit.c“ kann hier mit Leichtigkeit herausgefunden werden, welche gemeinsamen Header diese haben, wie in Abbildung 5 gezeigt. Wenn man alleine für „usage.c“ die Dateien einblendet, deren Funktionen durch Funktionen innerhalb von „usage.c“ aufgerufen werden wird deutlich, dass eine starke bidirektionale Kopplung zwischen den Dateien besteht. Dies ist in Abbildung 6 zu erkennen.

Datentyp-Diagramme sind sehr hilfreich, um komplexe Typen zu verstehen, was ohne Hilfsmittel eine sehr zeitraubende Aufgabe ist. Imagix 4D bietet die Möglichkeit, Beziehungen zwischen Typen und Variablen aufzuzeigen. Typen selber werden unterteilt in Kategorien wie Pointer, Enums, Classes, Structs, Typedefs und Templates, welche grafisch unterschiedlich dargestellt werden. Aus dem Beispielprojekt wird der Rückgabewert der Funktion „read_commit_extra_header_lines“ aus der Datei „commit.c“ gezeigt. Wie in Abbildung 7 zu sehen, besteht „struct commit_extra_header“ aus 4 Variablen von Skalar- und Pointer-Typen mit einer Referenz auf ein nächstes Element, was auf eine

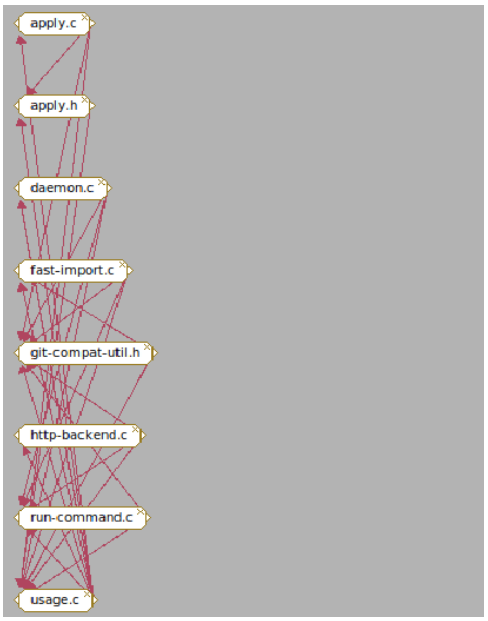

Abbildung 6. Dateien, deren Funktionen von Funktionen aus „usage.c“ aufgerufen werden mit ihren Aufrufen untereinander.

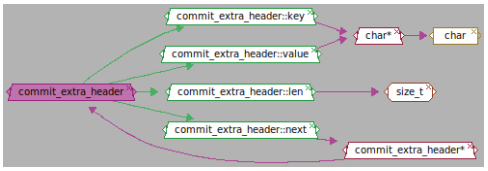

Abbildung 7. Zusammensetzung der Struktur „commit_extra_header“.

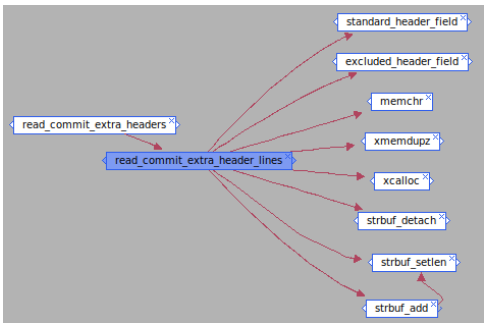

Abbildung 8. Ein- und ausgehende Aufrufe der Funktion „read_commit_extra_header_lines“.

unidirektionale Listenimplementierung hinweist.

Funktions-Diagramme beantworten die Frage der Aufrufhierarchie von Funktionen innerhalb eines Projektes. Hier werden Funktionen mit ihren eingehenden wie ausgehenden Aufrufen gezeigt. Auch Funktionspointer können in ein solches Diagramm einfließen. Abbildung 8 gibt Aufschluss über die direkten Aufrufe nach und von „read_commit_extra_header_lines“. Da die Funktion nur von „read_commit_extra_headers“ verwendet wird, handelt es sich auch des Namens nach um eine Hilfsfunktion, die Zeilen liest.

Kontrollfluss-Diagramme fokussieren den Kontrollfluss zwischen den Funktionen und zeigen, an welchen Stellen in andere Funktionen gesprungen wird. Wichtig zur Nachverfolgung des Kontrollflusses sind Kenntnisse über Verzweigungen und Schleifen

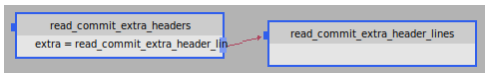

Abbildung 9. Der interfunktionale Kontrollfluss von „read_commit_extra_headers“ nach „read_commit_extra_header_lines“.

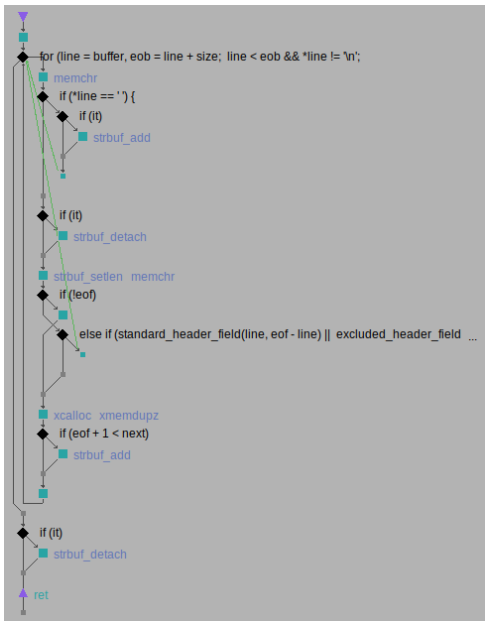

Abbildung 10. Das Flussdiagramm der Funktion „read_commit_extra_header_lines“.

im Programm, welche den Kontrollfluss steuern. Deshalb zeigen die Diagramme nicht nur die Stellen mit Funktionsaufrufen, sondern auch jene mit Kontrollflusssteuerelementen. Wie schon zuvor festgestellt ist „read_commit_extra_header_lines“ eine ausgelagerte Hilfsfunktion von „read_commit_extra_headers“, da sie einmalig im Hauptkontrollfluss aufgerufen wird. Abbildung 9 zeigt diesen Umstand. Im linken Block wird der eine Aufruf zur Hilfsfunktion gezeigt. Weitere Kontrollkonstrukte werden in diesem Fall nicht gezeigt, da nur Beziehungen zwischen den aktiven Funktionen eingeblendet werden.

Fluss-Diagramme ermöglichen die detaillierte Inspektion des Kontrollflusses einer einzelnen Funktion. Imagix 4D beinhaltet 4 Darstellungsvarianten und die Möglichkeit Zusatzinformationen ein und auszublenden. Damit können verschiedenen Anforderungen bedient werden. Von einer groben Übersicht der Verzweigungen bis hin zu Implementierungs- und Kommentierungsdetails direkt im Diagramm. Abbildung 10 gibt einen groben Überblick des Kontrollflusses von „read_commit_extra_header_lines“ mit eingeblendeten Funktionsaufrufen. Wie man erkennt, gibt es eine Hauptschleife, welche für jeden Eintrag eines Datenfeldes durchlaufen und abgearbeitet wird.

Quelltext-Inspektionen können mit Imagix 4D wesentlich verbessert werden. Neben der hauptsächlich manuellen Durchsicht bietet Imagix 4D Optionen, automatisiert nach Auffälligkeiten im Quelltext zu suchen um ein gezielteres Vorgehen zu gewährleisten oder eine Argumentationsgrundlage für weitere Nachforschungen zu liefern. Orientierung für eine Inspektion liefern Berichte zu Metriken von Programmelementen als auch zu Anomalien im Daten- und Kontrollfluss bis hin zu möglichen Redundanzen im Quelltext.

III. Rückblick

Diese Arbeit demonstrierte anhand eines praktischen Beispiels, mit welchen Mitteln Imagix 4D dabei unterstützt unbekannten Quelltext zu erkunden und zu verstehen. Der Schwerpunkt lag auf den grafischen Sichten, welche verschiedene Aspekte jeder Granularität des Projektes belichten. Damit zeigt sich auch, dass Imagix 4D ein geeignetes Werkzeug für die zu Anfang definierte Problemstellung ist.

Literatur

[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