Automated Software Engineering manuscript No.
(will be inserted by the editor)

Synchronous Development in Open-Source Projects: A Higher-Level Perspective

Thomas Bock · Claus Hunsen · Mitchell Joblin · Sven Apel

Received: date / Accepted: date

Abstract Mailing lists are a major communication channel for supporting developer coordination in open-source software projects. In a recent study, researchers explored temporal relationships (e.g., synchronization) between developer activities on source code and on the mailing list, relying on simple heuristics of developer collaboration (e.g., co-editing files) and developer communication (e.g., sending e-mails to the mailing list). We propose two methods for studying synchronization between collaboration and communication activities from a higher-level perspective, which captures the complex activities and views of developers more precisely than the rather technical perspective of previous work. On the one hand, we explore developer collaboration at the level of features (not files), which are higher-level concepts of the domain and not mere technical artifacts. On the other hand, we lift the view of developer communication from a message-based model, which treats each e-mail individually, to a conversation-based model, which is semantically richer due to grouping e-mails that represent conceptually related discussions. By means of an empirical study, we investigate whether the different abstraction levels affect the observed relationship between commit activity and e-mail communication using state-of-the-art time-series analysis. For this purpose, we analyze

Thomas Bock
Saarland University, Saarland Informatics Campus, Germany
E-mail: bockthom@cs.uni-saarland.de

Claus Hunsen
University of Passau, Germany
E-mail: hunsen@fim.uni-passau.de

Mitchell Joblin
Siemens AG / Saarland University, Saarland Informatics Campus, Germany
E-mail: joblin@cs.uni-saarland.de

Sven Apel
Saarland University, Saarland Informatics Campus, Germany
E-mail: apel@cs.uni-saarland.de

a combined history of 40 years of data for three highly active and widely deployed open-source projects: QEMU, BusyBox, and OpenSSL. Overall, we found evidence that a higher-level view on the coordination of developers leads to identifying a stronger statistical dependence between the technical activities of developers than a less abstract and rather technical view.

Keywords open-source software projects · developer coordination · synchronous development

1 Introduction

The success of large software projects relies on the extent to which developers coordinate their efforts. This is especially true for large-scale open-source software (OSS) projects, to which often numerous globally distributed and independent developers contribute (Herbsleb, 2007). When multiple developers contribute to interrelated source-code fragments, changes that lack coordination often introduce unintentional side effects. Developers must coordinate their interdependent activities to prevent conflicting changes, to avoid bugs, or to keep the code simple and maintainable (Cataldo et al., 2008, 2009; Bird, 2011; Kwan et al., 2011). In large-scale projects, developer coordination is absolutely crucial to ensuring high-quality software and to supporting high developer productivity (Cataldo and Herbsleb, 2013).

Since software developers in OSS projects are often globally distributed, they mostly communicate via the Internet to discuss software issues or enhancements or to review code changes (Wu et al., 2003). Mailing lists, issue trackers, and instant messengers are the most commonly used communication channels for coordination of developers in OSS projects (Storey et al., 2017). We dedicate attention to analyzing developer communication on mailing lists because they are historically rich and well-established sources of data for discussions regarding software architecture and reviewing of code changes (Rigby et al., 2008; Ramsauer et al., 2019). In a recent study on 37 OSS projects, Mannan et al. (2020) have shown that about 89% of such discussions take place on the project's mailing list. Mailing lists are a greater source of longitudinal data than more recently introduced social-coding platforms (e.g., GitHub), because their usage dates back more than 10 years (see Table 2). Mailing lists are also used to discuss the outcomes of developer conferences and similar events where complex issues and long-term plans for feature development are discussed. Even developers in OSS projects who work for corporations may use mailing-list discussions to communicate their intentions to others as public communication is one of the basic concepts in OSS projects (Riehle, 2015).

To obtain deeper insights into the fundamentals of developer coordination and the role communication plays in OSS projects, we investigate the relationship between co-editing activities on source-code artifacts and communication activities on the developer mailing list. For this purpose, we replicate and extend an empirical study of Xuan and Filkov (2014) on synchronous development in OSS projects, which we will refer to as the *original study*. The authors

of the original study identified pairs of developers co-editing files to explore the relationship between developer productivity and communication activities. Their major finding was that time intervals rich in co-editing activities are correlated with time intervals rich in e-mail activities and, more importantly, that during these synchronized periods developer productivity was higher.

The original study already provided interesting and useful insights on developer collaboration and developer communication. Nonetheless, they relied on a rather technical, low-level view. Regarding developer collaboration, they limited their perspective to co-edits of individual files. There is reason to believe, though, that this perspective only covers technical edits to files which are likely to be a noisy indication of the content-wise relationship between the edits. Developers co-editing a file may not change any interrelated source code because a file can contain lots of independent functionality. Conversely, highly interrelated source code that is scattered across multiple files will also not be captured by a file-level abstraction. To raise the abstraction level, we analyze co-edits on related source code in terms of features. A *feature* is a characteristic, user-visible behavior or configuration option of a software product (Czarnecki and Eisenecker, 2000; Apel et al., 2013). The information a feature usually conveys is richer and more closely mirrors a developer's mental model of the software than files. For that reason and also due to the fact that the concept of features is apparently used by developers (Berger et al., 2015; Queiroz et al., 2017; Hunsen et al., 2016, 2020), our overarching research question is whether there is a difference in developers' collaboration and coordination on features and files. Technically, the code belonging to a feature may be scattered across several files and several features may be tangled within a file (Apel et al., 2013), which needs to be taken into account when developers coordinate.

Compared to the original study, we also take a more nuanced view on communication activity by grouping individual e-mails together according to the thread of communication they belong to. In the original study, all e-mails sent to the mailing list are considered equally likely to be related to each other. We extend the original study by lifting this message-based view of developer communication to a *conversation-based* view, which incorporates the context of e-mails by grouping e-mails according to threads. Since e-mails belonging to the same thread address a relatively narrow topic space, the likelihood of these e-mails being content-wise related is higher (Bird et al., 2008). A heuristic solely based on temporally close-by e-mails sent to the mailing list likely misses meaningful communicative associations between developers. Hence, we investigate the question of whether there is a difference in the dependence of social and technical activities using a message-based or a conversation-based view of the complex processes involved in developer coordination.

By means of an empirical study, we investigate whether the different abstraction levels (file-based vs. feature-based and message-based vs. conversation-based) affect the relationship between commit activity and e-mail communication observed in the original study using state-of-the-art time-series analysis. More specifically, to learn whether developers engineer their mutual contributions on features, we investigate whether synchronous development oc-

curs more frequently or with a higher degree of synchronicity on features than on files. Knowing about differences between abstraction levels could be exploited for improving developer coordination (e.g., to predict on which parts of the source code a developer is likely to work on next). Furthermore, we investigate whether synchronous development is temporally aligned with coordination on the mailing list. To find out whether developers working on the same file or same feature contemporaneously also communicate, that is, to measure synchronization, we use dynamic time warping (Rabiner and Juang, 1993), a state-of-the-art time-series analysis technique.

It is important to note that, when we investigate whether co-editing activity is accompanied by communication on the mailing list, we cannot be sure that the mailing-list communication is related to the co-editing activity. However, it is a difficult task to find out which e-mails are related to the co-editing activity and which not, as e-mails sent by a developer shortly before or after a commit could also cover completely unrelated topics (especially if there are many commits and e-mails within a short period of time); when relating only e-mails whose subject is related to the commit we may omit related e-mails that have a different subject. For that reason, we propose two different approaches, which we call the lower-bound approach and the upper-bound approach: Whereas the *upper-bound approach* considers all e-mails sent to the mailing list to identify time intervals rich in e-mail activities (as in the original study), the *lower-bound approach* considers only e-mails whose subject is topically related to the co-editing activity following a very strict matching procedure. We call them upper-bound and lower-bound because the former considers all messages without restrictions, ending up in the maximum amount of considering communication activity, and the latter considers only messages related to co-editing activity, which is a very small subset of the total set of e-mails. Hence, the actual amount of the communication that is content-wise related to the co-editing activity lies in-between these bounds. For the upper-bound approach, we additionally perform manual checks to explore to which extent the content of e-mail communication is related to temporally close-by collaboration on the source code.

For the purpose of the study, we analyze a combined history of 40 years of data for three highly active and widely deployed open-source projects: QEMU, BusyBox, and OpenSSL. We investigate synchronous collaboration on source code and coordination on mailing lists using different abstraction levels. Overall, we found evidence that a more abstract and higher-level view describes developer collaboration and coordination more accurately than a less abstract and more technical view. That is, developers collaborate more frequently and more synchronously on features than on files. For some of our approaches and projects, a conversation-based representation of developer coordination reveals a stronger statistical relation to co-editing source-code artifacts than a message-based representation.

In summary, we make the following contributions:

- We replicate the original study on a different data set: three highly active and widely deployed open-source projects. Regarding the existence of syn-

chronous development, we are able to confirm the results of the original study. However, we cannot confirm the results of the original study regarding code growth and implementation effort in synchronous development nor the relationship between the number of synchronous collaboration activities and the number of synchronous communication activities.

- We propose two methods for raising the abstraction level of exploring synchronization between developers' collaboration and communication activities:
  - Instead of viewing files as the primary artifacts on which developers are expected to coordinate, we lift the abstraction level to the higher-level perspective of features (which often crosscut the underlying file decomposition).
  - We lift the view of developer communication from a message-based model, which treats each e-mail individually, to a conversation-based model, which is semantically richer due to grouping e-mails that represent conceptually related discussions.
- We introduce the continuous variable *synchronicity degree* to quantify the significance of co-editing artifacts. (Previously only binary variables were used.)
- We propose an upper bound and a lower bound for determining whether e-mail communication is related to co-editing activity, as relating e-mail communication to co-editing activity is not trivial.
- We manually investigate whether e-mail communication is content-wise related to temporally close-by collaboration activities. Our results indicate that only between 29% and 47% (depending on the subject project) of temporally aligned collaboration and communication activities are content-wise related.
- We use a novel technique based on dynamic time warping to measure synchronization of activities across source code and mailing lists to adequately take care of the dynamic nature of socio-technical congruence.
- We report on an extensive empirical study of three highly active and widely deployed OSS projects. We found that feature-based collaboration captures developer collaboration more accurately than file-based collaboration. In general, our results indicate that a more abstract and higher-level view leads to a stronger statistical dependence between developers' pairwise technical activities than a less abstract, technical view.

A full replication package is available on a supplementary Web site[1].

2 Background

Xuan and Filkov (2014) define *synchronous development* as the situation where two developers contribute to the same source-code file within a short period of time. In the original study, they consider two different kinds of synchronous

[1] `https://se-sic.github.io/paper-coordination-bursts/`
Data and scripts are also available at `https://zenodo.org/record/5131282`.

activities: co-commit bursts and e-mail bursts. To explore the temporal relationship between co-commit bursts and e-mail bursts, they construct continuous curves by smoothing time series of bursts. In the end, they calculated the correlation of these curves to measure the synchronization of collaboration activities and communication activities.

In this section, we introduce the algorithms and concepts of co-commit bursts and e-mail bursts as well as the continuous curves in detail, as used by the authors of the original study.

2.1 Co-Commit Bursts

Version-control systems (VCS), such as Git, are frequently used to manage the codebase of software projects. In a VCS, developers can access the source code from a main repository, modify parts of the code, and submit their patches, for example, to the mailing list (Sommerville, 2010; Ramsauer et al., 2019; Draheim and Pekacki, 2003). Code changes can implement bug fixes, refactorings, or further enhancement of the software. Developers often discuss and review code changes on the project's developer mailing list (Mannan et al., 2020) and then someone else may merge the discussed changes into the main repository (Storey et al., 2017). The VCS stores all code changes in the form of commits together with meta-data such as author information and modification timestamps.

When two developers commit to the same source-code artifact (i.e., file) within a short period of time, Xuan and Filkov (2014) call this a co-commit burst (short, *C-burst*). For two commits to be included in a burst, the time difference between the commits must not exceed a specified *time window*, denoted by ξ. The time window resembles the fact that developers may have different preferences of how quickly and often they contribute code. Note that looking at only pairs of developers is not a limitation, as groups of more than two collaborating developers end up in separate C-bursts for each pair of developers that are part of such a group. Hence, group-wise collaboration can be considered as the composition of the collaborations of individual developer pairs.

As we describe in Algorithm 1 (adapted from Xuan and Filkov (2014)), for each pair of developers (Lines 2–22), it is checked whether the two developers are authors of mutual commits to the same source-code artifact that have a time[2] distance of, at most, ξ, and whether these commits have been made to, at least, one common artifact (Line 7). If so, these commits form a C-burst (Lines 4–10), where each burst is represented by a start time and an end time. Finally, overlapping bursts of the same developer pair are merged (Lines 11–19). This algorithm has a complexity of $\mathcal{O}(|\overline{D}|^2 \cdot |\overline{c_{max}}|^2)$, with $|\overline{D}|$ being the number of developers and $|\overline{c_{max}}|$ being the maximum number of commits of a single developer in the project.

[2] All timestamps are transferred to the Coordinated Universal Time (UTC) first.

Algorithm 1 Identification of C-bursts

Input: list of commits $\bar{c}$ (annotated with timestamps and developer names)

```
 1: bursts ← ∅
 2: for each pair of developers {A,B} do
 3:     bursts_AB ← ∅
 4:     for each commit c_A ∈ c̄ authored by developer A do
 5:         burst ← {c_A}
 6:         for each commit c_B ∈ c̄ authored by developer B do
 7:             if |time(c_A) − time(c_B)| ≤ ξ
                and artifacts(c_A) ∩ artifacts(c_B) ≠ ∅ then
 8:                 burst ← burst ∪ {c_B}
 9:             end if
10:         end for
11:         if burst ≠ {c_A} then
12:             for each burst b in bursts_AB do
13:                 if overlap(burst, b) then
14:                     burst ← merge(burst, b)
15:                     bursts_AB ← bursts_AB \ {b}
16:                 end if
17:             end for
18:             bursts_AB ← bursts_AB ∪ burst
19:         end if
20:     end for
21:     bursts ← bursts ∪ {bursts_AB}
22: end for
```

Output: *bursts* containing lists of C-bursts for all developer pairs

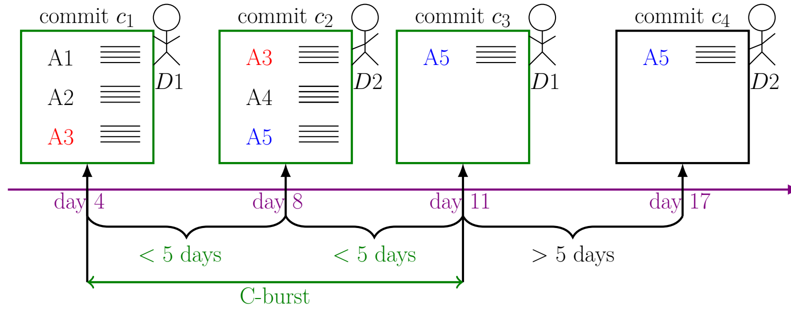

Fig. 1 An example containing four commits made by one pair of developers, $D1$ and $D2$. Commits c_1, c_2, and c_3 form a C-burst: In c_1 and c_2, both developers change one artifact synchronously within the time window ξ of 5 days; commits c_2 and c_3 also form a burst for the same reason. Since these two bursts overlap at c_2, they are combined to one burst. c_3 and c_4 do not form a C-burst as their temporal distance is larger than the time window.

In Figure 1, we show an example of four commits made by one pair of developers, $D1$ and $D2$. In the commits c_1 and c_2, both $D1$ and $D2$ change artifact $A3$. Using a time window $\xi = 5$ days, c_1 and c_2 were created within the time window and form a burst. Analogously, c_2 and c_3 form a C-burst due to the change of artifact $A5$. Since both bursts overlap at c_2, they are merged into one burst in the end. c_4 also changes the same artifact as c_3, but these commits have a larger distance than the time window. Hence, c_3 and c_4 do not form a C-burst.

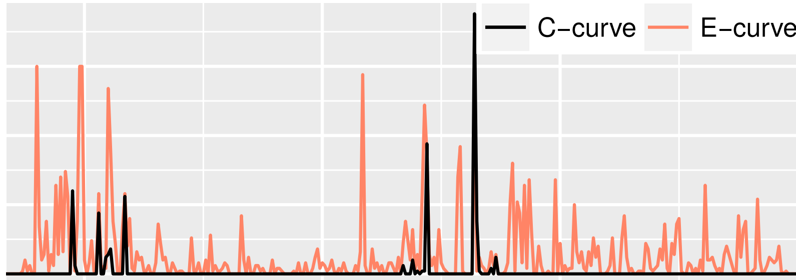

Fig. 2 Example for the C-curve and E-curve of a pair of developers. The horizontal axis represents the time dimension (days), the vertical axis the intensity of the bursts (number of commits and e-mails respectively within the burst).

In addition to identifying C-bursts, the original study analyzed how C-bursts are related to code growth ΔL and implementation effort ΔW, defined as follows: Let L_{Add} denote the number of added lines of code (LOC) per commit and L_{Delete} the number of deleted LOC per commit. Then, $\Delta L = L_{\text{Add}} - L_{\text{Delete}}$ and $\Delta W = L_{\text{Add}} + L_{\text{Delete}}$ (Xuan and Filkov, 2014).

2.2 E-Mail Bursts

Xuan and Filkov (2014) use a message-based model to identify e-mail bursts. An e-mail burst (short, *E-burst*) arises if two persons each send an e-mail to the mailing list within a defined time window ξ. For determining E-bursts, Xuan and Filkov use almost the same approach as for identifying C-bursts: For each pair of developers, they iterate over all the e-mails sent by one developer and search for all e-mails of the other developer whose creation dates have an absolute time difference of less than or equal ξ to the e-mail of the first developer. As opposed to the C-burst identification, there are no further conditions to be checked. Hence, all detected e-mails of two different developers within the time window ξ form an E-burst, where each burst is represented by a start time and an end time. Similar to C-bursts, overlapping E-bursts of the same developer pair are merged.

2.3 C-Curves and E-Curves

To check whether two developers coordinate their collaboration, that is, to check whether C-bursts and E-bursts of a developer pair are synchronized, Xuan and Filkov (2014) introduced the notions of C-curves and E-curves. They compute a C-curve (or E-curve) for each developer pair denoting the number of commits (or e-mails) that are part of a burst aggregated for each day of the time series, as we illustrate in Figure 2. By comparing the C-curve and the E-curve of a developer pair, they investigate whether synchronous development and communication activities of the developer pair are temporally related.

Since coding collaboration and e-mail communication do not take place at exactly the same time, it is not useful to directly compute the overlap of the resulting curves. Therefore, they applied Gaussian smoothing on each of the curves to also be able to align slightly off-set C-bursts and E-bursts. To compare the smoothed curves, they used the Pearson correlation coefficient to check whether C-curve and E-curve of a developer pair are dependent or independent from each other.

3 Research Approach

In our study, we extend the original study by lifting the abstraction level in two ways and by changing the methodology of comparing C-curves and E-curves. On source code, we consider synchronous development based on files and features. Additionally, we introduce a metric to quantify the synchronicity of C-bursts. On mailing lists, we differentiate between message-based communication (considering all synchronously sent e-mails from two developers) and conversation-based communication (considering only e-mails belonging to the same thread). When identifying E-bursts, we use two different approaches to determine a lower-bound and an upper-bound for identifiable coordination. Finally, we use a sophisticated time-series analysis technique to check whether C-bursts and E-bursts of a pair of developers are synchronized.

3.1 Research Questions

Before we state our research questions, let us reiterate the precise meaning of the terms collaboration, communication, and coordination:

Collaboration means that two developers work together by contributing to common source-code artifacts.

Communication means that two developers talk to one another on the mailing list (i.e., exchanging e-mails).

Coordination means developers are collaborating and communicating in (content-wise related) temporally aligned manners.

To obtain deeper insights into the fundamentals of developer coordination in OSS projects, we investigate the relationship between co-editing activities on source-code artifacts and communication activities on the mailing list. The idea is that developers rely on the characteristic information conveyed by features and conversation threads for building a mental model of the software and the processes around it, which in turn drives the communication and coordination with other developers (Espinosa et al., 2001; Scozzi et al., 2008; Cannon-Bowers et al., 1993). So, the overarching question is whether there is a difference in the statistical dependence of social and technical activities between a semantic, high-level view and a rather technical, low-level view of the complex processes involved in developer coordination. That is, we investigate whether developers collaborate more frequently and more synchronously

on features than on files and whether a conversation-based representation of developer coordination reveals a statistically stronger relation to co-editing source-code artifacts than a sole message-based representation. Specifically, we will address the following two research questions regarding each abstraction level of collaboration (files and features) and coordination (message-based communication and conversation-based communication):

RQ1: Which abstraction level of the source code captures the collaboration of developers best: files or features? That is, which of the two abstraction levels of the source code leads to identifying a stronger statistical dependence between technical activities of developer pairs?

RQ2: Which abstraction level of the mailing list captures the coordination of developers best: message-based communication or conversation-based communication? That is, which of the two abstraction levels of the mailing list leads to identifying a stronger statistical dependence between technical activities and social activities on the mailing list?

3.2 Files and Features

We perform the extraction of C-bursts, as defined in Section 2, in two separate analyses for files and features. In the file-based analysis, the commits from two developers within a certain time window form a C-burst if the commits change the same file. One could also think of considering a C-burst if the commits just change a file in the same folder, as files in the same folder may be semantically related to each other. However, projects differ in how they organize files into folders. Folders may be deeply nested, having files at different nesting levels. High-level folders may be too coarse-grained (co-editing code in the same folder may be not related at all), whereas low-level folders may be too fine-grained (missing the relations between files at different levels of nested folders). As it is not obvious and mostly project-dependent which nesting level of folders would be appropriate for C-burst identification, we stick to a file-based analysis, which has been established in the original study.

In the feature-based analysis, the commits from two developers within a certain time window form a C-burst if the commits change the same feature. A feature is a characteristic, user-visible behavior or configuration option of a software product (Czarnecki and Eisenecker, 2000). There are different ways of implementing features in source code, one common way is the use of preprocessor directives (Apel et al., 2013; Ernst et al., 2002). For feature extraction from the source code, we rely on C preprocessor directives (#ifdef, #endif, etc.) (Kernighan and Ritchie, 1988). In Figure 3, we demonstrate a short example: All the code which is in-between #ifdef and #endif belongs to the feature stated in the same line as the #ifdef directive (in the example, the feature is called CLOCK_MONOTONIC). Note that one line of code can belong to multiple features, for example if nested #ifdef directives are used or more than one feature is stated at the beginning of #ifdef. All code changes that affect one of the lines between #ifdef and #endif account for the change of the

```
1  int use_rt_clock;
2
3  static void __attribute__((constructor)) init_get_clock(void)
4  {
5      use_rt_clock = 0;
6  #ifdef CLOCK_MONOTONIC
7      {
8          struct timespec ts;
9          if (clock_gettime(CLOCK_MONOTONIC, &ts) == 0) {
10             use_rt_clock = 1;
11         }
12     }
13 #endif
14 }
```

Fig. 3 Example for feature code using preprocessor directives (#ifdefs), taken from the source code of QEMU (file `util/qemu-timer-common.c`[3]). Source code belonging to feature CLOCK_MONOTONIC is surrounded by an #ifdef directive (Lines 6–13). Hence, Lines 7–12 belong to this feature whereas Lines 1–5 and Line 14 do not belong to this feature.

corresponding feature(s). Note that features may be scattered across multiple files, possibly tangled with other features (Apel et al., 2013). All the changes surrounded by #ifdef directives together with the same feature name belong to the same feature, even if they are part of different files. When analyzing co-edits to features, in our study, code changes which do not belong to a feature (i.e., not surrounded by #ifdef directives) are ignored. We introduce the tools we use to extract feature information in Section 4.2.

3.3 Synchronicity Degree

The method to identify synchronous development described in Section 2 is limited because it does not quantify the magnitude of the overlap among the commits of a C-burst. Essentially, the variable denoting synchronous development is binary. To gain precision, we model the overlap of synchronously changed artifacts within a burst using a continuous variable. This is beneficial because synchronous commits from two developers can contain changes to one common artifact while most of the other changes are to artifacts that are touched by only one of the developers (Bird et al., 2011). For this reason, we introduce the *synchronicity degree*, a metric capturing the overlap based on the number of lines of code (LOC) each of the two developers adds to the artifacts changed in a C-burst. We calculate the synchronicity degree individually for each C-burst. Formally, we define the synchronicity degree deg_{sync} for a C-burst c of the developers A and B as follows:

$$deg_{sync}(c) = \sqrt{\frac{add(A, syncArt(c))}{add(A, art(c))} \cdot \frac{add(B, syncArt(c))}{add(B, art(c))}}, \qquad (1)$$

[3] `https://github.com/qemu/qemu/blob/master/util/qemu-timer-common.c` (accessed: 2019-02-12)

Table 1 Examples of the synchronicity degree deg_{sync} for different numbers of added LOC by developers A and B in C-burst c.

$add(A, syncArt(c))$	$add(B, syncArt(c))$	$add(A, art(c))$	$add(B, art(c))$	deg_{sync}
10	10	10	10	1.00
10	10	20	20	0.50
10	10	15	2 010	0.06

where $add(A, x)$ denotes the number of code lines added by developer A to the list of code artifacts x in C-burst c, $syncArt(c)$ denotes the list of synchronously changed artifacts in C-burst c (i.e., the set of all artifacts changed by both A and B in their respective commits), while $art(c)$ is the set of all artifacts changed in C-burst c. In other words, to determine the synchronicity degree, we calculate the geometric mean of the code changes made by the two developers involved in a C-burst. Specifically, the metric incorporates the size of changes to synchronously changed artifacts of each developer, normalized by the changes to all artifacts in the C-burst. To let the synchronicity degree assign high values only to C-bursts that have a high portion of synchronously changed artifacts, and to down-weight C-bursts that have a highly imbalanced number of changes to non-synchronously changed artifacts, we use the geometric mean, this way reducing the weight of higher values compared to the arithmetic mean (as we also show in the following examples).

In Table 1, we provide examples how the synchronicity degree treats the size of mutual contributions in a C-burst: If both developers change all artifacts touched in the commits of a C-burst synchronously, $deg_{sync} = 1$. When both developers change synchronous artifacts and individually changed artifacts of the C-burst in a balanced way, $deg_{sync} = 0.5$. Finally, if the proportion of synchronously added lines over all added lines is highly imbalanced, the synchronicity degree is very low (e.g., $deg_{sync} = 0.06$).[4]

3.4 E-Mails and E-Mail Threads

We analyze the mailing list of the selected software projects by identifying message-based E-bursts, as described in Section 2. For identifying conversation-based E-bursts, we introduce the additional constraint that only e-mails belonging to a common thread can appear in an E-burst. The rationale is, if two e-mails belong to the same e-mail thread, then this is a more reliable indicator of coordination due to the topical scope in e-mail threads. We identify e-mail threads by cross-referencing e-mail headers. That is, we consider the

[4] In this example, only about 0.5% of developer B's changed lines are made to a synchronously changed artifact, meaning that there is almost no synchronicity. If we would use the arithmetic mean instead of the geometric mean for the calculation of the synchronicity degree, we would get a value of 0.34 instead of 0.06, as the imbalance of the non-synchronously changed lines would not be respected. As a consequence, we use the geometric mean, since a value of 0.06 better describes that there is almost no synchronicity.

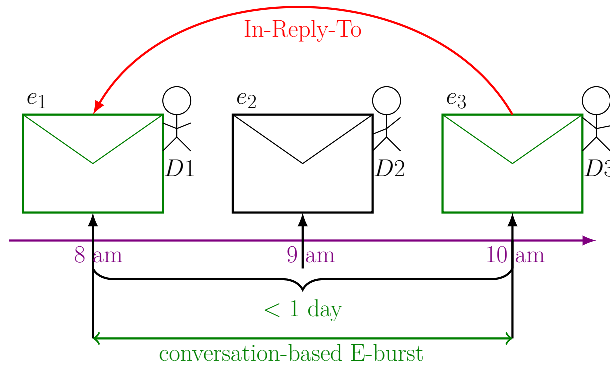

Fig. 4 An example containing three e-mails to the mailing list sent by three different developers $D1$, $D2$, and $D3$. All three e-mails were sent on the same day, therefore, each pair of e-mails forms a message-based E-burst of the involved developer pair. Incorporating thread information, we see that e_1 and e_3 belong to the same thread since e_3 is sent in reply to $e1$. Therefore, e_1 and e_3 form a conversation-based E-burst.

`<In-Reply-To>` and `<References>` tags in e-mail headers to group e-mails belonging to the same thread. We used the threading algorithm of the R package TM-PLUGIN-MAIL[5], which basically implements the standard RFC 5256.

For example, consider the situation illustrated in Figure 4: Developers $D1$, $D2$, and $D3$ each write an e-mail to the mailing list on the same day. Without considering thread information, each pair of e-mails forms a message-based E-burst of the corresponding developer pair, as all three e-mails were sent on the same day. However, e-mail e_2 may address a completely different topic than e_1 and e_3, e_2 may not be related to e_1 and e_3 at all. When considering thread information, we see that e_1 and e_3 belong to the same thread since e_3 is sent in reply to e_1. Therefore, these e-mails can be considered as content-wise related (as defined by the thread). As a consequence, we consider e_1 and e_3 as a conversation-based E-burst, which is a more robust indicator of coordination between developers $D1$ and $D3$ due to the conceptual relation of their e-mails. In this example, there is no conversation-based E-burst including e_2.

3.5 Upper-Bound and Lower-Bound Approach for Determining Coordination

To search for coordination between two developers, we check whether C-bursts and E-bursts of a developer pair are temporally aligned. However, we cannot be certain whether temporally aligned C-bursts and E-bursts are related to each other or whether they are completely unrelated and just are temporally aligned by coincidence. We elaborate on this later in detail and manually check in Section 6 for a small sample of our data whether and for which percentage of the E-bursts such relationships exist. As it is prohibitively time-consuming

[5] `https://r-forge.r-project.org/projects/tm-plugin-mail/` (accessed: 2019-02-12)

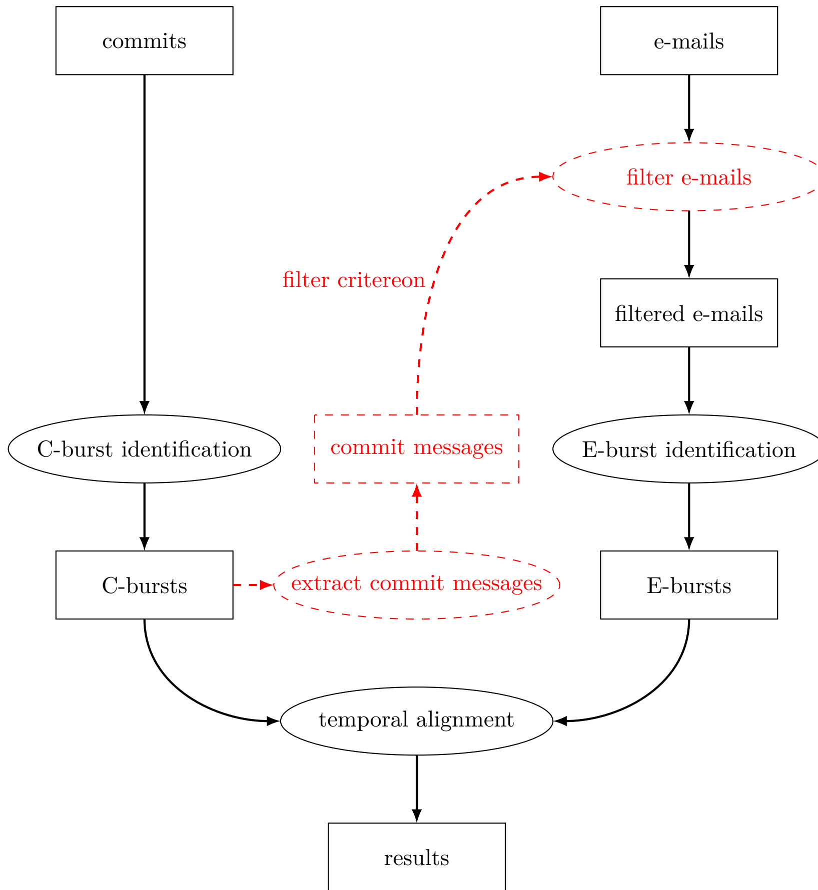

Fig. 5 Workflow for one pair of developers using the upper-bound approach (black, solid only) and the lower-bound approach (red, dashed, and black, solid, together). Commit data are used to identify C-bursts. For the upper-bound approach, all e-mail data are used to identify E-bursts. For the lower-bound approach, the e-mails are first filtered based on the commit messages, which are extracted from the identified C-bursts of the developer pair. Only those e-mails whose subjects match one of those extracted commit messages are kept. Then, the E-bursts are extracted from the filtered e-mails. In the end, independent of the approach, C-bursts and E-bursts of a developer pair are temporally aligned.

to manually decide for each pair of temporally aligned C-bursts and E-bursts whether they are related, we here use two automatic approaches, which we illustrate in Figure 5:

There is an upper bound for coordination, that is, we assume that all the temporally aligned C-bursts and E-bursts are content-wise related and therefore represent coordination (see Algorithm 2 for the trivial e-mail filtering in the upper-bound approach).

Alternatively, in many OSS projects, there is information on the relation between e-mails and commits. For example, if code changes (which form a commit) have to be submitted to the mailing list in form of a patch (like

Algorithm 2 E-Mail Filtering in the Upper-Bound Approach

Input: list of e-mails of a developer pair $\{A,B\}$ $\overline{e}_{\{A,B\}}$
1: $\triangleright$ *no filtering needs to be performed*
2: $\overline{e}_{\{A,B\},\text{filtered}} \leftarrow \overline{e}_{\{A,B\}}$
Output: list of e-mails of a developer pair $\{A,B\}$ $\overline{e}_{\{A,B\},\text{filtered}}$

Algorithm 3 E-Mail Filtering in the Lower-Bound Approach

Input: list of e-mails of a developer pair $\{A,B\}$ $\overline{e}_{\{A,B\}}$,
list of C-bursts of the developer pair $\{A,B\}$ $\overline{cb}_{\{A,B\}}$
1: $\overline{e}_{\{A,B\},\text{filtered}} \leftarrow \emptyset$
2: **for each** C-burst $cb \in \overline{cb}_{\{A,B\}}$ **do**
3: $\overline{m}_{cb} \leftarrow$ extract commit messages from all commits belonging to cb
4: **for each** e-mail $e \in \overline{e}_{\{A,B\}}$ **do**
5: $s_e \leftarrow$ extract subject from e-mail e
6: $s_e \leftarrow$ remove auto-generated prefixes like 'Re:' or 'Fwd:' or '[PATCH]' from s_e
7: **for each** commit message $m \in \overline{m}_{cb}$ **do**
8: **if** m starts with s_e **then**
9: $\overline{e}_{\{A,B\},\text{filtered}} \leftarrow \overline{e}_{\{A,B\},\text{filtered}} \cup e$
10: **break**
11: **end if**
12: **end for**
13: **end for**
14: **end for**
Output: filtered list of e-mails of a developer pair $\{A,B\}$ $\overline{e}_{\{A,B\},\text{filtered}}$

in QEMU), the e-mail subject is often automatically generated out of the heading of the commit message. This way, we can learn that contents of e-mail threads whose subject is also the beginning of a commit message are related to the respective commit. Therefore, temporally aligned C-bursts and E-bursts for which one e-mail of the E-burst has a subject which belongs to a commit message of the temporally aligned C-burst are content-wise related and, hence, indicate coordination. Nevertheless, this might not be the only kind of coordination as e-mails that do not follow this convention could also contain content-wise related information. This is why we call the approach lower-bound approach (see Algorithm 3 for the e-mail filtering based on commit messages in the lower-bound approach). E-mail filtering in the lower-bound approach has a complexity of $\mathcal{O}(|\overline{D}|^2 \cdot |\overline{c_{max}}| \cdot |\overline{e_{max}}|)$, with $|\overline{D}|$ being the number of developers, $|\overline{c_{max}}|$ being the maximum number of commits of a single developer, and $|\overline{e_{max}}|$ being the maximum number of e-mails of a single developer in the project.

Both the upper-bound and the lower-bound approach will not represent the actual amount of coordination, but by using an upper-bound and a lower bound we are able to narrow down the problem and know that the truth must be somewhere in-between these bounds.

3.6 Time-Series Analysis of C-Curves and E-Curves

To check whether C-bursts and E-bursts of a developer pair are synchronized, we need to measure the similarity between both sequences of bursts. For this

purpose, we construct C-curves and E-curves for each developer pair. The curves denote the number of commits and e-mails that are contained in a burst aggregated for each day of the time series. That is, we build a histogram of the numbers of these commits and e-mails per day and derive a curve from that, as depicted in Figure 2.

Since commit activities and e-mail activities rarely occur at the same instant of time, the comparison of C-curves and E-curves needs to be error-tolerant such that we are able to tolerate slight temporal shifts between C-bursts and E-bursts. For example, as developers need some time to create a commit and also some time passes until an e-mail is written, we aggregate the number of commits or e-mails belonging to a burst on a daily basis so that we can perceive bursts of developer activity more clearly. To incorporate also latent times of activity (such as time for implementation, testing, or planning) into this line of thought, we use rectangular smoothing, which achieves two objectives: (1) We reduce noise in the curves and alleviate the intensity of a burst at a specific day (as the aggregation on a daily basis is sensitive to the distribution of the commits or e-mails among several days), and (2) we slightly broaden bursts in the curve to cover that developers may prepare or deal with source-code changes or communication activity longer than the actual work on commits and e-mails lasts, to be robust to a shift of several days between C-bursts and E-bursts. We use a smoothing parameter of 2ξ (with ξ being the time-window parameter used for burst identification). That is, we also take ξ days before and after a burst into account to check whether C-bursts and E-bursts are synchronized. (We also tried other smoothing parameters, which led to similar results, though. For more information, please refer to the supplementary Web site.)

To compare C-curves and E-curves, we use dynamic time warping (DTW) (Rabiner and Juang, 1993), effectively calculating their distances. The DTW algorithm transforms one time series (the query) into the other (the reference) and measures the transformation costs. The higher the resulting transformation costs, the greater the distance between the compared time series. In addition, we use a Sakoe-Chiba band (Sakoe and Chiba, 1978) to constrain the maximum allowed time deviation between two matched data points. The Sakoe-Chiba band prohibits global deformations to match the time series restricting the optimization algorithm to only local transformation operations. With that, we have a global constraint that allows only close-by bursts of the two time series to get matched. Consequently, when using a band-window size of 2ξ, we restrict C-bursts and E-bursts to get matched when their data points have a maximal distance of 2ξ days. We provide further information regarding DTW and the Sakoe-Chiba band in Appendix A.1.

The outcome of the DTW calculation is a distance value describing how different the C-burst time series and the regarding E-burst time series are under the defined transformation restrictions. So, using the DTW, we can measure the temporal correlation between C-bursts and E-bursts of a developer pair.

Table 2 Analyzed time range and size (in terms of numbers of developers, commits, LOC, files, features, e-mails, and e-mail threads) of the subject projects.

		QEMU	BusyBox	OpenSSL
analyzed time range	start:	2003-02-18	2003-01-14	2002-02-18
	end:	2016-07-27	2016-02-19	2016-02-19
# developers		951	230	168
# developer pairs		451 725	26 335	14 028
average # developers active per year		151	34	26
# commits		35 608	10 087	7 887
average # commits per year		3 484	831	791
# LOC (at the latest analyzed commit)		1 106 794	229 087	334 149
# files		3 165	1 362	1 378
# features		1 739	2 498	1 107
# e-mails (messages)		374 815	23 527	10 228
# e-mail threads (conversations)		52 170	7 320	6 280

4 Study Design

In our empirical study, we consider coordination in synchronous development for different abstraction levels: file-based and feature-based collaboration as well as message-based and conversation-based communication on the mailing list. For this purpose, we analyze the OSS projects QEMU, BusyBox, and OpenSSL. In this section, we provide information on our subject projects, describe our data-extraction procedure, give a description of the experiment variables, and formulate our hypotheses.

4.1 Subject Projects

We analyze three different OSS projects: QEMU, BusyBox, and OpenSSL. As these projects differ in size, commit policies, and application domain, they already provide enough insights. Although, due to high computation time and high memory consumption when identifying bursts, we cannot analyze more projects with reasonable effort. All projects are developed in the programming language `C`, using `C` preprocessor directives to annotate feature-specific code (Liebig et al., 2010; Hunsen et al., 2016). Moreover, all projects used a mailing list as well-established and–in the time range we analyze–persistent contribution system to discuss patches and share developer knowledge. For all projects, we analyze all commits and e-mails (sent by developers which also contributed to the source code) from 2002 until 2016. This sums up to about 54 000 commits and 409 000 e-mails across all projects. We provide more details for each project in Table 2.

QEMU is an open-source virtual-machine emulator. The QEMU project has a policy[6] that forces developers to send patches to the mailing list first. Within the analyzed time range of more than 13 years, 951 developers of QEMU created 35 608 commits that changed 3 165 different files and 1 739

[6] `https://wiki.qemu.org/Contribute/SubmitAPatch/` (accessed: 2019-02-12)

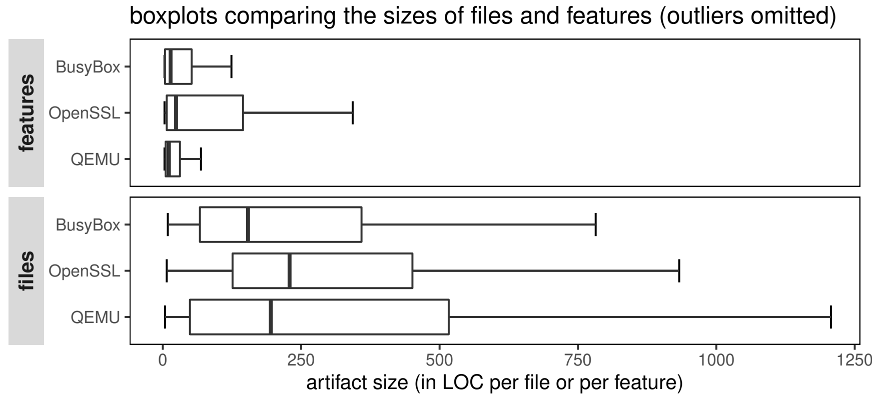

Fig. 6 Box plots of the sizes of files and features in terms of LOC; outliers are omitted. These plots refer to the latest analyzed commit for each of the three subject projects. We do not provide these sizes for the complete project history, as these sizes vary after each single commit but the whole distribution only differs slightly.

different features. The developers, who contributed to the source code, sent 374 815 e-mails in 52 170 different e-mail threads to the mailing list.

BusyBox is a UNIX command-line tool suite, having 230 developers. They created 10 087 commits in the analyzed time range and changed 1 362 different files and 2 498 different features. The developers of BusyBox sent 23 527 e-mails in 7 320 e-mail threads to the mailing list.

OpenSSL is an open-source encryption library to secure connections on the Internet, having 168 developers and 7 887 commits. The developers changed 1 378 different files and 1 107 different features. On the corresponding mailing list, the developers sent 10 228 mails in 6 280 e-mail threads.

In Figure 6, we show the distribution of the sizes of files and features (in terms of LOC) of our three subject projects (for the latest analyzed commit). In general, the size of a feature is smaller than the size of a file. On the one hand, this is because features cover mostly small related units, whereas files may contain code of several features. On the other hand, files also contain some base code, which not necessarily belongs to any feature (e.g., defining constants, copyright headers, glue code).

4.2 Data Extraction

For our study, we gathered commit and e-mail data for each of the subject projects. To obtain commit data, we cloned the publicly available Git repositories hosted on GitHub. For the extraction of the commit data, we use the framework Codeface[7], which analyzes social and technical aspects of development in software projects based on the Git version history. Internally,

[7] `https://siemens.github.io/codeface/` (accessed: 2019-02-12)

Codeface uses the tool cppstats (Liebig et al., 2010) for determining which code belongs to which feature, based on preprocessor directives. That is, cppstats identifies code blocks which are surrounded by preprocessor directives belonging to the feature which is named in the surrounding preprocessor directives. cppstats is regularly used for locating features (Hunsen et al., 2016, 2020; Berger and Guo, 2013; Feigenspan et al., 2013) and has been extended in various ways (Fenske et al., 2015; Schulze and Fenske, 2018; Medeiros et al., 2015). Using Codeface, we are able to extract meta-data of all commits of a software project, as has been demonstrated in prior studies (Joblin et al., 2015, 2017a,b). The commit meta-data contain 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 or features), together with the number of added lines per artifact. We limit the extraction of commit data to files that are implementation-related, so header files, documentation files, and build files are not considered in our study (see Section 8).

Beside commit data, we have collected e-mail data from each subject project. We downloaded the e-mails from the publicly available mailing-list archive Gmane[8]. For each of our subject projects, we only downloaded the e-mails of the corresponding developer mailing list (not user mailing lists), as we focus on investigating the coordination of source-code changes of developers. We used again Codeface to extract information from e-mail headers: the date on which an e-mail was created, the author of the e-mail, and the thread ID of the e-mail (e-mails belonging to the same thread have the same thread ID). In our study, we only include e-mails of developers who also appear in commits, because we want to investigate the relation between C-bursts and corresponding E-bursts.

Authors can use different names and several e-mail addresses, which makes it challenging to map e-mails and commits onto real persons. Codeface first attempts to assign e-mails to a certain author by matching the author's names. If names of authors do not match, Codeface proceeds with matching e-mail addresses. To put it simply, authors of e-mails which use the same name or the same e-mail address are mapped to one real person. Regarding name and e-mail disambiguation, Codeface implements the heuristics of Oliva et al. (2012), which provides good results compared to other heuristics as it has a median recall of approximately 0.5 and a median precision of about 0.9 (Wiese et al., 2016).

4.3 Variables

In Table 3, we provide an overview of the independent and dependent variables of our study.

As independent variables, we vary the considered artifact types (files and features) for identifying the C-bursts. For identifying E-bursts, we distinguish

[8] `https://gmane.org/` (accessed: 2019-02-12)

Table 3 Independent and dependent variables of our empirical study.

independent variables	dependent variables
– abstraction of co-editing (files, features)	– number of C-bursts
– abstraction of communication (message-based, conversation-based)	– synchronicity degree deg_{sync}
– filtering of the e-mails: (none (upper-bound), by C-bursts' commit messages (lower-bound))	– classification of commits (synchronous and non-synchronous)
– time window ξ (1 day, 5 days, 10 days, 15 days)	– code growth ΔL
	– code effort ΔW
	– number of E-bursts
	– DTW distances describing the temporal correlation between C-curves and E-curves

between message-based E-bursts and conversation-based E-bursts. On top of that, we also vary the e-mail filtering: Whereas we use all e-mails and perform no filtering for our upper-bound approach, we filter the e-mails by commit messages of the C-bursts before identifying E-bursts for the lower-bound approach. In addition, we vary the time window for burst identification: We consider time windows ξ of 1 day, 5 days, 10 days, and 15 days, based on a response-time analysis: In Figure 7, we show the response times for subsequent e-mails and subsequent commits to a common artifact for each pair of developers, for each level of abstraction and subject project considered in our study. Depending on project and abstraction level, at least, 70% of the different response times are shorter than 15 days. When omitting outliers, all response times on features are shorter than 15 days. Especially on mailing lists, at least, 70% of the response times are even shorter than 1 day. Hence, our chosen time windows ξ are reasonable time distances that synchronous development can deliberately last, because developers mostly reply to e-mails or commits within a few days.

The results of our study depend on the variations of the above described independent variables (cf. Table 3). In particular, the number of bursts, the synchronicity degree of C-bursts, and the temporal correlation between C-bursts and E-bursts depend on the independent variables. Also the differentiation of commits into synchronous and non-synchronous as well as code growth and code effort depend on the independent variables. Notice that we consider a commit to be synchronous if, at least, one of its changed artifacts is mutually changed within a C-burst.

4.4 Null Model

To determine whether the bursts and their synchronicity degrees are just artifacts of a purely random process (and thus uninteresting for us as we expect that collaboration and communication are dependent and correlated processes), we use a simulation technique based on synthetic data sets drawn from a null model. The null models, which represent random time series, allow us to test whether empirically observed bursts are significantly different from purely random bursts (i.e., whether they convey information). That is, by us-

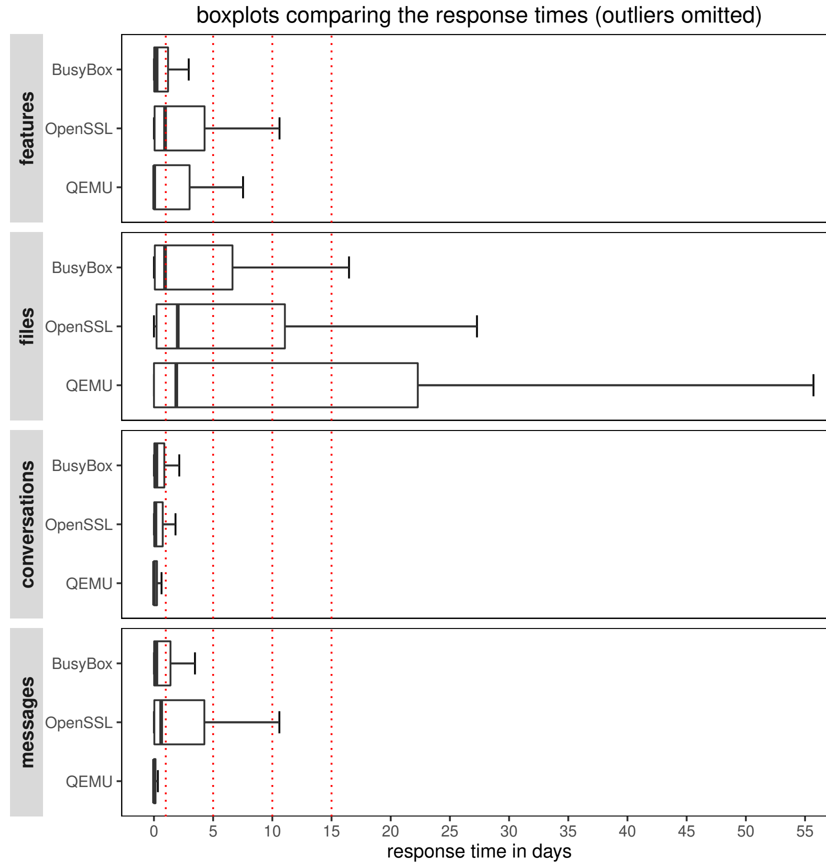

Fig. 7 Box plots of the response times for each pair of developers; outliers are omitted. File-based and feature-based response times represent the time distances between subsequent commits to a common file or feature of a developer pair. Message-based response times represent the time distances between subsequent e-mails of a developer pair, conversation-based response times only represent the time distances between subsequent e-mails of a developer pair within the same thread. The red vertical lines represent the time windows ξ chosen for our study.

ing a null model, we check whether our results are dependent on our variables or arise randomly. Specifically, we use the null models of the original study, as we explain next (Xuan et al., 2012).

For commit data, we generate synthetic data based on a null model by purely randomizing the time intervals between two successive commits for each developer. The randomization operation is performed by randomly permuting the time intervals between all commits of the considered developer (see Algorithm 4 in Appendix A.2). This way, for each developer, the distribution of the time intervals, the order of the commits, and the artifacts changed by this developer are preserved. C-bursts generated from the purely randomized time series are referred to as *simulated C-bursts*.

For e-mail data, we use a similar approach. The only difference is that we do not randomize the time intervals between the e-mails of each developer, but the time intervals between successive e-mails of each pair of developers to preserve the order of e-mails sent by two different developers. So, each pair of developers has their own simulated e-mail time series (see Algorithm 5 in Appendix A.2). E-bursts of the purely randomized time series are referred to as *simulated E-bursts*.

Generating a simulated commit time series (without burst detection) has a complexity of $\mathcal{O}(|\overline{D}| \cdot |\overline{c_{max}}|)$, generating a simulated e-mail time series has a complexity of $\mathcal{O}(|\overline{D}|^2 \cdot |\overline{e_{max}}|^2)$, with $|\overline{D}|$ being the number of developers, $|\overline{c_{max}}|$ being the maximum number of commits of a single developer, and $|\overline{e_{max}}|$ being the maximum number of e-mails of a single developer in the project.

For both commit and e-mail time series, we generate 100 simulated time series each, except for one subject project. For QEMU, we only generate 2 simulations of the e-mail time series due to computational limitations.[9] See Section 8 for a discussion of threats to validity.

4.5 Hypotheses

Next, we introduce our hypotheses. Each hypothesis is evaluated by varying time window (1 day, 5 days, 10 days, 15 days) and abstraction levels.

Before introducing our hypotheses, let us explain our numbering scheme for hypotheses: Our main hypotheses (i.e., H1, H2, and H3) are related to the comparison of the different abstraction levels. However, before comparing abstraction levels, we first check for each of them whether the underlying hypotheses of the original study hold. We introduce the underlying hypotheses of the original study as sub-hypotheses of our main hypotheses (i.e., H1.1, H1.2, H1.3, H1.4, H2.1, and H3.1). To answer our main hypotheses (i.e., to compare the different abstraction levels), we compare the outcomes of the corresponding sub-hypotheses on the different abstraction levels. So, for comparing abstraction levels, we lift the particular sub-hypotheses and define composed hypotheses (i.e., H1+H1.1, H1+H1.2, H1+H1.3, H1+H1.4, H2+H2.1, and H3+H3.1).

4.5.1 Hypotheses Related to C-Bursts

First, we define hypotheses regarding the collaboration of developers. Since features are higher-level units and are a common concept in software engineering, we formulate the following hypothesis:

H1 Feature-based collaboration captures developer collaboration more accurately than file-based collaboration.

[9] To identify E-bursts in 100 simulations of QEMU for four different time windows and two abstraction levels, we would need about 22 months using 40 nodes, having a 2.2 GHz processor, with 20 cores each in parallel and 128 GB RAM per node.

Specifically, due to the nature of features and the extensive use of features by the developers, we hypothesize that feature-based C-bursts appear more often and with higher synchronicity degree than file-based C-bursts. Moreover, for the same reason, we hypothesize that synchronous commits result in higher code growth and lower implementation effort, since synchronously working on features is more productive and less laborious. In particular, we define the following four sub-hypotheses, which we check for file-based and feature-based C-bursts separately:

H1.1 The number of empirical C-bursts is higher than the number of simulated C-bursts.
H1.2 The synchronicity degree of empirical C-bursts is higher than the one of simulated C-bursts.
H1.3 The code growth ΔL is higher in synchronous commits than in non-synchronous commits.
H1.4 The implementation effort ΔW is lower in synchronous commits than in non-synchronous commits.

We check these sub-hypotheses to examine that developer collaboration is not a purely random process and has an effect on the number of C-bursts and the considered characteristics. After checking the four sub-hypotheses for file-based and feature-based C-bursts separately, we use the results of all four sub-hypotheses and compare the outcomes for the different abstraction levels to answer H1 conclusively. In particular, H1 comprises the following hypotheses:

(H1+H1.1) The number of empirical feature-based C-bursts is higher than the number of empirical file-based C-bursts.
(H1+H1.2) The synchronicity degree of empirical feature-based C-bursts is higher than the synchronicity degree of empirical file-based C-bursts.
(H1+H1.3) The code growth ΔL in synchronous commits is higher on features than on files.
(H1+H1.4) The implementation effort ΔW in synchronous commits is lower on features than on files.

If one of the four sub-hypotheses H1.1, H1.2, H1.3, or H1.4 does not hold, we neglect the corresponding part of H1 since the corresponding characteristic appears purely random.

4.5.2 Hypotheses Related to E-Bursts

Second, as developers converse via mailing lists, we formulate hypotheses regarding which abstraction level of mailing-list communication captures coordination of developers best. That is, we expect that conversations capture the coordination activity among developers more accurately than considering individual messages only, as e-mail conversations represent the conceptual relationship between e-mails.

H2 Conversation-based communication captures developer coordination more accurately than message-based communication.

In particular, we check the following sub-hypothesis for both conversation-based and message-based communication, as a high amount of collaboration activity should be aligned with a high amount of coordination:

H2.1 The relation between the number of C-bursts and the number of E-bursts is described by a linear relationship.

For this hypothesis, we consider only developer pairs that have, at least, one C-burst and, at least, one E-burst, since we aim at analyzing developer pairs that contribute to the source code *and* communicate on the mailing list. Here, we use both our upper-bound and our lower-bound approach and evaluate the hypothesis separately for both approaches. After checking this sub-hypothesis for all abstraction levels separately, we compare the strengths of the linear relationships of message-based and conversation-based coordination to answer H2 conclusively:

(H2+H2.1) The linear relation between the number of C-bursts and the number of E-bursts has a higher goodness of fit for conversation-based E-bursts than for message-based E-bursts.

Also, here, we evaluate the hypothesis separately for the upper-bound and the lower-bound approach.

4.5.3 Hypotheses Related to C-Bursts and E-Bursts

Finally, we investigate the temporal relationship between C-bursts and E-bursts of developer pairs with the following hypothesis:

H3 The temporal correlation between C-bursts and E-bursts is higher for feature-based C-bursts than for file-based C-bursts and higher for conversation-based E-bursts than for message-based E-bursts.

If a C-burst and an E-burst of the same developer pair are temporally related, this is an indicator of a relationship between these bursts. That is, an E-burst that appears right before or after a C-burst may address the discussion of the code changes applied in the C-burst. To answer H3, we check the following sub-hypothesis for each of the abstraction levels of source code and mailing list to examine that the empirical DTW distances are not purely random and empirical C-bursts and E-bursts are dependent processes:

H3.1 C-bursts and E-bursts are temporally correlated, that is, the DTW distances between empirical C-curves and empirical E-curves are smaller than between simulated curves.

That is, we expect related C-bursts and E-bursts to appear temporally close to each other, resulting in smaller DTW distances than for simulated bursts. Again, we analyze the temporal correlation only of developer pairs which have, at least, one C-burst and, at least, one E-burst.

After checking the sub-hypothesis, we compare the empirical DTW distances of the different abstraction levels with each other to answer H3 conclusively:

(H3+H3.1) The DTW distances between C-curves and E-curves are smaller for feature-based C-bursts than for file-based C-bursts and also smaller for conversation-based E-bursts than for message-based E-bursts.

We use the E-bursts of our two different approaches separately and evaluate our hypothesis for both the upper-bound and the lower-bound approach.

4.5.4 Statistical Tests

To test the hypotheses, we use a suite of statistical methods. All the above stated hypotheses are alternative hypotheses.

For H1.1, we use a one-tailed, paired Wilcoxon signed-rank test to compare the numbers of empirical and simulated C-bursts for each pair of developers. Therefore, for each of the developer pairs, we compute the median of the numbers of C-bursts of the 100 simulations and compare this median with the empirical number of C-bursts of the developer pair. (We use an aggregated measure of the 100 simulations to be able to use a paired test for comparing the real and empirical numbers of C-bursts per developer pair. To be robust to outliers, we use the median.) Here, we consider also developer pairs that have no C-burst at all, since the number of developer pairs having no burst can be different and, therefore, can affect the comparison. Together with the Wilcoxon signed-rank test, we compute the corresponding effect size r.[10]

For H1.2, H1.3, and H1.4, we use a one-tailed, unpaired Mann-Whitney U test: We compare the whole population of the synchronicity degrees of all empirical bursts with the whole population of the synchronicity degrees of all 100 simulations together. We also compute Cliff's Delta, which quantifies the effect size that corresponds to the Mann-Whitney U test.

For H2 and H2.1, we fit a linear regression model and compare the fitted models by comparing their adjusted R-square values and p-values.

For H3.1, we use a one-tailed, unpaired Mann-Whitney U test to compare the empirical DTW distances of all considered developer pairs with the simulated DTW distances of all simulations together. Note that we use a Mann-Whitney U test because the number of available data points can be rather small (especially when using the lower-bound approach, we have only few developer pairs that have, at least, one C-burst and, at least, one E-burst in some cases), and the data are not necessarily normally distributed, which we measured using the Shapiro-Wilk test. Corresponding to the Mann-Whitney U test, we again use Cliff's Delta to quantify the effect size.

Finally, for the comparison of the different abstraction levels using a Mann-Whitney U test in the hypotheses H1 and H3, we use False Discovery Rate (FDR) correction to account for multiple testing.

[10] `https://www.rdocumentation.org/packages/rcompanion/versions/2.3.7/topics/wilcoxonPairedR` (accessed: 2019-02-12)

5 Results

Next, we present our results.[11] To generate the results (including simulations), we used the following hardware in parallel over several weeks: 40 computation nodes having a 2.2 GHz processor, with 20 cores each and 128 GB RAM per node. In the tables and plots that follow, we concentrate on the results for our largest and, therefore, most representative subject project QEMU, which also exemplifies the results of the other projects. For all data and plots, we refer the reader to the supplementary Web site[12].

5.1 C-Bursts

In H1, we state that feature-based collaboration captures developer collaboration more accurately than file-based collaboration. Before investigating this hypothesis, we check our four corresponding sub-hypotheses separately for both abstraction levels.

As we show in Table 9 in the appendix, we found for QEMU that the C-bursts per developer pair based on files or features occur significantly more frequently than purely by chance ($p < 0.05$), independent of the time window ξ. However, the corresponding effect size is very low. The reason for the low effect size is that most of the developer pairs have no C-burst at all. For example, in QEMU, only ~1% of the developer pairs have, at least, a C-burst, due to the combinatorial explosion of developer pairs. Nevertheless, when we restrict our analysis to developer pairs that have, at least, one C-burst, then we still get significant results (empirical C-bursts occur more frequently than purely by chance), but we get higher effect sizes (absolute values between 0.59 and 0.90, see also the corresponding tables on our supplementary Web site). Regardless of that, also the overall number of empirical C-bursts (file-based or feature-based) is higher than the overall number of simulated C-bursts (using the median of the 100 simulations to get one number per developer pair). For example, in the file-based analysis with $\xi = 5$, there are 5 185 empirical C-bursts, but only 3 122 simulated ones. There are similar results for BusyBox and OpenSSL. Therefore, *we accept H1.1.*

Also, the synchronicity degrees of the empirical C-bursts are significantly higher than the synchronicity degrees of the simulated C-bursts (see Table 10 in the appendix; $p < 0.05$) for all abstraction levels and time windows. This holds for each subject project; for BusyBox and OpenSSL the corresponding effect sizes are even higher than for QEMU. Hence, *we accept H1.2.*

Regarding the code growth of synchronous and non-synchronous commits, we observe in Table 11 in the appendix that code growth (ΔL) is, according to the Mann-Whitney U test ($p < 0.05$), only in the file-based analysis with $\xi = 1$

[11] When we state that we accept a certain hypothesis (all our hypotheses are alternative hypotheses, as stated above), we actually mean that we reject the corresponding null hypothesis.

[12] `https://se-sic.github.io/paper-coordination-bursts/`

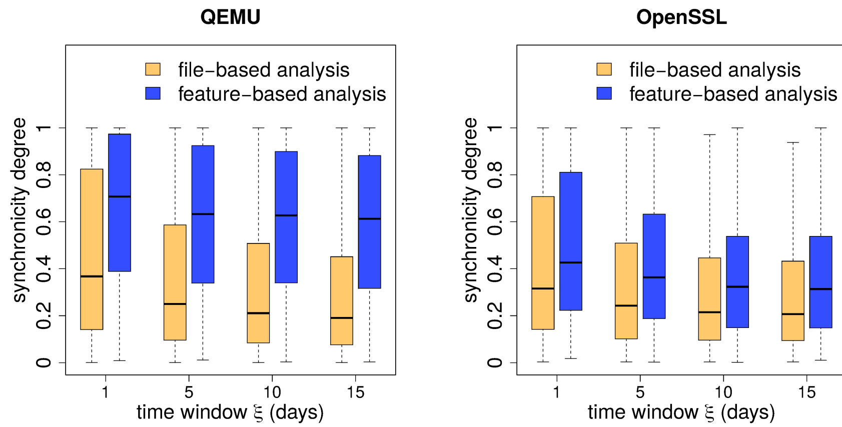

Fig. 8 Box plots of the synchronicity degrees of file-based and feature-based C-bursts at different time windows for QEMU (left) and OpenSSL (right). Outliers are omitted.

or $\xi = 5$ higher in synchronous commits than in non-synchronous commits of QEMU. For other time windows and for the feature-based analysis of QEMU, this does not hold. In BusyBox, code growth in synchronous commits is higher than the code growth in non-synchronous commits only in the feature-based analysis with $\xi = 10$ or $\xi = 15$. In OpenSSL, code growth in synchronous commits is not higher than in non-synchronous commits for both file-based and feature-based analyses for all chosen time windows ξ. As we can observe a higher code growth in synchronous commits than in non-synchronous commits only in some rare cases, but not in general, *we reject H1.3.*

When we look at code effort (Table 12 in the appendix), we find that ΔW is not significantly lower in synchronous commits than in non-synchronous commits, in all cases except for the feature-based analysis of QEMU. For BusyBox and OpenSSL, ΔW is not lower in synchronous commits than in non-synchronous commits for all feature-based and all file-based analyses. Hence, as there is no statistically significant difference, *we reject H1.4.*

Finally, to test H1, we compare the outcomes of H1.1 to H1.4 for file-based and feature-based C-bursts. For H1+H1.1, we see that the number of C-bursts per developer pair is higher for feature-based C-bursts than for file-based C-bursts. The Mann-Whitney U tests with FDR correction in Table 13 (left) in the appendix also indicate that the number of C-bursts per developer pair is significantly higher for feature-based C-bursts than for file-based C-bursts ($p < 0.05$). This holds for all projects, except for BusyBox with $\xi = 1$. The low effect sizes arise from the huge number of developer pairs that have no C-burst at all (due to combinatorial explosion), as stated above. Comparing the results of H1+H1.2, as we show in Figure 8, the synchronicity degrees of feature-based C-bursts are always higher than of file-based C-bursts, for all projects. From the corresponding Mann-Whitney U tests with FDR correction in Table 13 (right) in the appendix we can see that this is also a significant

Table 4 Overview of the results regarding H1 and its sub-hypotheses. ✓ denotes that we accept a (sub-)hypothesis, ✗ the denotes that we reject a (sub-)hypothesis.

	H1.1	H1.2	H1.3	H1.4	
files	✓	✓	✗	✗	
features	✓	✓	✗	✗	
	H1+H1.1	H1+H1.2	H1+H1.3	H1+H1.4	H1
files vs. features	✓	✓	✗	✗	✓

result ($p < 0.05$), having medium to large effect sizes. As we have already seen that H1.3 and H1.4 do not hold, we do not need to compare the respective data for H1. (For the sake of completeness, Table 14 in the appendix contains the results of the corresponding Mann-Whitney U tests with FDR correction.)

So, overall, *we accept H1* (see Table 4), as it holds for the comparison of files and features of H1+H1.1 in all projects (except for the comparably small project BusyBox with $\xi = 1$), and also for the comparison of files and features of H1+H1.2 in all projects. As H1.3 and H1.4 both do not hold, neither for features, nor for files, we do not need to take these sub-hypotheses into account for checking our main hypothesis H1. So, based on H1+H1.1 and H1+H1.2, we conclude that feature-based collaboration captures collaboration statistically more accurately than file-based collaboration in terms of a higher number of bursts and a higher synchronicity degree.

5.2 E-Bursts

For the comparison of message-based and conversation-based E-bursts, we first test for H2.1 whether the number of C-bursts and the number of E-bursts for a developer pair (having, at least, one burst each) are linearly dependent. We do this separately for the E-bursts extracted with our lower-bound approach and with our upper-bound approach respectively. Using the upper-bound approach, fitting a linear model on the message-based E-bursts results in a very small adjusted R-square value of 0.03 (see left plot in Figure 9), that is, only 3% of the variance is described by the linear model. Only a high adjusted R-square value would indicate that the model describes the data points well.

In contrast, still using the upper-bound approach, the linear model of conversation-based E-bursts and C-bursts fits significantly better, as the adjusted R-square value of 0.32 in Figure 9 (right) illustrates. That is, 32% of the variance is described by the linear model, and hence there is a stronger linear relationship between the number of C-bursts and the number of E-bursts for conversation-based E-bursts than for message-based E-bursts.

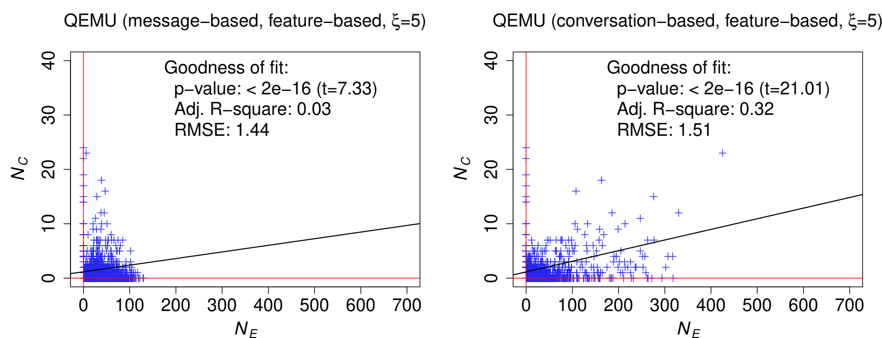

Fig. 9 Upper-bound approach: Linear model fitting for the relationship between the number of C-bursts (N_C) and the number of message-based E-bursts (N_E) (left) and conversation-based E-bursts (right). Every data point represents one developer pair. Model fitting was applied only to developer pairs having, at least, one C-burst and, at least, one E-burst (data points above and right respectively of the red horizontal and vertical lines respectively).

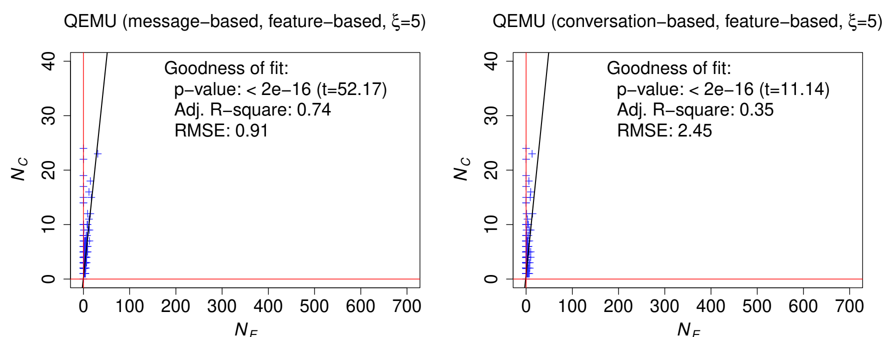

Fig. 10 Lower-bound approach: Linear model fitting for the relationship between the number of C-bursts (N_C) and the number of message-based E-bursts (N_E) (left) and conversation-based E-bursts (right). Every data point represents one developer pair. Model fitting was applied only to developer pairs having, at least, one C-burst and, at least, one E-burst (data points above and right respectively of the red horizontal and vertical lines respectively).

Table 5 Overview of the results regarding H2 and its sub-hypothesis. ✓ denotes that we accept a (sub-)hypothesis, ✗ the denotes that we reject a (sub-)hypothesis.

	H2.1 (lower bound)	H2.1 (upper bound)
messages	✓	✗
conversations	✓	✓
	H2+H2.1 (lower bound)	H2+H2.1 (upper bound)
messages vs. conversations	✗	✓

Switching to the lower-bound approach, we get significant linear models for both message-based and conversation-based E-bursts, as the adjusted R-square values of 0.74 (message-based) and 0.35 (conversation-based) indicate (see Figure 10). This result holds only for QEMU, though. In BusyBox and OpenSSL, there are too few data points to fit a significant linear model. Instead of fitting a linear model, we also tried to compute Spearman's rank correlation, which also led to similar, non-significant results for these subject projects. Therefore, *we accept H2.1 for conversation-based E-bursts*, as it holds for both the upper-bound and the lower-bound approach of QEMU. However, *we reject H2.1 for message-based E-bursts*, as we get a significant linear model only for the lower-bound approach.

Eventually, comparing the results for message-based and conversation-based E-bursts shows that conversation-based E-bursts have a better linear relationship to the number of C-bursts than message-based E-bursts for most of the cases when we use the upper-bound approach (see adjusted R-square values in Figure 9). Nevertheless, when we use the lower-bound approach, it is the other way round (see adjusted R-square values in Figure 10). So, overall, *we are inconclusive regarding H2* (see Table 5).

5.3 C-Bursts & E-Bursts

Finally, we search for temporal correlation between C-bursts and E-bursts by computing the DTW distances of the C-curves and E-curves of each developer pair.

First of all, we investigate whether the temporal correlation of the empirical C-bursts and E-bursts is different from the correlation in the null model. In Table 15 in the appendix, we state the results of using the upper-bound approach. For QEMU, message-based E-bursts have significantly smaller DTW distances to C-bursts for all abstraction levels (files, features) and time windows than in the null model. For conversation-based E-bursts, we also obtain significantly smaller DTW distances to the C-bursts than in the null model. Similar results hold for BusyBox: Here, we have significantly smaller DTW distances for the empirical bursts than for the null model for all abstraction levels, except for message-based E-bursts and feature-based C-bursts with $\xi = 5$, $\xi = 10$, or $\xi = 15$. Also for OpenSSL we obtain similar findings: There are significantly smaller DTW distances for the empirical bursts than for the null model for all abstraction levels, except for feature-based C-bursts and conversation-based E-bursts with $\xi = 1$ or $\xi = 5$ and file-based C-bursts and conversation-based E-bursts with $\xi = 1$. When we use the lower-bound approach instead, the empirical DTW distances are smaller than the simulated ones in all cases of QEMU and BusyBox (see Table 16 in the appendix). For OpenSSL, this holds only for message-based C-bursts and file-based E-bursts with $\xi = 15$. The reason for that is that there are too few data points: When using the

Table 6 Overview of the results regarding H3 and its sub-hypothesis. ✓ denotes that we accept a (sub-)hypothesis, ✗ the denotes that we reject a (sub-)hypothesis.

	H3.1 (lower-bound)	H3.1 (upper-bound)
	✓ (see Table 16, appendix)	✓ (see Table 15, appendix)
	H3+H3.1 (lower-bound)	H3+H3.1 (upper-bound)
files vs. features	inconclusive	inconclusive
messages vs. conversations	✗	inconclusive

lower-bound approach, we often have zero E-bursts per developer pair, resulting in empty DTW curves. Hence, for OpenSSL, we cannot state any results regarding H3. When we neglect the cases where we have insufficient data, the empirical DTW distances are (with few exceptions) shorter for empirical bursts than for the null model, for both the upper-bound and the lower-bound approach. Therefore, *we accept H3.1 for all abstraction levels.*

After comparing the empirical DTW distances with our null models, we now compare the outcomes of the different abstraction levels of the empirical data with each other. We provide a general overview of the corresponding results for the comparison of files and features in Table 17 in the appendix and the corresponding results for the comparison of messages and conversations in Table 18 in the appendix.

When we compare feature-based C-bursts and file-based C-bursts, we can see that using feature-based C-bursts leads to significantly lower DTW distances than using file-based C-bursts only if we analyze QEMU with message-based E-mails (see Table 17 in the appendix). For all the other cases and subject projects, this is not the case. Note that these results are almost identical for the upper-bound and the lower-bound approach.

For the comparison of message-based E-bursts and conversation-based E-bursts we obtain a complex picture (see Table 18 in the appendix). When analyzing QEMU with the upper-bound approach, using conversation-based E-bursts leads to significantly lower DTW distances than using message-based E-bursts, independent from ξ and independent from whether using file-based of feature-based C-bursts. However, when using the lower-bound approach, this does not hold at all. Also for BusyBox, no matter which approach and which kind of C-bursts are used, this does not hold (except for the file-based C-bursts with $\xi = 5$ in the upper-bound approach). As we have seen for H3.1, there are too few data points to state valid results for OpenSSL.

Since feature-based C-bursts have only in some cases significantly lower DTW distances with the E-bursts than file-based ones, and conversation-based E-bursts do not have significantly lower DTW distances with the C-bursts than message-based ones (except for QEMU with the upper-bound approach), *we have inconclusive results regarding H3* (see Table 6).

6 What is Discussed within E-bursts?

In Section 5, we presented the results of our quantitative analysis of the relation of C-bursts and E-bursts. The weakness of our quantitative analysis is that it does not capture whether there is actual coordination of source-code changes in temporally close-by discussions on the developer mailing list. To alleviate this threat to validity, we conducted a qualitative analysis to investigate whether our notion of coordination is reliable. We performed this qualitative analysis only for E-bursts identified via our upper-bound approach, but not for E-bursts identified via our lower-bound approach. The reason is that, in the lower-bound approach, for each developer pair, we filter the e-mails already by comparing the commit messages of temporally close-by C-bursts with the e-mail subjects. Hence, in the lower-bound approach, we only get E-bursts which are content-wise related to the C-bursts of the developer pair. As we cannot assume anything regarding the relation of C-bursts and E-bursts for the upper-bound approach, we perform this qualitative analysis to find out to which extent temporally close-by C-bursts and E-bursts are indeed content-wise related.

In a first step, we manually checked for all developer pairs with, at least, five C-bursts and five E-bursts whether the content of e-mails of an E-burst is related to the commits of the temporally close-by C-bursts. As this is a very time-consuming manual task, we only performed this for conversation-based E-bursts and only for $\xi = 15$. So, we manually looked at 56 E-bursts of BusyBox, 49 E-bursts of OpenSSL, and 766 E-bursts of QEMU. In BusyBox, 29% of these E-bursts are content-wise related to a C-burst of the same developer pair; in OpenSSL we found that 41% of the E-bursts are related to a C-burst, and in QEMU this holds even for 47%. Throughout our manual analysis, we identified different kinds of how C-bursts and E-bursts are related: In most cases, the e-mail subject is related to the commit message or the e-mail content even contains parts of the commit message or the patch itself. Also, certain key words are often used in commit messages or code patches that are also used in the content or subject of an e-mail. In addition to that, we also were able to match C-bursts and E-bursts by inspecting the file names of the changed files and searching for them in the e-mail subject or content. However, in cases where we decided that all the e-mails of an E-burst are not related to C-bursts of the same developer pair, we had a closer a look at the content of these E-bursts to find out what these conversations are about. It turned out that these conversations are mostly about future plans of the software project or organizational matters (e.g., coding conventions, contribution guides, workflows, or future releases). However, there are also lots of discussions regarding bugs or problems identified by users (even though we analyzed only developer mailing lists). This is also supported by the results of previous research: Guzzi et al. (2013) analyzed the communication in mailing lists of OSS projects and found that only about 35% of the discussions are related to actual source-code changes. Nevertheless, in our study, up to one half of all the E-bursts are di-

Table 7 The results of our qualitative analysis using our mention rate, that is, the percentage of C-bursts whose artifacts (file names and feature names respectively) are mentioned in a temporally close-by E-burst of the same developer pair.

	ξ	QEMU		BusyBox		OpenSSL	
		messages	convers.	messages	convers.	messages	convers.
files	1	34%	9%	10%	5%	1%	0%
	5	50%	13%	15%	8%	4%	1%
	10	55%	15%	19%	10%	6%	1%
	15	59%	17%	24%	12%	9%	2%
features	1	41%	8%	14%	12%	0%	0%
	5	62%	12%	24%	15%	2%	0%
	10	69%	14%	30%	21%	4%	0%
	15	72%	16%	33%	24%	6%	1%

rectly related to co-edits, which justifies our assumption that the developer mailing lists are used to coordinate source-code changes.

In a second step, we used an automatic approach to determine whether C-bursts and E-bursts are content-wise related. We calculated a *mention rate* for file names and feature names in E-bursts. That is, we determine the percentage of C-bursts whose artifacts (at least, one) are mentioned in temporally close-by E-bursts. In Table 7 we present the results of this analysis. As we can clearly see, the results differ between subject projects and abstraction levels. The mention rates are higher for message-based E-bursts than for conversation-based E-bursts. This is not surprising as the message-based E-bursts contain also single e-mails that are not related to other e-mails and, therefore, the chance for a file or feature of a C-burst to be mentioned is higher than for conversation-based ones, due to the potentially higher number of e-mails that are contained in E-bursts. Mention rates for file-based C-bursts are higher than for feature-based C-bursts as file names are used more often due to technical reasons (source code is organized in files). Overall, we can see that the mention rate ranges from 0 to 72%, which indicates that temporally close-by C-bursts and E-bursts are related in many cases. However, the exact file names or exact feature names need not be mentioned when coordinating software changes. Sometimes, developers may paraphrase which feature or file they are talking about without directly stating the name of the file or feature. Hence, our mention rate only covers a (possible small) part of the actual relation of C-bursts and E-bursts.

In a third step, we evaluated whether the e-mail filtering of our lower-bound approach is reasonable. So, we automatically checked for each commit of a C-burst whether there is, at least, one e-mail in the E-bursts of the same developer pair whose subject is equal to the beginning of the commit message. (Notice that we removed auto-generated prefixes of the e-mail subjects that match standard patterns like *Re:* or *Fwd:* or *[PATCH]* and alike before performing this analysis). We present the results of this check in Table 8: For QEMU, almost all commits of a C-burst are related to, at least, one e-mail of an E-burst of the same developer pair, using message-based e-mails. When

Table 8 The results of determining the percentage of commits of C-bursts whose commit messages map with the subject of, at least, one e-mail of an E-burst of the same developer pair.

	ξ	QEMU		BusyBox		OpenSSL	
		messages	convers.	messages	convers.	messages	convers.
files	1	86%	31%	0%	0%	0%	0%
	5	97%	43%	2%	2%	0%	0%
	10	98%	44%	28%	23%	2%	0%
	15	98%	45%	24%	21%	5%	0%
features	1	84%	21%	0%	0%	0%	0%
	5	95%	32%	0%	0%	0%	0%
	10	97%	36%	6%	3%	2%	0%
	15	98%	41%	8%	4%	4%	0%

only investigating e-mails belonging to the same thread, as in conversation-based E-bursts, the percentage of commits of C-bursts whose commit messages match the subject of, at least, one e-mail of an E-burst is much lower. This might be the case because we neglect single e-mails not belonging to a thread: Some of the automatically generated e-mails, whose subjects match commit messages, just contain the patch, whereas the discussion of this patch can take place in an e-mail thread different from the patch. So, the corresponding thread for the discussion of the patch can have a slightly different subject, which cannot be matched in this analysis. For BusyBox and OpenSSL it is only rarely the case that commit messages and e-mail subjects can be mapped to each other, as those do not have such a strict commit policy to send patches to the mailing list as QEMU. As a consequence, our lower-bound approach (in which we also map commit messages with e-mail subjects) seems to be a reasonable filtering of the e-mails before E-burst identification, at least, for the subject project QEMU.

7 Discussion

7.1 C-Bursts (H1)

Our study confirms that, for all abstraction levels and time windows, synchronous development is not a purely random process where developers' activities are statistically independent. Knowing that our operationalization of synchronous development does not occur purely by chance, we analyze the synchronicity degree of the identified C-bursts, which is also higher than for randomly generated C-bursts. Altogether, our results show that considering the concept of synchronous development in OSS projects is well-founded. This is in line with the original study.

Code growth in synchronous development is, in most cases, lower than in non-synchronous development. This is contrary to the outcomes of the original study. The reason for this is that, in synchronous development, not only

the number of added lines is higher, but also the number of deleted lines. H1.3 ignores that high coding activity in synchronous development does not necessarily incur high code growth. This also affects code effort, which is, in most cases, higher in synchronous commits than in non-synchronous commits. This is in contrast to H1.4 and, therefore, also in contrast to the results of the original study. As code additions and deletions are both higher in synchronous commits, we conclude that, in synchronous development, more lines are changed than in non-synchronous development. Hence, due to higher coding activity in terms of LOC, analyzing synchronous development is useful for understanding developer collaboration.

Finally, there are differences between abstraction levels. First, the number of feature-based C-bursts per developer pair is significantly higher than the number of file-based C-bursts per developer pair, which supports our reasoning that collaboration on features is more common than on files. Feature-based collaboration is not only more common, but also has higher synchronicity degrees. These results suggest that developer collaboration rather takes place at the level of features, which seem to represent the developers' mental model of the software system more appropriately. Consequently, further studies on the collaboration of developers should pursue a more higher-level view.

7.2 E-Bursts (H2)

According to our results, the number of C-bursts and the number of E-bursts per developer pair correlate only weakly. We use the goodness of the linear fit to assess whether message-based or conversation-based E-bursts lead to identifying a stronger statistical dependence between the amount of collaboration and the amount of communication. Contrary to the original study (which only considered message-based E-bursts of the upper-bound approach), the goodness of the linear fit is lower in our analysis, but when we use conversation-based E-bursts, we get a similar linear fit as the original study reported for message-based E-bursts. The difference between our message-based results and the message-based results of the original study may be due to different sizes of the projects in terms of developers and in terms of e-mails, as our largest subject projects has 951 developers and 374 815 e-mails, whereas the largest subject project of the original study had 72 developers and 11 865 e-mails. As there is huge difference in the size of the projects, there may be also differences in the organizational structure of the projects, resulting in different numbers of E-bursts. In the following, we compare message-based and conversation-based communication. We discuss our outcomes regarding H2 here only based on the results of QEMU, as we cannot draw reliable conclusions from BusyBox and OpenSSL. See Section 8 for the discussion of the corresponding threats to validity.

When we look at message-based E-bursts arising from the upper-bound approach, where we keep all e-mails for E-burst identification, we see that the number of resulting E-bursts is too high, as the goodness of fit of the linear

model is rather weak. Keeping the upper-bound approach but identifying E-bursts only among e-mails which are content-wise related (conversation-based), we get a lower number of E-bursts, resulting in a better goodness of fit with the number of C-bursts than for message-based E-bursts.

When using the lower-bound approach, we recognize the opposite behavior: Conversation-based E-bursts lead to a lower goodness of fit than message-based E-bursts. An explanation might be that, in the lower-bound approach, we filter the e-mails before E-burst identification by checking whether their subject is part of the beginning of a commit message of a C-burst of the same developer pair. Due to this filtering, we already narrowed down the number of E-bursts to keep only those for which we are sure that they are related to a C-burst. When we then construct conversation-based E-bursts, the number of E-bursts is narrowed down again as e-mails need to belong to the same thread to form an E-burst. Hence, the number of E-bursts may be too small in the end.

As a lesson learned, only focusing on conversations of e-mails, of which we can be sure that they are related to C-bursts, is a too severe restriction of the view of coordination. The actual truth with respect to the identification of E-bursts is somewhere in-between our lower-bound and upper-bound approach.

When looking at the goodness of fit in general, the linear model fitted on conversation-based E-bursts in the upper-bound approach describes only less than 40 percent of the variance, the linear model fitted on the message-based E-bursts in the lower-bound approach describes around 75 percent. Such low percentages are not unexpected since developers may often send e-mails to the mailing list without contributing to the source code at temporally close time. For instance, they can comment on other issues or discuss topics independent from the source code (Guzzi et al., 2013). Furthermore, multiple C-bursts may be discussed in one E-burst, whereas several E-bursts can address one single C-burst. That is, the number of C-bursts does not necessarily need to correlate with the number of E-bursts, as bursts can last differently long and discussions on inter-related topics among co-edits are possible. For the lower-bound approach, in particular, we can draw two subsequent conclusions from that: (1) Coordination of co-edits may not always happen within the same e-mail thread, as using thread information leads to a lower correlation with the number of C-bursts, and (2) coordination may not only take place via e-mails that are related to specific commits indicated by the e-mail subject and the commit message, as considering only commit-related e-mails is a strong restriction, which drastically reduces the number of E-bursts compared to the upper-bound approach.

7.3 C-Bursts & E-Bursts (H3)

Even though we found that features are a suitable abstraction (H1) and that a more nuanced view on e-mail communication (a message-based view for the lower-bound approach and a conversation-based view for the upper-bound approach) is valuable (H2), we often cannot find significant differences between

the DTW distances on different abstraction levels (although the empirical DTW distances are significantly smaller than for the corresponding null models, which is in line with the results of the original study, where the authors used correlation coefficients instead of DTW distances).

When we compare file-based coordination with feature-based coordination, we can see that there is mostly no significant difference. When using message-based E-bursts in QEMU, feature-based C-bursts have smaller DTW distances to the E-bursts than file-based ones. This shows us that message-based communication in QEMU is more related to coordinating feature-based collaboration than to coordinating file-based collaboration. As this does not hold for conversation-based communication and also not for other projects, we cannot generally assume that feature-based coordination reveals a stronger statistical dependence between developers' pairwise technical activities and their social activities than file-based coordination.

Also the comparison of message-based and conversation-based abstraction does not show significant differences, at least, when using our lower-bound approach. That is, in this case, a lot more C-bursts than E-bursts occur per developer pair, so the number of matched C-bursts and E-bursts is comparably small. Even if there are more message-based E-bursts than conversation-based ones, in the end, the differentiation here does not matter, since, in both cases, many C-bursts do not have a corresponding E-burst. For the upper-bound approach, in most cases, conversation-based E-bursts lead to identifying a stronger statistical dependence between collaboration and communication than message-based E-bursts. Thus, if there are enough communication data, a more nuanced view on the communication of developers describes coordination among developers more precisely than simply looking at single messages.

Nevertheless, the correlation between C-bursts and E-bursts is for both the upper-bound and the lower-bound approach higher than in the null model. This demonstrates a significant statistical dependence between collaboration and communication, which implies that developer coordination is actually taking place, but depends on many variables.

7.4 Research Questions (RQ1 and RQ2) and Perspectives

In RQ1, we asked for the abstraction level of source code which captures collaboration of developers best in terms of a stronger statistical dependence between technical activities of developers. As discussed in Section 7.1, collaboration takes place mostly and most synchronously at the level of features. This is intelligible given the characteristic, user-visible dimension of features. So, a higher-level view on developer collaboration should be pursued in further work.

In RQ2, we looked for the best abstraction level of developer coordination in terms of a stronger statistical dependence between the technical activities and the social activities on the mailing list. Our results suggest that lifting the study of developer coordination to a conversation-based model is worthwhile

and sometimes even required. However, for temporal alignment of coding and communication, many different aspects matter, which shall be explored in further studies.

To summarize, the correlation of collaboration and communication activities of developers depends on many variables. We did not observe any universal developer behavior when comparing the coordination using feature-based and file-based C-bursts, nor when comparing the coordination using message-based and conversation-based E-bursts. We noticed that, to some extent, the relationship between collaboration and communication is project dependent, which needs to be taken into account to refine the general measurement method toward a specific setting.

As the null model essentially captures the case in which developers' technical activities and social activities are performed independently at random and, since we see a departure from the null model in our empirical data, there is a stochastic dependence between developers' technical activities and their social activities. This is in line with the outcomes of the original study and indicates some extent of alignment in terms of time and structure between the technical realm and the social realm, which is sometimes referred to as socio-technical congruence. Our approach relies on a more dynamic interpretation of socio-technical congruence than in most previous work: We use dynamic time warping to identify the alignment of technical and social activities, whereas previous work adequately ignored the dynamic nature of the phenomenon and searched for an alignment within static time windows, as, for instance, in the work of Joblin et al. (2017b) (see Section 9 for more details on that).

The stochastic dependence between collaboration and coordination that we have identified has practical value because it helps to reduce the uncertainty when making predictions. For example, if we know that a pair of developers was involved in a huge number of C-bursts and we know that C-bursts and E-bursts are dependent, we may build a model that predicts for their joint technical activity the need for coordinating their work. In addition, we could think about exploiting the relationship between two developers represented by a C-burst to predict which files or features a developer is likely to work on next by considering past C-bursts. We could also search for missing dependences between collaboration and coordination and investigate how the quality of the development process and the developed artifacts are affected. For instance, one could check whether there are more bugs and other issues on a specific artifact if there is no E-burst related to a C-burst on the considered artifact, to get an even more detailed view on how software development is influenced by coordinating activities. Finally, as we have identified stronger statistical dependences when using higher-level views on the abstraction levels of collaboration (i.e., features) and coordination (i.e., a more content-related view), it is more feasible to execute the described ideas using these abstraction levels.

8 Threats to Validity

8.1 Internal Validity

The results of our study rely on the performance and correctness of CODEFACE as we extract all the commit and e-mail data using this tool. For the extraction of feature code, CODEFACE relies in turn on CPPSTATS, which has already been used to extract feature code in other empirical studies (Liebig et al., 2010; Hunsen et al., 2016, 2020; Berger and Guo, 2013; Fenske et al., 2015; Schulze and Fenske, 2018; Feigenspan et al., 2013; Medeiros et al., 2015). Even if preprocessor annotations are not used the same way in different projects, preprocessor annotations are a well-established means to denote feature-specific code.

In our study, we do not consider changes to header files, documentation files, and build files. This affects our results only barely, though, as the number of changes of build files or documentation files is comparably low. In addition, in non-implementation files, mostly, there is no variability implemented. Thus, considering non-implementation files would cause an imbalance among the abstraction levels.

As the studied projects predate the existence of the version-control system GIT, the commits in the GIT history at the beginning of the analyzed time range of all three subject projects had originally been imported from the previously used version-control system SVN, which had a slightly different operationalization of commits. However, to the best of our knowledge, this does not threaten our notion of C-bursts, as the import of SVN commits into GIT did preserve authoring timestamp, author, and code changes.

The validity of our results could potentially be threatened by the occurrence of bots that automatically send e-mails to the mailing list (e.g., when a commit is added to the VCS). This could lead to identifying E-bursts between bots and developers, which could be mitigated by detecting and filtering bots' e-mails (e.g., based on specific e-mail patterns used by the respective bot). However, this is not relevant in our study as the subject projects we analyzed, to the best of our knowledge, do not use such bots, which we have checked on a sample basis using the information from both the projects' Web sites and e-mail headers of the e-mails sent to the mailing lists.

For both commit and e-mail time series, we generated 100 simulated time series each per project. Due to the sheer size and complexity (that is, generating an individual simulation of the e-mail time series separately for each developer pair), we were not able to achieve this for the e-mail time series of QEMU (which has 451 725 developer pairs). While this threatens the validity of the results of H3.1, all other results confirm that the involved processes are significantly different from the null model and thus not purely random.

We introduce synchronicity degree and DTW distances describing the temporal correlation between C-curves and E-curves. For the synchronicity degree, we designed a metric that reasonably considers the size of commits. For the DTW distances, we rely on the fact that the technique of DTW is well-

established and can be properly restricted regarding the distance measurement (i.e., the Sakoe-Chiba band (Sakoe and Chiba, 1978)).

The sparseness of the data threatens the validity of our study: The number of developer pairs having, at least, one C-burst and, at least, one E-burst is low ($\sim 0.2\%$), although, this is expected since not all developers actually collaborate. This is also the reason why we also consider developer pairs that have no bursts in our C-burst-related analysis.

BusyBox and OpenSSL have only few developer pairs involved in, at least, one C-burst and one E-burst. So, we cannot draw reliable conclusions from these projects regarding the relationship between the number of C-bursts and the number of E-bursts. Nonetheless, as the number of E-bursts is narrowed down to zero with the lower-bound approach in some cases of OpenSSL, we should take a closer look at the different code-contribution practices of the different projects: Whereas there is a policy in QEMU[13] to send patches to the mailing list and discuss them there, other projects, such as OpenSSL, do not have such a strict code-contribution policy. As a consequence, on the QEMU mailing list, there are lots of e-mails that contain a patch and therefore, automatically, contain the title of the corresponding commit message in their e-mail subjects, whereas in other projects like OpenSSL this is not the case that often. Hence, our lower-bound approach is limited to the strictness of the patch-contribution policy of the respective subject project.

We rely on mailing lists as the only communication channel, although developers may happen to use further channels (e.g., personal e-mails or verbal communication) (Storey et al., 2017). We mitigate this threat by selecting only projects that have a historically rich and well-established mailing list for discussing software architecture and reviewing code changes and also subject projects which have mandates regarding patch submission to the mailing list prior to being accepted (Rigby et al., 2008; Sommerville, 2010; Ramsauer et al., 2019; Draheim and Pekacki, 2003). In addition, more recently introduced social-coding platforms (e.g., GitHub) are too young for history analysis, whereas the mailing lists of our subject projects date back more than 10 years (see Table 2).

Finally, we did not perform a linguistic analysis of the e-mail data. Hence, the communication on the mailing list may partly concern other issues than coordinating source-code changes. We alleviate this by considering only e-mails of developers who also contributed to the source code of the corresponding project. Moreover, we performed a qualitative analysis to check whether there is a content-wise correlation between temporally close-by C-bursts and E-bursts (see Section 6).

8.2 External Validity

We have analyzed three different OSS projects, which differ in size, commit policies, and application domain. Due to the high computation time and huge

[13] `https://wiki.qemu.org/Contribute/SubmitAPatch/` (accessed: 2019-02-12)

memory consumption of our approach, we cannot analyze more than these three subject projects within a reasonable amount of time and memory. While one cannot generalize our findings arbitrarily–as always in such a study–we have substantial data for three large, highly active, and widely deployed OSS projects, which gives us relevant insights into the behavior of collaboration and coordination at different abstractions levels.

The restriction to preprocessor annotations as means for locating feature-specific code may threaten external validity, as they are mostly specific to `C`. Preprocessor annotations are well-established in OSS to implement features (e.g., Ernst et al. (2002); Liebig et al. (2010); Apel et al. (2013); Hunsen et al. (2016, 2020)), but the findings may vary in detail for other feature implementation techniques, though not the big picture.

9 Related Work

Beside the study of Xuan and Filkov (2014), which we reproduce and extend, there has been various research on the relationship between development and communication between developers. Herbsleb and Grinter (1999) conducted a study on coordination in geographically distributed software projects. They found that ad-hoc communication between developers is one of the most important parts of today's well-working software development. Related studies (Cataldo and Herbsleb, 2013; Herbsleb and Mockus, 2003; Mockus et al., 2002; Crowston and Howison, 2005; de Souza et al., 2005) showed that coordination in software projects affects software quality and that considering social aspects, such as communication of developers, is essential for understanding OSS projects. We extend on these by considering the temporal and content-wise dependency between communication and technical activities from a higher-level perspective.

The authors of the original study enhanced their work by identifying patterns on the time series of working activity in terms of source-code commits and communication activity in terms of replying to e-mails on the mailing list for developer pairs, which indicate that collaboration on source-code artifacts and coordinating events on the mailing list are temporally related (Xuan et al., 2016).

Gharehyazie and Filkov (2017) extended the original study by not investigating pairs of developers but groups of developers working on the same source-code artifacts temporally close-by. In their work, they analyzed whether the size of such groups is purely random and how often developers work in groups rather than working alone. Contrary to our work, they chose an even more coarse-grained level of abstraction and analyzed source-code changes on package level. Similarly to our qualitative analysis, they manually checked for coordination of a developer group by searching for file names of temporally close-by edited files within e-mails of the developer group, resulting in finding actual coordination on the mailing list. In addition, they performed developer surveys which confirmed their results, which is also in line with the hypotheses

of our study. They also analyzed if the code growth is higher and the corresponding effort is lower for developers working in groups than solely. In line with our results, they only identified rare projects where this hypothesis holds.

In previous work, researchers used network approaches to describe the collaboration and coordination of developers: López-Fernández et al. (2006) constructed networks representing mutual contributions of developers to the same software module, that is, to files contained in the same directory. Jermakovics et al. (2011) built networks based on co-editing files, and Toral et al. (2010) analyzed social communities on e-mail networks that arose from software development. Joblin et al. (2015) constructed developer networks based on co-commits on source-code artifacts, especially at the more fine-grained level of functions, and used network analysis techniques to gain more information on collaboration. In contrast to our work, none of these analyzed features, and they also neglected temporal aspects.

Joblin et al. (2017b) analyzed structural and evolutionary trends of developer coordination using an evolutionary network approach, though synchronicity between collaboration at the code level and communication via e-mail was not in their focus, nor the characteristic, user-visible level of features.

Bacchelli et al. (2010) analyzed the e-mail communication of OSS projects, determined the much discussed source-code artifacts, and investigated the defect-proneness of those artifacts. Bird et al. (2006, 2008) investigated whether coding activities on files are related to communication on the mailing list by focusing on collaboration within sub-communities and analyzing e-mail social networks. In our study, we measure the synchronicity of co-edits and directly compare synchronous development on files or features to e-mail communication using state-of-the-art time-series analysis.

Jiang et al. (2014) linked commits to certain e-mails on the mailing list by comparing commits to previously submitted patches on the mailing list. Also, Ramsauer et al. (2019) use a similar approach. Compared to our study, they explicitly trace back commits (that is, changed source-code lines) to e-mails that contain these changed source-code lines as part of a patch, whereas we are interested in all kinds of e-mails that are in some way related to certain code changes, not necessarily containing patches changing exactly the lines which are changed in a commit. Coordination of certain changes may cover more discussions and high-level design decisions than just submitting or discussing patches. Ali et al. (2013) mined software repositories to build traceability links between source code and textual requirements documents using advanced information-retrieval techniques. As we look for relationships between code changes and communication among developers, we also link textual documents (sets of e-mails) to source-code changes, but as we are more interested in the coordination activities than the concrete content, we do not use advanced information-retrieval techniques.

10 Conclusion

The success of software projects, in particular, large-scale, globally distributed projects, relies essentially on the coordination of co-edits to the source code, as previous work has shown (Crowston and Howison, 2005; Cataldo and Herbsleb, 2013; Kwan et al., 2011). Co-editing source code is a common way of performing bug fixes, refactorings, enhancements, and adding new features concurrently (Singh, 2010). We investigated the relationship between co-editing activities and communication on the mailing list for three highly active and widely deployed OSS projects using different abstraction levels.

In a nutshell, we demonstrated that a more abstract, higher-level view on source code (features) captures the notion of developer collaboration and synchronicity of co-edits more precisely than a less abstract, technical view (files). Furthermore, we found that a more nuanced view of communication substantially increases the correlation between co-editing and coordinating e-mail activities compared to a simple message-based view, which is reasonable since coordination of developers comprises collections of conceptually related e-mails. We did not observe a general picture regarding the temporal correlation between co-editing code and e-mail communication, though, which depends to a good extent on the project setting at hand.

Overall, we found evidence that a more abstract and higher-level perspective captures the developers' collaboration and coordination activities more accurately than a sole technical perspective. This is not unexpected because developers typically think in terms of features and topics when building mental models of software rather than in terms of technical artifacts or individual text messages. Further studies in this area should take this perspective into account.

Acknowledgements This work has been supported by the German Research Foundation (AP 206/5-1&2, AP 206/6-1&2, AP 206/11-1, and AP 206/14-1).

A Appendix

A.1 Dynamic Time Warping and Sakoe-Chiba Band

Dynamic Time Warping (DTW) tries to align two time series of equal length with each other by traversing a matrix D beginning in $D(0,0)$ and ending in $D(n,n)$, where 0 and n are the earliest and latest time of the two time series. Using dynamic programming and calculating cumulative sums of distances on the path, DTW explores the whole matrix space to find the path of the shortest distance (Rabiner and Juang, 1993; Berndt and Clifford, 1994; Keogh and Pazzani, 2001). The Sakoe-Chiba band only allows exploring cells in the matrix at which the absolute distance of the compared data is less than or equal to the chosen band-window size. So, only data points of the two time series get matched that have an absolute distance less than or equal to the chosen band-window size (Sakoe and Chiba, 1978). In our study, this prohibits that a C-burst and an E-burst that occur temporally extremely distant to each other get matched by the DTW algorithm. (How we use DTW is described in Section 3.6.)

A.2 Algorithms

Algorithm 4 Generation of simulated C-bursts

Input: list of commits $\overline{c}$ (annotated with timestamps and developer names)

```
1:  ▷ generate simulated commit time-series
2:  c̄_sim ← ∅
3:  for each developer D do
4:      c̄_D ← commits in c̄ authored by D
5:      t(D) ← sorted list of timestamps of c̄_D
6:
7:      ▷ create an ordered list intervals to store the lengths
8:      ▷ of intervals between two subsequent commits of D
9:      for i in 2 : length(t(D)) do
10:         interval ← t(D)_i − t(D)_{i−1}
11:         intervals_i ← interval
12:     end for
13:
14:     ▷ randomize the order of the intervals, but keep the same distribution
15:     randomizedIntervals ← shuffle the elements in intervals
16:
17:     ▷ generate simulated list of timestamps s(D)
18:     s(D)_1 ← t(D)_1
19:     for i in 2 : length(t(D)) do
20:         s(D)_i ← s(D)_{i−1} + randomizedIntervals_{i−1}
21:     end for
22:     c̄_{D,sim} ← update the timestamps in the list of commits c̄_D according to s(D)
23:     c̄_sim ← c̄_sim ∪ c̄_{D,sim}
24: end for
25:
26: ▷ extract simulated C-bursts
27: apply Algorithm 1 to the simulated commit list c̄_sim
```

Output: simulated C-bursts for each pair of developers

Algorithm 5 Generation of simulated E-bursts

Input: list of e-mails $\overline{e}$ (annotated with timestamps and developer names)
1: ▷ *generate a simulated e-mail time-series for each pair of developers*
2: **for each** pair of developers $\{A,B\}$ **do**
3: $\overline{e}_{\{A,B\}} \leftarrow$ e-mails in $\overline{e}$ sent by A or B
4: $t(A,B) \leftarrow$ sorted list of timestamps of $\overline{e}_{\{A,B\}}$
5:
6: ▷ *create an ordered list* intervals *to store the lengths*
7: ▷ *of intervals between two subsequent e-mails sent by A or B*
8: **for** i in $2 : length(t(A,B))$ **do**
9: $interval \leftarrow t(A,B)_i - t(A,B)_{i-1}$
10: $intervals_i \leftarrow interval$
11: **end for**
12:
13: ▷ *randomize the order of the intervals, but keep the same distribution*
14: $randomizedIntervals \leftarrow$ shuffle the elements in $intervals$
15:
16: ▷ *generate simulated list of timestamps* $s(A,B)$
17: $s(A,B)_1 \leftarrow t(A,B)_1$
18: **for** i in $2 : length(t(A,B))$ **do**
19: $s(A,B)_i \leftarrow s(A,B)_{i-1} + randomizedIntervals_{i-1}$
20: **end for**
21: $\overline{e}_{\{A,B\},sim} \leftarrow$ update the timestamps in the list of e-mails $\overline{e}_{\{A,B\}}$ using $s(A,B)$
22:
23: ▷ *extract simulated E-bursts*
24: apply E-burst extraction to pair $\{A,B\}$ using $\overline{e}_{\{A,B\},sim}$, as described in Section 2.2
25: **end for**
Output: simulated E-bursts for each pair of developers

A.3 Result Tables

Here, we only present some selected result tables for our largest subject project QEMU. For all data and results, we refer the reader to our supplementary Web site:
`https://se-sic.github.io/paper-coordination-bursts/`

Table 9 Paired, one-tailed Wilcoxon signed-rank test for comparing empirical simulated numbers of C-bursts per developer pair (H1.1). (We use the median of all simulations to get one value per developer here.) s denotes the standard deviation. V represents the V-statistic of the Wilcoxon signed-rank test, the corresponding p-value indicates whether the alternative hypothesis H1.1 is accepted ($p < 0.05$) or not. r denotes the effect size corresponding to the paired Wilcoxon signed-rank test.

		# bursts (mean±s)				
QEMU	ξ	empirical	simulated	V	p-value	r
files	1	0.005±0.117	0.002±0.079	706 200	< 0.05	-0.041
	5	0.011±0.235	0.007±0.213	2 948 700	< 0.05	-0.045
	10	0.017±0.305	0.012±0.280	5 925 700	< 0.05	-0.047
	15	0.022±0.340	0.015±0.310	8 491 000	< 0.05	-0.048
features	1	0.008±0.163	0.004±0.136	257 870	< 0.05	-0.040
	5	0.023±0.296	0.016±0.302	1 908 400	< 0.05	-0.038
	10	0.036±0.343	0.026±0.352	4 973 200	< 0.05	-0.036
	15	0.044±0.373	0.035±0.377	8 208 900	< 0.05	-0.032

Table 10 One-tailed Mann-Whitney U test for comparing synchronicity degrees (H1.2). s denotes the standard deviation. U represents the U-statistic of the Mann-Whitney U test, the corresponding p-value indicates whether the alternative hypothesis H1.2 is accepted ($p < 0.05$) or not. Cliff's Delta denotes the corresponding effect size.

		deg_{sync} (mean±s)				Cliff's
QEMU	ξ	empirical	simulated	U	p-value	Delta
files	1	0.466±0.352	0.340±0.306	169 020 000	< 0.05	0.207
	5	0.369±0.325	0.296±0.284	1 403 200 000	< 0.05	0.123
	10	0.332±0.308	0.270±0.269	3 296 700 000	< 0.05	0.110
	15	0.307±0.297	0.254±0.260	5 078 600 000	< 0.05	0.097
features	1	0.642±0.307	0.601±0.311	47 697 000	< 0.05	0.076
	5	0.609±0.310	0.584±0.309	418 200 000	< 0.05	0.048
	10	0.603±0.307	0.575±0.307	981 100 000	< 0.05	0.054
	15	0.590±0.308	0.565±0.307	1 509 200 000	< 0.05	0.047

Table 11 One-tailed Mann-Whitney U test for comparing code growth ΔL of synchronous and non-synchronous commits (H1.3). s denotes the standard deviation. U represents the U-statistic of the Mann-Whitney U test, the corresponding p-value indicates whether the alternative hypothesis H1.3 is accepted ($p < 0.05$) or not. Cliff's Delta denotes the corresponding effect size.

		ΔL (mean±s)				Cliff's
QEMU	ξ	synchronous	non-synchronous	U	p-value	Delta
files	1	42.647±732.059	39.235±263.358	96 963 000	< 0.05	0.059
	5	34.986±515.489	42.769±285.679	152 310 000	< 0.05	0.023
	10	34.232±448.141	46.100±308.759	158 750 000	0.22	0.005
	15	32.979±415.286	51.160±339.906	147 620 000	0.91	-0.009
features	1	53.356±310.371	78.065±366.911	6 548 900	1.00	-0.062
	5	47.646±269.631	96.741±422.171	7 994 600	1.00	-0.104
	10	48.250±262.206	108.914±460.593	7 356 300	1.00	-0.129
	15	50.724±267.304	111.965±474.954	6 895 300	1.00	-0.139

Table 12 One-tailed Mann-Whitney U test for comparing code effort ΔW of synchronous and non-synchronous commits (H1.4). s denotes the standard deviation. U represents the U-statistic of the Mann-Whitney U test, the corresponding p-value indicates whether the alternative hypothesis H1.4 is accepted ($p < 0.05$) or not. Cliff's Delta denotes the corresponding effect size.

		ΔW (mean±s)				Cliff's
QEMU	ξ	synchronous	non-synchronous	U	p-value	Delta
files	1	87.903±764.969	74.342±318.517	99 455 000	1.00	0.086
	5	78.643±563.798	75.551±326.277	158 850 000	1.00	0.067
	10	72.276±493.614	78.330±349.167	166 410 000	1.00	0.053
	15	72.550±460.570	83.605±378.551	155 570 000	1.00	0.045
features	1	118.545±500.311	130.350±438.243	6 727 800	< 0.05	-0.036
	5	109.036±423.502	146.456±486.597	8 453 900	< 0.05	-0.053
	10	107.703±401.941	158.623±529.273	7 941 400	< 0.05	-0.059
	15	109.363±400.097	161.995±546.241	7 518 300	< 0.05	-0.061

Table 13 One-tailed Mann-Whitney U tests with FDR correction for comparing file-based and feature-based C-bursts (H1). The tests on the left (H1+H1.1) test whether the number of C-bursts per developer pair is higher in the feature-based analysis than in the file-based analysis. The tests on the right (H1+H1.2) test whether the synchronicity degree is higher in the feature-based analysis than in the file-based analysis. U represents the U-statistic of the Mann-Whitney U test, the corresponding p-value indicates whether the corresponding alternative hypothesis is accepted ($p < 0.05$) or not. Cliff's Delta denotes the corresponding effect size.

		H1+H1.1 (# C-bursts per developer pair)			H1+H1.2 (synchronicity degree)		
QEMU	ξ	U	p-value	Cliff's Delta	U	p-value	Cliff's Delta
features vs. files	1	25 891 734 214	< 0.05	0.003	681 840	< 0.05	0.298
	5	25 746 439 556	< 0.05	0.008	3 949 898	< 0.05	0.426
	10	25 612 361 104	< 0.05	0.014	8 290 824	< 0.05	0.486
	15	25 515 286 716	< 0.05	0.017	12 049 580	< 0.05	0.514

Table 14 One-tailed Mann-Whitney U tests with FDR correction for comparing file-based and feature-based C-bursts (H1). The tests on the left (H1+H1.3) test whether the code growth ΔL in synchronous commits is higher in the feature-based analysis than in the file-based analysis. The tests on the right (H1+H1.4) test whether code effort ΔW in synchronous commits is lower in the feature-based analysis than in the file-based analysis. U represents the U-statistic of the Mann-Whitney U test, the corresponding p-value indicates whether the alternative hypothesis is accepted ($p < 0.05$) or not. Cliff's Delta denotes the corresponding effect size.

		H1+H1.3 (code growth in sync. dev.)			H1+H1.4 (code effort in sync. dev.)		
QEMU	ξ	U	p-value	Cliff's Delta	U	p-value	Cliff's Delta
features vs. files	1	7 273 482	1.00	-0.037	7 013 639	1.00	0.000
	5	29 462 292	1.00	-0.009	28 321 505	1.00	0.030
	10	48 855 197	1.00	0.004	46 651 924	< 0.05	0.049
	15	61 288 970	0.30	0.011	58 353 021	1.00	0.058

Table 15 The results of testing H3.1 for different abstraction levels using the upper-bound approach. ✓ denotes that the empirical DTW distances are smaller than the corresponding simulated ones, ✗ the opposite.

	ξ	QEMU messages	QEMU convers.	BusyBox messages	BusyBox convers.	OpenSSL messages	OpenSSL convers.
files	1	✓	✓	✓	✓	✓	✗
	5	✓	✓	✓	✓	✓	✓
	10	✓	✓	✓	✓	✓	✓
	15	✓	✓	✓	✓	✓	✓
features	1	✓	✓	✓	✓	✓	✗
	5	✓	✓	✗	✓	✓	✗
	10	✓	✓	✗	✓	✓	✓
	15	✓	✓	✗	✓	✓	✓

Table 16 The results of testing H3.1 for different abstraction levels using the lower-bound approach. ✓ denotes that the empirical DTW distances are smaller than the corresponding simulated ones, ✗ the opposite. ? denotes that there are insufficient data (no E-bursts at all).

	ξ	QEMU		BusyBox		OpenSSL	
		messages	convers.	messages	convers.	messages	convers.
files	1	✓	✓	✓	✓	✗	?
	5	✓	✓	✓	✓	✗	✗
	10	✓	✓	✓	✓	✗	✗
	15	✓	✓	✓	✓	✓	✗
features	1	✓	✓	✓	✓	?	?
	5	✓	✓	✓	✓	✗	?
	10	✓	✓	✓	✓	✗	?
	15	✓	✓	✓	✓	✗	?

Table 17 The results of testing H3+H3.1 for comparing file and feature level using lower-bound or upper-bound approach respectively. ✓ denotes that the DTW distances using feature-based C-bursts are smaller then the DTW distances using file-based C-bursts, ✗ the opposite. ? denotes that there are insufficient data (no E-bursts at all).

files vs.	ξ	QEMU		BusyBox		OpenSSL	
features		messages	convers.	messages	convers.	messages	convers.
upper-bound approach	1	✗	✗	✗	✗	✗	✗
	5	✓	✗	✗	✗	✗	✗
	10	✓	✗	✗	✗	✗	✗
	15	✓	✗	✗	✗	✗	✗
lower-bound approach	1	✓	✗	✗	✗	?	?
	5	✓	✗	✗	✗	✗	?
	10	✓	✗	✗	✗	✗	?
	15	✓	✗	✗	✗	✗	?

Table 18 The results of testing H3+H3.1 for comparing message and conversation level using lower-bound or upper-bound approach respectively. ✓ denotes that the DTW distances using conversation-based E-bursts are smaller then the DTW distances using message-based E-bursts, ✗ the opposite. ? denotes that there are insufficient data (no E-bursts at all).

messages vs.	ξ	QEMU		BusyBox		OpenSSL	
conversations		files	features	files	features	files	features
upper-bound approach	1	✓	✓	✗	✗	✓	✗
	5	✓	✓	✓	✗	✗	✓
	10	✓	✓	✗	✗	✓	✓
	15	✓	✓	✗	✗	✓	✓
lower-bound approach	1	✗	✗	✗	✗	?	?
	5	✗	✗	✗	✗	✗	?
	10	✗	✗	✗	✗	✗	?
	15	✗	✗	✗	✗	✗	?

References

Ali N, Guéhéneuc Y, Antoniol G (2013) Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links. IEEE Transactions on Software Engineering (TSE) 39(5):725–741

Apel S, Batory D, Kästner C, Saake G (2013) Feature-Oriented Software Product Lines: Concepts and Implementation. Springer

Bacchelli A, D'Ambros M, Lanza M (2010) Are Popular Classes More Defect Prone? In: Proc. Int. Conf. Fundamental Approaches to Software Engineering (FASE), Springer, pp 59–73

Berger T, Guo J (2013) Towards System Analysis with Variability Model Metrics. In: Proc. Int. Workshop on Variability Modeling of Software-Intensive Systems (VaMoS), ACM, pp 23:1–23:8

Berger T, Lettner D, Rubin J, Grünbacher P, Silva A, Becker M, Chechik M, Czarnecki K (2015) What is a Feature?: A Qualitative Study of Features in Industrial Software Product Lines. In: Proc. Int. Software Product Line Conference (SPLC), ACM, pp 16–25

Berndt DJ, Clifford J (1994) Using Dynamic Time Warping to Find Patterns in Time Series. In: Proc. Int. Conf. Knowledge Discovery and Data Mining (KDD), AAAI Press, pp 359–370

Bird C (2011) Sociotechnical Coordination and Collaboration in Open Source Software. In: Proc. Int. Conf. Software Maintenance (ICSM), IEEE, pp 568–573

Bird C, Gourley A, Devanbu P, Gertz M, Swaminathan A (2006) Mining Email Social Networks. In: Proc. Int. Workshop Mining Software Repositories (MSR), ACM, pp 137–143

Bird C, Pattison D, D'Souza R, Filkov V, Devanbu P (2008) Latent Social Structure in Open Source Projects. In: Proc. Int. Symposium on Foundations of Software Engineering (FSE), ACM, pp 24–35

Bird C, Nagappan N, Murphy B, Gall H, Devanbu P (2011) Don't Touch My Code! Examining the Effects of Ownership on Software Quality. In: Proc. Europ. Software Engineering Conf. and the Int. Symposium Foundations of Software Engineering (ESEC/FSE), ACM, pp 4–14

Cannon-Bowers JA, Salas E, Converse S (1993) Shared Mental Models in Expert Team Decision Making. In: Individual and Group Decision Making: Current Issues, Lawrence Erlbaum Associates, chap 12, pp 221–246

Cataldo M, Herbsleb JD (2013) Coordination Breakdowns and Their Impact on Development Productivity and Software Failures. IEEE Transactions on Software Engineering (TSE) 39(3):343–360

Cataldo M, Herbsleb JD, Carley KM (2008) Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. In: Proc. Int. Symposium Empirical Software Engineering and Measurement (ESEM), ACM, pp 2–11

Cataldo M, Mockus A, Roberts JA, Herbsleb JD (2009) Software Dependencies, Work Dependencies, and Their Impact on Failures. IEEE Transactions on Software Engineering (TSE) 35(6):864–878

Crowston K, Howison J (2005) The Social Structure of Free and Open Source Software Development. First Monday 10(2)

Czarnecki K, Eisenecker UW (2000) Generative Programming: Methods, Tools, and Applications. ACM/Addison-Wesley

Draheim D, Pekacki L (2003) Process-Centric Analytical Processing of Version Control Data. In: Proc. Int. Workshop Principles of Software Evolution (IWPSE), IEEE, pp 131–136

Ernst MD, Badros GJ, Notkin D (2002) An Empirical Analysis of C Preprocessor Use. IEEE Transactions on Software Engineering (TSE) 28(12):1146–1170

Espinosa A, Kraut R, Lerch J, Slaughter S, Herbsleb J, Mockus A (2001) Shared Mental Models and Coordination in Large-Scale, Distributed Software Development. In: Proc. Int. Conf. Information Systems (ICIS), Association for Information Systems, pp 513–517

Feigenspan J, Kästner C, Apel S, Liebig J, Schulze M, Dachselt R, Papendieck M, Leich T, Saake G (2013) Do Background Colors Improve Program Comprehension in the #ifdef Hell? Empirical Software Engineering (EMSE) 18(4):699–745

Fenske W, Schulze S, Meyer D, Saake G (2015) When Code Smells Twice as Much: Metric-Based Detection of Variability-Aware Code Smells. In: Int. Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, pp 171–180

Gharehyazie M, Filkov V (2017) Tracing Distributed Collaborative Development in Apache Software Foundation Projects. Empirical Software Engineering (EMSE) 22(4):1795–1830

Guzzi A, Bacchelli A, Lanza M, Pinzger M, van Deursen A (2013) Communication in Open Source Software Development Mailing Lists. In: Proc. Int. Workshop Mining Software Repositories (MSR), IEEE, pp 277–286

Herbsleb JD (2007) Global Software Engineering: The Future of Socio-technical Coordination. In: Future of Software Engineering (FOSE), IEEE, pp 188–198

Herbsleb JD, Grinter RE (1999) Architectures, Coordination, and Distance: Conway's Law and Beyond. IEEE Software 16(5):63–70

Herbsleb JD, Mockus A (2003) Formulation and Preliminary Test of an Empirical Theory of Coordination in Software Engineering. In: Proc. Europ. Software Engineering Conf. and the Int. Symposium Foundations of Software Engineering (ESEC/FSE), ACM, pp 138–147

Hunsen C, Zhang B, Siegmund J, Kästner C, Leßenich O, Becker M, Apel S (2016) Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. Empirical Software Engineering (EMSE) 21(2):449–482

Hunsen C, Siegmund J, Apel S (2020) On the Fulfillment of Coordination Requirements in Open-Source Software Projects: An Exploratory Study. Empirical Software Engineering (EMSE) 25(6):4379–4426

Jermakovics A, Sillitti A, Succi G (2011) Mining and Visualizing Developer Networks from Version Control Systems. In: Proc. Int. Workshop Cooperative and Human Aspects of Software Engineering (CHASE), ACM, pp 24–31

Jiang Y, Adams B, Khomh F, German DM (2014) Tracing Back the History of Commits in Low-tech Reviewing Environments: A Case Study of the Linux Kernel. In: Proc. Int. Symposium Empirical Software Engineering and Measurement (ESEM), ACM, pp 51:1–51:10

Joblin M, Mauerer W, Apel S, Siegmund J, Riehle D (2015) From Developer Networks to Verified Communities: A Fine-grained Approach. In: Proc. Int. Conf. Software Engineering (ICSE), IEEE, pp 563–573

Joblin M, Apel S, Hunsen C, Mauerer W (2017a) Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics. In: Proc. Int. Conf. Software Engineering (ICSE), IEEE, pp 164–174

Joblin M, Apel S, Mauerer W (2017b) Evolutionary Trends of Developer Coordination: A Network Approach. Empirical Software Engineering (EMSE) 22(4):2050–2094

Keogh EJ, Pazzani MJ (2001) Derivative Dynamic Time Warping. In: Proc. Int. Conf. Data Mining (ICDM), Society for Industrial and Applied Mathematics, pp 1–11

Kernighan BW, Ritchie DM (1988) The C Programming Language, 2nd edn. Prentice-Hall

Kwan I, Schroter A, Damian D (2011) Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project. IEEE Transactions on Software Engineering (TSE) 37(3):307–324

Liebig J, Apel S, Lengauer C, Kästner C, Schulze M (2010) An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In: Proc. Int. Conf. Software Engineering (ICSE), ACM, pp 105–114

López-Fernández L, Robles G, Gonzalez-Barahona JM, Herraiz I (2006) Applying Social Network Analysis Techniques to Community-Driven Libre Software Projects. International Journal of Information Technology and Web Engineering (IJITWE) 1:28–50

Mannan UA, Ahmed I, Jensen C, Sarma A (2020) On the Relationship between Design Discussions and Design Quality: A Case Study of Apache Projects. In: Proc. Europ. Software Engineering Conf. and the Int. Symposium Foundations of Software Engineering (ESEC/FSE), ACM, pp 543–555

Medeiros F, Kästner C, Ribeiro M, Nadi S, Gheyi R (2015) The Love/Hate Relationship with the C Preprocessor: An Interview Study. In: Leibniz Int. Proc. in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp 495–518

Mockus A, Fielding RT, Herbsleb JD (2002) Two Case Studies of Open Source Software Development: Apache and Mozilla. ACM Transactions on Software Engineering and Methodology (TOSEM) 11(3):309–346

Oliva GA, Santana FW, de Oliveira KCM, de Souza CRB, Gerosa MA (2012) Characterizing Key Developers: A Case Study With Apache Ant. In: Proc. Int. Conf. Collaboration and Technology (CRIWG), Springer, pp 97–112

Queiroz R, Passos L, Valente MT, Hunsen C, Apel S, Czarnecki K (2017) The Shape of Feature Code: An Analysis of Twenty C-Preprocessor-Based Systems. Software and Systems Modeling (SoSyM) 16(1):77–96

Rabiner LR, Juang BH (1993) Fundamentals of Speech Recognition. Prentice-Hall

Ramsauer R, Lohmann D, Mauerer W (2019) The List is the Process: Reliable Pre-Integration Tracking of Commits on Mailing Lists. In: Proc. Int. Conf. Software Engineering (ICSE), IEEE, pp 807–818

Riehle D (2015) The Five Stages of Open Source Volunteering, Springer, pp 25–38

Rigby PC, German DM, Storey MA (2008) Open Source Software Peer Review Practices: A Case Study of the Apache Server. In: Proc. Int. Conf. Software Engineering (ICSE), ACM, pp 541–550

Sakoe H, Chiba S (1978) Dynamic Programming Algorithm Optimization for Spoken Word Recognition. IEEE Transactions on Acoustics, Speech, and Signal Processing (TASS) 26(1):43–49

Schulze S, Fenske W (2018) Analyzing the Evolution of Preprocessor-Based Variability: A Tale of a Thousand and One Scripts. In: Int. Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, pp 50–55

Scozzi B, Crowston K, Eseryel UY, Li Q (2008) Shared Mental Models among Open Source Software Developers. In: Proc. Hawaii Int. Conf. System Sciences (HICSS), IEEE, pp 306–306

Singh PV (2010) 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 (TOSEM) 20(2):6:1–6:27

Sommerville I (2010) Software Engineering, 9th edn. Addison-Wesley

de Souza C, Froehlich J, Dourish P (2005) Seeking the Source: Software Source Code As a Social and Technical Artifact. In: Proc. Int. Conf. Supporting Group Work (GROUP), ACM, pp 197–206

Storey MA, Singer L, Figueira Filho F, Zagalsky A, German DM (2017) How Social and Communication Channels Shape and Challenge a Participatory Culture in Software Development. IEEE Transactions on Software Engineering (TSE) 43(2):185–204

Toral SL, Martínez-Torres MR, Barrero F (2010) Analysis of Virtual Communities Supporting OSS Projects Using Social Network Analysis. Information and Software Technology (IST) 52(3):296–303

Wiese IS, Teodoro da Silva J, Steinmacher I, Treude C, Gerosa MA (2016) Who is Who in the Mailing List? Comparing Six Disambiguation Heuristics to Identify Multiple Addresses of a Participant. In: Proc. Int. Conf. Software Maintenance and Evolution (ICSME), IEEE, pp 345–355

Wu J, Graham T, Smith PW (2003) A Study of Collaboration in Software Design. In: Proc. Int. Symposium Empirical Software Engineering (ISESE), IEEE, pp 304–313

Xuan Q, Filkov V (2014) Building It Together: Synchronous Development in OSS. In: Proc. Int. Conf. Software Engineering (ICSE), ACM, pp 222–233

Xuan Q, Gharehyazie M, Devanbu PT, Filkov V (2012) Measuring the Effect of Social Communications on Individual Working Rhythms: A Case Study of Open Source Software. In: Proc. Int. Conf. Social Informatics (SocInfo), IEEE, pp 78–85

Xuan Q, Devanbu P, Filkov V (2016) Converging Work-Talk Patterns in Online Task-Oriented Communities. PLOS ONE 11(5):1–20