Bachelor's Thesis

SOFTWARE VARIABILITY OVER TIME

UNDERSTANDING THE EVOLUTION OF REVISION HISTORIES THROUGH DATA-FLOW INTERACTIONS

SIMON JONAS FRIEDEL

July 24, 2023

Advisors:

Sebastian Böhm	Chair of Software Engineering
Florian Sattler	Chair of Software Engineering

Examiners:

Prof. Dr. Sven Apel	Chair of Software Engineering
Prof. Dr. Jan Reineke	Real-Time and Embedded Systems Lab

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT DES SAARLANDES

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

ABSTRACT

The complexity and number of developers of software projects make judging the impact a change has on the overall system difficult because changes can affect seemingly unrelated parts and developers of the system. Recently, the use of data-flow analysis has been suggested to improve the understanding of the impact of changes. By mapping commit data to a data-flow graph, we can extract semantic links between commits and authors from a data-flow analysis. Up to now, commit interactions have been mainly applied to snapshots of projects, i.e., analyzing only a specific revision. However, software projects change over time and so does their data-flow structure. For example, a functionality may be sparsely used when it is introduced, but over time it is adopted throughout the project and becomes a central part. To fully understand the impact of changes, we need to consider the entire revision history of a project.

We investigate how data-flow-based commit interactions change over time, and how they differ from the traditional high-level metric of lines of code for the analysis of software evolution. We find that changes to commit interactions of a commit occur without direct changes to the lines of the commit. Furthermore, the amount of interactions that change with a new revision does not correlate with their size. Additionally, we investigate how these metrics evolve when viewed from an author level. Here we find that the lines and interactions of authors change in a similar manner. However, we observe some authors that contribute proportionally many interactions despite only contributing few lines to the project.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

IR Intermediate Representation

VCS Version Control System

LOC lines of code

1 INTRODUCTION

Over the last 20 years, software projects have experienced an increase in complexity and the teams and communities that develop them have grown rapidly. The high complexity and number of developers in software projects make judging the impact a change has on the overall system difficult, as changes can affect seemingly unrelated parts and developers of the system. This can lead to an increase in bugs, as developers are not always able to grasp the entire impact of the change they introduce [19]. Furthermore, changes that affect the code of other developers require coordination with them [9] to prevent duplication of work or breaking code.

To better understand the interactions between authors and changes in a project, Sattler et al. [21] have proposed a combined approach of data-flow analysis and repository mining called SEAL. With this approach, they are able to detect small but central changes to the code base [21] using a combination of data-flow and repository information. Furthermore, they are able to infer interactions between developers which can be used to facilitate developer collaboration. However, their approach currently lacks an evolutionary component. That is, SEAL currently only analyzes specific revisions of projects, but does not incorporate a notion of time. However, software projects are dynamic systems, which change frequently and continuously over time [17]. When revisions are made to a software project the interactions between parts of the system change. For example, functionality which is central to the project now, may have been only a peripheral part when it was introduced. This means the impact of a commit can change over time and we need to consider the revision history of a software project to fully understand the effects of changes.

Building on their work, we use SEAL to analyze a multitude of revisions throughout the lifetime of a software project. From this, we can analyze changes to the interactions between commits within in the history of a software project. Our goal is to identify patterns in the history of software projects that were not visible before. For this, we investigate multiple aspects of the evolution of commit interactions. First, we examine the impact of revisions on the commit interactions of a system. Because commit interactions are based on the data-flow in a software project, we can use them to detect the parts of a project affect by a change. Furthermore, we can use the changes in commit interactions to find the affect commits and developers. We investigate how the centrality of individual commits changes over time. Changes to the centrality of a commit are interesting, as a functionality introduced by the commit, which becomes more central to the project, may then be used in environments it was never intended for. Understanding how the commit interactions and therefore centrality of a commit evolve can help us detect such changes. We also investigate how the contributions of authors to the interactions of a project change and how this differs from their contributions to lines of code (LOC). While we observe a severe difference between the commit interactions and LOC of a commit, both in impact and evolution. We find only few differences in the evolution of contributions by authors.

1.1 GOAL OF THIS THESIS

In this thesis, we investigate how changes influence the data-flow structure of a project and how the interactions between commits in the data-flow graph change over time as a result of that. This could reveal interesting patterns in the development of software projects which are not visible in high-level metrics, as data-flow-based commit interactions are tied to the semantics of code. We explore the revision history of projects using data-flow-based commit interactions in order to better understand the relations and impact of commits and authors within a project. For this, we analyze multiple revisions throughout the lifetime of various projects using SEAL and calculate the changes in data-flow-based commit interactions. We further compare these to changes in a traditional metric of software evolution, LOC, to understand what new insights can be gained from changes to commit interactions.

Our goal is to highlight interesting findings in the changes to data-flow-based commit interactions, such as revisions that change the interactions in a software project drastically, or major changes to the interactions of a commit without changing the code of the commit. Through our work, want to provide a basis for future studies on the evolution of commit interactions. For this, we investigate patterns, which emerge when analyzing changes in commit interactions, as well as outliers which break these patterns. To find potential causes for these patterns and outliers we perform qualitative analysis on them. While we are not able to rigorously identify what causes the patterns we discuss what factors may influence them to provide a basis for further studies.

1.2 OVERVIEW

First, we provide background on the various concepts and terms that are necessary to understand this thesis in Chapter 2. In Chapter 3, we introduce our research questions and discuss how we intend to answer them. We then present our results in Chapter 4 and discuss and interpret our results. Afterwards, we discuss potential threats to the validity of our findings. Chapter 5 provides an overview of related work from the related fields and previous works. In Chapter 6, we present a conclusion and lay out future work that arises from the answers to our research questions.

2 BACKGROUND

In this chapter we provide background on various concepts needed to understand this thesis. First, we explain Version Control Systems (VCSs), what they are, how they are used and how we use them to recover past states and changes of software projects. We also provide an introduction to the field of software evolution, and key findings from it which influence our work. We then introduce and explain the concept of data-flow-based commit interactions which we use in our analysis. Next, we provide an overview of the field of socio-technical analysis and the concept of developer roles. Lastly, we explain the commit types we use to categorize commits in our qualitative analysis.

2.1 VERSION CONTROL SYSTEMS

Most software projects are not developed by a single author, but many, which often are not working from the same location. To manage changes from multiple authors Version Control Systems are employed. Most VCSs work by recording changes in a repository. These changes are generally not saved every x amount of time, but whenever a developer *commits* to a change they made. Because of this, such a change is referred to as a *commit*. A commit generally consists of a few changed lines and represents an addition or change of a single closed functionality. To manage them each commit is assigned an identifier. Because each commit represents the change from a prior state or revision to a new one we can use them to build every revision of a software project from an initial revision. To accomplish this, we only need the initial revision of a software project which can also be seen as a change from a blank state, and the order of commits. Because a revision is constructed from a series of commits, the last commit in this series can be used to represent and identify the revision. This also means that the order of commits we store to build revisions represents the history of the software project and can be used to restore–*checkout*–any revision. To use this order, in which revisions appear in the history of a software project, in our analysis, we assign each revision a *TimeID* t. The TimeID is a sequential number starting with 0 at the initial revision and counts up with each following revision.

To help developers to manage these changes most VCSs attach metadata to each commit. This metadata generally consists of a message stating the intention of the change, the timestamp when the change was made and the author who made the change. While there are many VCSs which are used for software development, the most prevalent especially in the open-source community is GIT [6].

GIT GIT is a distributed VCS, meaning it uses not a single central repository, but every developer has their own local repository. These repositories can be synced using a server which hosts the repository. Each developer can then copy–*clone*–this remote repository, upload–*push*–their changes and later download–*pull*–changes others made. The repositories of open-source software projects are, as the name implies, publicly available. We use this to

```
        Commit    Author           Timestamp       Code

1884    33ae4134 (Jean-loup Gailly 1993-08-19)     if (trunc != NULL) {
1885    3ea7fe86 (Jim Meyering     2010-02-03)         do {
1886    3ea7fe86 (Jim Meyering     2010-02-03)             trunc[0] = trunc[1];
1887    3ea7fe86 (Jim Meyering     2010-02-03)         } while (*trunc++);
1888    3ea7fe86 (Jim Meyering     2010-02-03)         trunc--;
1889    33ae4134 (Jean-loup Gailly 1993-08-19)     } else {
1890    3ea7fe86 (Jim Meyering     2010-02-03)         trunc = strrchr(name, PART_SEP[0]);
1891    3ea7fe86 (Jim Meyering     2010-02-03)         if (!trunc)
1892    3ea7fe86 (Jim Meyering     2010-02-03)           gzip_error ("internal error");
1893    3ea7fe86 (Jim Meyering     2010-02-03)         if (trunc[1] == '\0') trunc--;
1894    33ae4134 (Jean-loup Gailly 1993-08-19)     }
1895    33ae4134 (Jean-loup Gailly 1993-08-19)     strcpy(trunc, z_suffix);
```

Figure 2.1: Example of *git blame* output from GZIP. Each line is annotated with the commit which last changed it, the author of the commit and the time of the commit.

clone the repositories of the project we analyze and access their histories. Like most VCS GIT assigns an unique identifier to each commit. For this, it uses a 40 digit SHA-hash, in this thesis we use an abbreviated form of 10 digits. It also provides the general metadata to a commit.

Additionally, GIT provides *git blame*, a tool, which can be used to query the ownership or blame data of each line of a project at the currently checked out revision. This blame data contains the commit which last changed the line, the author of the commit and the timestamp of when the commit was made (cf. Figure 2.1). From this, we get the lines of a commit which we use to analyze the changes in LOCs as well as to construct commit interactions. To differentiate between commits as units of code, i.e., the lines which were last changed by the commit and points in time we refer to the points in time as *revisions* and units of code as *commits*.

2.2 SOFTWARE EVOLUTION

Software systems are constantly changing and evolving. We investigate what changes to data-flow-based commit interactions can provide to improve the understanding of software evolution. Software evolution has been a topic of research for at least the past 50 years. In this time a set of observations known as Lehman's Laws of Software Evolution [16, 17] have been established. While these observations have been based on the evolution of industrial projects, they have been found to generally hold for open-source software projects [7]. Lehman's laws state, among other things, that a software project experiences continuous changes, growth and increases in complexity over time. This means a software project is never finished and requires effort after its deployment. Furthermore, the constant changes in software introduce errors and security vulnerabilities [10] which need to be addressed further increasing the development effort. In fact, studies estimate the cost of maintenance for a software project to be 70% to 90% of its total cost [15]. To reduce this cost we need to

understand how software evolves to be able to better guide the maintenance of software projects [23].

To study the evolution of software we need a measure of time, in the field of software evolution there are two approaches to measure time. One considers releases as steps in time, i.e. versions of a project which have been declared by the developers to complete some larger change. The other considers each revision in a VCS as a step in time. In this work we consider revisions as steps in time because even minor changes can greatly affect the data-flow-based commit interactions of a software project.

Software evolution traditionally focused on changes in high-level metrics like module count and LOC [7]. More recently ideas of integrating semantic information, e. g., function calls [23] were introduced.

2.3 COMMIT INTERACTIONS

We explore how commit interactions, which represent semantic connections between regions of code change and how they may help improve our understanding of software evolution. The concept of commit interactions combines high-level data from the VCS with a low-level data-flow analysis. In our work we use this to analyze changes in the data-flow structure of projects.

Commit interactions are extracted from the data-flow between code regions associated with commits.

COMMIT REGIONS Code regions are an abstraction used to combine blocks of code using commonalities. Most often code regions are used to group code in blocks which represent a functionality. However, we use the blame data to group the lines, i. e., we build blocks of code which belong to the same commit. For example, in Figure 2.1 we would build 5 regions of code one for line 1884 associated with `33ae4134`, one from l. 1885 to l. 1888 associated with `3ea7fe86`, another in line 1889 again associated with `33ae4134` and so on. We then use data-flow analysis to find which regions interact with each other through data-flow.

DATA FLOW & TAINT ANALYSIS Data-flow analysis is used to prove facts about a software system by analyzing connections in a program that are caused by the exchange of data. For example, when a variable X is assigned a value by one part of a software program and later read by another, data has flown between the instructions. Further any code that is influenced by the value of X directly or indirectly, for example, by a prior decision made based on the value of X, is part of the data-flow of X. This results in a graph structure with instruction as nodes and edges representing the data-flow. This representation is used by SEAL to infer commit interactions.

A method to extract the data-flow from a program is *taint analysis*. Taint analysis works by tracking values that have been tainted by one or multiple sources through a program. For our analysis any declaration of a variable is a source as we are interested in all interactions between instructions. By tracing the path of these tainted values through the program we find instructions that interact with a tainted value and there for with the sources of this value.

SEAL SEAL is an approach to combine low-level program analysis with high-level repository information introduced by Sattler et al. [21]. The approach is based on a compiler's Intermediate Representation (IR), an abstraction layer used in most modern compilers and static analysis tools. First, the instructions in the IR need to be annotated with the commit, which introduced them. These annotations are the later used to reconstruct the commit regions in the IR. For this, the compiler queries *git-blame* during the compilation to find the last change for each source-code line. The commit hash of the last change of a line is attached to the corresponding instructions. The resulting annotated program files are then analyzed by a static taint analysis encoded in Interprocedural Distributive Environment (IDE)[22]. IDE is a framework to build a context- and flow-sensitive, interprocedural data-flow analysis. Based on the data-flow graph provided by the taint analysis and the blame annotations on the instructions of the IR, data-flow interactions between commits can be extracted. Commit interactions are defined such that two commits c_1, c_2 interact with each other if there are instructions i_1, i_2 between which data flows, and i_1 belongs to c_1 and i_2 to c_2. As there can be multiple instructions of two commits between which data flows, there can be multiple interactions between two commits. We define the commit interactions i_{c1} of a commit c_1 as the sum of all interactions between c_1 and other commits c_x. The commit interactions of a commit c then represent the number of data-flow connections to other commits. Changes to the code of these commits affect the commit interactions of c. Using this we can detect the commits which are indirectly affected by a change.

2.4 SOCIO-TECHNICAL ANALYSIS

Socio-technical studies are concerned with the relations between the stakeholders of a software project. A major focus for socio-technical studies is the open source community, as they often lack traditional coordination mechanisms [19]. This means structure within developer relations are more of an emergent phenomenon than a strictly planned one.

Using commit interactions and the metadata provided by the VCS we can infer interactions between authors. The interactions between developers and the collaboration between them, is a focus of socio-technical research. Understanding how developers collaborate and what problems they face in their collaboration, helps to guide future development. Because the development in the open-source space is not managed by a corporate environment, the roles within a project are not clearly defined. Regardless, there are still role structures we can observe. These roles are often not clear, especially to new authors starting to contribute to a project [2]. However, understanding the roles of different authors is necessary to collaborate with them.

DEVELOPER ROLES A categorization of *core* and *peripheral* developers has been established over the years [12]. Core developers are characterized by consistent, and dedicated involvement in the project. They often have a major influence on decisions within a project and possess extensive knowledge of the systems architecture [19]. In contrast, peripheral developers show more irregular and sporadic contributions. The majority of contributors on a project are peripheral developers, there are often only a few core developers. However, these core developers are responsible for most of the work performed on project [19]. We investigate how the commit interactions of authors evolve and how their contribution to

the overall interactions compares to the contribution to the LOCs of a project. From this we examine if commit interactions of authors could provide a new view on author roles.

2.5 COMMIT TYPES

Changes made to a software project fulfill different functions. For example, some changes correct errors in the system, others introduce new functionality or improve functionality. This could influence the impact on the commit interactions of the system and the evolution of the interactions of the change. To investigate this, we categorize the commit we investigate in our qualitative analysis. For this, we use the categories specified by *Conventional commits*[1].

As software projects often are a collaborative effort it is important to clearly label the intention and function of a change. *Conventional commits* is a specification for the format of commit messages. Among other things, it specifies two commit types:

- *fix:* correction of a bug in the codebase.
- *feat:* introduction of a new feature.

Additional types which are often used and recommended include:

- *build:* changes to the build system or external dependencies.
- *refactor:* code changes that do not affect functionality.
- *perf:* improvements to performance.
- *test:* addition or correction of tests.

Studies on commit classification [8, 20] find similar categories. For example, they categorize commits as, corrective, feature additions, nonfunctional, perfective and preventive. Corrective commits are fixes–*fix*–, feature additions add new functionality–*feat*–, perfective encompasses both performance improvements and refactors, non functional incorporate build changes and preventive commits change or add tests. While there is not much difference between these classifications we use the categories of *Conventional commits* because we want to differentiate between refactors to reduce complexity and changes to improve performance. Additionally, some of our subject projects already use *Conventional commits* to specify their commits.

1 https://www.conventionalcommits.org/en/v1.0.0/ (visited 15.07.23)

3 METHODOLOGY

We want to investigate what patterns emerge when we analyze the evolution of commit interactions throughout the history of software projects. For this, we investigate how commit interactions change over time in 12 open-source projects from various domains. We consider the evolution of commit interactions as well as interactions between authors which can be inferred from commit interactions. First, we look at patterns in the evolution of commit interactions on a project-wide level to understand what insights can be gained from the changes to the data-flow structure of a software project. We then study the evolution of individual commits, to understand if, and how the evolution of the interactions of a commit relates to the changes in the LOC of the commit. Afterwards, we investigate the evolution of interactions when viewed on an author level. With this, we want to see if the observations from the first research questions hold, and if commit interactions may provide insights into the evolution of author roles.

3.1 RESEARCH QUESTIONS

Changes in commit interactions could provide a new view on software evolution because they represent connections between parts of a project based on the semantics of code. This means, we are able to observe how changes to a part affect the rest of a project, and how the connections between parts of a project change over time. Understanding when and how commit interactions change could help us improve the understanding of software evolution and the impact of changes. Our goal is to investigate how changes to commit interactions and LOC differ from each other and what factors could facilitate changes in commit interactions. With this we want to build an understanding of the possibilities commit interactions offer as an evolutionary metric, and what new insights can be gained from them.

REVISION IMPACT Changes introduced by a revision can vary greatly in size, from single lines to hundreds of lines. At first glance, the size of a revision could seem to be a decent metric to measure the impact of the revision on the data-flow structure of the program. But small changes can have a large impact on a software system. For example, fixing the date issue of software systems before the year 2000, which was caused by date fields generally storing just the year and the decade, because the century had always been the same since the invention of computers. At first glance, this is a simple fix of extending the date field by two digits to accommodate the century, but such a small change effects every usage of such fields which were often not apparent [3]. This means we need to consider the centrality of the lines, affected by a change, in the data-flow structure of the program in order to judge the impact of the change.

This is related to the field of *Change Impact Analysis* which aims to predict the impact of changes to judge the work needed to implement it and identify affected parts of a

system [4, 18]. Some techniques from this field incorporate semantic information to estimate the impact of changes [18]. However, using changes in commit interactions to determine the impact of a change would also provide direct information on the authors whose code would be affected by the change. We address the following research question, to investigate what insights can be gained from a impact metric based in commit interactions:

RQ. 1: *How do changes affect the interactions of all commits?*

To answer this question, we investigate how an impact metric for revisions, derived from changes in commit interactions, relates to the size of the textual change the revision introduces and what other factors may influence the impact of a revision on the commit interactions of a project. We further investigate the patterns that emerge in the change history of projects when analyzing the impact of revisions on commit interactions, to find new insights on the effects of revisions, gained from the changes in commit interactions.

COMMIT EVOLUTION Software projects are complex systems with many interconnected parts, which change over time and with them the project. Commit interactions provide a view on the semantic connections between the parts of a software project. Understanding how the connections between parts of a project evolve, could help to guide the maintenance of software projects. For example, when the interactions of a commit drastically increase, i. e., its functionality is used more often throughout the project, it may be useful to reexamine the functionality. As it may now be used outside of its intended use case, which could introduce bugs. Over the lifetime of a commit its interactions may change as revisions are made to the commit's code or the rest of the code base.

We investigate the evolution of the commit interactions of individual commits to find when and why their interactions are affected and how changes to the code of a commit relate to the changes in interactions. Additionally, we are interested in cases where the code of a commit is reworked without affecting the interactions of the commit, as this means the parts responsible for the interactions were not changed. This could provide insights how the impact or centrality of commits evolves over time. We address the issue of changes in the centrality of commits in our second research question.

RQ. 2: *How do the interactions of individual commits change over time?*

To answer this question we analyze the changes and evolution of commit interactions and lines of individual commits. From this, we intend to find interesting patterns in the comparison of the evolution of commit interactions and lines. These patterns could help us understand which parts of a software project become more central and interconnected and what parts tend to lose their connections.

AUTHOR INTERACTIONS Interactions between commits can be helpful to understand changes to the structure of a software project. But to better understand the development process, we also need to consider who introduced the commits. By grouping commits by their author, we can analyze the evolution of interactions between authors. We want to understand how changes to the interactions between authors happen, what causes them, and how the amount of interactions of an author relates to the size of their contribution to the code-base. When projects evolve and change, so do the roles and participation of

Project	Domain	LOC	Commits	Authors	Analyzed Revisions
BZIP2	Compression	6 866	168	24	96
BROTLI	Compression	40 837	1 106	100	56
FILE	UNIX utils	19 708	4 530	8	168
GREP	UNIX utils	5 304	2 205	46	234
GZIP	Compression	8 121	605	17	48
HTOP	UNIX utils	30 018	2 827	193	292
LEPTON	Compression	72 445	968	31	105
LIBPNG	File format	74 646	4 199	78	163
LIBSSH	Protocol	99 866	5 482	143	0
LRZIP	Compression	18 296	978	31	103
LZ4	Compression	21 035	2 971	164	82
XZ	Compression	38 441	1 866	32	137

Table 3.1: Overview table of our subject projects.

developers (cf. Section 2.4). Joblin, Apel, and Mauerer [12] find that the core developers of a project generally remain stable, while peripheral developers tend to contribute irregularly. However, they also observe core developers leaving a project and peripheral ones rising to the role of a core developer. Their approach uses a heuristic-based metric to find semantic connections between artifacts, i. e., parts of a project which depend on and affect each other. The interactions between developers we infer from commit interactions contain actual semantic connections.

Core developers are often responsible for most of the work performed on a project. This means we would expect them to contribute the majority of lines and interactions of a project. We address the relations of developer contributions in our third research question.

RQ. 3: *How do author contributions to the interactions of a project change over time?*

To answer this question we analyze the contribution to the overall interactions of a project from each author. We then compare this to their contributions to the lines of a project. With this we aim to understand what new insights commit interactions can provide to the concept of author roles.

3.2 OPERATIONALIZATION

To evaluate our research questions, we analyze 12 open-source software projects (cf. Table 3.1[1]) from various domains. For our analysis, we first sample a set amount of revisions from each project excluding any revisions we cannot analyze due to bugs in the projects or outdated build requirements we are unable meet. For BROTLI, LEPTON, LRZIP and LZ4 we start with 50 revisions, because they have relatively short histories or we cannot analyze

1 Metrics taken on 19.07.2023

early parts of the history. For GZIP, we could only analyze the last 48 revisions because prior revisions require an outdated dependency we are unable to provide in our build environment. For BZIP2, we sampled the entire history but are only able to analyze 96 revisions, because of bugs in BZIP2. For all other projects, we started with an initial sample of 100 revisions. After we analyzed the sampled revisions we resample additional revisions for all projects to avoid spurious findings caused by the choice of revisions. We sample an additional revision between revisions that show drastic changes in our data but are not direct neighbors in the VCS. This process is repeated until we either smooth out the changes or find the revisions causing them. Revisions for which our analysis failed were removed, resulting in a total of 1610 revisions we analyze. Each revision is analyzed using SEAL to gather the data-flow-based commit interactions. Additionally, we collect data on the number of lines belonging to each commit at the analyzed revisions using GIT's blame tool. From this, we get the amount of interactions $i_{c,r}$ and number of lines $l_{c,r}$ for each commit c at all sampled revisions r.

3.2.1 *Revision Impact*

To answer our first research question (RQ. 1), we analyze changes in the number of interactions each commit is involved in. For this, we calculate the changes in commit interactions for every commit between each sampled revision and its predecessor. This gives us one value per revision r and commit c that represents how much the interactions involving c changed in r compared to the previous sampled revision. Using these changes, we calculate an *impact* score for each revision. We define this impact score as the number of commits whose interactions were affected by changes since the last revision relative to the total amount of commits.

Definition 1. *The commit interactions based impact of a revision. This metric indicates how many parts of a project are affected by a change.*

$$\mathsf{Impact}(r) = \frac{\mathsf{Commits\ which\ were\ affected\ by\ changes\ from\ r}}{\mathsf{All\ commit\ which\ contribute\ to\ the\ code\ at\ r}}$$

To relate the impact of a revision to its size we also need to calculate the *churn* of the revision, i. e., the total number of lines, which are changed by the revision. From this, we can calculate the relative change in LOC as the ratio of *churn* of a revision and the total LOC at this revision.

Definition 2. *The churn of a revision.*

$$\mathsf{Churn}(r) = \sum_c \left| l(c,r) - l(c,r-1) \right|$$

$$\mathsf{RelativeChurn}(r) = \frac{Churn(r)}{\sum_c l(c,r)}$$

We then calculate the correlation coefficient between the *RelativeChurn* of a revision and its *impact* for each project. We use the Pearson correlation coefficient because we are interested in the linear relation between these metrics. From this we want to see how much the amount of changed lines affect the impact of a revision. We also calculate a metric for the number of changed interactions similar to the *RelativeChurn*.

Definition 3. *The relative changes in interactions a revision causes.*

$$\mathsf{InteractionChange}(r) = \frac{\sum_c \left| i(c,r) - i(c,r-1) \right|}{\sum_c i(c,r)}$$

We again calculate the correlation of this metric to the *Impact* of a revision, to see how much they align.

Another factor which might affect the impact of a revision is its age within the project's history, to investigate this effect we calculate the correlation between a revisions impact and its TimeID. The TimeID is a sequential number assigned to each revision according to their order in the history of a project (see Section 2.1). We also perform a qualitative analysis on the 100 revisions with the highest impact, for which we also analyzed the revision directly prior in the VCS (i. e., we have the exact revision causing the change.). From this, we aim to find potential factors for their high impact. We investigate the author and commit message of the revisions as well as the changes which were introduced by the revision. We are interested in whether the author of the revision is one of the main authors of the project. For this, we use the 80th percentile threshold for the number of commits of each author, because this is widely used as a simple metric to determine author roles [11]. From the commit message and code changes introduced by a revision we categorize the revisions as bug fixes, feature additions, performance improvement, refactors or maintenance. These categories are mostly based on the *Conventional Commits* specification (cf. Section 2.5), as this is also used by some of our projects and provides an intuitive categorization. However, we group the addition of tests and changes to the build system as a maintenance category as we found few commits in these categories and the cause for the high impact of these revisions is similar. Through this, we want to find potential patterns which may cause revisions to have a high impact. For example, one could imagine revisions, that fix a bug in central functionality to have a high impact on interactions with a very small impact on LOC. We perform the same categorization on 100 randomly selected to be able to compare our results.

3.2.2 *Commit evolution*

To answer RQ. 2, we analyze the relation between changes to the LOC of a commit and changes to the number of interactions it is involved in. We are mostly interested in commits with a divergent evolution of commit interaction and LOC. To find commits that have a divergent evolution we calculate how the interactions per line of commits change. We calculate the interactions per LOC for each commit at each revision.

Definition 4. *Interactions per line of code.*

$$s(c,r) = \frac{i(c,r)}{l(c,r)}$$

To be able to better compare the changes in s, we then normalize s to the interactions per line of each commits at its introduction and calculate the change between sampled revisions.

Definition 5. *The normalized change of s.*

$$s_\Delta(c,r) = \left(\frac{s(c,r)}{s(c,c)}\right) - \left(\frac{s(c,r-1)}{s(c,c)}\right)$$

We then investigate commits that drastically increase or decrease in interactions per line. For this, we use the sum of s_Δ over a commits lifetime as a metric for the divergence of interactions and lines of commit over its lifetime.

Definition 6. *The divergence between the lines and interactions of a commit.*

$$d(c) = \sum_r s_\Delta(c, r)$$

We more closely examine commits below the 5% and above the 95% quantile for d of each project. This gives us 72 commits, which experience drastic changes in their interactions per line compared to other commits in their project. For these commits, we examine the evolution of the interactions and lines of these commits directly, and perform a qualitative analysis on them by investigating their commit message and change as well as who introduced them. We use this to categorize the commits like for Section 3.2.1 and use the categorization of the randomly selected revisions we performed to calculate an expected distribution for this sample.

3.2.3 *Author interactions*

To answer RQ. 3, we aggregate the commit interactions by the authors of the commits at each revision, to get the total interactions for each author. As before, we calculate the number of LOC of each author at every revision we analyze. We then normalize the number of interactions and LOC of each author to the total interactions and size of the project at each revision. This gives us the relative contribution of each author to the respective metric of the project. Furthermore, it removes any changes caused by the growth of the project and enables us to detect changes in the impact of each author on the software project. Changes to an author's share of overall interactions that are not reflected in the size of their contribution could indicate changes to the role of the author. As this indicated that the existing code of the author or their new contributions became more or less central to the project. We compare the evolution of the contributions of an author to the lines and interactions of a project. Through that, we aim to identify interesting instances of change in these metrics. We also investigate how the contributions to interactions and lines of authors relate to each other. Authors, who only contribute to very few lines of a project, but have larger contribution to the interactions of the project could be of interest. These authors contribute only in small ways to the project so they would be classified as *peripheral* developers, but because of the high centrality of their code they are more important to the project than most *peripheral* developers.

4 EVALUATION

In this chapter, we present the results of our evaluation and answer our research questions. First, we present the results for each research questions. Afterwards, we discuss and interpret the results. In the end, we discuss potential threats to the validity that arise from our experiment design.

4.1 RESULTS

We analyze over 1600 revisions across 12 projects to understand how commit interactions change and evolve over time. We first look at the effect of changes on each project. We then investigate how the interactions of individual commits change over time. Lastly, we focus on the interactions of authors to find what we can learn about changes in author roles.

4.1.1 *Revision Impact*

We want to understand how the impact of a revision on the interactions of a project differs from the size of code changes it introduces. Furthermore, we want to understand what other factors influence the impact a revision has on the software project. For this, we calculate the changes in interactions and LOC for each commit between revisions.

To investigate if there is a direct relation between the churn of a revision and its impact, we calculate the Pearson correlation coefficient between *RelativeChurn(r)* and *Impact(r)* of a revision. Table 4.1 shows the correlation we calculate for each project and over all revisions. A high positive correlation would indicate that revisions that change large parts of their project have a high impact. While the corelation vastly differs between projects the highest correlation coefficient we find is in LEPTON with 0.6. For one project–BZIP2–we calculate a slightly negative correlation coefficient, which would indicate that smaller revisions have high impact, but with an absolute value smaller than 0.05. Over half of our projects show an absolute value of the correlation coefficient of less than 0.2. Overall we observe a slight positive correlation of 0.134. This shows that there is only a small correlation between the size of a revision and its impact. This means the impact of a revision does not depend on its size and other factors have more influence.

To investigate how our *Impact* score relates to the number of changed interactions, we calculate the correlation coefficient between the *Impact* and *InteractionChange* of revisions. We find an even greater difference between project for this correlation. For BROTLI, FILE, GZIP, LIBSSH and LRZIP we calculate a high correlation coefficient of over 0.8. This indicates that in these projects revisions that change a large portion of commit interactions generally have a high impact. In contrast, HTOP, LIBPNG and XZ show a relatively low correlation of less than 0.5, indicating that the *Impact* of a revision does not depend on the number of interactions it changes.

Projects	TimeID	InteractionChange	*RelativeChurn*
BROTLI	−0.028	0.870	0.555
BZIP2	−0.070	0.765	−0.018
FILE	−0.082	0.954	0.248
GREP	−0.066	0.707	0.149
GZIP	−0.370	0.937	0.077
HTOP	0.038	0.466	0.151
LEPTON	−0.426	0.612	0.599
LIBPNG	−0.181	0.468	0.444
LIBSSH	−0.016	0.806	0.168
LRZIP	−0.035	0.890	0.184
LZ4	−0.039	0.707	0.602
XZ	−0.273	0.219	0.132
overall	—	0.3381	0.134

Table 4.1: Pearson correlation of the *Impact* of a revision to various metrics. A negative correlation with TimeID indicates, older revisions have a higher *Impact*. *InteractionChange* measures the number of changed interactions, a high positive correlation indicates that revisions which change many interactions also have a high impact. *RelativeChurn* measures the size of a change, a high positive correlation would indicate larger revisions to have more impact.

We also calculate the correlation between the impact of a revision and its TimeID, which is a sequential number assigned to each revision with the initial revision being assigned 0. A high negative correlation would indicate that early revisions have a higher impact. Again, the highest correlation is observed in LEPTON with −0.64. This again indicates that the age of a revision does not greatly affect its impact.

We find that 90% of revisions change less than 1% of the LOC of their project, while their impact can vary greatly. Figure 4.1[1] shows the *Impact* of revisions on the overall project in relation to their *RelativeChurn*. Here we see an accumulation of revisions which have a very small impact and change less than 0.5% of the projects lines. This accumulation contains revisions from all projects. We find some revisions which change over 1% of lines but have no impact on the interactions. These revisions mostly belong to FILE. There are two more regions with a high number of revisions, one at an impact of slightly more than 50% and one between 80% and 90%. These two accumulations mostly contain revisions from different projects. We observe relatively few revisions with an impact between 20% and 50%.

Figure 4.2 shows the distribution of the impact of revisions for each project, with the overall distribution as a kernel density estimation on the margin. Overall, we observe a bimodal-distribution for the impact of revisions, with the least frequent impact around 40%. We find that most projects follow a bimodal-distribution with one mode at 0% impact. The location of the second mode as well as the width of the modes varies between projects. GZIP

1 The plot excludes a few commits, that have a very large *RelativeChurn* and can be considered outliers(e. g., large scale code reformating). We use Tukey's fence($k = 3$) to exclude these outliers, this removes data point which lie over k times the interquartile range above the third quartile.

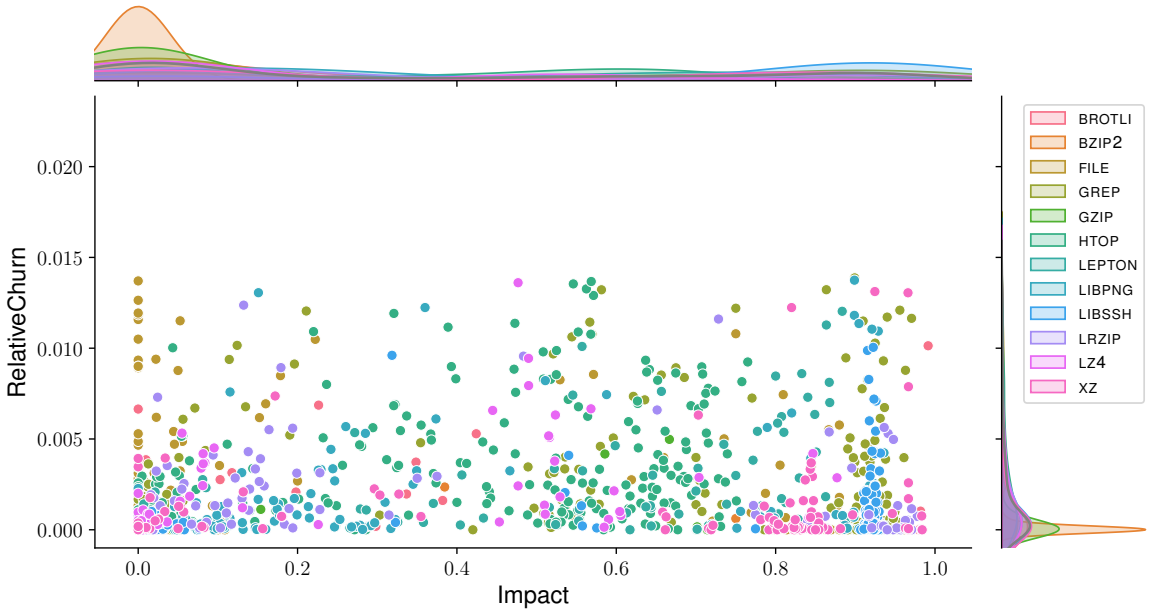


Figure 4.1: Scatterplot comparing the *Impact* and *RelativeChurn* of revisions. The margins show kernel density estimations for the metrics. Point in the lower left corner represent revisions that have little impact and change few lines. Overall, most revisions change few lines, but their impact varies greatly.

and BZIP2 show almost no revisions with an impact of more than 1%. LZ4 shows more of a tri-modal distribution with a mode of 50% and another at 80%.

To find potential factors that cause a revision to have a high impact we perform a qualitative analysis on 100 revisions with the highest impact, where we found the exact revision causing the change. For this, we examine the author of the revision as well as its commit message and the change it caused. Of the 100 revisions we examined, 82 were made by one of the main authors of the project, this is close to the distribution we find for the 100 randomly selected revisions of 88 revisions made by a main author. We then categorize the revisions to see if the type of change influences its impact. Table 4.2 shows the number of revisions we found for each category, as well as the distribution we found from a 100 randomly selected revisions. Of the 100 revisions with high impact 44 implement a bug fix, e. g., revision `1a604a05a5`[2] of HTOP. 17 were identified to add a new feature, for example, revision `19d86fb9a6`[3] of BROTLI. We found 18 revisions that refactored some part of the project, e. g., `9365ed6536`[4] of GREP, 14 performed some type of maintenance, e. g., `3909cddc8e`[5] from GZIP, 2 made changes in order to improve performance, e. g., `a9477d1e0c`[6] in XZ, 4 contained changes from multiple categories.

2 Message:"*BUGFIX: behavior of 'F' (follow) key was broken, also affecting the persistence of mouse selections. Closes #3165065.*"
3 Message:"*Merge-in SharedDictionary feature (#916)*"
4 Message: "*grep: prepare search backends for thread-safety To facilitate removing mutable global state from search backends, . . .*"
5 Message: "*maint: update gnulib to latest; also update bootstrap and init.sh*"
6 Message: "*liblzma: SHA-256: Optimize the way rotations are done. . . .*"

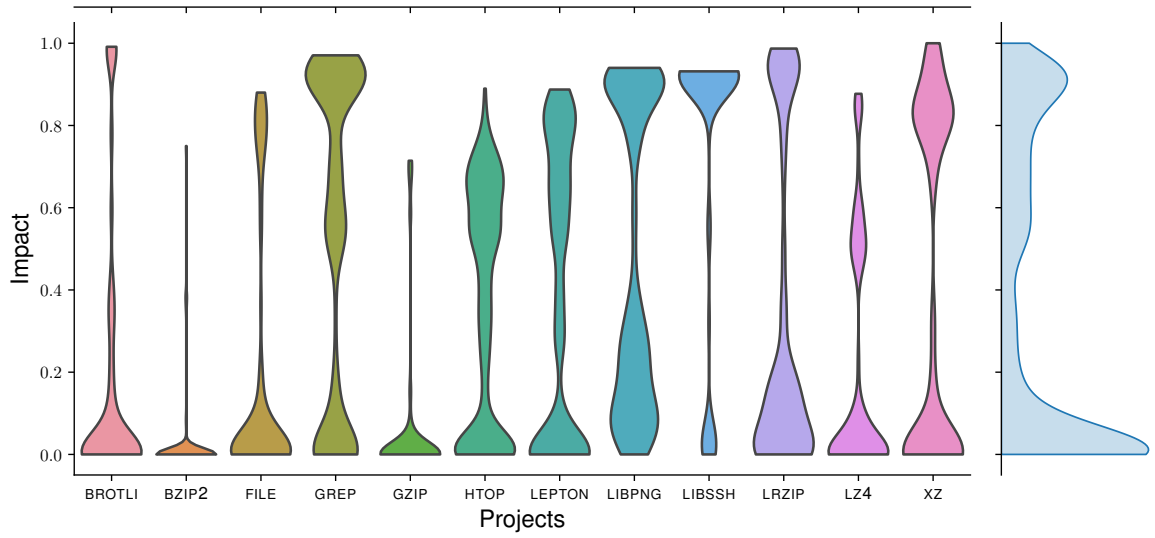


Figure 4.2: Violin plot showing the distributions of the *Impact* of revisions on commit interactions. The margin contains the overall distribution over all projects.

Category	RandomSelection	High Impact
bug fix	34	45
feature	10	17
refactor	15	18
maintenance	37	14
performance	3	2
multiple	0	4

Table 4.2: Table shows the categorization of commits with a high impact. RandomSelection shows the distribution we got from 100 randomly selected revisions.

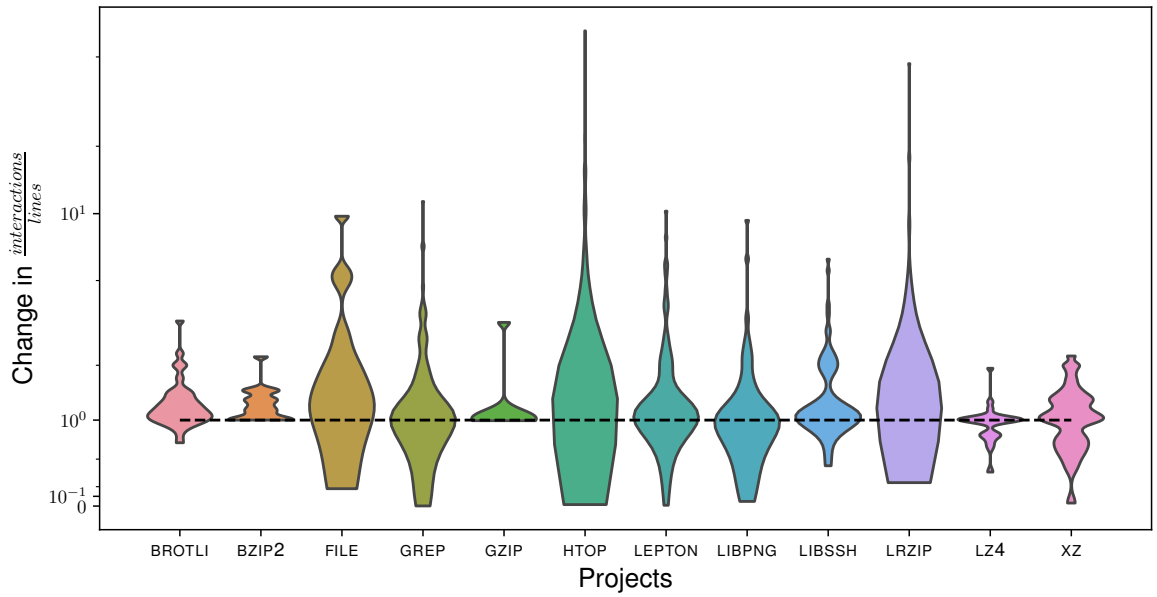
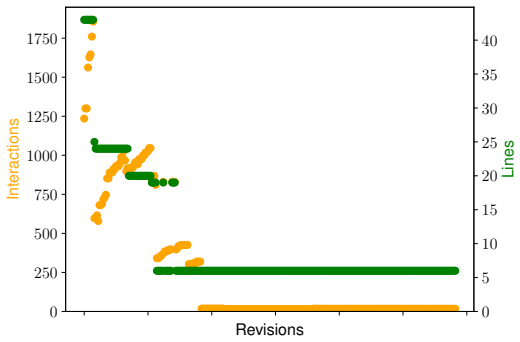


Figure 4.3: Violin plot of the change of interactions per line relative to the ratio at introduction of each commit by project. Dotted line represents no change over the life-time of a commit.

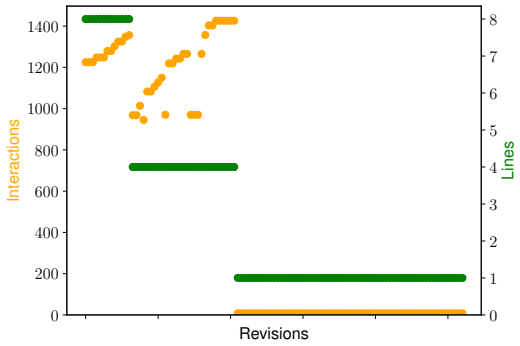
4.1.2 *Commit Evolution*

As a project evolves, code that was introduced by a commit is often rewritten and then belongs to a new commit. Similarly, the interactions of a commit change over time as the functionality it introduced is adopted more widely or superseded by other code. Figure 4.3 shows the distribution of the divergence d for the evolution of lines and interactions of commits for each project. We find that for the majority of commits the ratio of interactions per LOC remains relatively stable over the commits lifetime. In HTOP, FILE, LRZIP and LEPTON we observe a wide spread of evolutions. For the other projects we mostly find a large mode of commits, which experience little to no change, and a few commits which change drastically. While we observe an overall trend of a slight increase of interactions per LOC, the extrema of change are almost equally positive and negative. We find multiple commits which almost stop producing interactions at some point in time, while some lines of the commit remain. This drop in interactions does not always occur with change to the commits lines is made. For example, the number of interactions of HTOP@`855d9eaf9a` (see Figure 4.4a) drop from roughly 250 to nearly 0, while there are no changes made to its code. We also find examples where it does coincide with a change to the lines of the commit, e. g., HTOP@`cb297af848` (see Figure 4.4b) where we observe a rework which removes all but one line of the commit, this removes all but 8 of the previously 1400 interactions. For most commits, we observe an increase of interactions in periods with no change to the lines of the commit. However, some commits show extreme growth in interactions even when their lines are reworked. For example, in the evolution of HTOP@`93f091c47e` shown in Figure 4.4d, we observe a growth form around 1000 interactions to 5000 interactions.

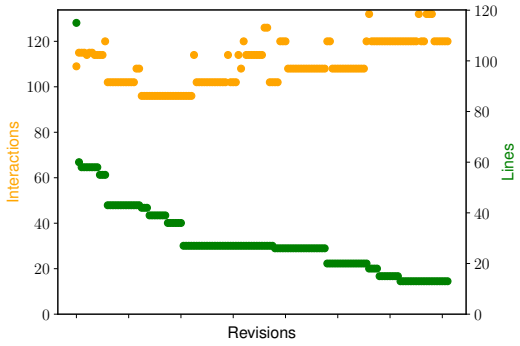
We also observe multiple commits with an evolution similar to Figure 4.4c, which shows the evolution of FILE@`ace9da574a`. Here we observe regular and often drastic reworks of



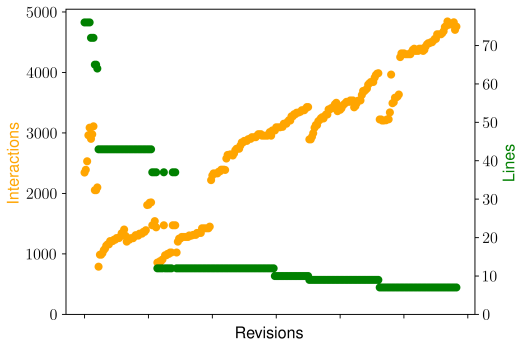
(a) HTOP `855d9eaf9a`, interactions disappear without change to lines.



(b) HTOP `cb297af848`, interactions disappear with change to lines



(c) FILE `ace9da574a`, interactions remain stable but lines are continously reworked



(d) HTOP `93f091c47e`, the lines disappear initially but are the stable while the interactions increase.

Figure 4.4: Plots comparing the evolution of lines and interactions of individual commits.

Category	RandomSelection	Increasing	Decreasing
bug fix	12, 24	12	15
feature	3.6	3	2
refactor	5.4	12	15
maintenance	13.32	5	3
performance	1	4	1

Table 4.3: Table shows the categorization of commits with high change in interactions per line. RandomSelection shows the distribution we would expect based on 100 randomly selected revisions.

their lines while their interactions remain mostly unaffected by that change. In the commits with this behavior, we find multiple instances where the code of a commit is reworked, immediately after its introduction. Further investigation shows that such reworks are mostly performed by the main author of the project.

We then perform a qualitative analysis on 72 commits which represent the 5% of commits, of each project, with the highest increase and decrease in interactions per LOC. We categorize the commit based on their commit message and the change they implement (cf. Table 4.3) to find potential relations between the type of change and the divergent evolution. We find the majority of commits with a high change in interactions per line to be bug fixes and code refactoring. We found 5 commits which add a new feature, 3 with increasing interactions per line. 8 of the commits performed a form of maintenance 5 of them increased in interactions per line and another 4 with increasing and one with decreasing interactions per line made some performance improvement.

4.1.3 *Author Interactions*

Software projects are often developed by multiple authors. Over time the authors and their contributions to the the project can change. We use the ownership data attached to commits to infer the interactions of authors from the commit interactions of a software project. With this, we investigate the changes in the author contributions to the interactions of a project.

Most of our projects are maintained by a single author with very small contributions by others. Because of this, we focus on GREP, HTOP and LIBSSH which have more diverse contributions. Figure 4.5 compares the contributions of authors to the lines and interactions of HTOP and GREP. Each area plot shows the relative contributions from authors for one of the metrics and projects. We find that changes to the contributions of an author are generally reflected in both metrics.

While the main author of a project generally has the highest contribution for both the lines and interactions of a project, their relative contribution to the interactions of the project is often higher. For *peripheral* developers we mostly observe the contribution to the lines of the project to be larger than to the interactions. However, for some *peripheral* authors we observe a substantially higher contribution to the interactions of the project. For example, in GREP Jim Meyering contributes roughly 10% of the interaction but less than 1% of lines.

Similarly, Atis Adamantiasdis has roughly 25% of lines of LIBSSH but around 50% of the interactions, while Anderson Toshiyuki, for example, contributes roughly 7% of the lines but only 1% of interactions.

The overall distribution of contributions remains mostly stable for each author, with only small changes, but we also find instances where the contributions in lines and interactions of authors drastically change. For example, we observe a change in HTOP when Christian Göttsche starts contributing to the project, and their contribution quickly grows to 40% of the interactions (cf. Figure 4.5a) of HTOP. This coincides with a documented change in main author from Hisham Muhammad to Christian Göttsche and others[7]. In the beginning, we find that Hisham Muhammad has 80% of lines but almost 100% of interactions. When Christian Göttsche takes over as a main author their lines only grow to 20% of the project but they quickly take over almost 40% of the interactions, while other authors gain contribution in lines but not as much in interactions. A similar change in the contributions can be observed for GREP where Paul Eggert becomes the author with the most contributions to lines and interactions, replacing Alain Magloire and Paolo Bonzini (cf. Figure 4.5b). This change is more pronounced in the change in contributions to the lines.

4.2 DISCUSSION

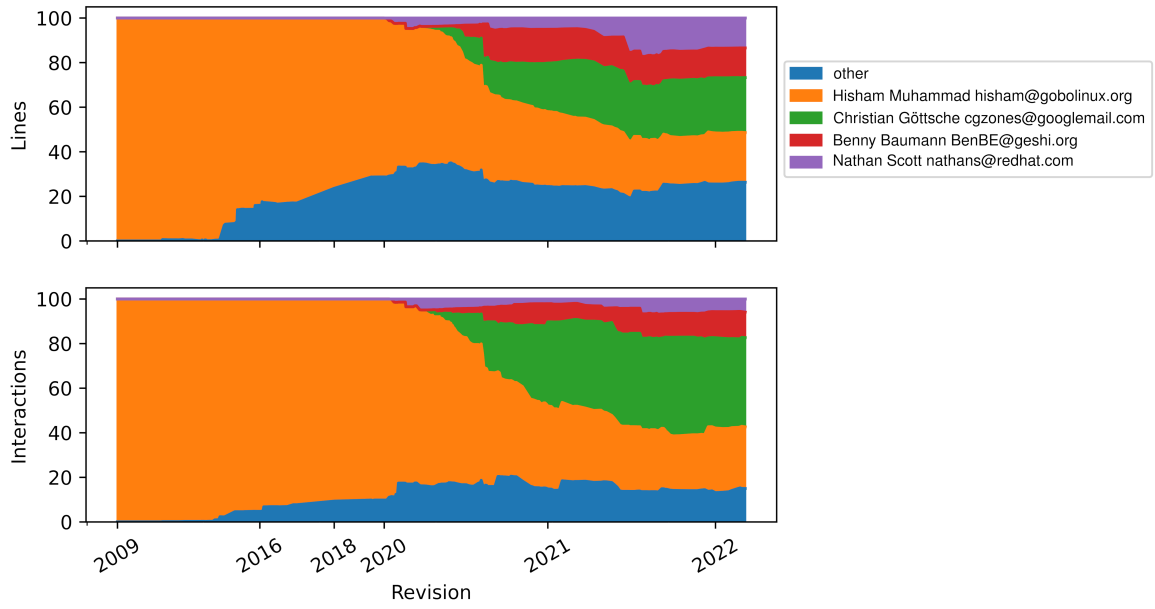
In this section, we discuss our results and highlight insights that we gained from our work.

4.2.1 *Revision Impact*

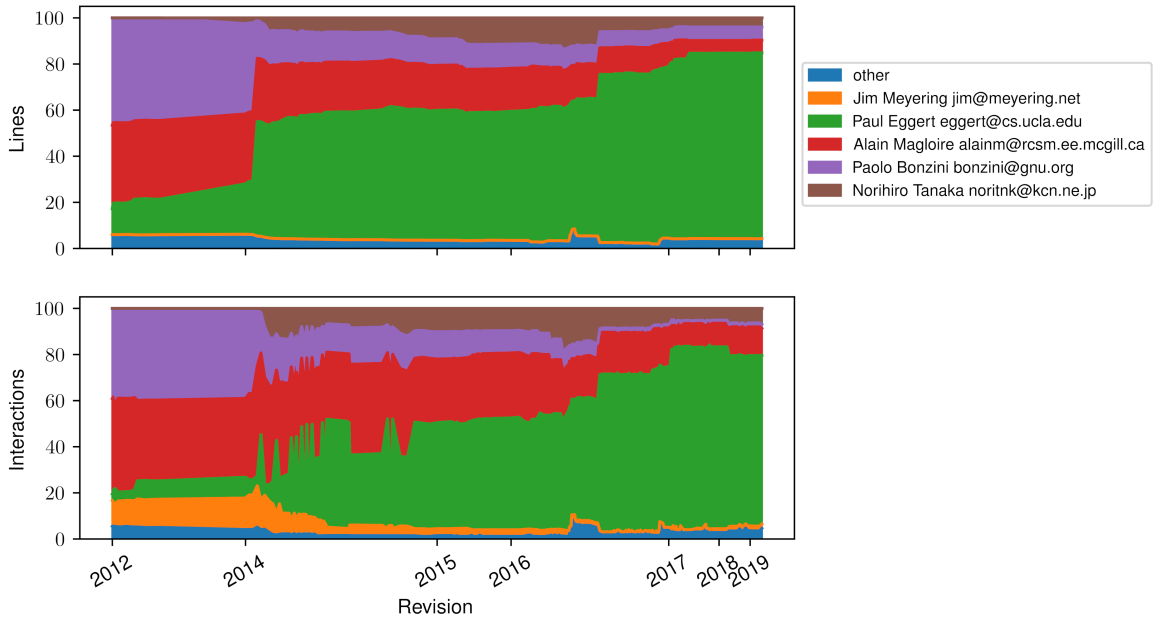
We want to understand which factors influence the impact of a revision on the data-flow structure of a project. For the traditional metric, LOC we find only a low correlation between the size of a change in LOC and its impact on the commit interactions of the system. This indicates that some lines have a higher impact on the project than others. This can also be seen in Figure 4.6, which shows the changes to the LOCs and interactions of each commit in a project. While we see some cases where changes to lines and interactions of a commit happen at the same time, this is not the case in general. This indicates that there could be two types of code regions, central and peripheral ones, where small changes to central code affect the interactions of most of the project. This aligns with the findings of our qualitative analysis where we find that 37 of the 44 bugfixes, with high impact, implement small changes to central parts of the respective projects. On the contrary, large changes to peripheral code may not even affect the commits whose lines where changed. This can also be seen in Figure 4.1 where we find multiple revision with little to no impact, which change relatively large parts of the project.

This categorization in central and peripheral code is also reflected in the distribution of the impact of revisions shown in Figure 4.2. Here we find a bimodal distribution in most projects, the first mode around 0% impact could represent changes to peripheral code, and the second mode with a high impact represents changes to central code.

7 Authorship declaration of htop: https://github.com/htop-dev/htop/blob/main/AUTHORS (Last visited 14.07.23)



(a) HTOP



(b) GREP

Figure 4.5: Area plots comparing the evolution of author contributions to the lines and interactions. Authors with a maximum contribution of less than 10% in both metrics are grouped as *other*.

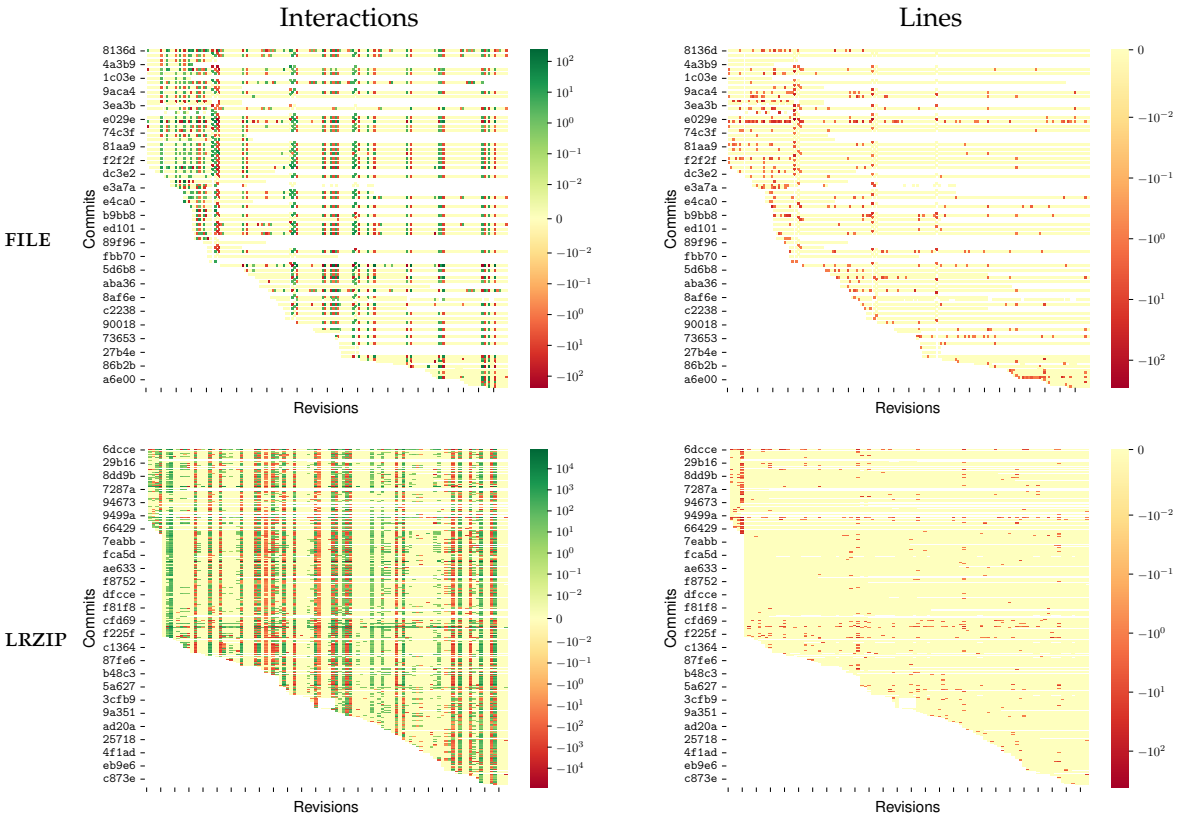


Figure 4.6: The plots show changes in the interactions or lines of commits caused by revisions. The x-axis represents the revisions we sampled as points in time, the y-axis shows the commits of which we find some contribution at the points in time. Each cell then represents the change in the metric for a commit compared to the previous revision.

One factor we examine is the age of a revision, we find that there is also no correlation between the impact and age of a revision. This could relate to Lehman's forth Law, which states that the work performed on a project remains invariant over time (i. e., while the project growths and increases in complexity the work performed on it remains constant).

Because we only sample additional revisions in regions where we observe a changes in interactions, the steps between revisions with low impact are often large. Additionally, because we measure the change in interactions per commit, we detect changes even if the total number of interactions remains constant. This means there are large regions in time where changes do not affect the interactions of the project. One cause for these low impact regions could be the release of a new version, for example, when all features of a new version are added and only minor changes are made to polish them for release. For example, at the start of the first large region with low impact revisions in FILE we find revision `b293f989933`[8] which introduces version 5.04 of FILE. This again indicates that the location and other factors of a change within the projects architecture affects its impact more, than the size of the change. While we find that the impact of revisions follows a bimodal distribution for most projects, the exact shape of this distribution drastically varies between projects. Similarly, the correlation between the size of a revision and its impact, while generally low, also shows drastic differences between project. Furthermore, we would expect the number of changed interactions and the impact of a revision to correlate, but we find multiple projects for which we calculate only a small correlation coefficient. This could be caused by architectural differences, as in a monolithic design changes would be more likely to have a high impact, while in a highly modular design with only a small core part the impact of changes would most often be restricted to the affected module.

Our qualitative analysis shows that for the most part the simple categorization does not have much relation to the impact of a revision. However, we observes much fewer revisions performing maintenance and a few more feature additions, refactors, and bugfixes, than in the randomly selected revisions. This is expected, because, for example, changes to documentation would also be maintenance, but do not change any code lines and therefore do not affect commit interactions.

In conclusion, we find most revisions either have little to no effect on the commit interactions of a project or greatly affect them. The size of a revision has very little influence on its impact, so does the age. Our simple categorization also shows only small differences to a random sample, indicating that the factors influencing the impact of a revisions are more complex.

4.2.2 *Commit Evolution*

Changes to the interactions of commits do generally not occur as a result of changes to their lines. We want to understand how the interactions of commits evolve and what factors might facilitate a divergent evolution from their lines.

We find that for most commits the ratio of interactions per LOC remains relatively stable with only a slight overall increase. This slight increase is expected, as Lehman's Laws (cf. Section 2.2) state that projects are constantly growing and increasing in complexity. The growth of a project would mean that the overall interactions of a project also increase and

8 Message: "welcome to 5.04"

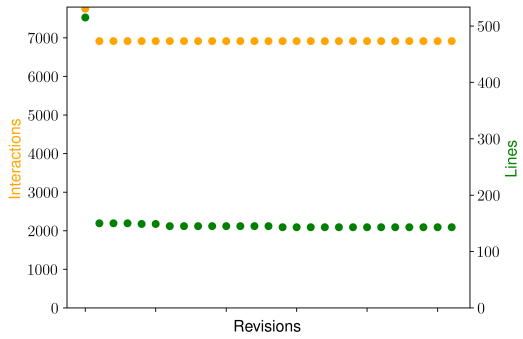


Figure 4.7: Figure shows the evolution of GZIP@7a6f9c9c32 which implements compatibility with a new hardware acceleration. The changes of this revision are directly reworked by Paul Eggert, aligning the changes to the style of the project. While this changes most of the lines, the majority of interactions of the original commit remain.

with that the interactions of individual commits. Similarly an increase in complexity could cause an increase in interactions within a project as a higher complexity is often related to more interconnections within a program.

While the number of interactions and LOC do not drastically diverge for most commits, we are interested in cases where they do. Divergent evolution of interactions and LOC of a commit occur as an overall increasing or decrease of interactions per LOC.

We find two types of commits which show an overall decrease in interactions per LOC. One which behaves like HTOP@cb297af848 (cf. Figure 4.4b), where a change in interactions occurs with a change in lines. For these changes the lines that caused the majority of interactions are reworked of removed, while the remaining lines produce little to no interactions. In contrast, we find commits where the interactions shrink without a change in LOC, for example HTOP@855d9eaf9a (cf. Figure 4.4a). This means the functionality implemented by the commit loses importance. When this occurs because the functionality was superseded by something, the drop in interactions could be used to detect this and notify the authors of the remaining usages. A complete removal of interactions would then indicate that the remaining lines are basically dead code and could potentially be removed.

For commits which show an overall increase of interactions per line, we again find two basic types.

Commits that increase in interactions while their lines are mostly unchanged, this is the expected behavior when a project grows. We see this behavior in most commits for periods with no changes to their lines, but in some cases like HTOP@93f091c47e we observe an extreme increase in interactions. These commits are most likely in parts of the project that experience few changes, but are central, or become central to the growing project. When projects grow and new functionality is added, this increases the interactions with central code as the code that was central before is most likely also used by the new functionality.

The other type we observe, are commits that shrink in LOC but stay constant in interactions or even grow, e. g., FILE@ace9da574a shown in Figure 4.4c. For these commits the central LOC seem to remain while the rest is reworked. Instances of this can be found as commits, which are reworked immediately after their introduction, here we observe that often the control flow is changed but the central behavior is untouched. For example, in Figure 4.7 we see a change by Ilya Leoshkevich, implementing new compatibility with a hardware

accelerator. This change is immediately reformated and changed by Paul Eggert to use the style of the project. While the rework changes many of the lines it does not seem to affect the central lines, as the majority of interactions remain. These reworks are often made by the main author, as they posses deeper knowledge of the project.

Our qualitative analysis shows that commits which introduce a bug fix or refactor a part of the project make up most of the commits with large changes in their interactions per LOC. Overall, this result is close to the distribution we expect based on our random sample. With the only difference, that we find fewer maintenance commits than in the random sample, which is expected, and more commits which refactor parts of the code. For commits which increase in interactions per LOC we also find a few more commits that improve performance, while this is not statistically significant because of our low sample size, it could be that after a functionality is improved it becomes more widely used in the project resulting in an increase of interactions. We further observe only small differences between commits which increase in s and those which decrease in s. This further indicates that these simple categories do not relate to the divergent evolution of interactions and lines of a commit.

In conclusion we find, the interactions and lines of commits to evolve in a similar manner for most commits, with only a slight increase in interactions per lines across all projects. However, we find some commits for which the evolution of interactions and lines diverges drastically. In these divergent evolutions we often observe changes to the interactions of a commit which are not related to a change in its lines. Our qualitative analysis again shows only small differences to the random sample, indicating that the type of commit we categorize has no influence on the evolution of the commit.

4.2.3 *Author Interactions*

We observe that changes to the contributions of an author in interactions and lines generally evolve somewhat similar to each other but their compositions can differ greatly. The main authors often have a higher share of interactions than lines. We also observe many of the peripheral authors to contribute a sizable share in lines but almost no interactions, this indicates that the contribution on interactions of an author may be a useful measure to differentiate between core an peripheral authors. However, we also find some authors with comparatively small contributions to the lines of a project and sizable contributions to its interactions. These cases, like Jim Meyering in GREP and Atis Adamantis in LIBSSH, are most likely authors which contribute to the core of a project in small ways. In the case of Jim Meyering this is most likely because they are also involved in other GNU projects which share core parts so they have a good understanding of the architecture of the project. Atis Adamantis shows the behavior we could expect from a core developer, who mostly works on the core of the project, which may be a small part in lines of the project but interacts with virtually every aspect of the project.

In the change of main authorship in HTOP from Hisham Muhammad to Christian Göttsche we observe that Christian Göttsche quickly contributes to over 40% of interactions while only contributing around 20% of lines, this indicates a change in development structure, as the contribution in lines becomes more diverse while the contribution of interactions remains mostly from Hisham Muhammad and Christian Göttsche. We observe a similar change in GREP where Paul Eggert becomes the author with the highest contribution in lines

and interactions. But this change is more pronounced in the contributions to of lines. This could be caused by a slower change, as the contribution of Paul Eggert grow over multiple years while the change in HTOP takes only around half a year.

We find that the contributions of authors to the lines and interactions of a project generally evolve similarly. However, we observe instances of drastic change one of which we can relate to a change in ownership of a project. The main author of each project often contributes more to the interactions of a project than they contribute to the lines. The other authors generally have a very small contribution to the interactions of a project. However, we observe two authors which show a relatively large contribution to the interactions of a project while contributing only few lines.

4.3 THREATS TO VALIDITY

INTERNAL VALIDITY Since our analysis requires the compilation of the project, we are limited to analyzing buildable revision of projects. This means, that older revisions, especially before the first release of a project, can often not be analyzed. Similarly revisions containing major bugs that break the build system can also not be analyzed.

Because we initially sample only 50 to 100 revisions per project, we miss large parts of the history of each project. Large changes we detect between sampled revisions could be caused by multiple smaller changes. To combat this, we sample additional revisions in between data points with large changes. Changes which happen and are mitigated between sampled revisions are difficult to detect this way. To combat this, we always consider changes to the metrics on a commit level as these are more volatile than the overall project.

EXTERNAL VALIDITY The expensiveness, especially in regards to memory consumption of the analysis, limits our scope to smaller software projects. This may introduce anomalies, especially with project consisting of a single main author with few others contributing in small ways. Larger projects with many authors often have a more complex structure and a higher need for collaboration. This influences the architecture of the project as well as the development structure. It could be that there are stricter rules on the form of commits which could influence how these changes impact the system.

The limit of our project selection to only *C/C++* based projects limits the generalizability of our observations, as other languages often employ different programming paradigms. The paradigm of a language affects the architecture of projects and therefore the data-flow structure. This could change our observations as a different data-flow structure could cause commit interactions to change in different ways.

These issues are important for future studies, since we aim to only explore the possibilities of the approach. While our insights may not be generally applicable, we demonstrate that data-flow-based commit interactions present a useful metric to study the evolution and development of software projects.

5 RELATED WORK

In this chapter we provide an overview of related work from related fields and previous works.

COMMIT INTERACTIONS Sattler et al. [21], proposed the concept of commit interaction as a combination of high-level repository information with low-level semantic information. By adding commit information from the VCS to nodes of a data-flow graph the data-flow connections between commits can be extracted. They show that the resulting graph of data-flow based commit interactions can provide a useful metric for code and author centrality. They have used this to analyze individual revisions of projects, we build on that by exploring the capabilities of the approach for the analysis of revision histories.

The evolution of software projects has been a subject of research in many fields studying different aspects of software projects.

ARCHITECTURAL EVOLUTION Architectural decay is a generally recognized phenomenon in the study of software architecture. Despite its prevalence there is relatively little empirical data on the nature of architectural changes, which may lead to decay [14]. Behnamghader et al. [1] investigated the evolution of software architecture using multiple approaches. They found, semantic based architecture views to produce more accurate results. Commit interactions may provide a useful metric to analyze the architectural changes of a project as they provide semantic interactions with high-level repository information. For example, we expect a revision which impacts the interactions of many commits to be an architectural change. And it could be that architectural decay coincides with and overall increase in interactions between commits which did not interact before. Understanding how and when the data-flow interactions in a project change could therefore provide new insights in the architectural evolution of software projects.

SOFTWARE REPOSITORY MINING. Mining Software repositories(MSR) consists of gathering, modeling, and exploiting the data produced by developers and other stakeholders in the software development process as they create a software system [5]. Kagdi, Collard, and Maletic [13], investigated a wide range of approaches to software repository mining with the focus on software evolution. They found, that the majority of current MSR approaches operate either the physical level (e. g., system,subsystems, directories, files, lines) or at a fairly high level of logical/syntactic entities (e. g., classes). Using data-flow interactions as an additional source of data could provide additional information, since they are tied directly to the semantics of code. Zimmermann et al. [24] investigated coupling between entities in a software project, by analyze which parts of a project are changed together, using textual changes. They find their approach to produce helpful information to guide developers on changes which may be necessary given an intended change. One aspect they highlight as a potential improvement is to incorporate program analysis as an additional

data source, in order to find semantic coupling between changes. Commit interactions could provide such data, as they represent semantic coupling between code regions. The evolution of commit interactions could also provide an insight into indirect coupling, by analyzing which commits are generally affect together.

SOFTWARE EVOLUTION The field of software evolution studies evolutionary patterns in software projects. They often use high-level metrics [13], such as the file, module and line count [7] to analyze the history of projects. In contrast to these high-level approaches, we analyze how the data-flow structure in a software project evolves. This gives deeper insights into the effect and role of changes in the overall project, as we can detect changes to the semantic connections between regions.

SOCIO-TECHNICAL RESEARCH. Socio-technical studies aim to understand the relations between code authors and the organizational structure of the development process [12]. One aspect of relations between developers is artefact coupling, as it creates dependencies between developers [12]. Joblin, Apel, and Mauerer [12] investigate evolutionary trends in the organizational structure of software projects. They extend a classical approach, for the study of developer relations, which assumes a relation when two developers edit the same code region, with semantic-artefact coupling information. From these extended relations they study evolutionary changes in an developer network. They define a classification for the roles of developers based on their node degree in this network. Using commit interactions to extrapolate interactions between authors also provides a view on semantic coupling relations between developers. We investigate how the data-flow interactions between code of different authors evolve. For this we mostly focus on the amount of interactions a developer is involved in, this is very similar to the node degree described by Joblin, Apel, and Mauerer [12].

CHANGE IMPACT Understanding the impact a change has on a software system is important to judge the work required to implement the change [4]. The field of *impact analysis* tires to: "identify the potential consequences of a change, or estimate what needs to be modified to accomplish a change [4]". Many techniques to provide such estimates have been developed. These approaches use a variety of metrics from high-level models of the architecture of a system to low-level call graph information [18]. Studying the impact of past changes may help us improve our understanding and predictions of the impact of future changes. We use changes in commit interactions to build an metric for the impact of revisions, in order to hopefully provide additional information on the impact of changes.

6 CONCLUDING REMARKS

6.1 CONCLUSION

We analyzed 1610 revision across 12 open source projects from different domains, in order to explore how commit interactions change over time. From this, we want to investigate what insights a impact metric based on commit interactions can provide, how the commit interactions of individual commits evolve and what commit interactions provide when viewed on an author level.

We find that there is very little correlation between the size of a revision and its impact on the data-flow structure of the software project. While most revisions have small impact on the lines of a project, they have either little to no impact on the interactions of the project or affect large parts. This indicates that there are central and peripheral parts of a project.

When investigating changes to individual commits we find that generally the amount interactions increases when code remains unchanged. For most commits this effect is minor. However, there cases in which the interactions drastically increases over the lifetime of the commit. In contrast, we also find commits which nearly stop interacting with the rest of the code base despite having lines remaining. These changes often happen independent of changes to the lines of the commit. This indicates that there are regions which experiences drastic changes in usage.

By grouping commit interactions by the authors of the commits we can analyze the interactions of authors. This shows that while the evolution of the contribution to lines and interactions of authors follows similar trends, there are differences which could relate to the roles of authors. Especially drastic changes in author roles like a change of the main author of a project show that commit interactions may be a useful metric to analyze changes in author roles. Furthermore, we find authors which contribute to the interactions pf a project in a sizable fashion, with only a small contribution in lines of code. These authors do not fully fit the categorization of core and peripheral authors as they do not contribute in large amounts, but in central locations which would indicate a somewhat deep knowledge of the project.

Overall we find that analyzing software evolution through data flow based commit interactions provides new information and patterns which are not found in traditional high-level metrics.

6.2 FUTURE WORK

We show the potential of data-flow-based commit interactions as a tool to analyze software evolution. While we find patterns in the evolution of commit interactions, our qualitative analysis shows that the causes of these patterns are not obvious, and are most likely rooted in more complex structures like the architecture of a project or socio-technical relations between authors. The connections between these factors and the evolution of commit

interactions requires further research. Additionally, an analysis of a wider variety of projects especially larger ones and projects based in languages other than *C* and *C++* may reveal very different patterns. Because the development structure of larger project may greatly affect how commit interactions evolve. Similarly, different programming paradigms of other languages would most likely also affect our findings, as the programming paradigm of a language and project often shape its architecture.

While commit interactions present a useful metric when analyzing the evolution of software projects, it is difficult to measure continuous changes to a single part with them. Because, when a change is made to the code of a commit the code is no longer associated with that commit. Including a region metric which is more static over time like function scopes could help with this.

APPENDIX

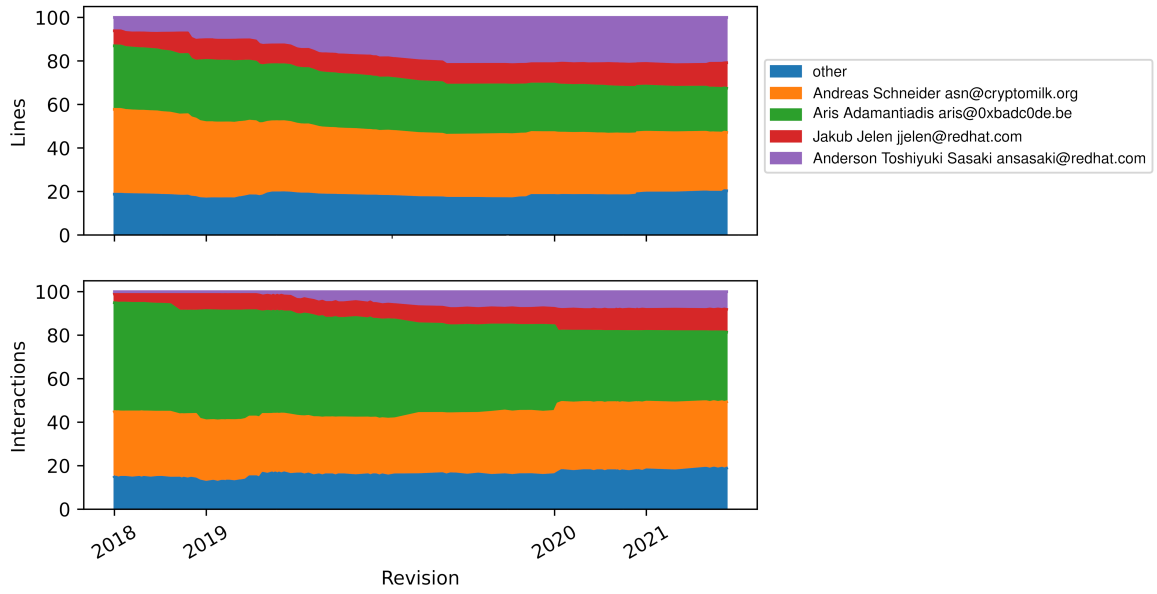


Figure A.1: Area plots comparing the contribution of authors to the lines and interactions of LIBSSH. We see that of the 4 authors with the most contribution Anderson T.S. has a comparatively small contribution to the interactions of the project. In contrast Aris A. contributes around 20% of the lines but roughly 40% of the interactions of LIBSSH. The contributions of the other authors are similar between lines and interactions.

BIBLIOGRAPHY

[1] Pooyan Behnamghader, Duc Minh Le, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. "A large-scale Study of Architectural Evolution in Open-Source Software Systems." In: *Empirical Software Engineering* (2017), pp. 1146–1193.

[2] Thomas Bock, Nils Alznauer, Mitchell Joblin, and Sven Apel. "Automatic Core-Developer Identification on GitHub: A Validation Study." In: *ACM Trans. Softw. Eng. Methodol.* (2023).

[3] S. Bohner. "Impact analysis in the software change process: a year 2000 perspective." In: *1996 Proc. of Int. Conf. on Software Maintenance (ICSM)*. IEEE, 1996, pp. 42–51.

[4] S. Bohner and R.S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[5] M D'Ambros and Romain Pierre Julien Robbes. "Effective Mining of Software Repositories." In: IEEE Computer Society, 2011, pp. 598 –598.

[6] N. Deepa, Prabadevi B, Krithika Lb, and B.Deepa. "An analysis on Version Control Systems." In: 2020, pp. 1–9.

[7] Israel Herraiz, Gregorio Robles, Jesús M. Gonzalez-Barahona, Andrea Capiluppi, and Juan F. Ramil. "Comparison between SLOCs and Number of Files as Size Metrics for Software Evolution Analysis." In: *Conf. Software Maintenance and Reengineering (CSMR)*. IEEE, 2006, 8 pp.–213.

[8] Abram Hindle, Daniel M. German, and Ric Holt. "What Do Large Commits Tell Us? A Taxonomical Study of Large Commits." In: *Proc. Int. Working Conf. on Mining Software Repositories (MSR)*. Association for Computing Machinery, 2008, 99–108.

[9] Claus Hunsen, Janet Siegmund, and Sven Apel. "On the Fulfillment of Coordination Requirements in Open-Source Software Projects: An Exploratory Study." In: *Empirical Softw. Engg.* (2020), 4379–4426.

[10] Emanuele Iannone and Fabio Palomba. "The Phantom Menace: Unmasking Security Issues in Evolving Software." In: *2022 IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*. 2022, pp. 612–616.

[11] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. "Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics." In: *Proc. of the 39th Int. Conf. on Software Engineering (ICSE)*. IEEE Press, 2017, 164–174.

[12] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. "Evolutionary Trends of Developer Coordination: A Network Approach." In: *Empirical Software Engineering (EMSE)* (2015).

[13] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution." In: *J. Softw. Maint. and Evol.: Research and Practice* 19.2 (2007), pp. 77–131.

[14] Duc Minh Le, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. "An Empirical Study of Architectural Change in Open-Source Software Systems." In: IEEE, 2015, pp. 235–245.

[15] M.M. Lehman. "Programs, life cycles, and laws of software evolution." In: *Proceedings of the IEEE* 68 (1980), pp. 1060–1076.

[16] M.M. Lehman. "Laws of software evolution revisited." In: *Software Process Technology*. Springer Berlin Heidelberg, 1996, pp. 108–124.

[17] M.M. Lehman and L.A. Belady. *Program evolution: processes of software change.* Academic Press Professional Inc., 1985.

[18] Steffen Lehnert. "A review of software change impact analysis." In: 2011.

[19] Audris Mockus, Roy Fielding, and James Herbsleb. "Two Case Studies of Open Source Software Development: Apache and Mozilla." In: *ACM Transactions on Software Engineering and Methodology* (2002), pp. 309–346.

[20] Christoffer Rosen, Ben Grawi, and Emad Shihab. "Commit Guru: Analytics and Risk Prediction of Software Commits." In: *Proc. Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. Bergamo, Italy: Association for Computing Machinery, 2015, 966–969.

[21] Florian Sattler, Sebastian Böhm, Philipp Schubert, Norbert Siegmund, and Sven Apel. "SEAL: Integrating Program Analysis and Repository Mining." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* To appear (2023).

[22] Sagiv Shmuel, Reps Thomas, and Horwitz Susan. "Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation." In: *TAPSOFT '95: Theory and Practice of Software Development*. Springer Berlin Heidelberg, 1995, pp. 651–665.

[23] Guowu Xie, Jianbo Chen, and Iulian Neamtiu. "Towards a better understanding of software evolution: An empirical study on open source software." In: *Int. Con. on Software Maintenance (ICSM)*. IEEE, 2009, pp. 51–60.

[24] Thomas Zimmermann, Peter Weibgerber, Stephan Diehl, and Andreas Zeller. "Mining Version Histories to Guide Software Changes." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2004, pp. 563–572.