University of Passau

Department of Informatics and Mathematics

Master's Thesis

Synchronous Development in Open-Source Systems: File-based, Feature-based, and Function-based

Author:

Thomas Bock

September 28, 2016

Advisors:

Prof. Dr.-Ing. Sven Apel
Chair of Software Engineering

Claus Hunsen
Chair of Software Engineering

Mitchell Joblin
Chair of Software Engineering

Bock, Thomas:
Synchronous Development in Open-Source Systems: File-based, Feature-based, and Function-based
Master's Thesis, University of Passau, 2016.

Abstract

In open-source software (OSS) projects, often many developers from different locations contribute to the source code. To coordinate their collaboration, developers frequently use email communication. In summary, this study aims to get more insight into the coordination of collaboration in OSS projects. In this thesis, we investigate whether synchronous development on the source code of the OSS project QEMU is reflected in the corresponding communication on the mailing list. Therefore, we use the concept of co-commit bursts and email bursts to analyze the synchronous development of pairs of developers. To this end, we replicate an empirical study of Xuan and Filkov [XF14]. As an enhancement, we consider three different levels of abstraction from the source code, namely files, features, and functions, on which we detect synchronous development within a certain time window. Usually, source code is organized in files, whereas features and functions are semantic units of program parts. In our study, we found that synchronous development based on features appears more frequently than synchronous development on files or functions. In addition, we quantify the synchronicity of commits that form a co-commit burst. For each of the abstraction levels, we examine whether there is a timely correlation between commit activities and email communication. We also analyze whether there is a significant difference in code growth and implementation effort within periods of synchronous development and outside of synchronous development. Finally, we compare the differences of file-based, feature-based, and function-based collaboration of developers.

Contents

List of Figures ix

List of Tables xii

1 Introduction 1

2 Background 3
- 2.1 Synchronous Development . . . 3
- 2.2 Abstraction Levels of Source-Code Artifacts . . . 4
  - 2.2.1 Files . . . 5
  - 2.2.2 Functions . . . 5
  - 2.2.3 Features . . . 6

3 Research Questions and Hypotheses 7
- 3.1 Replication of Burst Study of Xuan and Filkov . . . 7
- 3.2 Hypotheses . . . 8
  - 3.2.1 C-Burst-related Hypotheses . . . 8
  - 3.2.2 E-Burst-related Hypotheses . . . 9
  - 3.2.3 Comparative Hypotheses . . . 10

4 Methodology and Implementation 11
- 4.1 Subject System: QEMU . . . 11
- 4.2 Extraction of Commit and Email Data using Codeface . . . 12
- 4.3 Identification of Bursts . . . 13
- 4.4 Synchronicity Degree . . . 14
- 4.5 Null Models . . . 17
- 4.6 Code-Growth and Code-Effort Metrics . . . 18
- 4.7 Correlation between C-Bursts and E-Bursts . . . 19
- 4.8 Limitations and Differences to the Original Study . . . 20

5 Results and Discussion 23
- 5.1 Results . . . 23
  - 5.1.1 General Results . . . 23
  - 5.1.2 C-Burst-related Hypotheses . . . 26
  - 5.1.3 Correlation of C-Bursts and E-Bursts . . . 36
  - 5.1.4 Comparison of File-based, Feature-based, and Function-based Synchronous Development . . . 39

5.2 Discussion . . . 43
5.2.1 C-Burst-related Hypotheses . . . 43
5.2.2 E-Burst-related Hypotheses . . . 44
5.2.3 Comparative Hypotheses . . . 45
5.3 Threats to Validity . . . 46

6 Conclusion **49**
6.1 Summary . . . 49
6.2 Future Work . . . 50

A Appendix **53**

Bibliography **57**

List of Figures

4.1 Randomization of commit time series. 17

5.1 Box-and-whisker diagrams of the synchronicity degree of real C-bursts and simulated C-bursts. 29

5.2 Box-and-whisker diagrams of the number of added lines per commit within synchronous and non-synchronous commits. 30

5.3 Box-and-whisker diagrams of the code growth per commit within synchronous and non-synchronous commits. 32

5.4 Box-and-whisker diagrams of the code effort per commit within synchronous and non-synchronous commits. 33

5.5 Box-and-whisker diagrams of the number of deleted lines per commit within synchronous and non-synchronous commits. 35

5.6 Relationship between the number of C-bursts and the number of E-bursts per developer pair for time window ξ=5. 37

5.7 Box-and-whisker diagrams comparing the synchronicity degrees of the file-based, feature-based, and function-based analyses. 40

A.1 Relationship between the number of C-bursts and the number of E-bursts per developer pair for time window ξ=1. 54

A.2 Relationship between the number of C-bursts and the number of E-bursts per developer pair for time window ξ=5. 55

A.3 Relationship between the number of C-bursts and the number of E-bursts per developer pair for time window ξ=10. 56

List of Tables

4.1 Commit meta-data which we extract from the GIT repository. 12

4.2 Email meta-data which we extract from the mailing list archive. . . . 12

4.3 Example for a C-burst having low synchronicity degree. 16

4.4 Example for a C-burst where both developers change the same amount of lines on the synchronously changed artifact. 16

4.5 Example for a C-burst having very low synchronicity degree. 16

5.1 General data on the identified C-bursts of QEMU for a time window of 1 day. 24

5.2 General data on the identified C-bursts of QEMU for a time window of 5 days. 24

5.3 General data on the identified C-bursts of QEMU for a time window of 10 days. 25

5.4 General data on the identified E-bursts of QEMU for the time windows of 1 day, 5 days and 10 days. 26

5.5 Paired t-tests of the differences in the number of bursts per developer pair between real and simulated C-bursts. 27

5.6 T-tests of the differences in the synchronicity degrees between real and simulated C-bursts. 28

5.7 Numbers of synchronous and non-synchronous commits for the different time windows and analysis types. 28

5.8 T-tests of the differences of the number of added lines in synchronous development and in non-synchronous development. 31

5.9 T-tests of the differences of code growth in synchronous development and in non-synchronous development. 31

5.10 T-tests of the differences of code effort in synchronous development and in non-synchronous development. 34

5.11 T-tests of the differences of the number of deleted lines in synchronous development and in non-synchronous development, estimating lower numbers of deleted lines in synchronous development. 34

5.12 T-tests of the differences of the number of deleted lines in synchronous development and in non-synchronous development, estimating higher numbers of deleted lines in synchronous development. 36

5.13 Correlation tests for the numbers of C-bursts and the numbers of E-bursts per pair of developers. 38

5.14 Mann-Whitney U tests for the real and simulated correlation between C-curves and E-curves per developer pair. 38

5.15 T-tests with FDR correction for the comparison of the numbers of C-bursts per developer pair of file-based, feature-based, and function-based analyses. 39

5.16 T-tests with FDR correction for the comparison of the synchronicity degrees of file-based, feature-based, and function-based analyses. . . . 40

5.17 T-tests with FDR correction for the comparison of the number of added lines in synchronous commits of file-based, feature-based, and function-based analyses. 41

5.18 T-tests with FDR correction for the comparison of the number of deleted lines in synchronous commits of file-based, feature-based, and function-based analyses. 41

5.19 Correlation coefficients for the real C-curves per developer pair and the real E-curves per developer pair. 42

1. Introduction

Software projects are often carried out by several developers which implement the source code of the software. To produce software of high quality, the source code should be simple and easy to maintain. However, if many developers contribute to the same parts of a software project, it is not that easy to implement good code since every developer submits their own changes. Additionally, changes of various developers can introduce conflicts or mismatched enhancements. Hence, some developers collaborate and arrange with each other, e.g., to prevent conflicting changes, to reduce the amount of bugs, or to keep the code simple and maintainable. Collaboration of developers and communication among them to coordinate collaboration is widely-used [CHC08, MFT02], particularly in software companies.

Especially in open-source software (OSS) projects, often a huge number of independent different developers, who may be distributed all over the world, contributes to the project [MFT02]. Since they have different workplaces, they often communicate via the Internet to discuss software issues or enhancements or to review code changes [WGS03].

In this work, we investigate if collaboration of software developers on source-code level is correlated to their email communication. For that purpose, we replicate an empirical study of Xuan and Filkov [XF14] on synchronous development in open-source software. In addition, we analyze a larger case study than the original authors have used, namely the virtual-machine monitor QEMU, and enhance their work by focussing on three different levels of abstraction in the source code: files, features, and functions. Moreover, we introduce a new metric to quantify the synchronicity of collaborative development.

In accordance with Xuan and Filkov, we extract time intervals in which two developers commit to the same source-code artifact synchronously within a certain time window. These time windows are called *co-commit bursts*. In addition, we also gather time intervals in which two developers write emails within a certain time window, which are called *email bursts*. We also check if there is a significant difference between the code growth within co-commit bursts and outside of co-commit bursts.

As we look at three different levels of abstraction, we also compare the outcomes of the three analyses to each other and check whether synchronous development appears more frequently respectively the effects of synchronous collaboration occur more strongly on one of these artifact-based analyses compared to the others. This will give us more insight into which level of abstraction is the most appropriate developers organize their collaboration on.

This thesis is organized as follows: In Chapter 2, we introduce the background of developer collaboration and explain the levels of abstraction, namely files, features, and functions. In Chapter 3, we present the study which we want to replicate and the research questions and hypotheses which we have derived. Chapter 4 contains a detailed description of our methodology, metrics, and limitations. We state the results of our study in Chapter 5, discuss them afterwards, and deal with our threats to validity. In Chapter 6, we conclude and list future work.

2. Background

In open-source software (OSS) projects, developer collaboration from programmers spread all over the world is common. We shortly explain some widely-used kinds of developer collaboration and explain the term *synchronous development*, which is coined by Xuan and Filkov [XF14]. Thereafter, we introduce three different levels of abstraction from the source code which we consider to identify synchronous development.

2.1 Synchronous Development

Collaboration on source-code level consists of bug fixes, code refactoring, enhancement of a software and adding new features, or even deleting unused or wrongly used functionality, for instance [Sin10]. If a pair of developers contributes to the same code artifact, i.e., to the same part of the source code, within a short period of time, their collaboration is called *synchronous development*. We consider two different levels of synchronous development, namely co-commit bursts and email bursts.

Often version-control systems (VCS), such as GIT, are used to manage the code base of a certain software online. In VCS, various developers can check out the current source code from a source-code repository, modify some parts of the software, and submit their code changes to the repository again. [Som10, DP03] In addition, VCS often provide tools to maintain and coordinate the submitted changes. The VCS usually stores all code changes, combined with their authors and modification time stamps. So, modifications to the source code are called commits. A commit is composed of a couple of code changes to parts of the source code made by one single developer at a certain point in time. If two developers commit to the same code artifacts within a short space of time, we call that a co-commit burst (short, *C-burst*) [XF14]. We analyze how co-commit bursts affect the code growth and implementation effort. There are several levels of abstraction from the source code to treat certain parts of the code as artifacts. The three artifact types which we use in this study are explained in Section 2.2.

Besides code-based collaboration, developer collaboration in many cases also includes communication to coordinate source-code changes [BGD+06, WSDN09]. In

our study, we focus on email communication of pairs of developers on a publicly available mailing list which belongs to the software project. With this, an email burst (short, *E-burst*) appears if two persons each write an email to the mailing list within a brief time period [XF14].

The maximal time difference between two commits respectively two emails to be considered as a burst is called *time window*. For example, if we want to determine bursts at a 1-day time window, the difference in time of commits of two different developers can maximally be one day to treat their synchronous development as a C-burst. The same holds analogously for E-bursts and the time difference between two emails of two different authors [XF14].

Further below, we investigate whether co-commit bursts and email bursts in OSS correlate. Therefore, we call a synchronous occurrence of co-commit bursts and email bursts *coordination bursts*.

Besides the study of Xuan and Filkov which we reproduce, there exists some other research on the investigation on the relationship between development and communication between developers. We just name a sample of the work which is done in this area of software engineering research. Herbsleb and Grinter [HG99] conducted a study on the coordination of a geographically distributed software project. Their result was that ad-hoc communication between developers is one of the most important parts of today's well-working software development. Toral et al. [TMTB10] analyzed the social communities which arose from collaborative development in OSS projects. Also Joblin et al. [JMA+15] constructed collaboration networks of developers based on co-commits on source-code artifacts and used network analysis techniques to gain more information on the collaboration. In contrast to our work, they did not analyze email-based collaboration and also neglect the time-wise synchronicity of co-commits. Bacchelli et al. [BDL10] analyzed the email communication of OSS projects and determined much discussed source-code artifacts. Afterwards, they investigated the defect-proneness of those artifacts. In contrast to all mentioned related work, the work of Xuan and Filkov and our replication study directly compare synchronous development on source-code level to synchronous email communication.

2.2 Abstraction Levels of Source-Code Artifacts

To identify periods of synchronous development of a pair of developers, we consider code changes on source-code artifacts. Though, there are several options how to decompose the code of a software project into artifacts. Different types of artifacts represent various levels of abstraction from the source code: files, functions, and features. Each of these abstraction levels depicts another way of seeing software and therefore we expect them to have different effects on developer collaboration. Moreover, the different levels of abstraction have different levels of granularity of decomposing source-code into artifacts.

All three mentioned levels of abstraction (files, functions, and features) are considered in our study. We perform our research work on all of them in separate and compare the different results afterwards. In Chapter 3, we state our concrete research questions and hypotheses. Below, we describe the individual characteristics of each of the three abstraction levels in detail.

2.2.1 Files

Generally, a software project consists of multiple files. Some of the files just contain documentation or images, but most of the files contain source code. In our study, we just focus on files containing source code. Often the source code itself is spread among several files. Usually each file contains code written in a certain programming language.

Depending on the programming language, a distinction is made between several types of source-code files. For instance, projects which use the programming language C differentiate between implementation files (`*.c`) and so-called header files (`*.h`), which just comprise a couple of definitions and declarations which can be used and included by several other code files. Usually, header files contain common definitions and function declarations, which are included within several other files [KR88]. We omit header files because they usually do not contain a lot of source code and programming elements which are specific to the software functionality itself. Commonly, the functionality of software is contained in implementation files. So, header files are not necessary when analyzing the collaboration of developers on source-code level.

Since the decomposition of source code into files is obvious due to the separated implementation itself, we treat files as different artifacts within our file-based analysis.

Collaboration of developers on the same file can be semantically random or logically intended: Depending on the length and semantic content of an implementation file, it is possible that one file contains only semantic related parts of the source code. However, a source-code file can also contain the implementation of various distinct units of the software which are not related to each other. So, it might be that two developers change one file and modify completely independent software parts, as well as it is possible to change parts that belong to the same semantic unit of the program. In short, synchronous development on the file level of abstraction is easy to detect because of separated implementation, but does not tell us if there actually is a semantically intended collaboration of developers.

2.2.2 Functions

A more fine-grained level of abstraction from the source code are functions, which are small parts of files and consist of a couple of statements to perform a certain task within the software. In other words, functions are subroutins which encapsulate computations that can be used in various parts of the program regardless of the concrete implementation [KR88]. The code of a function is completely contained within one file. Therefore, a function is the most fine-grained viewpoint to group statements into semantic units. In our study, we use functions as an artifact type since functions are one of the smallest detectable code parts which can be seen as logic units. Usually, almost every line of a file belongs to one function.

However, there are some macro definitions, variable declarations, or `#include` statements, for example, which are not part of a function. We group all the statements not belonging to a function as code at file level. So, to not distort the way of seeing functions, we disregard code changes made at the file level when we analyze on the function level.

From the perspective of synchronous development, functions are small semantic units developers can collaborate on. If two developers change the same function, they mostly adapt the same logic part of the software since a function encapsulates computation parts. Ordinarily, functions are comparably small in terms of lines of code and several functions are contained within one file. So, if two developer change the same function, they semantically collaborate on that function.

2.2.3 Features

Furthermore, we also look at features as a more coarse-grained viewpoint of abstraction. Features are semantic units of variable software, such as configurable parts in Software Product Lines. Software Product Lines (SPLs) are known as highly configurable software systems. The main component of a SPL is a set of features, which are also established as configuration options to provide variability. Each of these features is a characteristic of the system which implements a certain user requirement. [ABKS13, CN02, GBS01]. Hence, each feature forms a logical unit and is reflected somewhere in the source code. This relates to the implementation technique which is used to develop an SPL. For instance, one possible method is conditional compilation which uses preprocessor directives to mark variable feature code. In the programming language C, for instance, features can be implemented by using the preprocessor directive `#ifdef` [ABKS13, KR88]. The code belonging to a certain feature can be scattered across several files in the source code and is most commonly represented by a set of functions.

Code not belonging to a certain feature is called *base code* since it implements and supports the core functionality of the software which is part of every configuration. To keep the feature level of abstraction, we group all elements of the base code to a virtual feature called *base feature*. As the base feature is not really a feature in terms of a semantic unit, we completely disregard the base feature in our C-burst analysis. In other words, we remove the changes made to the base feature by arbitrary developers from all commits before identifying C-bursts in the feature-based analysis.

If two developers collaborate on the same features, they actually work on the same semantic units, which usually are more coarse-grained units than functions are. Therefore, feature-based collaboration is not only a collaboration of developers on the same source-code software artifacts, but also a substantial semantic collaboration.

3. Research Questions and Hypotheses

Our study is based on a study of Qi Xuan and Vladimir Filkov [XF14], which we replicate under different conditions and using slightly different metrics. In addition, we extend their research by new ideas and considering three different levels of abstraction from the source code. In what follows, the study goals and research hypotheses are extensively expounded.

3.1 Replication of Burst Study of Xuan and Filkov

In their work, Xuan and Filkov [XF14] analyzed whether file-based synchronous development in open-source software (OSS) projects exists and how synchronous development is related to coordination via emails. Additionally, they investigate how code changes in terms of lines of code (LOC) differ between synchronous and non-synchronous development. They acquired commit data from GIT[1] repositories of 31 different OSS projects from the Apache Software Foundation, which covered a time range between the years 2000 and 2012. In addition, they downloaded the emails from the corresponding developer mailing lists. From the case studies they analyzed, each case study has at most 72 developers within the analyzed time range, up to 17,808 emails per case study, and, at most, 14,697 commits per case study.

In contrast, we analyze the synchronous development of a considerably larger OSS case study, namely QEMU, which had 918 developers in the time interval from the beginning of April in 2009 to the mid of June in 2016. These developers were authors of 29,312 commits and wrote 380,958 emails on the corresponding mailing list. However, due to computational and resource limitations, we just analyze the 346,714 emails which were written by persons which also authored commits to the source-code repository.

For the burst analyses – both co-commit and email analyses – Xuan and Filkov used different time windows ξ with $1 \leq \xi \leq 10$ days. In our replication study, we stick to

[1] https://git-scm.com/

their range of time windows and, therefore, we perform our study with time windows ξ of 1 day, 5 days, and 10 days.

Additionally, we transferred their approach to different levels of abstraction, i.e., we performed their study on a file-based abstraction of synchronous development, on function-based synchronous development, and on feature-based synchronous development.

3.2 Hypotheses

Mainly, we follow the hypotheses of Xuan and Filkov [XF14] and investigate whether they also hold for our study and for different levels of abstraction. Hereinafter, we present their hypotheses, which we take over as our hypotheses in a slightly adapted way. Notice that we check each of these hypotheses separately for each of our three different artifact types, namely synchronous development on files, on features, and on functions.

We aim to apply the same study setup and metrics Xuan and Filkov have used for their empirical study, to find out whether the hypotheses of Xuan and Filkov and our hypotheses can be accepted. Unfortunately, we do not have the same raw data as they had and, therefore, we have to slightly deviate from their approaches. The methodology we use in this study is precisely described in Chapter 4, our limitations to this study are reported in Section 4.8.

3.2.1 C-Burst-related Hypotheses

Initially, we check whether synchronous development is a random phenomenon or not. Otherwise, it would be meaningless to discuss further hypotheses on synchronous development. Therefore, we validate whether the number of bursts per developer pair is significantly not random, which leads us to our first hypothesis:

Hypothesis 1. *Synchronous development occurs significantly more frequently in OSS projects than by chance.*

As an enhancement to the study of Xuan and Filkov, we quantify the parts of synchronous commits of two developers which were changed by both developers. Remember that we (as well as Xuan and Filkov) treat commits of two different developers within a certain time interval as synchronous if there is, at least, one artifact which was changed by both commits. However, such synchronous commits also can contain changes on lots of different artifacts which only one developer committed to. In Chapter 4, we present a metric which quantifies the amount of actual synchronous development for each co-commit burst. We call that metric *synchronicity degree*, which is a numeric approach to determine synchronous development, compared to the binary approach of just identifying co-commit bursts. So, we also check whether the calculated synchronicity degree is random or not, and, thus, formulate our next hypothesis:

Hypothesis 1a. *The synchronicity degree in OSS projects is significantly higher than by chance.*

Further, Xuan and Filkov investigate how code growth and the effort of programming behave within synchronous development. Therefore, they use the number of lines of code (LOC) which are changed within synchronous commits. Thereto, they determine the number of added lines Δ_{add} and the number of deleted lines Δ_{delete} per commit and designed the following metrics to quantify the code growth and effort per commit:

$$\text{Code growth:} \quad \Delta L = \Delta_{add} - \Delta_{delete} \tag{3.1}$$

$$\text{Code effort:} \quad \Delta W = \Delta_{add} + \Delta_{delete} \tag{3.2}$$

Using these metrics, they expect that the code growth is higher in synchronous development than in non-synchronous development, but the code effort is lower in synchronous development than in non-synchronous one.

Hypothesis 2. *OSS projects grow more in change of LOC per commit if developers participate in synchronous development. In addition, in synchronous development the effort in deleted plus added lines per commit is lower than in non-synchronous development.*

Since synchronous development is assumed to be more effective, it is hypothesized that due to collaboration between two developers not as many lines need to be deleted than in non-synchronous development.

Hypothesis $3_<$. *The number of deleted lines per commit is smaller within co-commit bursts than outside co-commit bursts.*

In the case that this hypothesis of Xuan and Filkov will not be accepted, we will check whether the number of deleted lines is even higher in synchronous development:

Hypothesis $3_>$. *The number of deleted lines per commit is larger within co-commit bursts than outside co-commit bursts.*

3.2.2 E-Burst-related Hypotheses

To coordinate their collaboration, developers need to communicate with each other. Often they use emails to align their development. Xuan et al. already showed that communication and commit activities are related to each other [XGDF12]. Hence, we expect that synchronous development comes along with synchronous email communication.

Hypothesis 4. *More co-commit bursts are accompanied with more email bursts.*

Xuan and Filkov further investigate the association between co-commit bursts and email bursts and, therefore, argue:

Hypothesis 5. *Email bursts and synchronous development are positively coupled.*

3.2.3 Comparative Hypotheses

As a further enhancement to the study of Xuan and Filkov, we compare the differences of synchronous development based on our three different artifact types, namely files, features and functions. Since features are semantic units, we assume that feature-based synchronous development is significantly more distinct than file-based synchronous development. Moreover, we hypothesize that function-based synchronous development occurs less frequently than file-based and feature-based synchronous development since functions can be very small parts of the source code and synchronous changes to the same functions by different developers are not that effective as collaboration on bigger parts of software units is. One reason for that is that in a commit often more than one function is changed and hence the synchronicity of synchronous development on functions is less than on semantic units like features.

To investigate whether our assumptions on the relationship between file-based, feature-based, and function-based synchronous development are right, we compare the outcomes of the Hypotheses 1 to 5 of the three different artifact-type-based analyses to each other. Each of the following hypotheses refers to one hypothesis of the above ones.

Hypothesis 6.1 (refers to Hypothesis 1)**.** *Synchronous development on features occurs significantly more frequently than synchronous development on files and synchronous development on functions.*

Hypothesis 6.1a (refers to Hypothesis 1a)**.** *The synchronicity degree of synchronous development on features is significantly higher than the synchronicity degree of synchronous development on files or on functions.*

Hypothesis 6.2 (refers to Hypothesis 2)**.** *The code growth per commit is significantly higher in synchronous development on features than in synchronous development on files or on functions. In addition, the code effort in terms of added and deleted LOC is significantly smaller on features than on files or functions.*

Hypothesis 6.3$_<$ (refers to Hypothesis 3$_<$)**.** *The number of deleted lines per commit within co-commit bursts on features is significantly smaller than within co-commit bursts on files or on functions.*

Hypothesis 6.3$_>$ (refers to Hypothesis 3$_>$)**.** *The number of deleted lines per commit within co-commit bursts on features is significantly greater than within co-commit bursts on files or on functions.*

Hypothesis 6.4 (refers to Hypothesis 4)**.** *The effect "More co-commit bursts are accompanied with more email bursts" is stronger on feature-based synchronous development than on file-based or function-based synchronous development.*

Hypothesis 6.5 (refers to Hypothesis 5)**.** *Email bursts and synchronous development on features are stronger positively coupled than email bursts and file-based or function-based synchronous development.*

4. Methodology and Implementation

To determine synchronous development in open-source software (OSS) projects and to analyze the identified co-commit bursts and email bursts, we use the following methodology. After gathering the required commit and email data, we use the statistical programming language R[1] to determine coordination bursts and analyze them.

Firstly, we present our subject system QEMU. Secondly, we describe how we gather our commit and email data. Thereafter, we explain the procedure of identifying bursts, our null models, and our metrics to check our hypotheses. Finally, we deal with the limitations to our study.

4.1 Subject System: QEMU

We analyze QEMU[2], short for *Quick Emulator*, which is an open-source virtual-machine emulator which uses dynamic translation to virtualize hardware, i.e, it allows to run software made for a specific hardware on a machine with another hardware.

QEMU is developed in the programming language C, features are implemented by using C preprocessor directives. For the development of QEMU, there exists a Git repository[3] where the source code is managed. In addition, a developer mailing list exists, which can be downloaded from the Gmane mailing list archive[4] as well. Since QEMU is an OSS project, everyone can contribute to the software project. Nevertheless, QEMU has a policy[5] which forces developers to send patches, which can contain bug fixes or newly implemented features, to the developer mailing list.

[1]http://www.r-project.org/
[2]http://qemu.org/
[3]http://git.qemu.org/
[4]http://dir.gmane.org/gmane.comp.emulators.qemu/
[5]http://wiki.qemu.org/Contribute/SubmitAPatch/

Moreover, every patch-commit has to follow a certain coding style and has to comprise a meaningful commit message.

Between 2009-04-02 and 2016-06-15, 918 different developers contributed to the source code of QEMU and added 29,312 commits. Within the same time span, the corresponding mailing list collected 380,958 emails, 346,714 of them written by developers which also contributed to the source code.

4.2 Extraction of Commit and Email Data using Codeface

At the beginning, we gather the commit and email data for the QEMU OSS project. To get data about the commits of the project, we use the publicly available data of the git repository hosted on GitHub[6]. All software projects stored there use the version-control system git[7], which makes it possible for us to extract the relevant commit data. For the extraction, we use the framework Codeface[8], which analyzes social and technical aspects of development in software projects based on the version history of a git project. Codeface allows us to gather information on files, features and functions within the software. Internally, Codeface uses the tool cppstats [LAL+10] to determine which code belongs to which feature, based on preprocessor directives. Using Codeface, we are able to extract the meta data of all commits of a software project [JMA+15]. As given in Table 4.1, commit meta data contains information on the author of a commit, the date at which the author had finished the commit, the number of added and deleted lines as well as the changed artifacts (files, features, functions), together with the number of added lines per artifact.

Besides the commit data, we also need to gather data about the emails sent corresponding to the development of the OSS project. Therefore, we downloaded emails

Table 4.1: Commit meta-data which we extract from the git repository.

meta-data extracted from each commit
author
author date
number of added lines
number of deleted lines
changed artifacts
number of added lines per artifact

Table 4.2: Email meta-data which we extract from the mailing list archive.

meta-data extracted from each email
author
creation date

[6]https://github.com/qemu/qemu/
[7]https://git-scm.com/
[8]https://siemens.github.io/codeface/

from a publicly available developer mailing list corresponding to the project to analyze. In particular, we get the emails from the mailing list archive GMANE[9]. Then, we also use the framework CODEFACE to extract data out of the email headers, namely the date an email was created on as well as the author of an email, as stated in Table 4.2.

Nevertheless, authors often use different names and several email addresses, which makes it hard to map emails with different names and addresses of the sender to one real person. CODEFACE first tries to assign emails to a certain author by matching names, which was developed by Oliva et al. [OSdO+12]. If names of authors do not match, CODEFACE proceeds with matching email addresses. According to Wiese et al., the heuristic of Oliva et al. to disambiguate the authors of emails provides good results compared to other email disambiguation heuristics [WTdSS+16].

Since CODEFACE stores the outcome of its analyses in a database on a remote machine, we deploy an extension[10] to CODEFACE to extract the data relevant for our study from the database. So, we are machine independent in the end.

4.3 Identification of Bursts

After gathering the required meta-data of commits and emails, the identification of coordination bursts can begin. In our study, we performed three different burst identifications with different time windows on each of the three artifact-based analyses. We consider time windows ξ of 1 day, 5 days, and 10 days. To identify coordination bursts, we exactly follow the algorithm developed by Xuan and Filkov [XF14].

Co-commit bursts are a set of commit activities of a pair of developers within a certain time interval ξ, in which both developers change, at least, one artifact together. Therefore, we determine the C-bursts for each pair of developers. (1) So, for each pair of developers A and B, we check in a first step if they have artifacts which they both commit to at any time.[11] If A and B never changed the same artifact, it is pointless to search for co-commit bursts. (2) In the case that they both have, at least, one changed artifact in common, we iterate over all commits of developer A and compare each commit of A to all commits of developer B which have an absolute time difference to the considered commit of developer A of at most ξ. So, it does not matter whether the commit of A was finished before the commit of B or the other way round. (3) In the next step, we check whether the identified commits of B within a time window of ξ to the considered commit of A change, at least, one artifact which also is changed in the considered commit of A. (4) If so, we determine a burst consisting of one commit of A and all the commits of B that co-change, at least, one artifact changed by the commit of A within the chosen time window. (5) As a final step, we merge overlapping bursts of the same developer pair, i.e., we check whether the newly identified burst starts before another burst has ended or ends after another burst has started. In that case, we combine the two bursts and

[9] http://dir.gmane.org/gmane.comp.emulators.qemu/

[10] https://github.com/clhunsen/codeface-extraction/

[11] In the file-based analysis, we check if they both commit to any file. In the feature-based analysis, we only look for features they both commit to, and in the function-based analysis we search for functions they have changed both.

get one burst with a longer time interval. (6) In addition, we also determine the occurrence time of each burst, which is just the mean of start time and end time. In the end, for each pair of developers we have a separate list of co-commit bursts where each burst is represented by its starting, ending, and occurrence time.

For determining email bursts, almost the same approach as for co-commit bursts is used: (1) For each pair of authors A and B, we iterative over all the emails created by author A and search for all emails of author B whose creation date has an absolute time difference of less or equal ξ to the considered mail of A. Here, we have no further conditions to check, so all detected emails within the time window form an email burst. (2) We also merge overlapping email bursts of the same pair of authors in the same way as we merge overlapping co-commit bursts. (3) Finally, for each pair of developers we get a separate list of email bursts where each burst is represented by its starting, ending, and occurrence time.

4.4 Synchronicity Degree

The above introduced methodology to identify synchronous development is kind of superficial because it does not consider how big the overlap between two commits of two developers actually is. As an enhancement to the work of Xuan and Filkov, we quantify the synchronicity of all the commits within a co-commit burst. Thus, we developed a metric, called *synchronicity degree*, which measures the overlap based on the number of lines of code (LOC) each of the two developers adds to several artifacts of the commits within a co-commit burst.

We calculate the synchronicity degree for each co-commit burst. Since every burst refers to a pair of developers, the synchronicity degree determines the degree of synchronicity between the two developers within a burst.

Let x denote a certain burst of the developer pair consisting of developers A and B. With that, we define the synchronicity degree deg_{sync}:

$$deg_{sync}(x) = \sqrt[2]{\frac{added(A, syncArtifacts(x))}{added(A, x)} \cdot \frac{added(B, syncArtifacts(x))}{added(B, x)}} \tag{4.1}$$

where

$added(A, x)$ returns the amount of lines added by developer A within co-commit burst x,

$syncArtifacts(x) = changedArtifacts(x, A) \cap changedArtifacts(x, B)$

with

$changedArtifacts(x, A)$ returning all the artifacts which are changed by developer A in co-commit burst x.

In other words, to determine the synchronicity degree of a co-commit burst, we calculate the geometric mean over a product consisting of two factors where each

factor refers to one of the two developers forming the burst. Each of the factors is the fraction of the number of added lines to synchronous artifacts over the number of added lines to all artifacts made by the single developer the factor refers to. To compute the synchronicity degree, we do not look at single commits within bursts, but we gather the changed artifacts and the added lines of all commits of one developer within the considered burst.

The geometric mean is used here since higher values get not that much weight compared to the arithmetic mean. That is the effect we need since we want the synchronicity degree to be lower if one developer adds a lot more lines within a burst than the other one.

We explain some characteristics of our synchronicity degree on three examples given in Table 4.3, Table 4.4 and Table 4.5. The first example, stated in Table 4.3, considers a co-commit burst of the developers A and B which contains changes to four different artifacts. However, there are only synchronous changes on two artifacts, namely Artifact 2 and Artifact 3, whereas Artifact 1 was only changed by developer A and Artifact 4 only be developer B. So, one may assume that the synchronicity degree may be 0.5 since half of the changed artifacts get synchronously changed. If one considers the number of lines which were added to each of the artifacts, one can see that developer A only adds 20 lines to the synchronously changed artifacts, whereas developer B adds 8,000 lines to them. That gives one the idea to directly compare the 20 lines to the 8,000 lines based on the arithmetic mean, which would result in a very low degree of synchronicity. However, this idea does not contemplate that developer B adds a lot more lines to all changed artifacts than developer A does. Therefore, we look at the fraction of what each developer adds to the synchronous artifacts compared to the additions to all her changed artifacts, and then use the geometric mean to reduce the weight of higher fractions, which results in our definition of synchronicity degree in Equation 4.1. So, in the example in Table 4.3 the synchronicity degree is computed as follows:

$$
\begin{aligned}
deg_{sync}(c_1) &= \sqrt[2]{\frac{added(A, syncArtifacts(c_1))}{added(A, c_1)} \cdot \frac{added(B, syncArtifacts(c_1))}{added(B, c_1)}} \\
&= \sqrt[2]{\frac{10+10}{100+10+10} \cdot \frac{4000+6000}{4000+6000+1000}} \\
&\approx 0.39
\end{aligned}
\tag{4.2}
$$

To also show some other scenarios, in Table 4.4 we depict a C-burst in which the two developers change one of three artifacts synchronously and every developer adds the equal amount of lines to each artifact, which results in a synchronicity degree of $deg_{sync}(c_2) = \sqrt[2]{\frac{10}{20} \cdot \frac{10}{20}} = 0.50$.

Lastly, in Table 4.5 we provide an example where developer B completely dominates the co-commit burst in terms of added lines, but only added few lines to the synchronous artifacts and therefore the synchronicity degree approximately is $deg_{sync}(c_3) = \sqrt[2]{\frac{10}{15} \cdot \frac{10}{2010}} \approx 0.06$.

Table 4.3: Example for a C-burst, named $\mathbf{c_1}$, of the developers A and B, having low synchronicity degree though both are contributing to almost the same artifacts, where developer B changes much more lines within the burst than developer A.

Developer	Artifact	Added LOC	Synchronicity Degree
A	Artifact 1	100	0.39
	Artifact 2	10	
	Artifact 3	10	
B	Artifact 2	4,000	
	Artifact 3	6,000	
	Artifact 4	1,000	

Table 4.4: Example for a C-burst, named $\mathbf{c_2}$, of the developers A and B where both developers change the same amount of lines and also change the same amount on the synchronously changed artifact.

Developer	Artifact	Added LOC	Synchronicity Degree
A	Artifact 1	10	0.50
	Artifact 2	10	
B	Artifact 2	10	
	Artifact 3	10	

Table 4.5: Example for a C-burst, named $\mathbf{c_3}$, of the developers A and B, having very low synchronicity degree though both are contributing the same amount of LOC to the synchronously changed artifact.

Developer	Artifact	Added LOC	Synchronicity Degree
A	Artifact 1	5	0.06
	Artifact 2	10	
B	Artifact 2	10	
	Artifact 3	2,000	

4.5 Null Models

In the previous sections, we denoted how we identify bursts and how we quantify the synchronicity of co-commit bursts. Nevertheless, we do not know whether the results are statistically significant. To ascertain that our identified bursts and their synchronicity do not appear randomly, we build null models. We stick to the null models illustrated by Xuan and Filkov [XF14].

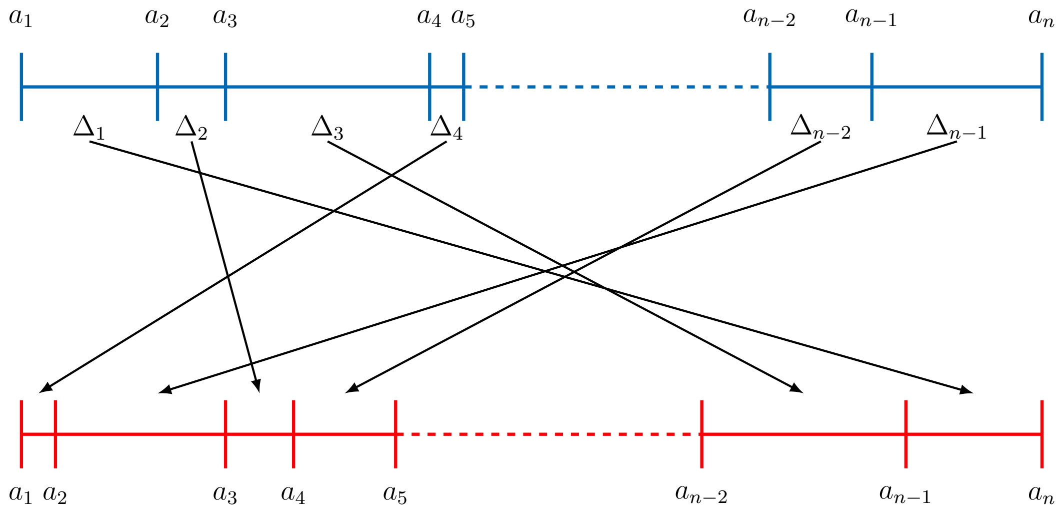

Figure 4.1: Randomization of commit time series. Adapted from [XGDF12].

First, we generate a null model for the commit time series. Therefore, we randomize the time intervals between two successive commits for each developer by just transposing the order of the intervals. Thus, the distribution of the time intervals stays the same as well as the order and the changed artifacts of the commits of each developer [XGDF12, XF14]. To randomize the time intervals, we use the `sample` function of the base package of R without replacement. In Figure 4.1, we exemplarily perform a randomization for commit activities $a_1, \ldots, a_n$ of a single developer by transposing the intervals $\Delta_1, ..., \Delta_{n-1}$ between two successive commits.

Based on the randomized time series of the commits of each developer, the co-commit bursts on the randomized time series, which are called *simulated C-bursts* [XF14], are determined as described in Section 4.3.

To check Hypothesis 1 stating that the real C-bursts are not generated by chance, we compare the real C-bursts to the simulated ones. For the null model, we generate fifty different simulations. On that point, we count the number of C-bursts per developer pair for real and all simulated C-bursts and use a paired t-test to statistically compare the real number of bursts per developer pair and the median of the number of bursts per developer pair for the simulations. We use Cohen's d [Coh88] as an effect size to measure the magnitude of the differences.

Besides the null model for the commit time series, also a null model for the email communication is necessary to avoid that the identified email bursts just appeared by chance. Hereto, we use almost the same approach as for null model of the commit time series. In contrast, we do not randomize the time intervals between the emails of

each author, but the time intervals between successive emails of each pair of authors to keep the order of emails sent by two different authors. So, each pair of authors has its own simulated email-time series. Among them, the simulated E-bursts are determined as described in Section 4.3.

The email null model is used to check whether a noticed correlation between C-bursts and E-bursts does not appear randomly. This is the topic of Section 4.7.

4.6 Code-Growth and Code-Effort Metrics

Analyzing synchronous development comprises to investigate whether there is a significant difference between the code growth in synchronous and non-synchronous development. Additionally, it is assumed that the effort in synchronous development is significantly smaller than in non-synchronous development.

To examine Hypothesis 2 and Hypothesis 3, Xuan and Filkov [XF14] established some metrics, which we also use for our study.

The two basic metrics are the number of added (L_{add}) and deleted (L_{delete}) lines per commit. In our study, we use CODEFACE to extract the meta data for each commit. That already gives us the information how many lines were added and how many lines were deleted within a commit.

However, CODEFACE provides only the number of added lines per artifact (file, feature, function), but not the deleted lines per artifact. Since we do not consider changes on header files within our analyses, we have to subtract the added lines to the header files from the added lines of the whole commit. This also holds for the base-feature lines within the feature-based analysis as well as for the file-level lines within function-based analysis. Moreover, we also should subtract the number deleted lines of those files from the overall number deleted lines of a commit, but this is not possible for us since CODEFACE does not tell us the number of deleted lines on artifact level. Thus, we develop heuristics which aim to mitigate the issue. We assume that lots of changes to the header files (resp. base feature, file level) are just changes, i.e., there are as many added lines as deleted lines in a commit on these files. Besides these changes, we also assume that there are more commits that just add new functionality than commits deleting it. Hence, we estimate that the number of deleted lines on the disregarded artifacts is half as high as the number of added lines to these artifacts, which leads to the following equation:

$$L_{delete(disregardedArtifact)} = 0.5 \cdot L_{add(disregardedArtifact)} \tag{4.3}$$

To quantify the code growth and effort in terms of lines of code (LOC), Xuan and Filkov defined the following metrics [XF14]:

$$\text{Code growth:} \qquad \Delta L = L_{add} - L_{delete} \tag{4.4}$$

$$\text{Code effort:} \qquad \Delta W = L_{add} + L_{delete} \tag{4.5}$$

All the metrics $L_{add}, L_{delete}, \Delta L,$ and ΔW are determined per commit.

The comparison of synchronous to non-synchronous development presumes that we categorize which commits are synchronous and which not. In our study, we consider

a commit to be synchronous if there is, at least, one burst for any of the developer pairs which the commit is part of. That is to say, a commit is treated as synchronous if there is, at least, one artifact in the commit that is synchronously changed by another commit of another developer. Each commit which does not contain such synchronously changed artifacts is considered to be non-synchronous.

Lastly, we use a statistical t-test to compare added lines, deleted lines, code growth, and code effort of synchronous commits to the regarding metrics of non-synchronous commits.

4.7 Correlation between C-Bursts and E-Bursts

To investigate whether the occurrences of co-commit bursts and email bursts are related, we again use the methodology presented by Xuan and Filkov [XF14]. In Hypothesis 4, we assume that more C-bursts are accompanied with more E-bursts. To check this hypothesis, we perform a correlation test, using Pearson correlation coefficient [CP91], for the numbers of C-bursts and E-bursts per pair of developers. In addition, we fit a linear model with parameters α and β between the number of C-bursts, denoted by N_C, and the corresponding number of E-bursts, denoted by N_E:

$$N_C = \alpha N_E + \beta \tag{4.6}$$

Furthermore, we additionally check Hypothesis 5, saying that email bursts and co-commit bursts are positively timely correlated, using the Pearson correlation coefficient. Therefore, we only consider developer pairs which have more than five C-bursts and more than five E-bursts, according to the study of Xuan and Filkov. For each pair of developers fulfilling the above preconditions, we perform the following steps to get the curves and calculate the correlation coefficient:

(1) First, let Γ_C be the set of C-bursts of the current pair and Γ_E be the set of E-bursts of the current pair. Let L denote the day on which the first burst in $\Gamma_C \cup \Gamma_E$ begins and let U denote the day on which the last burst in $\Gamma_C \cup \Gamma_E$ ends. So, let T be a series of time stamps, more precisely days, beginning at L and ending at U.
(2) For each C-burst of the considered pair, we count the commit activities which form the burst of both developers together for each day within the burst. So, for each day $t \in T$, we get the corresponding number of commit activities of the considered burst. We perform the same step also for E-bursts by counting the emails of the considered developer pair per day within the considered burst. Hence, for each C-burst of the current developer pair we have a time series x_C containing the amount of commit activities of the analyzed burst per day $t \in T$, and also for each E-burst we have a time series x_E containing the amount of email activities of the analyzed burst per day $t \in T$.
(3) Afterwards, we combine all C-bursts of one developer pair to a C-curve φ_C, and we also sum all E-bursts of the developer pair to get an E-curve φ_E as follows:

$$\varphi_C(t) = \sum_{x_C \in \Gamma_C} x_C(t) \tag{4.7}$$

$$\varphi_E(t) = \sum_{x_E \in \Gamma_E} x_E(t) \tag{4.8}$$

(4) Finally, we use the determined curves to calculate the Pearson correlation coefficient of the currently considered developer pair, which Xuan and Filkov defined as follows:

$$R = \frac{\int_L^U \varphi_C(t)\varphi_E(t)\ dt}{\sqrt{\int_L^U [\varphi_C(t)]^2 dt}\sqrt{\int_L^U [\varphi_E(t)]^2 dt}} \tag{4.9}$$

In our implementation, we compute the integrals by the sum of the function values of the integrand. In the end, we have calculated R for each pair of developers having more than five C-bursts and more than five E-bursts.

To be sure that the determined correlation between co-commit bursts and email bursts appears not by chance, we use our null model for email time series, which we already have described in Section 4.5. Thus, we detect the email bursts on our simulated email time series and compute the corresponding E-curves for the simulation. Hence, for each pair of developers, we calculate the correlation coefficient R between real C-curves and real E-curves as well as between real C-curves and simulated E-curves. In both cases, we just consider developer pairs having more than five C-bursts and more than five E-bursts. Afterwards, we use a Mann-Whitney U test (we cannot use a t-test due to the reduced amount of considered developer pairs here) to compare real and simulated correlation coefficients. For measuring the effect size of the correlation, we use Cliff's Delta [Cli96] here to measure the magnitude of the differences. Hypothesis 5 will be accepted if the correlation coefficients using the real email time series are significantly higher than the correlation coefficients on the simulated email data.

4.8 Limitations and Differences to the Original Study

Our above denoted methodology deviates from the one stated by Xuan and Filkov [XF14] since we had to deal with some technical limitations. Therefore, in this section, we will shortly summarize our restraints and differences to the replicated study.

First of all, we analyze the QEMU OSS project, which has a comparably larger number of developers and emails than the projects Xuan and Filkov have analyzed. Since our amount of emails to handle (380,958) is much higher than theirs (73,965 added up for their six investigated case studies), it is not feasible for us to analyze all of them. Therefore, we decided to only consider emails which were written by an author which is also an author of a commit.

To generate their null model, Xuan and Filkov used 100 simulations of commit bursts and also 100 simulations of email bursts. Due to the above reasons, we were only able to generate 50 simulations for the commit time series and one simulation for the email time series, which limits the validity of our null-model, at least, for the email-burst analysis.

Since commit activities and email communication activities may not take place at exactly the same time and, therefore, may happen slightly deferred, the original

authors use Gaussian smoothing to slightly level the appearance of co-commit and email bursts. Due to the lack of time, we postpone the smoothing to future work. Instead, we compare the C-curves and E-curves without any smoothing or normalization.

Another limitation is the correctness of the code-growth and code-effort metrics, which depend on the number of deleted lines per artifacts which we do not consider, such as header files. Since the framework CODEFACE does not provide the deleted lines on artifact level, we use an heuristic to estimate the deleted lines of not-considered artifacts, which distorts the outcome of the regarding metrics. Due to the large number of commits of QEMU (29,312), it is not feasible to extract the number of deleted lines on each artifact within each commit. For the same reason, we cannot determine added and deleted lines for synchronous and non-synchronous commits as Xuan and Filkov do because we only know these values on the commit level but not on artifact level within commits and, therefore, we always analyze the whole commit en bloc.

Furthermore, the original authors use a different algorithm to disambiguate the authors of emails. Xuan and Filkov use the email disambiguation algorithm of Bird et al. [BGD+06], which uses a similarity metric between pairs of email addresses and names and links pairs where the result of the similarity metric is under a certain threshold. In contrast, CODEFACE uses the email disambiguation heuristic of Oliva et al. [OSdO+12], which groups email addresses if the names of the senders are equal. According to Wiese et al. [WTdSS+16], both heuristics show good results compared to other heuristics. However, the heuristic of Bird et al. is more influenced by the number of analyzed emails and authors than the heuristic of Oliva et al., which we use in our study.

5. Results and Discussion

In this chapter, we provide some data on the determined coordination bursts in QEMU, which we gathered during our analyses. Thereafter, we present the statistical results needed to accept or reject our hypotheses from Chapter 3. Afterwards, we discuss our results and also denote the threats to validity of our study.

5.1 Results

We analyze the commit and email data of the open-source software (OSS) project QEMU within the time interval from the start of April in 2009 (2009-04-02) to the mid of June in 2016 (2016-06-15). To determine synchronous development, we use the official mirror of the software repository on GitHub[1]. For the analysis of the devolpers' coordination via email, we have downloaded email data from the mailing-list archive Gmane[2]. In this section, we just provide the general results of our study and the results regarding our hypotheses.

5.1.1 General Results

General Results regarding the Number of Developers

Before checking our hypotheses, we obtain some general results regarding the identified C-bursts and E-bursts. Overall, in our file-based approach, we analyze the commits of 918 different developers, which results in 842,724 distinct pairs of developers. Within the feature-based analysis, we consider only 461 developers since the remaining 457 developers only contribute to the base feature, which we completely disregard in the feature-based analysis. Ultimately, due to disregarding file-level changes in the function-based analysis, we analyze 874 developers committing to functions.

General Results regarding C-Bursts

In Table 5.1, we present statistics such as the length or the synchronicity degree for the determined C-bursts of the 1-day time window. Analyzing the implementation

[1]https://github.com/qemu/qemu/

[2]http://dir.gmane.org/gmane.comp.emulators.qemu/

Table 5.1: General data on the identified C-bursts of QEMU for a time window of 1 day.

		Files	Features	Functions
#Developers		918	461	874
#Developer Pairs		420,903	106,030	381,501
#Bursts overall		1,755	765	819
#C-Bursts per Pair	min	0.00	0.00	0.00
	median	0.00	0.00	0.00
	mean	0.00	0.01	0.00
	max	15.00	9.00	9.00
Length of C-Bursts (in days)	min	0.00	0.00	0.00
	median	0.28	0.47	0.27
	mean	0.39	0.49	0.36
	max	1.96	1.98	1.26
Synchronicity Degree	min	0.00	0.01	0.00
	median	0.30	0.50	0.12
	mean	0.42	0.58	0.22
	max	1.00	1.00	1.00

Table 5.2: General data on the identified C-bursts of QEMU for a time window of 5 days.

		Files	Features	Functions
#Developers		918	461	874
#Developer Pairs		420,903	106,030	381,501
#Bursts overall		4,644	2,281	2,182
#C-Bursts per Pair	min	0.00	0.00	0.00
	median	0.00	0.00	0.00
	mean	0.01	0.02	0.01
	max	28.00	23.00	14.00
Length of C-Bursts (in days)	min	0.00	0.00	0.00
	median	2.41	2.69	2.18
	mean	2.63	2.79	2.39
	max	19.35	13.51	13.53
Synchronicity Degree	min	0.00	0.00	0.00
	median	0.20	0.50	0.08
	mean	0.32	0.53	0.17
	max	1.00	1.00	1.00

Table 5.3: General data on the identified C-bursts of QEMU for a time window of 10 days.

		Files	Features	Functions
#Developers		918	461	874
#Developer Pairs		420,903	106,030	381,501
#Bursts overall		7,261	3,721	3,604
#C-Bursts per Pair	min	0.00	0.00	0.00
	median	0.00	0.00	0.00
	mean	0.02	0.04	0.01
	max	35.00	22.00	21.00
Length of C-Bursts (in days)	min	0.00	0.00	0.00
	median	6.14	6.56	5.76
	mean	6.27	6.58	5.61
	max	82.47	54.79	30.87
Synchronicity Degree	min	0.00	0.00	0.00
	median	0.15	0.49	0.06
	mean	0.28	0.50	0.14
	max	1.00	1.00	1.00

files of QEMU, we identify 1,755 bursts for all pairs of developers. At maximum, there are 15 C-bursts per pair of developers, but most of the pairs have few or no C-bursts as the median of zero bursts per developer pair denotes. A file-based C-burst has a length of 0 up to 1.96 days. The median synchronicity degree is 0.42. In fact, there are bursts which have a synchronicity degree of 1, as well as bursts, which have a synchronicity degree of almost 0. Performing the feature-based analysis, we find a number of 765 C-bursts overall. A single pair of developers has up to 9 bursts and a synchronicity degree of 0.58 on average. Within the function-based analysis, we identify 819 C-bursts having a maximal length of 1.26 days with a median synchronicity of 0.22.

For the time window of 5 days we get similar results, given in Table 5.2. In difference to the 1-day time window, we get a higher number of bursts, namely 4,644 bursts for the file-based analysis, 2,281 for the feature-base one, and 2,182 for the function-based one. In addition, the length of C-bursts also is higher due to the larger time window.

In Table 5.3, we show the general results of the C-bursts using a time window of 10 days. Here, we determine 7,261 bursts on file-based collaboration, 3,721 on feature-based collaboration, and 3,604 on function-based collaboration. Within the time window of 10 days, a developer pair causes up to 35 C-bursts with a length up to 82.47 days for file-based bursts, up to 54.79 for feature-based bursts, and up to 30.87 days for function-based bursts.

Table 5.4: General data on the identified E-bursts of QEMU for the time windows of 1 day, 5 days and 10 days.

Time window		1 day	5 days	10 days
#Email authors (only authors of commits)		883	883	883
#Author Pairs (only authors of commits)		389,403	389,403	389,403
# Bursts overall		910,712	689,859	564,849
#E-Bursts per Pair	min	0	0	0
	median	0	0	0
	mean	2.34	1.77	1.45
	max	418	118	57
Length of E-Bursts (in days)	min	0.00	0.00	0.00
	median	0.99	6.37	14.79
	mean	1.25	9.02	20.82
	max	52.30	1,764.13	2,216.71

General Results regarding E-Bursts

In our email-burst analysis, we considered 883 different authors of emails, which also contributed to the source code as authors of commits. Thus, we analyzed 389,403 pairs of authors. The number of identified E-bursts is higher than the number of identified C-bursts, as depicted in Table 5.4. We determined 910,712 E-bursts for the 1-day time window respectively 689,859 for the 5-days time window, and 564,849 for the 10 days time window. In difference to the number of C-bursts, the number of E-bursts decreases as the time window increases. The reason for that is that increasing the time window often combines several smaller bursts. Contrary to the number of bursts, the length of E-bursts increases as the time window increases. Hence, the median length of an E-burst is 0.99 days for the 1-day time window, 6.37 days for the 5-days time window and 14.79 for the 10 days time window. An interesting fact is that there also exist E-bursts which last about five years or even more, namely 1,764.13 days for the 5-days time window or 2,216.71 days for the 10-days time window. Usually, pairs of developers taking part in such lengthy E-bursts belong to the lead developers of QEMU because they write emails to the mailing list, at least, every 5 days respectively 10 days for several years.

5.1.2 C-Burst-related Hypotheses

In our first hypothesis we state that synchronous development does not appear randomly. We use a paired t-test to compare the number of C-bursts per pair of developers of real and simulated C-bursts and estimate that the real number of bursts is larger than the simulated one. The result of the t-test is given in Table 5.5. For all three time windows ξ of 1 day, 5 days and 10 days we obtain that C-bursts based on files, features, or functions do significantly appear more frequently than by chance because the p-value is smaller than 0.05. However, the Cohen's d effect size between 0.02 and 0.04 is comparably small. This means that the mean of the differences between the number of real C-bursts and the number of simulated C-bursts per developer pair is small. Nevertheless, this is caused by the huge number

Table 5.5: Paired t-tests in the differences of the number of bursts per developer pair between real and simulated C-bursts for the different analyses and time windows.

	ξ	mean of differences	t	df	p-value	Cohen's d
files	1	0.003	24.67	420,900	< 0.05	0.04
	5	0.005	25.50	420,900	< 0.05	0.04
	10	0.006	26.72	420,900	< 0.05	0.04
features	1	0.005	10.89	106,030	< 0.05	0.03
	5	0.008	8.00	106,030	< 0.05	0.02
	10	0.010	7.46	106,030	< 0.05	0.02
functions	1	0.002	17.35	381,500	< 0.05	0.03
	5	0.003	14.23	381,500	< 0.05	0.02
	10	0.004	12.89	381,500	< 0.05	0.02

of pairs of developers which has no C-bursts. Therefore, we accept *Hypothesis 1.*

Hypothesis 1: Accepted.

Besides the number of bursts per pair of developers, we also check whether the determined synchronicity degrees are significantly higher than the synchronicity degrees of our simulations. Hereto, the conducted unpaired t-tests show that this is true for all time windows and for all levels of abstraction, which can be seen in the p-values in Table 5.6. Comparing real and simulated synchronicity degrees results in an effect size d between 0.18 and 0.39, which is comparably high compared to the effect size of comparing the number of real and simulated C-bursts per pair of developers. So, the differences between real and simulated synchronicity degrees are effectively higher than the differences between the number of real and simulated C-bursts per developer pair. In Figure 5.1 we depict the related box-and-whisker diagrams, showing that the synchronicity degree always is higher for the real C-bursts than for the simulated ones. Recapitulating, we accept *Hypothesis 1a*.

Hypothesis 1a: Accepted.

Since we know from the previous results that the determined synchronous development is not random, we further investigate in the code growth in synchronous development. We hypothesize that the code growth in terms of lines of code is higher in synchronous development than in non-synchronous development, whereas the code effort is lower in synchronous development than in non-synchronous development. Using a 1-day time window, we identified 5,493 commits as synchronous and 23,819 as non-synchronous in our file-based analysis, as given in Table 5.7. According to our performed unpaired t-test, denoted in Table 5.8, the number of added lines on files is significantly higher in synchronous development than in non-synchronous development since the p-values are smaller than 0.05. The same holds for the time window of 5 days, where we found 11,584 synchronous and 17,764 non-synchronous commits. Also for a 10-days time window, the number of added lines in synchronous development (15,980 commits) significantly exceeds the number of added lines in non-synchronous development (13,332 commits), as depicted in

Table 5.6: T-tests of the differences in the synchronicity degrees between real and simulated C-bursts for the different analyses and time windows.

	ξ	avg. real	avg. simulated	t	df	p-value	Cohen's d
files	1	0.42	0.30	19.56	3,661.60	< 0.05	0.39
	5	0.32	0.25	19.57	9,583.00	< 0.05	0.24
	10	0.28	0.23	20.24	14,979.00	< 0.05	0.20
features	1	0.58	0.55	3.95	1,594.00	< 0.05	0.10
	5	0.53	0.51	3.77	4,719.40	< 0.05	0.06
	10	0.50	0.48	5.67	7,704.70	< 0.05	0.07
functions	1	0.22	0.15	11.01	1,707.70	< 0.05	0.35
	5	0.17	0.13	12.71	4,483.60	< 0.05	0.25
	10	0.14	0.11	12.31	7,409.20	< 0.05	0.18

Table 5.7: Numbers of synchronous and non-synchronous commits for the different time windows and analysis types.

	ξ	# synchronous commits	# non-synchronous commits
files	1	5,493	23,819
	5	11,548	17,764
	10	15,980	13,332
features	1	1,758	4,082
	5	3,315	2,525
	10	3,927	1,913
functions	1	2,177	24,387
	5	4,445	22,119
	10	6,501	20,063

Figure 5.2 (a). In Figure 5.2 (b), we depict the comparison of the number of added lines within synchronous and non-synchronous commits for the feature-based analysis, where also the number of added lines per synchronous commit is higher than for non-synchronous commits. Nevertheless, the p-values of the performed t-tests (cf. Table 5.8) state that this difference is significant for feature-based analysis with 1-day time window, but not for the feature-based analysis with a time window of 5 or 10 days. Furthermore, in Table 5.8 we see that the the number of added lines (L_{add}) is significantly higher for synchronous development than for non-synchronous development, which also can be seen in Figure 5.2 (c).

Besides the number of added lines, in Hypothesis 2 we also consider the code-growth and code-effort metrics. In Figure 5.3, we show the results of the code-growth metric ΔL for all artifact types and all time windows. Based on these plots, the code growth in synchronous development seems to be larger than in non-synchronous development. Nonetheless, these differences are not significant since most of the p-values of the t-test comparing the code growth per commit in synchronous development

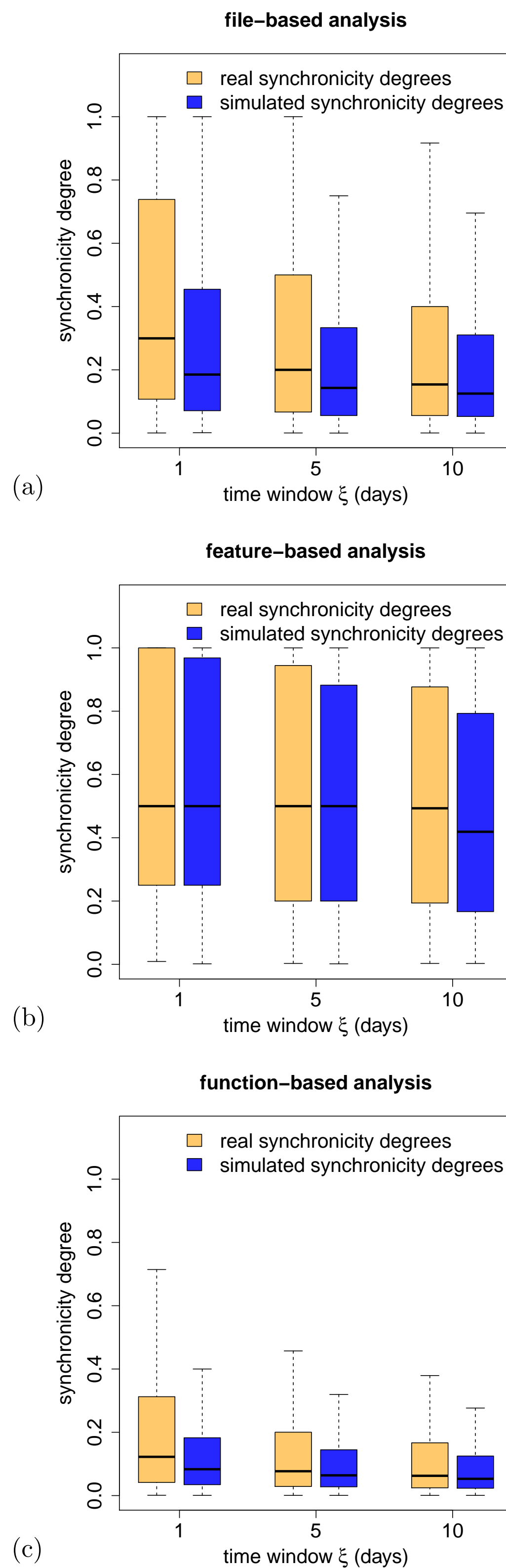

Figure 5.1: Box-and-whisker diagrams (without outliers) of the synchronicity degree of the real C-bursts and the simulated C-bursts for the (a) file-based, (b) feature-based, and (c) function-based analysis for the different time windows.

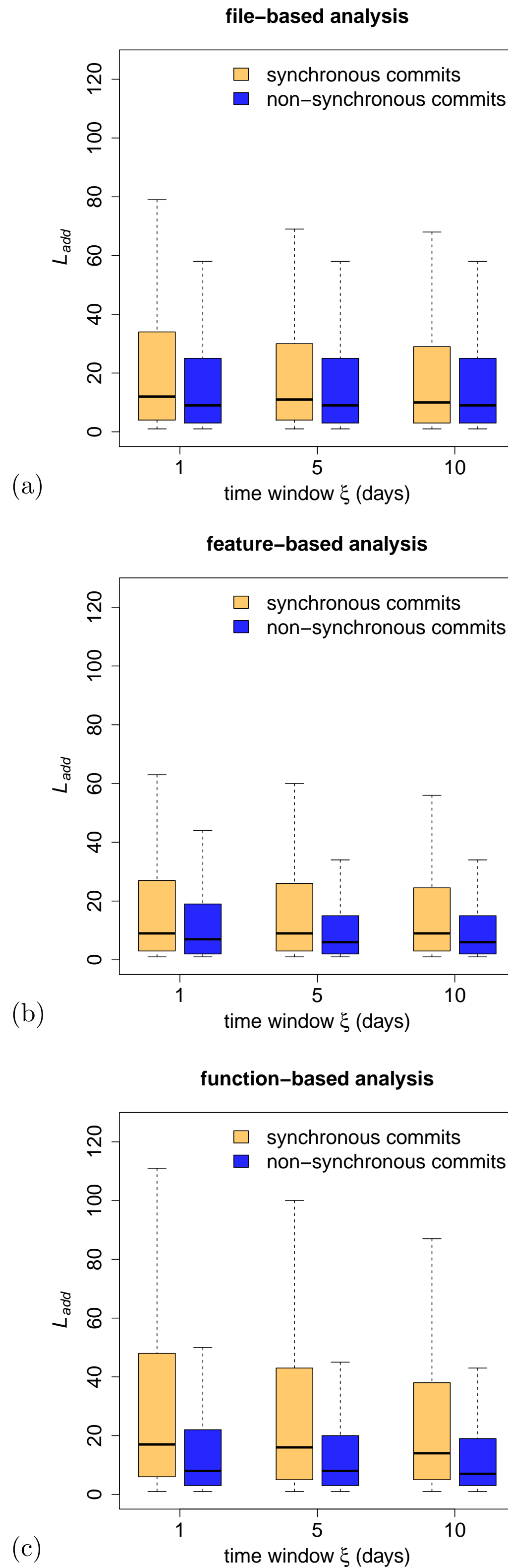

Figure 5.2: Box-and-whisker diagrams (without outliers) of the number of added lines per commit (L_{add}) within synchronous and non-synchronous commits for the (a) file-based, (b) feature-based, and (c) function-based analysis for the different time windows.

Table 5.8: T-tests of the differences of the number of added lines in synchronous development and in non-synchronous development for the different analyses and time windows.

	ξ	avg. sync.	avg. non-sync.	t	df	p-value	Cohen's d
files	1	40.67	37.13	1.66	13,797.00	< 0.05	0.02
	5	36.44	38.67	-1.09	27,989.00	< 0.05	0.01
	10	34.82	41.36	-2.70	17,604.00	< 0.05	0.03
features	1	41.84	28.84	2.45	2,350.60	< 0.05	0.08
	5	40.84	22.14	4.98	5,120.70	0.86	0.12
	10	38.57	20.82	5.12	5,837.50	1.00	0.12
functions	1	55.91	28.99	8.46	2,708.30	< 0.05	0.17
	5	55.77	26.26	6.52	4,696.60	< 0.05	0.19
	10	51.98	24.46	8.18	7,142.40	< 0.05	0.17

Table 5.9: T-tests of the differences of code growth in synchronous development and in non-synchronous development for the different analyses and time windows.

	ξ	avg. sync.	avg. non-sync.	t	df	p-value	Cohen's d
files	1	7.77	19.62	-2.00	6,210.90	0.98	0.04
	5	11.03	21.54	-3.05	21,149.00	1.00	0.04
	10	11.74	24.18	-3.82	27.942.00	1.00	0.04
features	1	-4.27	-3.97	-0.02	1,909.00	0.51	0.00
	5	-5.86	-1.71	-0.44	4,051.40	0.67	0.01
	10	-5.47	-1.19	-0.51	5,430.80	0.69	0.01
functions	1	21.87	13.44	0.67	2,209.80	0.25	0.04
	5	27.49	11.45	2.13	4,566.30	< 0.05	0.07
	10	25.54	10.44	2.83	6,874.30	< 0.05	0.06

and in non-synchronous development are greater than 0.05 (cf. Table 5.9). So, only for function-based synchronous development with time windows of 5 or 10 days, the code-growth in synchronous commits is significantly higher than in non-synchronous commits.

Next, we consider the code effort ΔW, which is depicted in Figure 5.4. For all time windows of file-based, feature-based, and function-based analyses, the effort in terms of lines of code is higher in synchronous commits than in non-synchronous commits. The conducted t-test for the comparison of code effort in synchronous and in non-synchronous commits also supports this outcome since the code effort is significantly higher in all cases due to a p-value larger than 0.05, as given in Table 5.10. Since code growth is not significantly higher in synchronous development than in non-synchronous development and also the code effort is not lower in synchronous commits than in non-synchronous commits, we reject *Hypothesis 2*.

Hypothesis 2: Rejected.

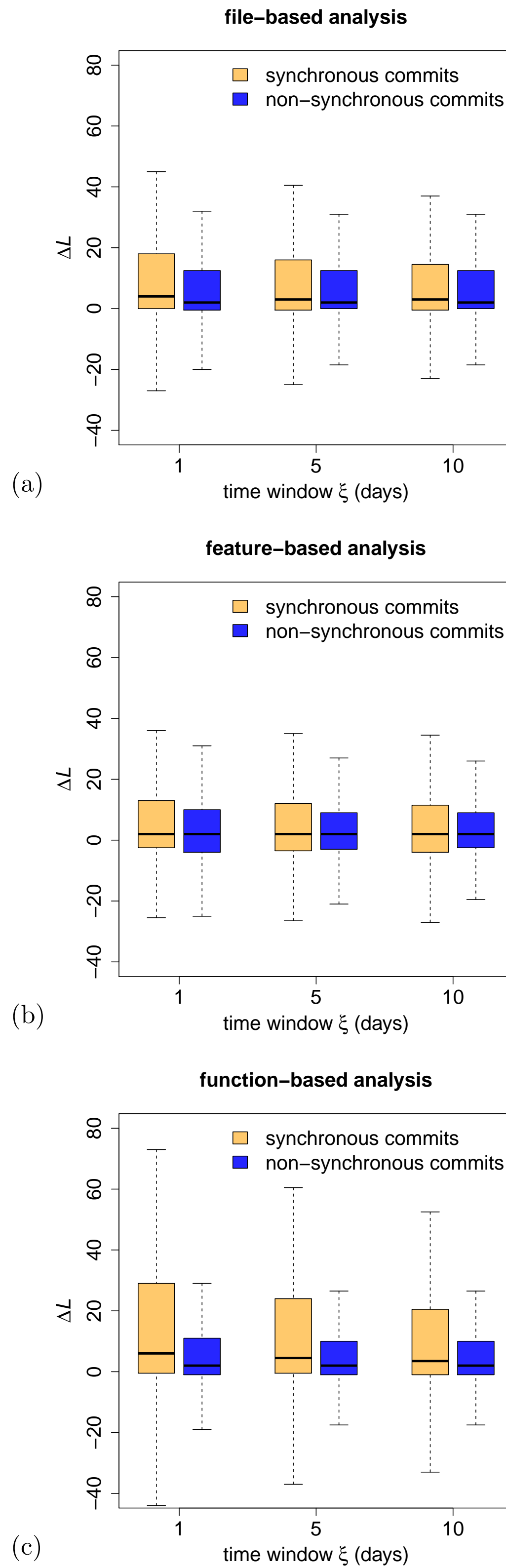

Figure 5.3: Box-and-whisker diagrams (without outliers) of the code growth ΔL per commit within synchronous and non-synchronous commits for the (a) file-based, (b) feature-based, and (c) function-based analysis for the different time windows.

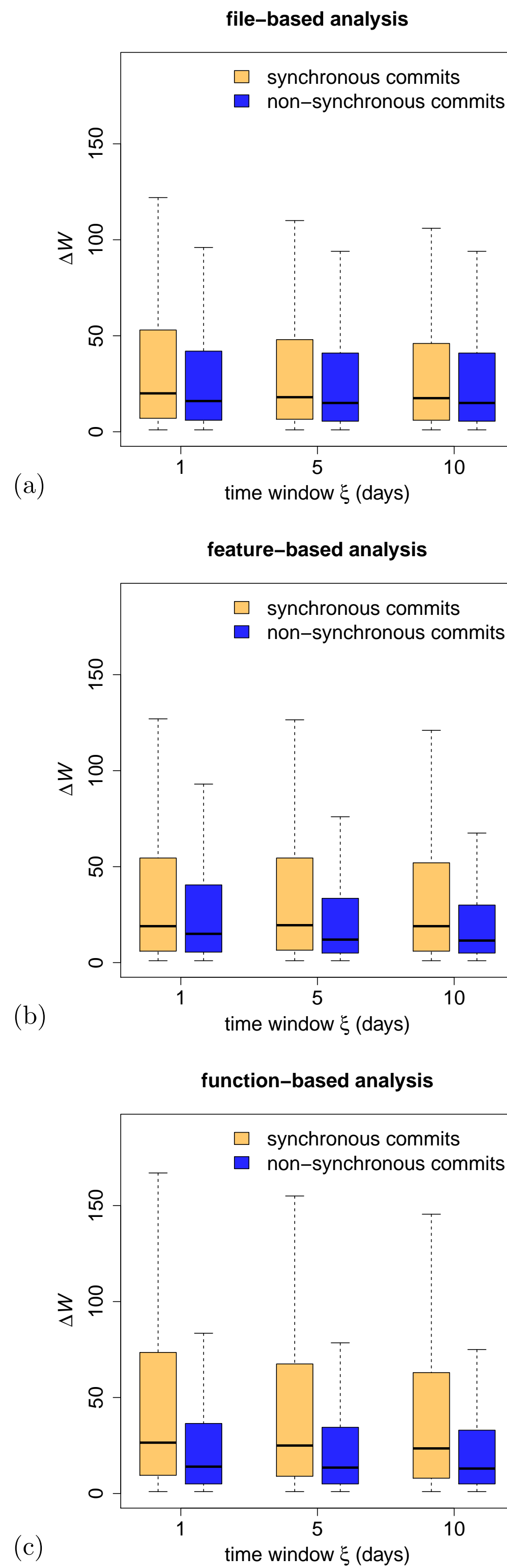

Figure 5.4: Box-and-whisker diagrams (without outliers) of the code effort ΔW per commit within synchronous and non-synchronous commits for the (a) file-based, (b) feature-based, and (c) function-based analysis for the different time windows.

Table 5.10: T-tests of the differences of code effort in synchronous development and in non-synchronous development for the different analyses and time windows.

	ξ	avg. sync.	avg. non-sync.	t	df	p-value	Cohen's d
files	1	73.57	54.64	2.91	6,221.00	1.00	0.06
	5	61.85	55.81	1.60	20,589.00	0.95	0.02
	10	57.89	58.54	-0.18	28,438.00	0.43	0.00
features	1	87.96	61.65	1.51	1,967.10	0.94	0.06
	5	87.53	45.99	3.96	4,047.60	1.00	0.09
	10	82.60	42.82	4.28	5,241.30	1.00	0.09
functions	1	89.94	44.54	3.46	2,214.50	1.00	0.18
	5	84.05	41.07	5.52	4,602.90	1.00	0.17
	10	78.42	38.49	7.14	6.949.50	1.00	0.16

Table 5.11: T-tests of the differences of the number of deleted lines in synchronous development and in non-synchronous development for the different analyses and time windows (estimation: number of deleted lines in synchronous development < number of deleted lines in non-synchronous development).

	ξ	avg. sync.	avg. non-sync.	t	df	p-value	Cohen's d
files	1	32.90	17.51	2.62	5,674.50	1.00	0.07
	5	25.41	17.13	2.77	14,068.00	1.00	0.04
	10	23.08	17.18	2.48	24.484.00	0.99	0.03
features	1	46.11	32.81	0.84	1,900.30	0.80	0.03
	5	46.69	23.85	2.48	3,872.30	0.99	0.06
	10	44.03	22.01	2.70	5,148.70	1.00	0.06
functions	1	21.87	13.44	1.48	2,184.70	0.93	0.10
	5	28.28	14.81	2.18	4,526.90	0.99	0.07
	10	26.44	14.03	2.88	6.778.40	1.00	0.07

The last metric we consider here is the number of deleted lines per commit. We hypothesize that this metric is higher in non-synchronous than in synchronous commits. However, as pictured in Figure 5.5, the number of deleted lines per commit is higher in synchronous development. This holds for all time windows and analysis types. Also the associated t-tests sustain this outcome, as given in Table 5.11. Thus, we reject *Hypothesis* $3_{<}$.

Hypothesis $3_{<}$: Rejected.

Checking the opposite hypothesis (cf. Table 5.12) results in a significantly higher number of deletions in synchronous development compared to the non-synchronous development. Hence, we accept *Hypothesis* $3_{>}$.

Hypothesis $3_{>}$: Accepted.

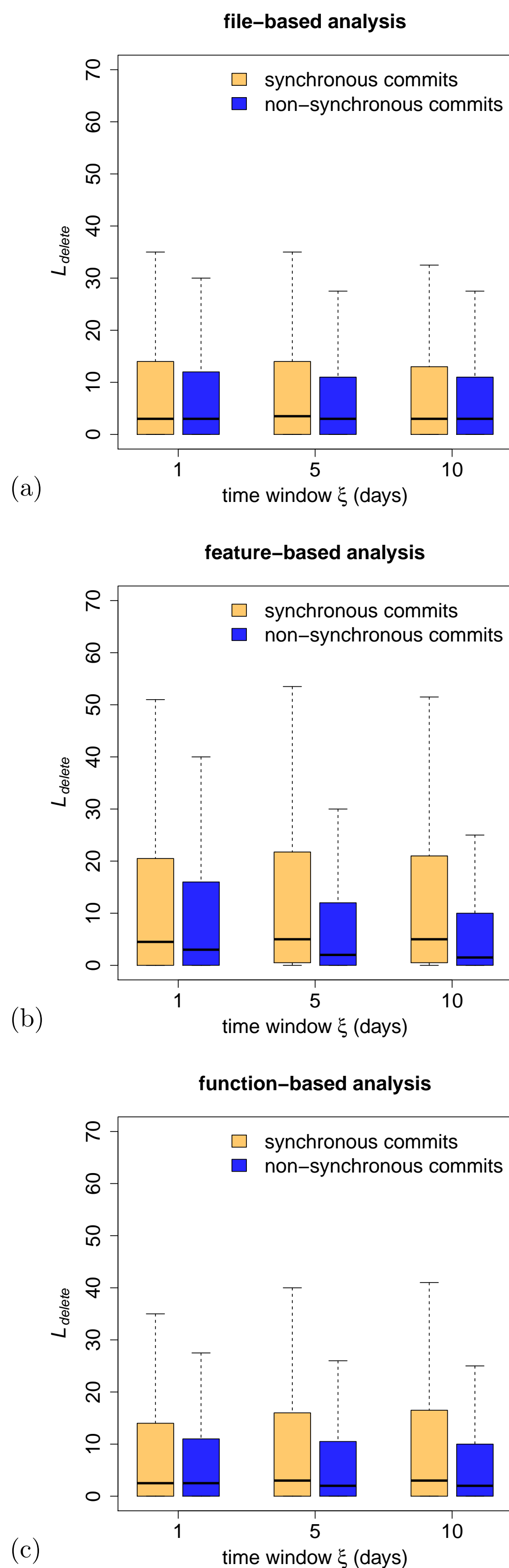

Figure 5.5: Box-and-whisker diagrams (without outliers) of the number of deleted lines per commit (L_{delete}) within synchronous and non-synchronous commits for the (a) file-based, (b) feature-based, and (c) function-based analysis for the different time windows.

Table 5.12: T-tests of the differences the number of deleted lines in synchronous development and in non-synchronous development for the different analyses and time windows (estimation: number of deleted lines in synchronous development > number of deleted lines in non-synchronous development).

	ξ	avg. sync.	avg. non-sync.	t	df	p-value	Cohen's d
files	1	32.90	17.51	2.62	5,674.50	< 0.05	0.07
	5	25.41	17.13	2.77	14,068.00	< 0.05	0.04
	10	23.08	17.18	2.48	24.484.00	< 0.05	0.03
features	1	46.11	32.81	0.84	1,900.30	0.20	0.03
	5	46.69	23.85	2.48	3,872.30	< 0.05	0.06
	10	44.03	22.01	2.70	5,148.70	< 0.05	0.06
functions	1	21.87	13.44	1.48	2,184.70	0.07	0.10
	5	28.28	14.81	2.18	4,526.90	< 0.05	0.07
	10	26.44	14.03	2.88	6.778.40	< 0.05	0.07

5.1.3 Correlation of C-Bursts and E-Bursts

We investigate whether C-bursts and E-bursts are correlated. First, we check whether the numbers of C-bursts per pair of developers and the numbers of E-bursts per pair of developers are correlated. We conducted correlation tests using the Pearson correlation coefficient. The results of these correlation tests are stated in Table 5.13. For all file-based, feature-based, and function-based analyses, the correlation coefficient is between 0.17 and 0.39, indicating that there is a positive correlation between the numbers of C-bursts per developer pair and the numbers of E-bursts per developer pair. Moreover, the correlation coefficient is higher for smaller time windows than for the 10-days time window, holding for all analyses.

In addition, we fitted a linear model for the pairwise number of E-bursts and corresponding number of C-bursts as stated in Equation 4.6. Considering the 5-days time window, we get a linear fit with parameters $\alpha = 0.00035$ and $\beta = -0.01$ for the file-based analysis, as depicted in Figure 5.6 (a), where we also denote the goodness of fit in terms of R-square, adjusted R-square, and Root Mean Square Error (RMSE). In Figure 5.6 (b) we illustrate the linear fit of the feature-based analyses, resulting in parameters $\alpha = 0.00082$ and $\beta = 0$. Last, Figure 5.6 (c) contains the illustration of the relationship between the number of pairwise C-bursts and corresponding E-bursts for the function-based approach. Here, we fit a linear model with parameters $\alpha = 0.00021$ and $\beta = 0$. The results of all analysis types are quite similar for time windows of 5 days or 10 days. We depict the regarding illustrations for all time windows in Figure A.1 to Figure A.3 in the Appendix.

Since in all cases more C-bursts are accompanied with more E-bursts, we accept *Hypothesis 4*.

Hypothesis 4: Accepted.

Furthermore, we analyze whether there is a timely correlation between C-bursts and E-bursts of each developer pair. Therefore, we generate C-curves and E-curves for each pair having more than five C-bursts and more than five E-bursts as stated

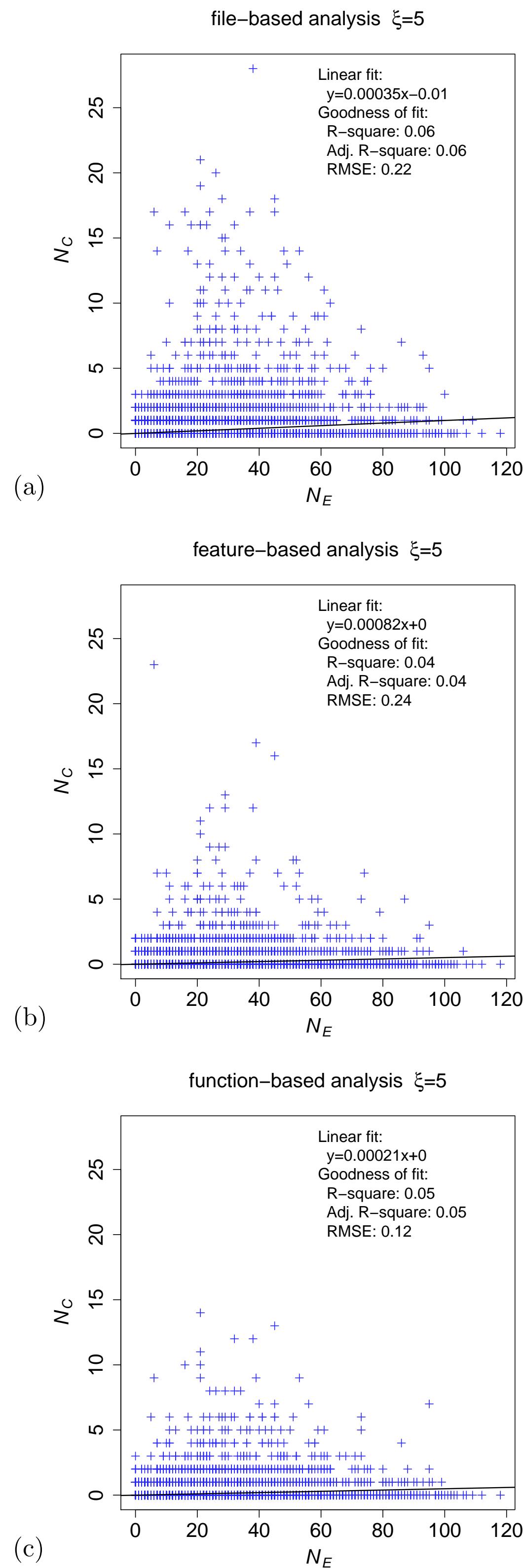

Figure 5.6: Relationship between the number of C-bursts (N_C) and the number of E-bursts (N_E) per developer pair for time window ξ=5 for the (a) file-based, (b) feature-based, and (c) function-based analysis. The black line depicts the corresponding fitted linear model.

Table 5.13: Correlation tests for the numbers of C-bursts and the numbers of E-bursts per pair of developers for the different analyses and time windows.

	ξ	correlation	t	df	p-value
	1	0.39	272.17	420,900	< 0.05
files	5	0.24	157.45	420,900	< 0.05
	10	0.18	121.55	420,900	< 0.05
	1	0.26	86.38	106,030	< 0.05
features	5	0.19	62.40	106,030	< 0.05
	10	0.17	55.35	106,030	< 0.05
	1	0.32	207.95	381,500	< 0.05
functions	5	0.22	137.81	381,500	< 0.05
	10	0.17	105.33	381,500	< 0.05

Table 5.14: Mann-Whitney U tests for the real and simulated correlation between C-curves and E-curves per developer pair for the different analyses and time windows. (Only developer pairs having more than five C-bursts and more than five E-bursts are considered.)

	ξ	real corr.	simulated corr.	W	p-value	Cliff's delta
	1	0.13	0.02	690	< 0.05	0.97
files	5	0.12	0.03	11,916	< 0.05	0.76
	10	0.12	0.04	30,182	< 0.05	0.68
	1	0.12	0.02	62	< 0.05	0.83
features	5	0.10	0.03	1,223	< 0.05	0.69
	10	0.11	0.04	2,383	< 0.05	0.52
	1	0.08	0.02	29	< 0.05	0.80
functions	5	0.11	0.03	1,129	< 0.05	0.61
	10	0.11	0.04	3,589	< 0.05	0.53

in Section 4.7. With these curves, we compute the Pearson correlation coefficient between the real C-curve and the real E-curve of each developer pair. To show that the correlation appears not by chance, we also compute the correlation coefficient between the real C-curves and the simulated E-curves for each pair of developers. Since only between 5 and 195 developer pairs (depending on time window and analysis type) have more than five C-bursts and more than five E-bursts, we cannot use a t-test here due to the few and not normally distributed data points. Instead, we conduct Mann-Whitney U tests to compare the correlation coefficients of both computations, given in Table 5.14. For all time windows and all case studies, the correlation coefficients using the real email time series are significantly higher than the correlation coefficients based on the simulated email time series. Moreover, the correlation coefficient is positive in all cases, saying that the C-curves and E-curves are positively correlated. Therefore, we accept *Hypothesis 5*.

Hypothesis 5: Accepted.

Table 5.15: T-tests with FDR correction for the comparison of the numbers of C-bursts per developer pair of file-based, feature-based, and function-based analyses for the different time windows.

	ξ	t	df	p-value
# C-bursts features > # C-bursts files	1	7.66	150,738.30	< 0.05
	5	12.54	151,332.80	< 0.05
	10	16.04	152,625.20	< 0.05
# C-bursts features > # C-bursts functions	1	13.44	123,277.00	< 0.05
	5	20.03	121,092.50	< 0.05
	10	24.38	123,037.30	< 0.05
# C-bursts files > # C-bursts functions	1	10.58	701,004.60	< 0.05
	5	13.41	673,415.00	< 0.05
	10	14.45	690,895.80	< 0.05

5.1.4 Comparison of File-based, Feature-based, and Function-based Synchronous Development

Hitherto, we have stated the results of file-based, feature-based, and function-based analyses. In the following, we compare the outcomes of the three different analyses.

First, we compare the number of C-bursts per developer pair of file-based, feature-based, and function-based approach. Therefore, we use a t-test with false-discovery-rate correction (FDR correction). In Table 5.15, we present the regarding outcomes: The number of C-bursts per pair of developers is significantly higher in the feature-based analysis than in the file-based analysis due to a p-value lower than 0.05. Moreover, the amount of C-bursts in the feature-based analysis is also higher than in the function-based analysis. Furthermore, the number of C-bursts per developer pair of the file-based analysis is significantly higher than the number of C-bursts per developer pair for the function-based analysis, too. Hence, we accept *Hypothesis 6.1*.

Hypothesis 6.1: Accepted.

Next, we compare the synchronicity degrees of the three approaches. Our hypothesis states that the synchronicity degree is higher for feature-based C-bursts than for file-based or function-based C-bursts. Thereto, we conduct a t-test with FDR correction, given in Table 5.16. Hence, the result reflects our hypothesis because the synchronicity degrees of feature-based C-bursts are significantly higher than the synchronicity degrees of file-based C-bursts which, in turn, are significantly higher than the synchroncity degrees of function-based synchronous development. This can also be seen in Figure 5.7. Consequently, we accept *Hypothesis 6.1a*.

Hypothesis 6.1a: Accepted.

Further above, we showed that the number of added lines per commit is significantly higher in synchronous development than in non-synchronous development. This holds for all analyses and all time windows except the 5-days and 10-days time window in the feature-based analysis. Here, we check whether the number of added lines in synchronous development is higher for synchronous development based on

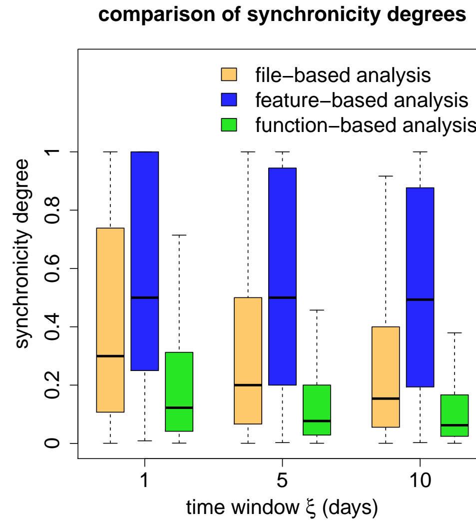

Figure 5.7: Box-and-whisker diagrams comparing the synchronicity degrees of the file-based, feature-based, and function-based analyses for time windows ξ of 1 day, 5 days, and 10 days. Outliers are omitted.

Table 5.16: T-tests with FDR correction for the comparison of the synchronicity degrees of file-based, feature-based, and function-based analyses for the different time windows.

	ξ	t	df	p-value
sync. degrees features > sync. degrees files	1	15.43	2,986.94	< 0.05
	5	33.51	8,478.26	< 0.05
	10	47.49	13,288.88	< 0.05
sync. degrees features > sync. degrees functions	1	33.10	2,782.62	< 0.05
	5	57.96	7,828.92	< 0.05
	10	77.93	12,031.55	< 0.05
sync. degrees files > sync. degrees functions	1	22.32	4,343.29	< 0.05
	5	31.83	11,763.75	< 0.05
	10	40.33	19,893.82	< 0.05

features than on files or functions. We again conducted a t-test with FDR correction, depicted in Table 5.17. Since the p-value in all tests is higher than 0.05, we have to reject our hypothesis. Moreover, we already have rejected Hypothesis 2 regarding the code-growth and code-effort metrics, so we have no significant results to compare here and therefore we also reject *Hypothesis 6.2*.

Hypothesis 6.2: Rejected.

Next, we compare the number of deleted lines per commit of the different results. Due to the rejection of Hypothesis $3_<$, we also reject *Hypothesis* $6.3_<$.

Hypothesis $6.3_<$: Rejected.

Table 5.17: T-tests with FDR correction for the comparison of the number of added lines in synchronous commits of file-based, feature-based, and function-based analyses for the different time windows.

	ξ	t	df	p-value
sync. L_{add} features > sync. L_{add} files	1	0.23	2,171.05	1.00
	5	1.28	4,099.42	0.30
	10	1.26	4,751.41	1.00
sync. L_{add} features > sync. L_{add} functions	1	-2.45	2,994.64	1.00
	5	-2.70	7,559.33	1.00
	10	-3.09	10,314.38	0.31
sync. L_{add} files > sync. L_{add} functions	1	-4.44	3,552.40	1.00
	5	-4.21	4,993.03	1.00
	10	-5.03	7,502.89	1.00

Table 5.18: T-tests with FDR correction for the comparison of the number of deleted lines in synchronous commits of file-based, feature-based, and function-based analyses for the different time windows (estimating that the number of deleted lines is higher in feature-based analysis than in the file-based or function-based analysis).

	ξ	t	df	p-value
sync. L_{delete} features > sync. L_{delete} files	1	0.80	2,273.38	0.41
	5	2.29	4,014.71	0.03
	10	2.69	4,548.80	0.01
sync. L_{delete} features > sync. L_{delete} functions	1	0.61	3,558.29	0.41
	5	1.71	6,204.53	0.07
	10	2.04	6,453.89	0.03
sync. L_{delete} files > sync. L_{delete} functions	1	-0.08	3,172.81	0.53
	5	-0.42	6,418.75	0.66
	10	-0.71	9,765.92	0.76

We already showed that the number of deleted lines is significantly higher in synchronous development than in non-synchronous development. Here, we compare the number of deleted lines per synchronous commits of file-based, feature-based, and function-based analyses. The result of the regarding t-tests with FDR-correction is depicted in Table 5.18. Only for the 10-days time window, the number of deleted lines per synchronous commits in the feature-based analysis is significantly larger than the number of deleted lines per synchronous commits of the file-based and function-based analysis. Since this does not hold for the 1-day and 5-days time windows, and also not for the comparison of file-based and function-based analysis, we reject *Hypothesis* $6.3_{>}$.

Hypothesis $6.3_{>}$: Rejected.

Table 5.19: Correlation coefficients for the real C-curves per developer pair and the real E-curves per developer pair (only pairs with more than five C-bursts and more than five E-bursts are considered) for the different analyses and time windows. Cliff's delta is the corresponding effect size to compare real and simulated correlation. For additional data, see Table 5.14.

	ξ	real correlation	Cliff's delta
files	1	0.13	0.97
	5	0.12	0.76
	10	0.12	0.68
features	1	0.12	0.83
	5	0.10	0.69
	10	0.11	0.52
functions	1	0.08	0.80
	5	0.11	0.61
	10	0.11	0.53

To compare the correlation between the number of C-bursts per developer pair and the number of E-bursts per developer pair of file-based, feature-based, and function-based analyses, we consider the slopes of the linear models depicted in Figure 5.6. According to *Hypothesis 6.4*, the slope of the fitted line of the feature-based analysis (0.00082) is higher than the slope of the fitted line of the file-based analysis (0.00035), which, in turn, is higher than the slope of the fitted line of the function-based analysis (0.00021). Hence, the effect that more C-bursts are accompanied with more E-bursts is stronger on feature-based development than on file-based and function-based development. Thus, we accept *Hypothesis 6.4*.

Hypothesis 6.4: Accepted.

Finally, we compare the correlation between the curves of C-bursts and E-bursts of file-based, feature-based, and function-based analyses. Considering the effect-size, Cliff's delta, as given in Table 5.19, we see that for all considered time windows the effect size between real and simulated correlation of C-curves and E-curves is higher for the file-based analysis than for the feature-based analysis. Also the correlation coefficient itself is larger for file-based synchronous development than for feature-based synchronous development. Moreover, comparing feature-based and function-based correlation coefficients, we notice that for the 1-day and 5-days time windows the effect size as well as the correlation coefficient is higher for the feature-based analysis than for the function-based analysis. However, this holds not for the 10-days time window of feature-based and functions-based analyses. Since the correlation of the feature-based C-curves and E-curves has not the strongest effect compared to the file-based and function-based curves, we reject *Hypothesis 6.5*.

Hypothesis 6.5: Rejected.

5.2 Discussion

After presenting the results of our study in the previous section, we now discuss all the results regarding C-bursts, E-bursts, and the comparison of file-based, feature-based, and function-based analyses.

5.2.1 C-Burst-related Hypotheses

To start with, we remind our C-burst-related hypotheses:

Hypothesis	Status
Hypothesis 1. *Synchronous development occurs significantly more frequently in OSS projects than by chance.*	**[Accepted]**
Hypothesis 1a. *The synchronicity degree in OSS projects is significantly higher than by chance.*	**[Accepted]**
Hypothesis 2. *OSS projects grow more in change of LOC per commit if developers participate in synchronous development. In addition, in synchronous development the effort in deleted plus added lines per commit is lower than in non-synchronous development.*	**[Rejected]**
Hypothesis $3_<$. *The number of deleted lines per commit is smaller within co-commit bursts than outside co-commit bursts.*	**[Rejected]**
Hypothesis $3_>$. *The number of deleted lines per commit is larger within co-commit bursts than outside co-commit bursts.*	**[Accepted]**

For all time windows ξ of 1 day, 5 days, and 10 days, the synchronous development occurs more than random within all file-based, feature-based, and function-based analyses, which is the initial position for our whole study. Knowing that synchronous development in QEMU appears not by chance, we can consider the synchronicity degree of the identified C-bursts, which is also higher than for randomly generated C-bursts. Altogether, these results show us that considering the concept of synchronous development in open-source software is well-founded.

Next, we consider the used code-growth metrics. Developer collaboration (regardless of whether it is based on files, features, or functions) is accompanied with a higher productivity in terms of added lines per commit. However, as opposed to the outcomes of the study of Xuan and Filkov [XF14], the number of deleted lines per commit is also higher in synchronous development than in non-synchronous development. A reason, therefore, might be that the expectation that collaborating developers delete less code than non-collaborating developers is wrong. For example, if developers refactor large parts of the source code, they probably delete a huge number of lines of code. Also, changing a huge amount of lines results in a huge amount of deleted lines. Consequently, expecting that the number of deleted lines per commit is lower in synchronous development than in non-synchronous development is not reasonable. In contrast, the higher number of deleted lines in synchronous development compared to non-synchronous development reveals that collaborative development is more productive in terms of deletions.

Combining the results of comparing the numbers of added lines and deleted lines in synchronous and non-synchronous development, we can conclude that in synchronous development more lines are changed (regardless of whether they are added

or deleted). Nevertheless, relating the number of added and deleted lines to each other to draw further conclusions on the productivity of developers is not meaningful here. Hence, the results of the comparison of the code-growth and code-effort metrics in synchronous and non-synchronous development are not convincing.

5.2.2 E-Burst-related Hypotheses

Before discussing the E-burst-related results, we also remind the corresponding hypotheses:

Hypothesis 4. *More C-bursts are accompanied with more E-bursts.*	[**Accepted**]
Hypothesis 5. *E-bursts and synchronous development are positively coupled.*	[**Accepted**]

According to our hypotheses, the number of C-bursts and the number of E-bursts per pair of developers correlate positively. However, the slopes of the fitted lines of the linear models describing the relationship between the numbers of C-bursts and E-bursts are comparably small. This holds for all time windows and all analysis types. Thus, on average, there is one C-burst for every 1,220 E-bursts of one developer pair in the feature-based analysis. Moreover, in the file-based analysis, there, roughly, is one C-burst for every 3,000 E-bursts, and in the function-based analysis we have one C-burst for every 5,000 E-bursts. For time windows of 1 day or 10 days the corresponding difference between the number of C-bursts and E-bursts per pair of developers is even worse. In addition to that, one should keep in mind that the overall number of E-bursts is more than 500 times higher than the overall number of C-bursts (just focusing on the 5-days time window). As illustrated in Figure 5.6, most of the developer pairs have 0 to 2 C-bursts, but up to 100 E-bursts, which biases our results. Therefore, the fitted linear model is much less expressive than expected. However, it is intelligible that developers more often send emails to the mailing list than they contribute to the source code because they also can comment on the commits or bug reports of others without contributing to the source code itself in a near time period.

Furthermore, there is a positive timely correlation between C-bursts and E-bursts of developer pairs for all file-based, feature-based, and function-based analyses. The computed correlation coefficient has a comparably small value between 0.08 and 0.12. One reason for this weak positive correlation is that the amount of E-bursts per pair of developers intensively exceeds the amount of C-bursts per pair of developers. Another reason, therefore, is that E-bursts and C-bursts do not occur at exactly the same time, but delayed. According to the commit policy of QEMU, developers are requested to send their commits to the corresponding mailing list. Then it is possible that a few weeks or even months elapse until the submitted commit is discussed. The other way round, it is also possible that a certain feature or bug will be discussed on the mailing list first, but the corresponding implementation or bugfix will be sent to the mailing list much later and then will potentially be pushed to the repository without any discussion on the mailing list.

5.2.3 Comparative Hypotheses

Finally, we discuss the comparison of all the results of the three different artifact-based analyses, namely file-based, feature-based, and function-based analyses. Right at the start, we remind the regarding hypotheses:

Hypothesis 6.1 (refers to Hypothesis 1)**.** *Synchronous development on features occurs significantly more frequently than synchronous development on files and synchronous development on functions.* [**Accepted**]

Hypothesis 6.1a (refers to Hypothesis 1a)**.** *The synchronicity degree of synchronous development on features is significantly higher than the synchronicity degree of synchronous development on files or on functions.* [**Accepted**]

Hypothesis 6.2 (refers to Hypothesis 2)**.** *The code growth per commit is significantly higher in synchronous development on features than in synchronous development on files or on functions. In addition, the code effort in terms of added and deleted LOC is significantly smaller on features than on files or functions.* [**Rejected**]

Hypothesis $6.3_{<}$ (refers to Hypothesis $3_{<}$)**.** *The number of deleted lines per commit within co-commit bursts on features is significantly smaller than within co-commit bursts on files or on functions.* [**Rejected**]

Hypothesis $6.3_{>}$ (refers to Hypothesis $3_{>}$)**.** *The number of deleted lines per commit within co-commit bursts on features is significantly greater than within co-commit bursts on files or on functions.* [**Rejected**]

Hypothesis 6.4 (refers to Hypothesis 4)**.** *The effect ”More co-commit bursts are accompanied with more email bursts” is stronger on feature-based synchronous development than on file-based or function-based synchronous development.* [**Accepted**]

Hypothesis 6.5 (refers to Hypothesis 5)**.** *Email bursts and synchronous development on features are stronger positively coupled than email bursts and file-based or function-based synchronous development.* [**Rejected**]

The number of feature-based C-bursts is significantly higher than the number of file-based or function-based C-bursts, which supports our assumption that collaboration on features is more common than file-based or function-based collaboration of developers. Functions are also semantic units, but mostly very small compared to features. In addition, functions usually do not encompass a certain feature in the application domain, but only a small set of statements capturing a certain task. Therefore, it is reasonable that feature-based collaboration occurs more frequently. Moreover, file-based collaboration appears not that often as feature-based collaboration since files are not necessarily semantic units and, therefore, file-based collaboration might not be that intended as feature-based collaboration. These ideas are

not only supported by the frequency of synchronous development, but also by the synchronicity of the regarding synchronous development.

We also aimed to compare the productivity, effectiveness, and programming effort of synchronous development on files, features, or functions. However, we did not get significant outcomes which allow us to draw conclusions from the comparison. As already stated in the previous subsection, judging the code growth and code effort in terms of added or deleted lines of code is not that meaningful.

Lastly, we compare the correlation between C-bursts and E-bursts of feature-based, file-based, and function-based analyses. In file-based collaborations, more E-bursts are needed to discuss code changes than in feature-based collaborations, because the average number of E-bursts per C-burst is lower in feature-based collaboration than in file-based collaboration. Since feature-based collaboration is collaboration on semantic units, feature-based code changes of two developers do not need as much coordination than in file-based collaboration. For function-based collaboration, more coordination in terms of email communication is necessary than on feature-based collaboration since functions may be used in several contexts, e.g., in several files or by several features.

Nevertheless, considering the timely correlation of C-bursts and E-bursts, file-based collaboration on commit level is slightly stronger correlated to email communication than feature-based collaboration. One reason therefore might be that file-based collaboration needs more coordination that feature-based collaboration, as stated above. Another reason therefore can be that communication about files is more selective and isolated, so E-bursts are related to certain C-bursts, whereas communication about features is more or less spread on larger period in time. Thus feature-based communication is not directly related to certain commits due to features being logic abstraction from the application domain. The same reasons also hold for the comparison of feature-based and function-based correlation between C-bursts and E-bursts, where functions are sets of statements which are discussed more isolated and based on certain commits. Nonetheless, the differences of the correlations of the three artifact-based analyses are biased by not considering the content of emails and not considering time shifts between commits and emails, as already stated in the previous section.

5.3 Threats to Validity

After the above discussion of our results, we denote the threats to the validity of our study.

One threat to validity is that our null models consists of few simulations, namely 50 random simulations of commit time series and only 1 random simulation of the email time series for our case study. Another threat to validity is that only less than 0.2 percent of the considered developer pairs of QEMU have more than five C-bursts and more than five E-bursts, which holds for all of our analyses and all considered time windows. However, since we analyze a huge amount of developer pairs, we still find significant differences between the different analyses.

Considering the C-burst analysis, our results regarding the differences in terms of lines of code (LOC) between synchronous and non-synchronous development may

be biased since therefore we treat commits as synchronous even if the corresponding synchronicity degrees are close to zero. The reason for that is that we replicated the study of Xuan and Filkov [XF14], which did not measure the degree of synchronicity. Moreover, we cannot easily determine the actual number of deleted lines per artifact. Since we know the correct number of deleted lines per commit, we use a heuristic to guess the number of deleted lines of unconsidered artifacts, assuming that the number of deleted lines is half of the regarding number of added lines per artifact. This primarily influences the feature-based analysis, in which we completely disregard contributions to the base feature why we have to determine the number of deleted lines of the base feature to subtract them from the overall number of deleted lines of a commit. In addition, the incorrect determination of the number of deleted lines also affects all the code-growth and code-effort metrics. Aside from that, the numbers of added and deleted lines themselves are biased: If developers just change lines without adding additional lines or removing lines, each changed line is considered as both deleted and added. So, changing a huge amount of lines also results in a huge amount of added lines and in a huge amount of deleted lines. On top of this, the extracted number of added and deleted lines per commit also considers blank lines, which cannot be used to value the productivity of developers. Besides, using code-growth metrics and differences in the number of added or deleted lines of code to quantify the productivity or implementation effort might be not a valid approach since these measurements do not consider the code complexity of a line of code and also disregard the time which is needed for the regarding implementation. However, the number of added lines of code can be easily extracted from the source code repository, whereas we do not have information on the regarding implementation time, for instance. In general, using code-growth and code-effort metrics to assess the productivity of a developer is not valid since they also do not measure the quality of the produced source code.

Furthermore, there are also threats to the validity of the E-burst analysis. Since developers can use different email addresses with different names, we cannot correctly disambiguate them. Hence, Codeface uses a heuristic, which we have introduced and discussed in Section 4.8. So, it is possible that we consider two different authors to be identical due to having the same name. Opposed to that, there also might be authors which use different email addresses and names which we then consider to be distinct authors, whereas it is one and the same author.

Another thing to keep in mind is that we do not check whether C-bursts and E-bursts are semantically coupled. So, it is possible that developers write a lot on the mailing list, but on different topics than they contribute to the source code. In addition, we do not consider the topic or thread information of the sent emails for the E-burst identification, which can result in E-bursts consisting of completely unrelated emails. Such E-bursts will potentially not stand in any correlation to C-bursts. Moreover, developers may have different preferences how quickly or how often they check their emails and send replies, which also can bias our study regarding the relation of the number of C-bursts and the number of E-bursts per pair of developers. Furthermore, we did not include the differences in the lengths of bursts in terms of days in our corresponding analyses. Directly comparing C-curves and E-curves of developer pairs can also distort the results of our study since commit activities and email

activities may not take place at exactly the same time, but slightly shifted, which we do not cover within our study.

In addition to that, in GIT repositories, every commit has an author and a commiter which can possibly differ. Moreover, every commit is a assigned an author date, which is the point in time at which the author had finished the commit, as well as a commit date, which is the date at which the commit was committed to the project. In our study we use the author of a commit, together with the regarding author date, since that is the person that usually is the original creator of an commit. Nevertheless, a commit can get public much later than it was created and therefore the communication on the mailing list can also take place around the commit date instead of the author date, which is a threat to validity to our study. For the sake of completeness, we note that Xuan and Filkov did not specify which date they have used for their study.

While comparing the results of our analyses for file-based, feature-based, and function-based synchronous development, we disregard that we analyze different parts of the source code within the different studies. So, for the feature-based analysis, we omit the source code belonging to the base feature, which reduces the number of considered developers by half compared to the file-based analysis, for instance. Nonetheless, since we only want to analyze the contributions to actual features, it is intended to disregard the developers which do not contribute to actual features in the feature-based analysis.

Aside from that, in our study we just focus on the developer coordination which takes place by writing emails to the developer mailing list, according to the original study. However, developers which personally know each other may coordinate their synchronous development by private emails or verbal communication. Moreover, other tools can be used for coordination, such as instant messengers or social networks, which we do not consider in our study.

For time reasons, we only analyze the synchronous development of one case study. Thus, we cannot generalize our outcomes since for other case studies we might get different results.

6. Conclusion

In this chapter, we summarize the outcomes of our study. Thereafter, we denote possible enhancements and future work regarding this study.

6.1 Summary

In this thesis, we analyzed the collaboration and coordination of developers in the open-source software project QEMU. Therefore, we replicated an empirical study of Xuan and Filkov [XF14] on co-commit bursts and email bursts in open-source software, which we enhanced by analyzing and comparing file-based, feature-based, and function-based collaboration of developers. We analyzed more than seven years of the development of the case study QEMU and found that synchronous development of pairs of developers is more than a random phenomenon, regardless of considering the different time windows of 1 day, 5 days or 10 days between developers' activities. We also quantified the synchronicity of synchronous development which is also significantly higher than on random simulations. In addition, we identified more co-commit bursts based an features than based on files or functions. Also the synchronicity degree of feature-based collaboration is significantly higher than on file-based or function-based collaboration. As result, developers collaborate more frequently on semantic units like features since this is the level of abstraction from the source code developers think in.

Moreover, we investigated whether the code growth and code effort of synchronous and non-synchronous development significantly differ. Here, we found that the number of added lines per commit is larger in synchronous development than in non-synchronous development. In contrast to the study of Xuan and Filkov, also the number of deleted lines is higher in synchronous development than in non-synchronous development. In addition, the code growth is not significantly higher in time intervals of synchronous development than in time intervals of non-synchronous development. Additionally, the code effort in terms of lines of code is not smaller in synchronous development than in non-synchronous development. Nevertheless, we cannot draw any conclusions from these results since only measuring the productivity and implementation effort in terms of lines of code is not sufficient thereto.

Furthermore, we also compared the relation of co-commit bursts and email bursts of pairs of developers. We identified much more email bursts per pairs of developers than co-commit bursts of developers. On average, in feature-based collaboration less communication in terms of the number of email bursts takes place to coordinate synchronous commits than in file-based or function-based collaboration. Analyzing the timely correlation of co-commit bursts and email bursts, we identified a weak positive correlation between co-commit bursts and email bursts of pairs of developers. Notwithstanding that this correlation is weak, it is higher for file-based collaboration of developers than for feature-based collaboration of developers.

6.2 Future Work

For the future, we plan to extend our study in several ways. First, we will generate more simulations of the commit time series and also more simulations of the email time series than used in this study to get a more convincing null model. In addition, the null model for the commit time series can be extended by randomizing a second dimension: Besides randomizing the time differences between subsequent commits of a developer, the changed artifacts of commits can also be randomly permuted.

Furthermore, we aim to find a more suitable heuristic to determine the number of deleted lines per source-code artifact. One possible approach hereto could be to compute the ratio between the number of added and deleted lines per commit and than transfer this ratio to the number of changed lines of the artifacts within the commits.

Moreover, we aim to develop a more appropriate approach of how to classify synchronous and non-synchronous changes. Here, we aim to differentiate between synchronous and non-synchronous parts of commits instead of treating the whole commit as synchronous. So, only the artifacts of commits which are synchronously changed by both developers will be considered to be synchronous, whereas the rest of the commit will be considered to be non-synchronous. This could be a more exact way of comparing code growth of synchronous and non-synchronous development.

Since we developed a metric for the synchronicity degree of synchronous commits, we also plan to design a synchronicity-degree metric for synchronous emails. Here, we think of several different approaches: We could consider the thread id of each email to only group emails which belong the same email thread. Another solution could be to group semantically related emails based on the subjects of the emails or based on email content, e.g., by using text mining methods. Further, an email can also have more than one receiver. So, besides the mailing list, emails of developers can also be directly addressed to other developers. We can also use this information to determine directly collaborative developers, for instance. We also reflect about identifying the collaboration within groups of developers instead of pairs of developers to get a more accurate model for communication activities.

Due to existing time shifts between commit activities and email activities, we aim to find an appropriate solution to handle these shifts within our investigation of the correlation between C-curves and E-curves. Therefore, Xuan and Filkov [XF14] use an adapted way of Gaussian smoothing to deal with such time differences between

commits and emails. However, we also try to find other methods to consider time shifts within the comparison of C-curves and E-curves to get more insight into what their smoothing actually effects.

Another idea is to study bug reports corresponding to the analyzed case-study and draw further conclusions on the role of synchronous development. For example, we can investigate the amount or appearance of bugs or bug fixes within synchronous or non-synchronous development to find out more about the advances or drawbacks of synchronous development in open-source systems.

Finally, we contemplate analyzing the synchronous development and the corresponding coordination bursts for a larger number of open-source case studies such as BusyBox, OpenSSL, or the Linux kernel, having different sizes in terms of the number of developers, coming from various domains, and having different contribution policies.

A. Appendix

In the Appendix, we provide additional illustrations to several outcomes of our study.

Relationship between number of C-bursts and number of E-bursts per pair of developers

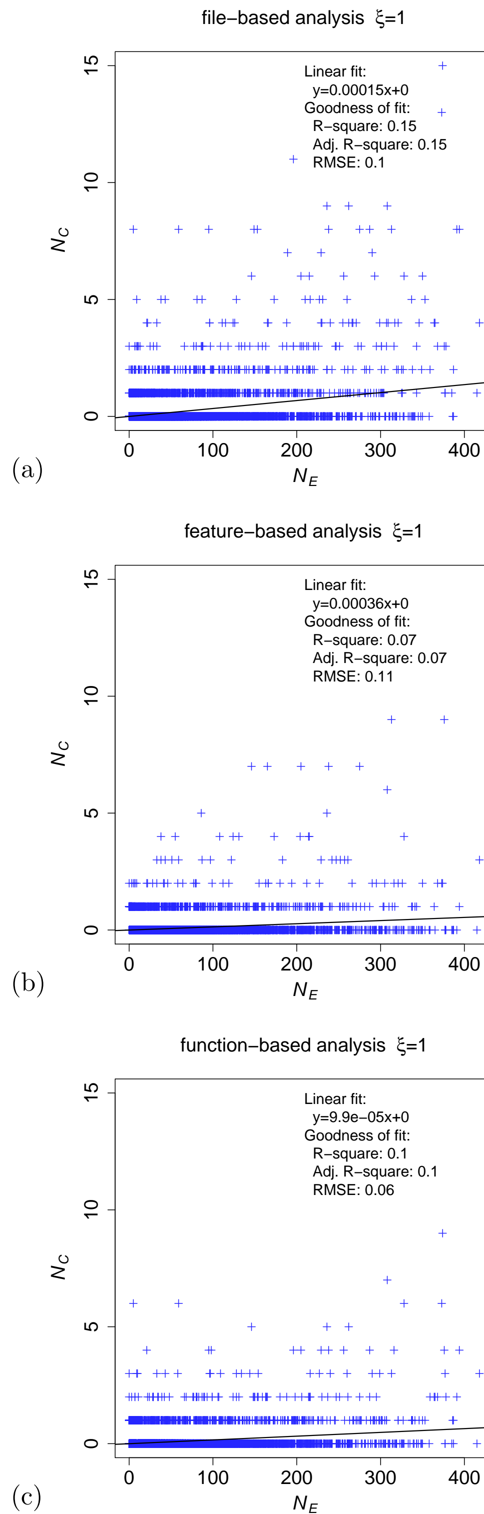

Figure A.1: Relationship between the number of C-bursts (N_C) and the number of E-bursts (N_E) per developer pair for time window ξ=1 for the (a) file-based, (b) feature-based, and (c) function-based analysis. The black line depicts the corresponding fitted linear model.

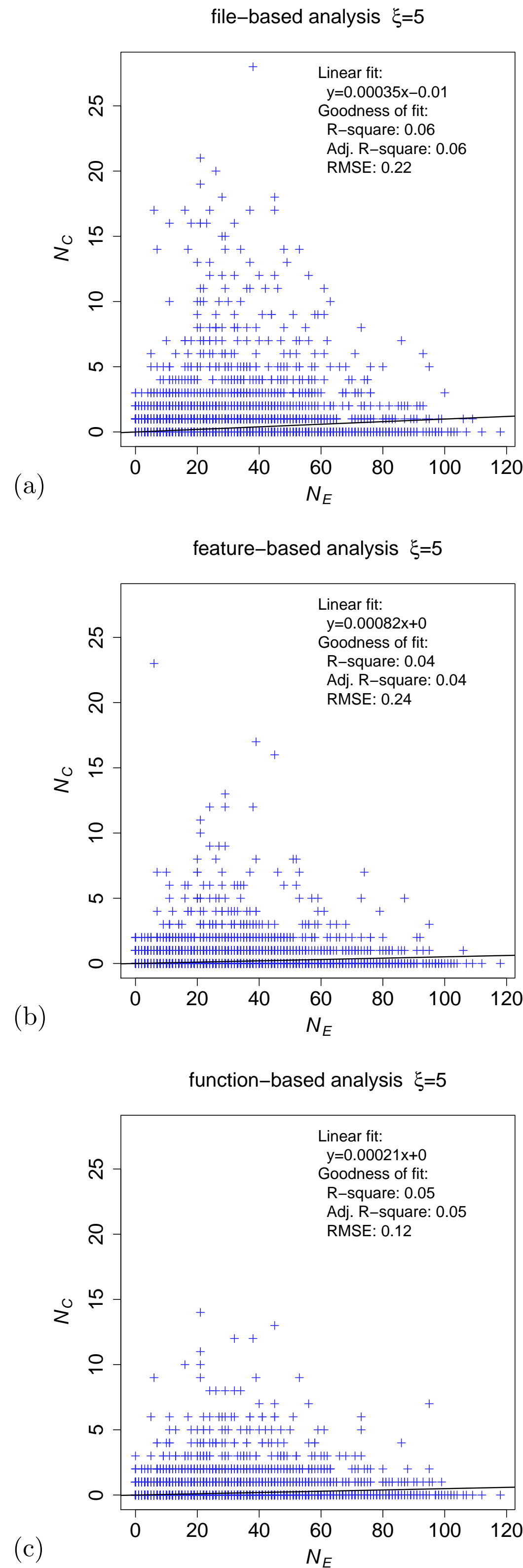

Figure A.2: Relationship between the number of C-bursts (N_C) and the number of E-bursts (N_E) per developer pair for time window ξ=5 for the (a) file-based, (b) feature-based, and (c) function-based analysis. The black line depicts the corresponding fitted linear model.

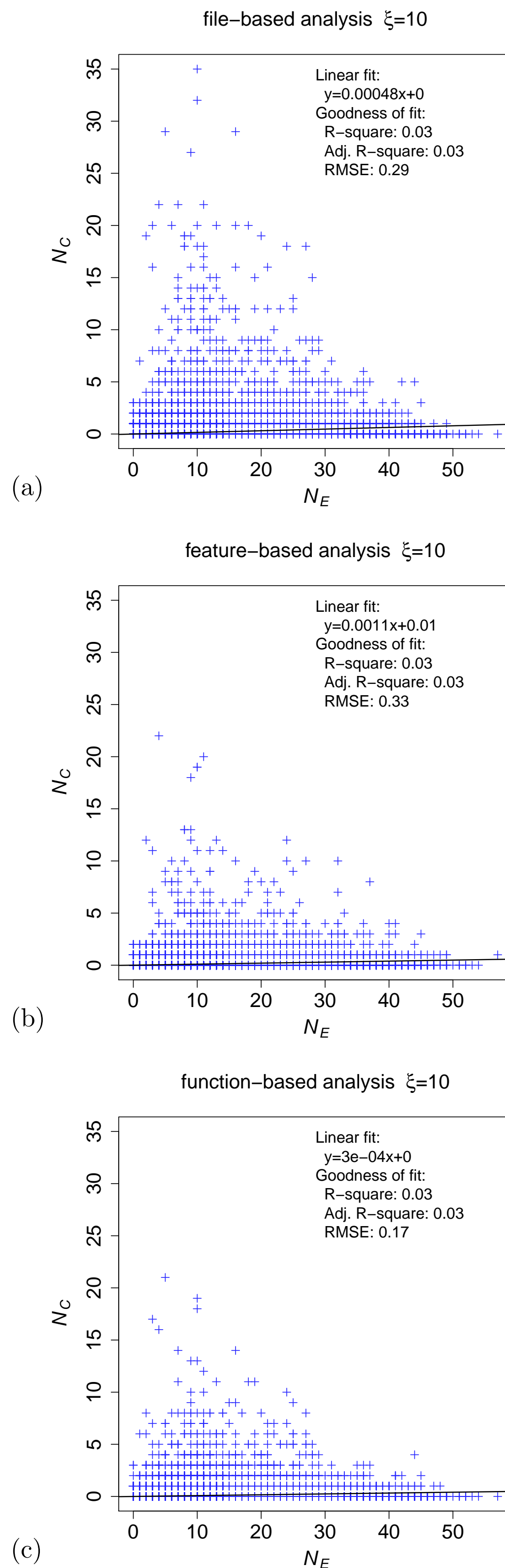

Figure A.3: Relationship between the number of C-bursts (N_C) and the number of E-bursts (N_E) per developer pair for time window ξ=10 for the (a) file-based, (b) feature-based, and (c) function-based analysis.

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer-Verlag, 2013. (cited on Page 6)

[BDL10] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. Are Popular Classes More Defect Prone? In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, FASE, pages 59–73. Springer, 2010. (cited on Page 4)

[BGD+06] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining Email Social Networks. In *Proceedings of the International Workshop on Mining Software Repositories*, MSR, pages 137–143. ACM Press, 2006. (cited on Page 3 and 21)

[CHC08] Marcelo Cataldo, James D. Herbsleb, and Kathleen M. Carley. Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 2–11. ACM Press, 2008. (cited on Page 1)

[Cli96] Norman Cliff. *Ordinal Methods for Behavioral Data Analysis.* Erlbaum, 1996. (cited on Page 20)

[CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002. (cited on Page 6)

[Coh88] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Routledge, 1988. (cited on Page 17)

[CP91] Samprit Chatterjee and Bertram Price. *Regression Analysis by Example.* John Wiley & Sons, 1991. (cited on Page 19)

[DP03] Dirk Draheim and Lukasz Pekacki. Process-Centric Analytical Processing of Version Control Data. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE, pages 131–136. IEEE Computer Society, 2003. (cited on Page 3)

[GBS01] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working*

IEEE/IFIP Conference on Software Architecture, WICSA, pages 45–54. IEEE Computer Society, 2001. (cited on Page 6)

[HG99] James D. Herbsleb and Rebecca E. Grinter. Architectures, Coordination, and Distance: Conway's Law and Beyond. *IEEE Software*, 16(5):63–70, 1999. (cited on Page 4)

[JMA+15] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. From Developer Networks to Verified Communities: A Fine-grained Approach. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, ICSE, pages 563–573. IEEE Computer Society, 2015. (cited on Page 4 and 12)

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988. (cited on Page 5 and 6)

[LAL+10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, ICSE, pages 105–114. ACM Press, 2010. (cited on Page 12)

[MFT02] Gregory Madey, Vincent Freeh, and Renee Tynan. The Open Source Software Development Phenomenon: An Analysis Based on Social Network Theory. In *Proceedings of the Americas Conference on Information Systems*, AMCIS, pages 1806–1813. Association for Information Systems, 2002. (cited on Page 1)

[OSdO+12] Gustavo A. Oliva, Francisco W. Santana, Kleverton C. M. de Oliveira, Cleidson R. B. de Souza, and Marco A. Gerosa. Characterizing Key Developers: A Case Study With Apache Ant. In *Proceedings of the International Conference on Collaboration and Technology*, CRIWG, pages 97–112. Springer, 2012. (cited on Page 13 and 21)

[Sin10] Param Vir Singh. The Small-World Effect: The Influence of Macro-Level Properties of Developer Collaboration Networks on Open-Source Project Success. *ACM Transactions on Software Engineering and Methodology*, 20(2):6:1–6:27, 2010. (cited on Page 3)

[Som10] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, 9th edition, 2010. (cited on Page 3)

[TMTB10] Sergio L. Toral, M. Rocío Martínez-Torres, and Federico Barrero. Analysis of Virtual Communities Supporting OSS Projects Using Social Network Analysis. *Information and Software Technology*, 52(3):296–303, 2010. (cited on Page 4)

[WGS03] James Wu, T.C.N. Graham, and Paul W. Smith. A Study of Collaboration in Software Design. In *Proceedings of the International Symposium on Empirical Software Engineering*, ISESE, pages 304–313. IEEE Computer Society, 2003. (cited on Page 1)

[WSDN09] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. Predicting Build Failures Using Social Network Analysis on Developer Communication. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, ICSE, pages 1–11. IEEE Computer Society, 2009. (cited on Page 3)

[WTdSS+16] Igor Scaliante Wiese, José Teodoro da Silva, Igor Steinmacher, Christoph Treude, and Marco Aurélio Gerosa. Who is Who in the Mailing List? Comparing Six Disambiguation Heuristics to Identify Multiple Addresses of a Participant. In *Proceedings of the International Conference on Software Maintenance and Evolution*, ICSME. IEEE Computer Society, 2016. Accepted. (cited on Page 13 and 21)

[XF14] Qi Xuan and Vladimir Filkov. Building It Together: Synchronous Development in OSS. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, ICSE, pages 222–233. ACM Press, 2014. (cited on Page iii, 1, 3, 4, 7, 8, 13, 17, 18, 19, 20, 43, 47, 49, and 50)

[XGDF12] Qi Xuan, Mohammed Gharehyazie, Premkumar T. Devanbu, and Vladimir Filkov. Measuring the Effect of Social Communications on Individual Working Rhythms: A Case Study of Open Source Software. In *Proceedings of the International Conference on Social Informatics*, SocInfo, pages 78–85. IEEE Computer Society, 2012. (cited on Page 9 and 17)

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich diese Masterarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Thomas Bock

Passau, den 28. September 2016