Noname manuscript No.
(will be inserted by the editor)

On the Fulfillment of Coordination Requirements in Open-Source Software Projects: An Exploratory Study

Claus Hunsen · Janet Siegmund · Sven Apel

Received: date / Accepted: date

Abstract In large-scale open-source software projects, where developers are often distributed across the entire planet, coordination among developers is crucial. To estimate whether a state of *socio-technical congruence* is achieved, which is associated with software quality and project success, we assess the alignment of collaboration and communication in such software projects in terms of *coordination requirements*. By means of an empirical study on a substantial set of large-scale open-source software projects—the development histories of all projects sum up to over 180 years—we aim at shedding light on this issue. To this end, to take a more semantic view on this phenomenon in comparison to previous work, we do not only identify coordination requirements arising from files and functions only, but also those arising from features. We found that open-source developers fulfill coordination requirements intentionally, but mostly those coordination requirements that arise from coupled source-code artifacts, while they resolve simpler ones independently. Furthermore, neither of the considered abstraction levels of source-code artifacts (files, functions, features) is more suitable to construct coordination requirements with respect to their fulfillment. This finding strongly indicates that features do not play an as important role in the development process as expected and commonly believed by the research community in the area of feature-oriented and feature-driven development. Finally, we identified notable evolutionary trends in the fulfillment of coordination requirements and showed that far-reaching social events (such as organizational issues) have a huge impact on their fulfillment,

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

Janet Siegmund
Chemnitz University of Technology
Chemnitz, Germany
E-mail: janet.siegmund@informatik.tu-chemnitz.de

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

both negatively and positively. The key findings of our empirical study are that socio-technical relations are important to understand open-source development communities and that the incorporation of different abstraction levels for developer collaboration does yield important insights to further improve the evolution in open-source software projects.

Keywords coordination requirements · socio-technical congruence · features · social-network analysis · CORONET · CODEFACE · open-source software systems · configurable systems · software product lines · feature-oriented software development

1 Introduction

Developing open-source software heavily relies on the participation and, mostly, voluntary contribution of many developers. To contribute efficiently, developers need to collaborate and coordinate their work with others [15, 43]. Today, open-source software (OSS) development projects, such as QEMU, LLVM, or OPENSSL, are complex and decentralized projects that require large development communities, which are often scattered across the entire planet. Such large and distributed teams make coordination and organization difficult, requiring considerable discipline, especially for communication [41].

To understand whether developers collaborate and communicate sufficiently, researchers have looked at how coordination requirements are fulfilled by communication [18, 65, 100]: A *coordination requirement* arises among two developers if they work on the same parts (e.g., the same files) of the project and, thus, are required to coordinate their work, for example, to resolve interdependencies [65], to avoid duplication of work, or even breaking code [15, 18, 45, 47, 102]. Developers can fulfill their coordination requirements by communicating with each other, for example, by exchanging e-mails or text messages, by talking via phone, or by informal conversations over coffee—basically, everything that enables them to coordinate their work. Cataldo et al. [17, 18, 19] argued that the gap between required coordination and the actual coordination should be low to achieve a state of *socio-technical congruence*. That is, the higher the socio-technical congruence, the better the chances for project success or successful evolution.

In line with these and other studies [1, 10, 11, 53], we are interested in understanding the actual coordination activities among developers—especially, the alignment with coordination requirements. Specifically, a better understanding of coordination requirements allows us to assess and facilitate how shared mental models emerge among developers, which, in turn, would increase the team performance during development [30, 66, 82]. However, going beyond previous studies, we argue it is reasonable to expect that a more *semantic perspective*, such as considering features instead of files, is more suitable as constructional argument for coordination requirements. The background is that, in the past, researchers mainly extracted coordination requirements from purely *technical aspects* of the software projects under consideration. For example, when two developers contributed code to the same file, a coordination requirement was defined among them [17, 18, 50]. Beyond the abstraction level of files, we additionally analyze the abstraction levels of functions and features. *Functions* are also a technical, but more fine-grained

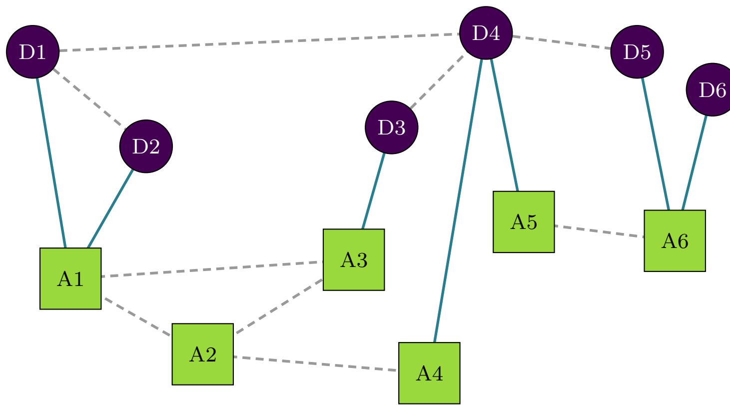


Fig. 1 An exemplary coordination-requirement network. Circles (●) represent developers, dashed edges among developers (●- -●) represent coordination effort. Squares (■) represent artifacts, dashed edges among artifacts (■- -■) represent coupling among the connected artifacts. Developers are connected to artifacts (solid edges, ●—■) if they worked on that artifact in a commit.

aspect of software compared to files and also provide small semantic chunks of code. By analyzing functions, we assume that a file may contain various independent functionality. In addition, we analyze coordination requirements arising from semantically related source code in terms of *features*. Technically, the source code belonging to a feature may be scattered across multiple files, and multiple features may be tangled within one file [3]. We hypothesize that the semantic view on source code that is induced by features reflects the developers' mental model of the software better than files, because features represent and implement requirements, providing the corresponding functionality [3]. Thus, the semantic meaning of features should induce more meaningful coordination requirements and developers should rely on features for communication and coordination.

To test this hypothesis, we conduct an empirical study of ten substantial open-source software projects (including QEMU and LLVM). Specifically, we answer three research questions targeting three different aspects of the state of socio-technical congruence: We investigate on (i) the state of socio-technical congruence for the subject systems by assessing the extent to which coordination requirements are fulfilled, (ii) the level of abstraction for source-code artifacts which aligns best with actual developer communication, and (iii) how the ratios of fulfilled coordination requirements evolve over time. To this end, we identify and analyze coordination requirements arising from different levels of abstraction (i.e., files, functions, and features), which represent different semantic levels or, rather, units of comprehension [51]. We extract coordination requirements from concurrently changed code artifacts (co-changed artifacts) [104], while their fulfillment is indicated by social interaction [94] (i.e., they are fulfilled via the contribution of the involved developers to the same thread on their project's mailing list [93]). To study coordination requirements and their fulfillment, we use a network-analytic approach [63, 98]. Specifically, we construct *coordination-requirement networks* with the following properties (see example shown in Figure 1): We represent developers (●) and source-code artifacts (■) as vertices, and we represent their relationship

as edges. By encoding coordination requirements as *network motifs* (i.e., as patterns to search for in the networks) [90], we can identify whether a coordination requirement is fulfilled or not. For example, both the developers $D1$ and $D2$ work on artifact $A1$ and, thus, give rise to a coordination requirement (●—■—●). In line with previous work [17, 18], we view a high fraction of fulfilled coordination requirements as an indicator for sufficient coordination and sustainable evolution.

In our study, we found three key results: (i) First, the identified coordination requirements of the subject systems are fulfilled *not* by chance. This indicates that the coordination process in open-source systems is guided by the needs in the development process. (ii) Second, the abstraction level does *not* matter, when assessing the fulfillment of coordination requirements. For most subject systems, the fraction of fulfilled coordination requirements is similar across all considered abstraction levels—although the set of identified coordination requirements may vary significantly. To this end, our results indicate that features are not as prominently placed in the development process of our subject systems as expected (especially, not be researchers in the area of feature-oriented software development [3] and feature-driven development [73]). (iii) Third, we found that the coordination in our subject systems changes substantially over time. We identified and discuss a number of far-reaching events that shaped the evolution of subject systems in terms of communication (e.g., the finding of the Heartbleed bug in OpenSSL). (iv) In the end, all our findings emphasize that the socio-technical analysis of software projects is valuable for understanding the dynamics in such projects and for influencing their evolution.

In summary, we make following contributions:

– Based on previous work, we devise an extended formal analysis framework to extract coordination requirements and construct corresponding coordination-requirement networks for the application of network-analytic methods at different levels of abstraction for source-code artifacts (*file*, *function*, and *feature*).
– We apply our analysis framework to mailing-list data and commit data from the history of ten open-source projects to identify (un-)fulfilled coordination requirements over time. Our results indicate that coordination requirements are fulfilled on purpose and do not differ significantly across abstraction levels. Nevertheless, we emphasize that a differentiation among abstraction levels *does* matter as different coordination requirements are identified.
– We reveal factors that may influence coordination in software projects over time (such as public announcement of critical bugs or a change in maintainership) by conducting a qualitative study on the evolution of our subject projects investigating the fulfillment of coordination requirements over time.
– We show that socio-technical analyses can yield useful insights into the communication and coordination of open-source developers and should be employed in further contexts.

All scripts and data are available on our supplementary website:
`https://se-sic.github.io/paper-coordination-requirements/`.

2 A Network Approach

In this section, we provide the necessary background on developer coordination and collaboration. We formalize our approach to construct coordination-requirement

networks, and we show how to encode coordination requirements as networks motifs, which enables us to analyze the coordination-requirement networks in an automated manner at different levels of abstraction.

2.1 Coordination Requirements

Communication for the sake of coordination is crucial for the progress and sustainability of large-scale software projects. However, coordination breakdowns (i.e., failing coordination) occur often and in various forms, and project teams need to spend effort and take precautions to prevent them [17, 25]. In today's open-source software projects, such as the LINUX kernel or QEMU, with decentralized and distributed development teams, coordination is even more important for the project's success [26, 41, 44, 45, 47, 57], but also more difficult to achieve (e.g., due to different time zones or geographical distance) [41]. As a consequence, it is important to know whom to contact [41] and, generally, whom to share knowledge with across the project [25, 43] to achieve the project's goals. In general, proper communication in projects will likely increase the speed and quality of development due to accurate planning and division of tasks [18, 47].

A central idea to guide coordination efforts in a software project is to align the communication with the tasks arising from the project's goals to guarantee progress and purposeful effort achieving a state of *socio-technical congruence* [18, 19, 47]. While Conway hypothesized that this alignment is a result directly arising from the software architecture [22], Parnas proposed the idea of manual modularization, where a module is a "work item" (not necessarily a subprogram) [74]. The basic idea behind both views is that the software artifacts and their decomposition guide the team structure and, consequently, the communication process [22, 26, 40].

As a pragmatic way to observe the state of socio-technical congruence [19], Cataldo and others proposed the notion of *coordination requirements* [18, 65]. If two developers work on the same part of the project (e.g., the same file or function), a coordination requirement arises among them—to fulfill that requirement, they need to communicate. The rationale for studying coordination requirements is that the lack of communication may lead to problems in the development process: To avoid duplication of work, breaking code due to conflicting changes, merge conflicts, and, as a consequence of such problems, project delay in any sense, developers need to resolve interdependencies by coordinating their activities [15, 18, 45, 47, 65]. In other words, developers should aim at fulfilling their coordination requirements to increase the chance for project success or successful evolution [17, 18]. Developers may exchange e-mails or text messages, or communicate via a bug tracker (e.g., using change requests, pull requests, or issues) to fulfill their coordination requirements [17, 18, 44].

In our study, we consider two types of coordination requirements, which can be formalized as network motifs: the triangle motif and the square motif, both illustrated in Figure 2. Both motifs have been studied (implicitly) before [17, 18, 98]. The *triangle motif* resembles the idea that two developers work on the same source-code artifact, so coordination is crucial to avoid conflicts or inconsistencies of any kind. The *square motif* incorporates information on interdependencies among source-code artifacts in addition to the information which developer works on which artifact. For example, in the case that one function relies on another func-

tion, this causes a problem if two developers each work on one of them without coordinating or even without being aware of the other's work.

We discuss the importance of the choice of which abstraction level to choose for the source-code artifacts when deriving coordination requirements in Section 2.4.

2.2 Coordination-Requirement Networks

To analyze the fulfillment of coordination requirements in a software project, we construct *coordination-requirement networks*, which we can analyze with network-analytic methods. We show an exemplary coordination-requirement network in Figure 1. Formally, such a network is defined as an undirected graph $G = (D \cup A, E)$, where we encode *developers* (●, set D) and *artifacts* (■, set A) as vertices; E is the set of edges among the vertices. Following previous work [98], we encode the following three relations in the edges:

Developer–artifact relation (●—■). Developers work on code artifacts while committing to the project's version control system. Such artifacts may be files, functions, or features. E_{DA} denotes the set of edges between developers and source-code artifacts (i.e., developers worked on artifacts): $E_{\mathrm{DA}} = \{\{d, a\} \in E \mid d \in D,\ a \in A\}$. In isolation, the set E_{DA} describes a bipartite graph between developers and artifacts and is the main source for the identification of coordination requirements.

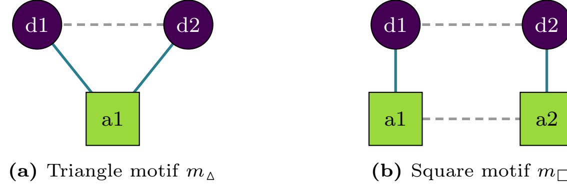
Artifact–artifact relation (■- -■). Artifacts can be related in various ways: For example, two functions can be related if one function calls the other, which is called dynamic coupling [5]; they can contain similar semantic functionality, which is called semantic coupling [78]; or the artifacts are related as they are concurrently changed in a commit or belong to the same change or pull request, which is called logical coupling [18, 35, 104]. Hence, edges in the set E_{AA} represent coupling among artifacts ($E_{\mathrm{AA}} = \{\{a_1, a_2\} \in E \mid a_1, a_2 \in A\}$).

Developer–developer relation (●- -●). Finally, developers may coordinate their work with other developers. In previous studies, researchers analyzed mailing lists [10, 42, 94], IRC [18, 42], bug trackers of any kind [18], or their combination [1, 85] as means for coordination. Accordingly, the set $E_{\mathrm{DD}} = \{\{d_1, d_2\} \in E \mid d_1, d_2 \in D\}$ represents coordination effort among developers and is the main source to assess the fulfillment of coordination requirements.

Ultimately, $E = E_{\mathrm{DD}} \cup E_{\mathrm{AA}} \cup E_{\mathrm{DA}}$. As E is a set with unique entries, G is consequently a simple graph, i.e., it does not contain any loops or multiple edges among any two vertices.

2.3 Network Motifs

To automatically identify coordination requirements in coordination-requirement networks, we encode coordination requirements or, rather, the patterns they represent as *network motifs*. Network motifs are recurrent sub-graphs in a given network [90]. Motifs can be described formally as a set of vertices (e.g., $\{d_1, d_2, a_1\}$) with specific edges connecting them, but are more illustrative and intuitive. We show two network motifs for coordination requirements in Figure 2, the triangle motif and the square motif.



(a) Triangle motif $m_\triangle$ **(b)** Square motif $m_\square$

Fig. 2 *Triangle* and *square motifs.* Edges among artifacts () represent coupling, while a developer is connected to an artifact (), if they worked on that artifact. An edge among two developers () represents coordination, where the edge's existence indicates the fulfillment of the encoded coordination requirement.

The *triangle motif* (Figure 2a) captures the idea that two developers () work on the same code artifact (). The motif applies, for example, in a situation where two developers work on the same file while committing code to the repository, independent of the fact whether they coordinate or not (see above). The corresponding graph representation of this motif is the following: An artifact vertex has two incident edges from two distinct developer vertices. We formally define the motif as

$$m_\triangle = \{d_1, d_2, a_1\}, \text{ where } \{d_1, a_1\} \in E_{\mathrm{DA}}, \{d_2, a_1\} \in E_{\mathrm{DA}}, \text{and } d_1 \neq d_2.$$

The *square motif* in Figure 2b represents a coordination requirement of two developers working on two related code artifacts: There are two developer vertices () connected to two artifact vertices (), while the two artifact vertices are connected by an edge. The square motif applies therefore, for example, in a situation in which two developers work on two distinct functions, where one function eventually calls the other function. Formally, we define

$$m_\square = \{d_1, d_2, a_1, a_2\}, \text{ where } \{d_1, a_1\} \in E_{\mathrm{DA}}, \{d_2, a_2\} \in E_{\mathrm{DA}}, \{a_1, a_2\} \in E_{\mathrm{AA}}, a_1 \neq a_2, \text{and } d_1 \neq d_2.$$

For both motifs, the edge $\{d_1, d_2\} \in E_{\mathrm{DD}}$ () among the developers representing coordination effort may exist or not, based on the premise that the coordination effort occurred within a reasonable time window (see Section 3.2.1 for details). If the edge exists, the coordination requirement is fulfilled; otherwise, it is not.[1]

[1] We could define "positive" and "negative" motifs to capture the fulfillment of coordination requirements directly; but to keep the analysis simple, we define only motifs for the coordination requirements as such and analyze whether the identified coordination requirements are fulfilled or not.

Looking at the example given in Figure 1, there are two fulfilled coordination requirements: $\{D1, D2, A1\}$ (triangle motif $m_{\triangle}$) and $\{D4, D5, A5, A6\}$ (square motif $m_{\square}$). Regarding unfulfilled coordination requirements, there are: $\{D5, D6, A6\}$ ($m_{\triangle}$), $\{D1, D3, A1, A3\}$ ($m_{\square}$), $\{D2, D3, A1, A3\}$ ($m_{\square}$), and $\{D4, D6, A5, A6\}$ ($m_{\square}$), as all sets of vertices miss the edge indicating coordination among the involved developers.

2.4 Levels of Abstraction

In previous work [12, 17, 18], researchers have tracked concurrent or simultaneous contributions of developers on the same *file* to derive coordination requirements. We conjecture that this view may be too technical to capture the richness of coordination. Thus, we use the two code-artifact abstractions *function* and, most importantly, *feature* to infer coordination requirements at different levels of abstraction. Although such abstraction levels are based on heuristics, some have been shown to be reliable in multiple previous studies [12, 50, 51, 63, 67]. Additionally, with regard to network construction, the abstraction *file* has been shown to produce dense networks that often hinder community detection [14, 50]. It has been already shown that a function-level view is more accurate [51]. While the function-level view has been used in previous studies [12, 51] (e.g., by live-tracking editing actions in an IDE), we revisit it for long-term analysis of coordination requirements based on commit information (see Section 4.1 for details).

Functions are also a technical, but more fine-grained aspect of software structure compared to files. Functions contain semantic chunks of code as they represent abstractions that modularize a dedicated functionality for re-use. To this end, when two developers working on the same function, they likely work on the same functionality in the source code. The inferred coordination requirements in this scenario are more specific than the ones derived at the abstraction level of files, as they exclude developers working on different functions in the same file.

Furthermore, *features* provide a semantic, behavior-driven view on a software system: Features represent and implement requirements, providing the corresponding functionality, often implemented in a cross-cutting manner [3]. They closely resemble the developers' shared mental model of the project structure, because they act as an interface between developers and implementation [30]. Using concurrent change on features as constructional argument, we assume to obtain more precise coordination requirements.

For illustration, we show how strongly the choice of abstraction level influences the extraction of coordination requirements by means of the triangle motif and its manifestation in the source-code excerpt listed in Figure 3. The source code contains two files (db.c, actions.c), four functions (persist, execute, delete, and lockOnAction), and two features (PERSIST and LOCKING, implemented with the help of the C preprocessor via #ifdef directives). In file db.c (Figure 3a), the functionality of a database connection is provided, while, in file actions.c (Figure 3b), the functionality of deleting data and locking is implemented. Over time, three developers (Dev A, B, and C) change parts of the code (illustrated by patches and boxes with the developer's name) due to changed requirements or bug fixes. Based on the performed changesets, coordination requirements arise at different abstraction levels, which we illustrate in Figure 4. Regarding co-changes on the same file, both

```
1  #ifdef PERSIST                                              Dev A
2  -void persist() {
3  -    // old code
4  -}
5  +void persist(char *filename) {
6  +    // completely re-written new code
7  +}
8  #endif
9
10 void execute(struct DBConn *conn,
11              struct DBAction *action) {                     Dev B
12 -    // old code
13 +    // code with bugfixes
14 }
```

(a) File db.c

```
1  void delete (struct DBconn *conn, char *command) {
2       // ...                                                  Dev C
3  +#ifdef PERSIST
4  +    persist(Config.filename);
5  +#endif
6  }
7
8  #ifdef LOCKING
9  void lockOnAction(struct DBAction *action) {                Dev C
10 -    // old code
11 +    // code with changes
12                                                             Dev B
13 -    if(data == NULL) {
14 +    if(data != NULL) {
15      // ...
16 }
17 #endif
```

(b) File actions.c

Fig. 3 Code example containing two files, four functions, and two features (controlled by #ifdef directives). Lines starting either with + or − indicate patch blocks applied by an developer, which are additionally annotated with the name of the responsible developer.

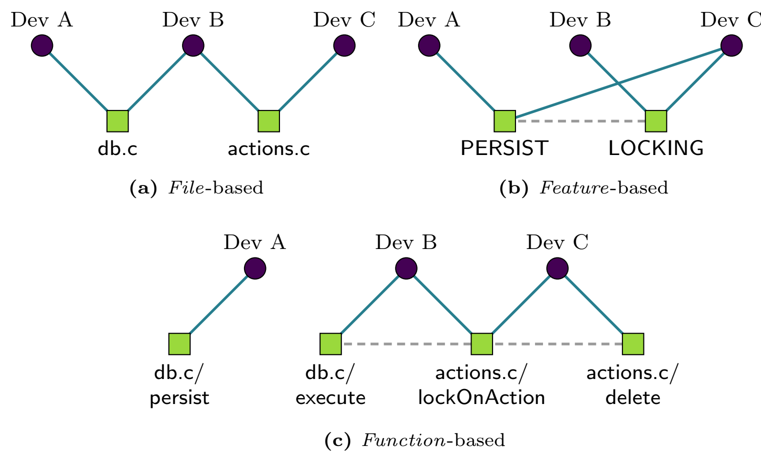


Fig. 4 Coordination-requirement networks (excluding coordination edges) extracted from the source code in Figure 3 for each of the abstraction levels *file*, *function*, and *feature*.

Devs A and B as well as Devs B and C should coordinate their work according to the triangle motif (Figure 4a), while, for functions (Figure 4c), only Devs B and C need to coordinate, as they both change the function `lockOnAction`. As illustrated in Figure 4b, the coordination requirements for features resemble the need for coordination arising from the actual performed changes in the code more naturally: Dev C relies on the functionality of the function `persist` of the feature `PERSIST`, which is completely changed by Dev A. The implied coordination requirement for the Devs A and C has not been identified for any of the other abstraction levels. Additionally, the other important coordination requirement (Devs B and C) is also identified. Thus, the identification of coordination requirements on more fine-grained and more semantically meaningful levels of abstraction seems reasonable.

3 Study Design

In this section, we provide details on our study design for analyzing the fulfillment of coordination requirements at different levels of abstraction. Based on our research questions and operationalization (Sections 3.1 and 3.2), we develop hypotheses in Section 3.3. In Section 3.4, we discuss the criteria for the selection of subject projects.

3.1 Research Questions

The overarching goal of our study is to understand coordination in open-source software projects. To this end, we extract coordination requirements and determine whether they are fulfilled or not using our network approach. In particular, we aim at answering the following research question, which is inspired by previous studies on coordination requirements [17, 18, 19]:

> **Research Question $\mathbf{RQ_1}$.** *Does developer communication align with artifact-based coordination requirements in real-world open-source software projects, such that the coordination requirements are fulfilled?*

Even more importantly, when using different levels of abstraction for source-code artifacts (i.e., files, functions, features) in our analysis, we are able to assess their semantics: The arising coordination requirements may vary significantly across abstraction levels. In Section 2.4, we specifically argued that the semantic abstraction level of features is a more precise means for studying coordination requirements. Thus, we formulate our second research question:

> **Research Question $\mathbf{RQ_2}$.** *Does developer communication align better with feature-based coordination requirements than with function-based or file-based coordination requirements?*

It is reasonable to assume that the coordination efforts in a project evolve and, eventually, mature while the project matures itself. Consequently, when considering the evolution of the considered software projects into our analysis, we can observe whether the communication also evolves, which means that the ratios

of fulfilled coordination requirements change over time, rendering socio-technical congruence a dynamic state [4, 8, 18, 84]. Especially, when looking at the different levels of abstraction regarding source-code artifacts, we potentially gain important insights on how the shared mental model of developers evolves along with the coordination requirements.

Research Question RQ$_3$. *Does the degree of fulfillment of coordination requirements change for different levels of abstraction during project evolution?*

3.2 Experiment Variables and Formal Framework

We provide an overview of the dependent and independent variables of our analysis framework (and our study) as well as their operationalization in Table 1. Our framework is based on previous work (e.g., Cataldo et al. [18, 19], Valetto et al. [98], and Kwan and Damian [55]; for more details, see Section 8), but provides a more formal and holistic approach to the analysis of coordination requirements (for example, a formal definition of all configurable relations and motifs, while directly incorporating the evolutionary perspective and, most importantly, different artifact abstraction levels), enabling easier configuration of future studies and also easier comparison of existing studies.

3.2.1 Independent Variables

The independent variables of our study are the choices that we have for our data extraction and network-construction procedures: (i) the length of the time window to consider for network construction and the identification of coordination requirements, (ii) the kind of developer–developer relation representing coordination efforts, (iii) the abstraction levels of artifacts to consider in the analysis, (iv) the kind of artifact–artifact relation representing coupling among artifacts, and (v) the kind of network motif to search for in coordination-requirement networks.

First, we consider only coordination requirements among developers if their concurrent activities occur within a reasonable *time window*. We consider a coordination requirement fulfilled if there is coordination effort among the developers involved also within the same time window. As a consequence, we consider only data from a specific time window for the construction of a coordination-requirement network and, thus, obtain several time-consecutive networks for a single subject system. While similar results have been found for one-month time windows [67], we use three months of time for such time windows, since this represents the sweet spot between sufficient data and avoiding overfitting [51, 52, 53, 67].

As *developer–developer relation*, we consider contributions of the developers to their project's mailing list: In line with the work by Bird et al., we assume that two developers coordinate their work if and only if they contribute to the same thread on the mailing list [10]. For that reason, we only analyze projects in our study that have a "strong" and central mailing list, which means that the mailing list is the primary way of communication and coordination in the project (see Section 3.2.3), which is supported by previous studies on mailing-list-based developer

networks [42, 53, 85, 94, 102]. Consequently, we can conclude that, with e-mails as a preferred channel of communication for developers [93] and "strong" mailing lists, concurrent contribution to threads on the mailing list are meaningful as means of coordination [102].

We consider three *abstraction levels for artifacts*: *files*, *functions*, and *features*. Files and functions are encoded by their project-relative path to avoid ambiguities and can be extracted directly from commit data. In detail, we assess the line numbers to map the changes in a commit to respective functions. We extract information on features by means of the C preprocessor that we compute with the help of CODEFACE and CPPSTATS. More detailed information on the extraction of feature information is given in Section 4.

For the *artifact–artifact relation*, we consider co-changes to describe logical coupling among artifacts. The term "co-changes" refers to artifacts that are concurrently changed in a single commit in the project's version control system. Co-changes have been used as a means to represent relations among code artifacts in various studies before [11, 18, 35, 51, 87, 100, 104] (by assessing commit information or merge requests, for example) and are considered to have more impact than call or data dependencies [19].

Table 1 Independent and dependent variables of our analysis framework, along with their corresponding description and the considered levels.

Variable	Description	Levels
	Independent variables	
Length of time window	Concurrent work and coordination effort within this time window induce coordination requirements and their fulfillment	3 months
Developer–developer relation *(coordination effort)*	Developers who coordinate fulfill their existing coordination requirements	E-mail exchange in the same thread
Code artifacts *(abstraction levels)*	Type of code artifact that is the constructional argument for coordination requirements	Files, functions, features
Artifact–artifact relation *(coupling relation)*	Coupling among code artifacts considered in the square motif $m_\square$: logical coupling arising from concurrently changing artifacts in a commit *(co-change)*	Co-change
Network motifs *(model for coordination requirements)*	Patterns of coordination requirements describing concurrent work on the same artifact *($m_\triangle$)* or coupled artifacts *($m_\square$)*	Triangle motif $m_\triangle$, square motif $m_\square$
	Dependent variables	
Fraction of fulfilled coordination requirements $frac_{\mathrm{cr}}(a,m) = \lvert cr_{\mathrm{full}}(a,m)\rvert \,/\, \lvert cr_{\mathrm{found}}(a,m)\rvert$	Indicator for "good" socio-technical congruence; parameters: code-artifact type a as constructional argument for coordination requirements and motif m to match coordination requirements in networks (see Section 3.2 and Equation 1)	ranges from *0* (bad) to *1* (good)

We use the motifs $m_{\triangle}$ and $m_{\square}$ as *network motifs* that encode coordination requirements, as defined and described in Section 2.3. A coordination requirement matched by either motif is labeled as fulfilled if the coordination edge in the motifs is present, otherwise, it is unfulfilled.

We give the more technical details on how to extract the described data for a project in Section 4.

3.2.2 Dependent Variables

To analyze the alignment of the email-based developer coordination and the actual artifact-based collaboration, we measure the *fraction of fulfilled coordination requirements.* Given a coordination-requirement network constructed using one type of code artifact (i.e., *file*, *function*, or *feature*) and a motif m to identify coordination requirements, we define the fraction $frac_{\text{cr}}(a, m)$ of fulfilled coordination requirements as follows:

$$frac_{\text{cr}}(a,m) = \frac{|cr_{\text{full}}(a,m)|}{|cr_{\text{found}}(a,m)|}, \text{ where} \tag{1}$$

$$\begin{aligned} cr_{\text{found}}(a,m) &= \{c \mid \text{matched instance } c \text{ of motif } m \text{ for artifact } a \\ &\qquad \text{in the current network}\}, \\ cr_{\text{full}}(a,m) &= \{c_f \mid c_f \in cr_{\text{found}}(a,m),\ c_f \text{ is fulfilled}\}. \end{aligned}$$

$cr_{\text{found}}(a, m)$ represents the set of identified coordination requirements for the given artifact (i.e., the list of matched motif instances) and $cr_{\text{full}}(a, m)$ is a subset of $cr_{\text{found}}(a, m)$ with only the fulfilled coordination requirements. Evidently, the range of the fraction $frac_{\text{cr}}(a, m)$ is $[0, 1]$, where 0 describes that not a single identified coordination requirement is fulfilled and 1 that all have been fulfilled. Note that the artifact–artifact relation (only effective for the square motif $m_{\square}$) is fixed to “co-change” in our study and, consequently, is directly encoded in the given coordination-requirement networks.

3.2.3 Selection Criteria

Finally, we select specific software projects from the list of potential candidates: All projects use Git as version-control system, C as programming language, and the C preprocessor for features and their configuration. This way, we increase internal validity. The selected projects are developed in a distributed manner (i.e., developers are often scattered across the entire planet) and are diverse in their application domain (including the embedded-systems domain, database management systems, servers, and a version control system). Furthermore, we select only software projects with a long history of mailing-list-based communication. Finally, the selected projects exhibit different patterns in commits per developer and commits per analyzed artifact abstraction.

Regarding *features*, we stick to software systems that heavily use #ifdef directives to implement feature-specific functionality: In line with previous studies [49, 61], we consider C macros that occur in #ifdef conditions inside a project’s source code and that are, hence, used in the sense of configuration constants to implement variability.

We discuss threats to validity arising from the selection criteria in Section 7.

3.2.4 Null Model

We evaluate the identified fulfillment of coordination requirements against a null model to control for the probability of obtaining results by chance and also to assess the validity of our results. If we find a statistically significant difference between the empirical coordination-requirement networks and their equivalent null models in terms of fulfillment of coordination requirements, we can conclude that it is improbable that the topological structure of the empirical network arose from a random process. In other words, if there is a significant difference, it is more likely explained by an organized process, such as a communication effort guided by coordination requirements.

We construct the null-model networks for a subject system for each time window and abstraction level of source-code artifacts that we analyze. For this purpose, we conduct a standard approach [37, 68] also used in previous studies [11, 51], where vertices are connected uniformly at random—under the condition that the degree distribution is identical to the empirical network. For this purpose, we employ a rewiring procedure that maintains the amount of communication (i.e., number of edges) for each developer, but ignores the idea that each communication effort may be guided by a real-world coordination requirement [8].

Most importantly, for constructing the null-model networks, we need to carefully respect all kinds of relations in our coordination-requirement networks separately, as described in the following.

Developer–artifact relation. Developers perform well-defined tasks on various code artifacts which they add to the repository by means of commits. Accordingly, randomizing these edges in the networks would destroy the semantic meaning of changes to the repository, thus, we do not change the edges within this relation for the null model.

Artifact–artifact relation. As the code artifacts and their coupling is defined by the source code itself, we do not randomize this part of the coordination-requirement networks for the generation of null models.

Developer–developer relation. Our main idea for the null model is that the coordination effort among developers can occur randomly and, hence, is not driven by present coordination requirements. Consequently, we randomize the corresponding edges among all developers while keeping the degree distribution in the network stable [37], so that the amount of coordination effort and the topology stay the same in the null-model networks compared to the empirical networks.

Technically, we construct the coordination-requirement networks by constructing the individual networks for the different relations before merging them to a single coordination-requirement network. This way, we are able to add any transformation step to any independent relation-related network before merging: In our analysis, we randomized the edges of the developer–developer network. For technical details, see Section 4.3.

In the following, to refer to the null-model data and results, we mark the metrics defined in Formula 1 (see Section 3.2.2) as $cr_{\text{found}}^{\text{null}}$, $cr_{\text{full}}^{\text{null}}$, and $frac_{\text{cr}}^{\text{null}}$, respectively.

In addition to validating our results against a null model as described above, we perform a sensitivity analysis following Kossinets [54] to investigate the stability of our results in the context of incomplete or missing information sources. For more details and the corresponding results, please see Section 5.2.

3.3 Hypotheses

We formulate four hypotheses that capture our assumption of a "good" alignment of coordination and actual collaboration, all of which are also supported by the literature (e.g., [8, 17, 18, 19]).

The rationale of our first hypothesis is straightforward: Who works on the same artifacts in the project repository should coordinate. The idea is that a high fraction of fulfilled coordination requirements is an indicator for a good socio-technical congruence [19]. On the other side, unrelated and independent co-changes on the same files, functions, or features may cause major conflicts in the code base, so coordination is necessary. We (and others [52]) assume that the probability that two collaborating developers coordinate via the mailing list (i.e., $frac_{\text{cr}}(a, m_{\triangle})$) is likely high for healthy and well-working projects that put a high emphasis on the mailing list (as is the case in our subject projects). Consequently, a high number of identified coordination requirements is likely fulfilled—such that the fulfillment is not just a product of random processes (e.g., noise), such as the null model defined in Section 3.2.4. Following research question RQ_1, our first hypothesis is:

Hypothesis $\mathbf{RH_1}$. *In comparison to the null model, a significantly higher number of developer pairs collaborating on the same artifacts* do *exchange e-mails on the same threads of the mailing list to fulfill arising coordination requirements.*

$$frac_{cr}(a, m) > frac_{cr}^{null}(a, m), \text{ with } a \in \{file, function, feature\} \text{ and } m \in \{m_{\triangle}, m_{\square}\}$$

Developers may be aware of other developers working on the same artifact, but they may not be aware of which artifacts are coupled with the artifacts they work on. Consequently, more coordination requirements with the triangle motif $m_{\triangle}$ might get fulfilled during mailing-list conversation than coordination requirements with the square motif $m_{\square}$.

Hypothesis $\mathbf{RH_2}$. *The fraction of fulfilled coordination requirements is lower for the square motif than for the triangle motif, independent of the observed artifact abstraction a.*

$$frac_{cr}(a, m_{\square}) < frac_{cr}(a, m_{\triangle})$$

Following research question RQ_2 and the discussion in Section 2.4, we assume that developers coordinate based on their shared mental model of the software project [30], which is resembled closely by the semantic concept of *features*. Features are used in the source code to represent and trace requirements of a user to

the software and they provide the corresponding functionality [3].[2] Therefore, the abstraction level of features makes them generally a better user-perceivable unit in the software project and, thus, we hypothesize that features are a more precise abstraction level to reason about coordination requirements. In contrast, *functions* are likely a too fine-grained unit of comprehension and may only be suitable to coordinate on if a possible error has been traced back to a specific function. In the same vein, *files* are likely too large and unstructured to include them as a whole into the coordination process as a standalone unit of comprehension.

Hypothesis RH₃. *The fraction of fulfilled coordination requirements differs for the different levels of abstraction, and is highest for the abstraction level of features.*

$$frac_{cr}(feature, m) > frac_{cr}(file, m) \text{ and } frac_{cr}(feature, m) > frac_{cr}(function, m), \text{ where } m \in \{m_\triangle, m_\square\}$$

Over time, the developers of a project tend to take over more responsibility (e.g., they become a subsystem maintainer) and also develop a more structured way to coordinate their work with the other developers [52], although their mental models may diverge due to the emergence of experts [59, 88]. Following from research question RQ_3, we argue that a consequence of this is that more coordination requirements get fulfilled in later stages of development, when the project and also the developers mature. A reason may be, when the project matures, the maintainers may publish a clear guideline on how to contribute and get in touch with the correct project member for a specific task. This way, potential chaos in developer coordination may stabilize or even decrease over time.

Hypothesis RH_4. *In later stages of development, the fraction of fulfilled coordination requirements $frac_{cr}(a, m)$ is higher than in earlier stages for all motifs $m \in \{m_\triangle, m_\square\}$ and artifacts $a \in \{file, function, feature\}$.*

3.4 Subject Projects

For our study, we selected ten projects using the selection criteria described in Section 3.2.3: Apache HTTP, BusyBox, FFmpeg, Git, LLVM, OpenSSL, PostgreSQL, QEMU, U-Boot, and Wine. We considered all commits and e-mails until 2016 (or early 2017), making up more than 12 years of project history for each project and over 180 years in total for all projects. In total, we analyzed roughly 600,000 commits and roughly 2,600,000 e-mails across all projects. Over time, more than 55,000 distinct developers contributed to the projects, either via e-mail or by committing to the respective repositories. We summarize the details on the subject projects in Table 2.

Furthermore, our subject projects exhibit a great diversity in terms of the number of commits per developer and commits per source-code artifact. On average, developers created 23.50 to 1,048 commits each, with maxima up to 26,580, while the distributions per subject project are heavily skewed with few developers creating a high number of commits. Similar properties hold for the number of

[2] There are various techniques to implement features, we analyze preprocessor annotations. More details in Section 4.1.

commits created per artifact abstraction: The number of commits ranges from 7 to 38 for files, from 2 to 5 for functions, and from 5 to 86 for features, all on average. All distributions are heavily skewed, with few source-code artifacts changed in a high number of commits. These data indicate that the contribution patterns of our subject projects differ significantly, corroborating our selection criteria and, thus, our analysis. We present more detailed statistics with regard to the number of commits on our supplementary website.

The subject projects have a strict policy that developers need to send their patches for review purposes to the mailing list first[3]—that is, the project forces developers to coordinate their work with others before it is accepted. In detail, in the review process, the relevant developers are asked to take part in the mailing-list discussion to fulfill inherent coordination requirements by either identifying potential problems with the proposed patches or accepting the patch.

4 Data Extraction and Study Execution

For data extraction, we mainly use the tool Codeface (Section 4.1). Based on the Codeface results, we construct and analyze coordination-requirement networks using our network-construction library coronet (Section 4.3). All scripts and data are available on our supplementary website.

4.1 Project Analysis using Codeface

We use Codeface[4] to perform commit and mailing-list analyses for any given software project [51]. Codeface stores all results in a linked database, so that we can extract the data to perform further analyses.

Codeface's *commit analysis* yields the following information for each commit: (i) the commit hash and further meta-data, (ii) the commit author, and (iii) the list of artifacts (*file*, *function*, *feature*) that have been touched by the commit author.

There are three points to note regarding the extraction of artifact data: (1) For the artifacts *function* and *file*, the artifact is defined by its relative path in the software project to avoid ambiguities. For example, the function delete in Figure 3b is encoded as actions.c/delete.[5] (2) To identify C-preprocessor annotations and features induced by their presence conditions, the tool cppstats[6] [49, 61] is used internally by Codeface (see also Section 4.2 for more details). (3) We limit the developer–artifact data to files that are implementation-related, so header files, documentation files, and build files are not considered in our study. This is necessary as the function-related and feature-related analyses inside Codeface do not work on non-implementation files and, thus, would cause an imbalance among the different abstraction levels.

Running the mailing-list analysis of Codeface, we obtain all meta-data for each e-mail on the mailing list of a given project: (i) the sender of the e-mail,

[3] E.g., see http://wiki.qemu.org/Contribute/SubmitAPatch (accessed: 2018-11-05).

[4] http://siemens.github.io/codeface/

[5] In our analysis implementation, we use "::" as separator, but for readability reasons, we write "/" in this paper.

[6] http://fosd.net/cppstats/

Table 2 List of subject projects, combined with their empirical data on fulfilled coordination requirements.

Project	Time	# Commits	# E-Mails	# Developers	Artifact	# Artifacts	Triangle motif $m_{\triangle}$			Square motif $m_{\square}$		
							cr_{found}	cr_{full}	$frac_{\text{cr}}$	cr_{found}	cr_{full}	$frac_{\text{cr}}$
Apache HTTP	1996–2017	29,704	54,921	2,146	*file*	1,366	5,710	2,818	0.49	316,417	134,993	0.43
					function	16,869	2,830	1,612	0.57	173,364	121,281	0.70
					feature	1,357	654	322	0.49	7,224	4,380	0.61
BusyBox	1999–2016	14,313	42,013	2,736	*file*	1,370	2,021	907	0.45	181,236	69,771	0.38
					function	10,942	1,661	756	0.46	164,853	29,122	0.18
					feature	2,534	670	312	0.47	18,384	3,407	0.19
FFmpeg	2000–2017	80,605	242,295	5,998	*file*	3,257	42,978	18,277	0.43	4,030,726	1,886,168	0.47
					function	33,078	19,303	8,658	0.45	1,154,399	438,927	0.38
					feature	2,079	5,742	2,369	0.41	110,600	51,972	0.47
Git	2005–2017	34,898	338,500	9,246	*file*	1,740	13,690	4,798	0.35	240,103	77,443	0.32
					function	11,937	7,151	2,473	0.35	128,572	39,928	0.31
					feature	175	185	119	0.64	209	177	0.85
LLVM	2001–2017	158,562	706,716	6,407	*file*	5,619	119,428	42,219	0.35	9,974,743	3,832,917	0.38
					function	50,201	17,647	8,457	0.48	1,007,912	571,735	0.57
					feature	937	8,495	2,432	0.29	26,508	9,420	0.36
OpenSSL	1998–2016	18,143	32,659	4,786	*file*	1,444	5,295	1,922	0.36	712,756	307,113	0.43
					function	12,941	3,044	1,195	0.39	183,091	88,851	0.49
					feature	1,132	3,445	1,134	0.33	153,236	68,464	0.45
PostgreSQL	1996–2017	44,062	320,711	4,647	*file*	2,192	21,798	16,671	0.76	5,147,123	4,216,000	0.82
					function	34,960	14,237	10,797	0.76	3,193,322	2,551,078	0.80
					feature	1,061	1,764	1,339	0.76	19,753	13,158	0.67
QEMU	2003–2016	46,633	430,561	7,205	*file*	3,227	36,467	23,096	0.63	3,466,452	2,486,040	0.72
					function	57,955	15,394	10,365	0.67	1,166,730	810,775	0.69
					feature	1,753	13,892	5,433	0.39	156,289	59,658	0.38
U-Boot	1988–2017	44,736	319,160	7,924	*file*	8,257	11,096	5,755	0.52	307,816	185,906	0.60
					function	63,067	4,664	2,680	0.57	162,257	100,415	0.62
					feature	7,065	20,711	6,931	0.33	423,328	147,434	0.35
Wine	1993–2017	121,815	111,333	4,087	*file*	5,568	45,088	20,019	0.44	1,463,843	817,654	0.56
					function	164,073	23,665	12,506	0.53	1,431,211	818,966	0.57
					feature	1,687	6,348	2,328	0.37	32,501	12,884	0.40

(ii) the date and the time-zone offset, (iii) the subject, and (iv) the thread ID. While the first three parts of information are extracted directly from each e-mail, we extract thread information based on the `In-Reply-To` and `References` fields in the e-mail headers. Each identified thread of e-mails is assigned a unique number.

4.2 Feature Identification

We concentrate on C-preprocessor annotations (#ifdefs) as a central mechanism for implementing configurable features. That is, based on the specific set of configured configuration constants (macros) used for preprocessing and compilation, one can obtain different compiled programs or, more accurately, variants of the subject system. Configuration constants can control whether annotated functionality is included (features, in this sense), decide among hardware-specific characteristics, and may affect other options. In this study, we do not distinguish between these different uses of configuration constants (i.e., kinds of features; see also our previous study [49]).

Shared code among features is identified by incorporating tangling and nesting information when analyzing #ifdefs. The tool CPPSTATS, which we use, rewrites all nested #ifdefs as separate #ifdef blocks, each with a condition that conjoins their own conditional expression with the enclosing ones. As a consequence, we are able to attribute shared feature code to the features involved and, thus, distinguish them from two separate and perfectly modularized pieces of feature code.

While this definition of features is one among many (see Berger et al. [7]) and, thus, possibly threatens validity, as we discuss in Section 7, the variability implemented using preprocessor annotations is a major source for feature implementation in our subject projects and is still widely used in open-source software projects implemented using the programming language C [49, 61].

4.3 Network Construction and Motif Analysis

For the construction of the coordination-requirement networks based on the collected data from Section 4.1, we encode the three types of relations in the network: developers–artifact relations, artifact–artifact relations, and developer–developer relations. We perform the encoding of the relations for each revision range that we consider as described in the following.

Using the data of CODEFACE's commit analysis, we can directly construct both the developer–artifact relations (which developer touched which artifact in a commit) and artifact–artifact relations denoting co-change (which artifacts change concurrently in a commit), as the desired information for the edges is directly present in the data. Using the e-mail meta-data from CODEFACE, we are able to construct developer–developer relations for our coordination-requirement networks: We consider developers to coordinate their work if they send e-mails to the same thread in the same revision range. In line with previous work [72, 104], we skip any e-mail thread or commit information that would create more than

1 million edges in a coordination-requirement network, as they would likely stem from non-implementation-related changes [48].[7]

For data processing and network construction, we use the statistical-computing language R [80] and our network-construction library CORONET[8]. Internally, we use the package IGRAPH [24] for network representation and analysis: We use its functionality to construct the individual networks for each relation defined in Section 3.2, their merge to coordination-requirement networks, and any subsequent network-analytic measures. This also includes motif matching: We define the motifs as small networks directly arising from the sets of vertices $m_\triangle$ and $m_\square$ and the inherent set of edges, as defined in Section 2.3. The small networks representing the coordination requirements can be passed to IGRAPH's function igraph::subgraph_isomorphisms to obtain results (i.e., motif matchings), for a given coordination-requirement network. To construct the null-model networks (see Section 3.2.4), we use the R package BIREWIRE [39] and its predefined function birewire.rewire.undirected for edge randomization that is needed before merging.

For both the commit and the mailing-list handling, we use three-month time ranges, called *revision ranges*, which are considered appropriate in previous work [51, 53, 67] and can be used as a basis for a historical analysis. This way, CORONET divides the projects' version history into windows of three months each, starting with the very first commit. In general, each revision range is defined by two commit hashes (i.e., start and end) and their associated author date. Using the same division into revision ranges, also the mailing-list data of our subject projects are split into three-month revision ranges by assigning e-mails from the mailing list to a revision range based on their date. All e-mails not falling into a revision range are omitted by the network-construction algorithm and, thus, our analysis. We also omit all revision ranges at the start and the end of the subject projects' considered time frame, for which no e-mails are available. At the end, we obtain commit data and mailing-list data grouped by revision range, which enables us to construct coordination-requirement networks with the help of CORONET for consecutive time frames to analyze our subject projects' evolution, instantiating the time-window variable of our study (see Section 3.2).

5 Results

In this section, we present our results, guided by the research hypotheses presented in Section 3.3. In the plots (but not the statistics and tables) that follow, we will focus on the triangle motif $m_\triangle$; the complete set of data and plots is available on the supplementary website.

We discuss data relevant for our hypotheses mainly by means of violin plots and bar plots, along with the results of our statistical analysis. All data presented in this section are characterized either by absolute values or arithmetic mean values and standard deviations ($a \pm s$).

[7] As a simple scenario, changing indentation from tabs to spaces in 1,415 files at once gives rise to more than 1 million edges representing logical coupling among these files. However, such far-spreading changes are likely *not* functionality changes [48], so we aim at reducing their impact by omitting them during network construction.

[8] `http://github.com/se-passau/coronet/`

5.1 Basic Statistics

In general, the extracted data for all hypotheses are shaped as follows: For each subject project and each of their analyzed three-month revision range, we assess the statistics defined in Section 3.2.2: the number of identified coordination requirements (cr_{found}), the number of fulfilled coordination requirements (cr_{full}), and the fraction of fulfilled coordination requirements ($\mathit{frac}_{\text{cr}}$). We collect each of these statistics for each of the defined network motifs ($m_{\triangle}$ and $m_{\square}$) and each artifact (*file*, *function*, and *feature*).

When considering the Hypotheses RH_1, RH_2, and RH_3—for which we view the data on a subject-project level—, we sum up cr_{found} and cr_{full} while grouping the data by subject project, motif, and artifact. Consequently, we need to recalculate the $\mathit{frac}_{\text{cr}}$ value to obtain a single value for each subject project. For the Hypothesis RH_4, we use the raw data and do not perform any aggregation.

In Table 2, we provide the collected raw data of our analysis (on the right side of the table).

5.2 Hypothesis RH_1: Are Most Collaborations Coordinated on Purpose?

> **Hypothesis RH_1.** *In comparison to the null model, a significantly higher number of developer pairs collaborating on the same artifacts* do *exchange e-mails on the same threads of the mailing list to fulfill arising coordination requirements.*
>
> $\mathit{frac}_{cr}(a, m) > \mathit{frac}_{cr}^{null}(a, m)$, *with* $a \in \{\mathit{file}, \mathit{function}, \mathit{feature}\}$ *and* $m \in \{m_{\triangle}, m_{\square}\}$

For RH_1, we need to determine $\mathit{frac}_{\text{cr}}$ for each motif and artifact (see also Section 5.1)—for both the empirical data and the null-model data. On the data that we use for each of the violin-plot pairs in Figure 5, we perform a paired two-sample Wilcoxon signed-rank test[9] to determine the inequality of the empirical distributions' mean to the null-model distributions' mean, independently for each artifact a and motif m (statistical null hypothesis H_0: $\mathit{frac}_{\text{cr}}(a, m) \leq \mathit{frac}_{\text{cr}}^{\text{null}}(a, m)$). As presented in Table 3, the results for the statistical tests show that the fraction of fulfilled coordination requirements for all motifs and artifacts is indeed significantly higher than for the corresponding null-model data (all p-values ≤ 0.02). Cliff's δ effect size shows values ≥ 0.30 for the triangle motif $m_{\triangle}$ and slightly lower values ≥ 0.12 for the square motif $m_{\square}$. We can also confirm this result by examining Figure 5. The mean value $\mathit{frac}_{\text{cr}}(a, m_{\triangle})$ for all subject projects and artifacts is 0.48 ± 0.14, while the respective mean for $\mathit{frac}_{\text{cr}}(a, m_{\square})$ is slightly higher with 0.50 ± 0.18, although there is slightly more deviation for the square motif. In more detail, the mean values for the null model are $\mathit{frac}_{\text{cr}}^{\text{null}}(a, m_{\triangle}) \approx 0.41 \pm 0.13$ and $\mathit{frac}_{\text{cr}}^{\text{null}}(a, m_{\square}) \approx 0.44 \pm 0.15$, respectively. In the end, the empirical values illustrate that roughly half of all identified coordination requirements in all subject projects are fulfilled by corresponding coordination effort. Consequently, we can

[9] We use the Wilcoxon signed-rank test because the number of available data points is rather small in our analysis and the data for some subject projects cannot be assumed to be normally distributed (Shapiro-Wilk test, $p < 0.1$). This also holds for other hypotheses and corresponding statistical analyses.

Table 3 Paired Wilcoxon signed-rank test regarding Hypothesis RH_1, data paired by motif m and artifact a (H_0: $frac_{cr}(a,m) \leq frac_{cr}^{null}(a,m)$, $N = 10$ for all tests).

Paired Wilcoxon signed-rank test		
H_0:	$frac_{cr}(\bullet, m_{\triangle}) \leq frac_{cr}^{null}(\bullet, m_{\triangle})$	$frac_{cr}(\bullet, m_{\square}) \leq frac_{cr}^{null}(\bullet, m_{\square})$
$frac_{cr}(file, \bullet)$	$W \approx 54, p < 0.01*, \delta = 0.42$	$W \approx 48, p \approx 0.02*, \delta = 0.12$
$frac_{cr}(function, \bullet)$	$W \approx 53, p < 0.01*, \delta = 0.48$	$W \approx 53, p < 0.01*, \delta = 0.24$
$frac_{cr}(feature, \bullet)$	$W \approx 54, p < 0.01*, \delta = 0.30$	$W \approx 51, p < 0.01*, \delta = 0.26$

(W = test value W, p = p-value, $*$ for p < 0.05, δ = Cliff's δ effect size)

accept Hypothesis RH_1. We give some more insights on the empirical data in the following.

For further illustration, we present the data of Hypothesis RH_1 per subject project in Figure 6. As shown there, the $frac_{cr}$ values differ per subject project: The values for the artifact *file*, $frac_{cr}(file, m)$, range from 0.25 (LLVM) to 0.82 (PostgreSQL), while the values for the artifact *function*, $frac_{cr}(function, m)$, range from 0.18 (BusyBox) to 0.80 (PostgreSQL). The value range for the artifact *feature* goes from 0.19 (BusyBox) to 0.85 (Git). For most subject projects, the feature level appears to have the lowest fulfillment rate of all artifact levels, although there are differences from subject project to subject project (see Sections 5.5 and 5.5.7 for more details).

Furthermore, we performed a sensitivity analysis following Kossinets [54] to investigate the stability of our results. In detail, we used the simulation algo-

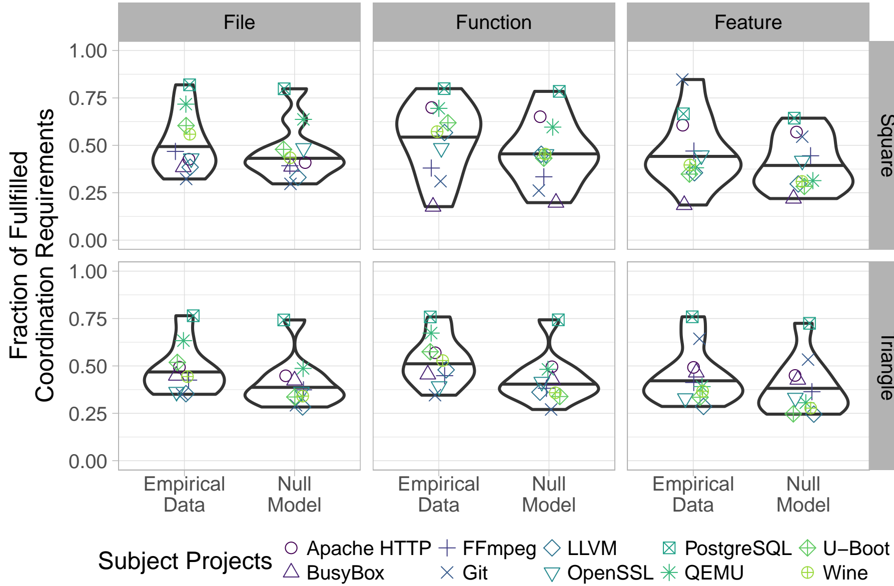


Fig. 5 Fraction of fulfilled coordination requirements $frac_{cr}(a, m)$ per motif m and artifact a as violin plot

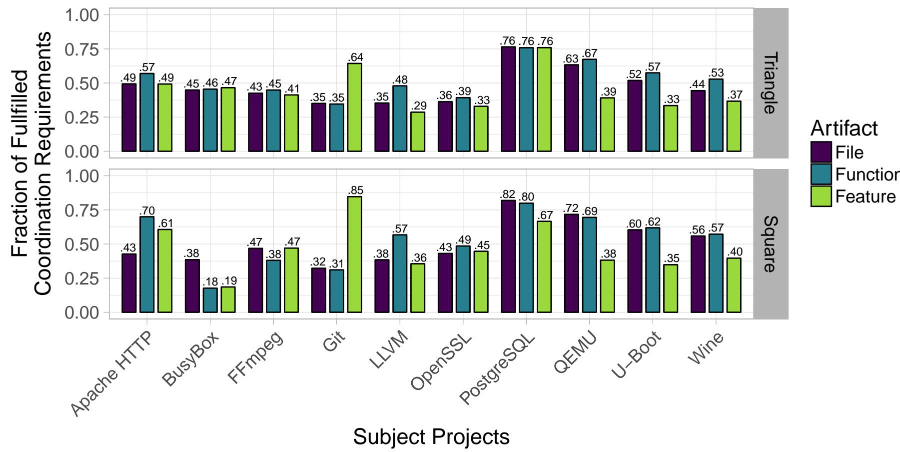


Fig. 6 Fraction of fulfilled coordination requirements $frac_{\mathrm{cr}}(a, m)$ per subject project and artifact a as bar plot

rithm "BSPC" (boundary specification problem for contexts) to simulate the absence of coordination effort from the mailing lists (which may occur on different platforms such as face-to-face meetings or chats instead) and, thus, incomplete information sources (i.e., mailing-list data) – similar to the null-model networks (see Section 3.2.4). The algorithm removes a defined number of random e-mail threads before constructing coordination-requirement networks and calculates the metric $frac_{\mathrm{cr}}$ as previously defined. In the end, for BusyBox, Git, LLVM, and OpenSSL,[10] we performed 25 iterations for better randomization, calculated mean values over all randomized iterations, and analyzed the final results. In short, we found for the selected projects that the removal of 10 % of all e-mail threads produces a relative error of about 15 % for $frac_{\mathrm{cr}}$ across all revision ranges and for all motifs and source-code artifacts, indicating strongly connected development communities. With 20 % of all e-mail threads being randomly removed, the metric exhibits a relative error of about 25 %, on average. These results indicate that the coordination process of the subject systems is driven by coordination requirements, while a larger error would have indicated potentially more random communication on the mailing list (e.g., in terms of rather larger threads with many participating developers). Furthermore, assuming that developers may use further communication channels to coordinate, in addition to the mailing list (see Section 7.1), our sensitivity analysis suggests that the empirical results of our study are an underestimation of the state of coordination-requirement fulfillment in our subject projects. We present more details and plots on our supplementary website and discuss potential reasons and implications in Section 6.1.1.

[10] We did not analyze further projects as the obtained results do not fully compensate for the large amount of computing time for the additional data. Nevertheless, we argue that the selected subset of projects is sufficient to identify indicators.

Hypothesis RH_1: Accepted. The comparison of the empirical data on the fulfillment of coordination requirements (for both types of motifs and across all artifacts) to the the respective values of the null model shows that the identified *coordination requirements are indeed not fulfilled by chance*. This result is supported by the performed sensitivity analysis.

5.3 Hypothesis RH_2: Triangle Motif $m_\triangle$ vs. Square Motif $m_\square$

Hypothesis RH_2. *The fraction of fulfilled coordination requirements is lower for the square motif than for the triangle motif, independent of the observed artifact abstraction a.*

$$frac_{cr}(a, m_\square) < frac_{cr}(a, m_\triangle)$$

To test RH_2, we perform a paired Wilcoxon signed-rank test on the empirical values of the metric $frac_{\text{cr}}$, with the data paired on the subject project and the artifact a (H_0: $frac_{\text{cr}}(a, m_\triangle) \leq f(a, m_\square)$). We illustrate the data related to this hypothesis in Figure 7.

As the results of the statistical test in Table 4 show, the fulfillment of identified coordination requirements is not higher for the triangle motif $m_\triangle$ compared to the square motif $m_\square$. With a p-value of about 0.97 ($W = 325$), we cannot reject the null hypothesis H_0. We can confirm on the violin plot presented in Figure 7 that the mean value of the metric $frac_{\text{cr}}(a, m_\triangle)$ is lower than the mean value of the metric $frac_{\text{cr}}(a, m_\square)$. In more detail, the mean value for $frac_{\text{cr}}(a, m_\triangle)$ is approximately 0.48 ± 0.14 while the mean value for $frac_{\text{cr}}(a, m_\square)$ is slightly higher with 0.50 ± 0.18. Furthermore, the values for the square motif are more scattered, as the higher standard deviation indicates. This impression is even more supported when we split the data by the artifact abstractions, as presented in Figure 8: For all abstraction levels independently, the average fraction of fulfilled coordination requirements is higher for the square motif than for the triangle motif. In summary, we cannot find evidence that supports Hypothesis RH_2.

Table 4 Results regarding Hypothesis RH_2: Paired Wilcoxon signed-rank test for $frac_{\text{cr}}(a, m_\triangle)$ and $frac_{\text{cr}}(a, m_\square)$, paired by subject project and artifact a.

Paired Wilcoxon signed-rank test H_0: $frac_{\text{cr}}(a, m_\triangle) \leq f(a, m_\square)$	
	$frac_{\text{cr}}(a, m_\triangle)$
$frac_{\text{cr}}(a, m_\square)$	$N = 30, W = 325, p \approx 0.97$

(N = number of pairs, W = test value W, p = p-value)

Hypothesis RH_2: Rejected. The comparison of empirical data on the fulfillment of coordination requirements for the triangle motif $m_\triangle$ and the corresponding data for the square motif $m_\square$ shows that the fulfillment of the identified coordination requirements is *not* higher for the triangle motif.

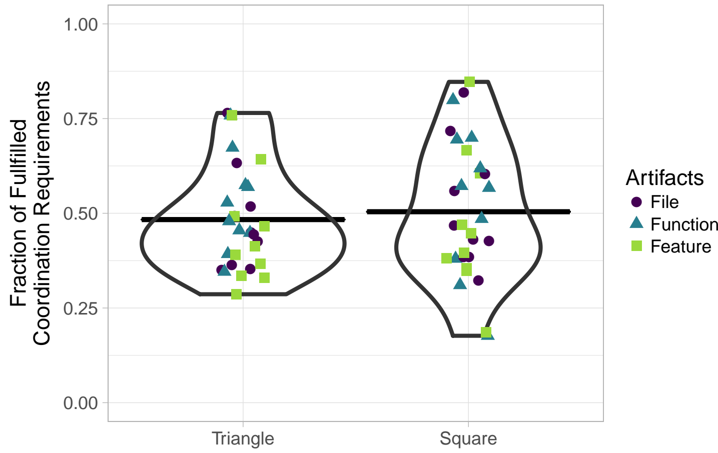


Fig. 7 Fraction of fulfilled coordination requirements $frac_{\text{cr}}(a, m)$ per motif (each scatter-plot dot represents a combination of a subject project and an artifact)

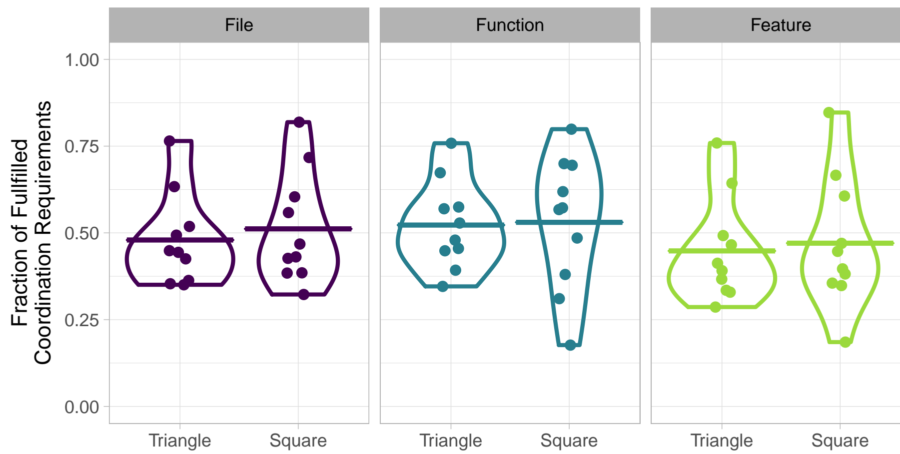


Fig. 8 Fraction of fulfilled coordination requirements $frac_{\text{cr}}(a, m)$ per motif and artifact

5.4 Hypothesis RH3: Does The Level of Abstraction Matter?

Hypothesis RH3. *The fraction of fulfilled coordination requirements differs for the different levels of abstraction, and is highest for the abstraction level of features.*

$$frac_{cr}(feature, m) > frac_{cr}(file, m) \text{ and } frac_{cr}(feature, m) > frac_{cr}(function, m),$$
$$\text{where } m \in \{m_{\triangle}, m_{\square}\}$$

Regarding RH3, we perform paired Wilcoxon signed-rank tests on the empirical data, which we paired on the subject project and the motif m, to find evidence for this hypothesis (H_0: $frac_{\text{cr}}(feature, m) \leq frac_{\text{cr}}(function, m)$ and $frac_{\text{cr}}(feature, m) \leq frac_{\text{cr}}(file, m)$). We illustrate the data regarding this hypothesis in Figure 9.

As presented in Table 5, the statistical tests suggest no significant differences among the artifact abstractions ($p > 0.12$ for all comparisons), so that we cannot reject the null hypothesis H_0. As evident also from the violin plots in Figure 9, the function-related statistics are, on average, highest ($\mathit{frac}_{\mathrm{cr}}(\mathit{function}, m) \approx 0.53 \pm 0.16$). The mean values for the artifacts *file* and *feature* are approximately 0.50 ± 0.14 and 0.46 ± 0.17, respectively. In more detail, the function-related values are more evenly distributed, resulting in a lower density estimation shown in the violin plots, while the data for other artifact abstractions show evident normal distributions around the median value. In summary, we cannot find evidence in support of Hypothesis RH_3.

Table 5 Results regarding Hypothesis RH_3: Paired Wilcoxon signed-rank test for the metrics $\mathit{frac}_{\mathrm{cr}}(\mathit{file}, m)$, $\mathit{frac}_{\mathrm{cr}}(\mathit{function}, m)$, and $\mathit{frac}_{\mathrm{cr}}(\mathit{feature}, m)$, paired by subject project and motif m.

Paired Wilcoxon signed-rank tests		
H_0:	$\mathit{frac}_{\mathrm{cr}}(\mathit{file}, m) \geq \mathit{frac}_{\mathrm{cr}}(\bullet, m)$	$\mathit{frac}_{\mathrm{cr}}(\mathit{function}, m) \geq \mathit{frac}_{\mathrm{cr}}(\bullet, m)$
$\mathit{frac}_{\mathrm{cr}}(\mathit{function}, \bullet)$	$N = 10, W = 152, p \approx 0.12$	
$\mathit{frac}_{\mathrm{cr}}(\mathit{feature}, \bullet)$	$N = 10, W = 64, p \approx 0.98$	$N = 10, W = 52, p \approx 0.98$

(N = number of pairs, W = test value W, p = p-value)

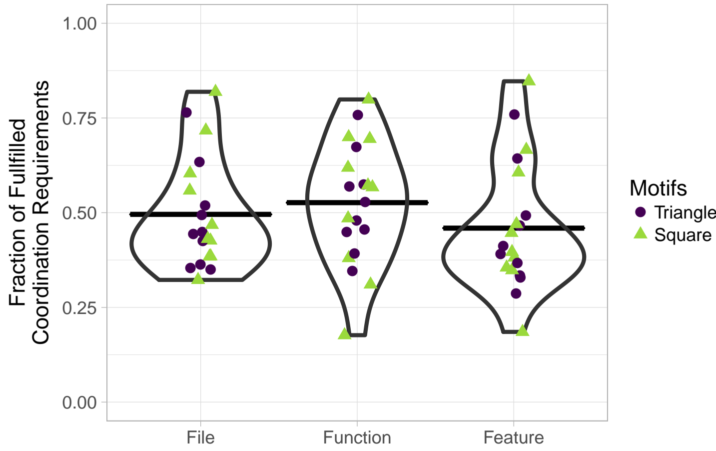


Fig. 9 Fraction of fulfilled coordination requirements $cr_{\mathrm{full}}(a, m)$ per artifact (each scatter-plot dot represents the combination of a motif and a subject project)

To shed some further light on the importance of analyzing different abstraction levels, we computed the unique number of coordination requirements per

Table 6 Number of unique coordination requirements per artifact abstraction level for Apache HTTP and the triangle motif $m_{\triangle}$, identified only for the abstraction level of the column (subcolumns cr_{found}), but *not* for the abstraction level of the row. The columns cr_{full} indicate how many of these unique coordination requirements are fulfilled.

	File		***Function***		***Feature***	
not identified by ...	cr_{found}	cr_{full}	cr_{found}	cr_{full}	cr_{found}	cr_{full}
File		–	0	0	91	30
Function	1,074	438	–	–	183	72
Feature	1,718	753	736	357	–	–
Combined	982	396	0	0	91	30

abstraction level.[11] We illustrate the data on Apache HTTP (triangle motif $m_{\triangle}$) in Table 6. For Apache HTTP, a total of 91 coordination requirements were identified for the abstraction level *feature* that have not been identified for the other two abstraction levels *file* and *function*. In relation to the total number of identified coordination requirements for the *feature* level (654, see Table 2), these coordination requirements make up nearly 14 % in total. Note that these may contain crucial coordination requirements that are missed by the other abstraction levels. Analogously, for the *file* level, 982 out of 5,710 coordination requirements were identified that have not been found on the other levels (≈17 %). The set of identified coordination requirements for the *function* level appears as a subset of the *file* level—as we expected due to the interdependency of both abstraction levels, i.e., that each function is always contained in a file. In the end, we found similar data for the other subject projects. Furthermore, a correlation analysis on the $frac_{\text{cr}}$ values of the three considered artifact abstract levels reveals a connection among the levels *file* and *function*, showing a strong correlation (Pearson correlation coefficient $r = 0.80$). But, the *feature* level only shows low correlations with the other artifact levels (*file*: $r = 0.20$, *function*: $r = 0.22$), supporting our hypothesis regarding the importance of analyzing different abstraction levels and, in particular, the *feature* level. We discuss the results and potential implications in Section 6.2.

Hypothesis RH$_3$: Rejected. The hypothesis that coordination requirements at the *feature* level are significantly more often fulfilled than for the other artifact abstractions is not supported by our data.

5.5 Hypothesis RH$_4$: Does Coordination Increase over Time?

Hypothesis RH$_4$. *In later stages of development, the fraction of fulfilled coordination requirements $frac_{cr}(a, m)$ is higher than in earlier stages for all motifs $m \in \{m_{\triangle}, m_{\square}\}$ and artifacts $a \in \{file, function, feature\}$.*

[11] To be able to compare coordination requirements among different abstraction levels—they include information on two developers and, at least, one source-code artifact, as we define in Section 2.3—, we stripped the artifact information from them.

To evaluate our empirical data regarding Hypothesis RH_4, we perform a qualitative analysis to obtain detailed insights into the subject projects' evolution. A mere quantitative analysis is not sufficient to grasp all details for the diversity of evolution scenarios that exist in our subject projects. In detail, we lack a clear statistical model to check against that is able to handle the amount of historical data that we have extracted. To avoid overfitting or underfitting the actual empirical data, we categorize the subject projects visually into overlapping and (probably) non-exhaustive groups in a qualitative assessment, to describe the specific characteristics in the software projects' history. Nevertheless, we support this analysis by additional lightweight quantitative statistics.

In our analysis of the subject projects' evolution, we mostly ignore the data points at the start and at the end of the considered historical data, as we assume that there may be data inconsistencies such as delayed usage of the mailing list for development reasons.

We present the evolutionary plots for all subject projects and the triangle motifs in Figure 10.[12] In general, for each plot, we visualize the individual values of $\mathit{frac}_{\mathrm{cr}}(m, a)$ (i.e., three lines per motif) in the history of each project over time to illustrate values' evolution. Additionally, we add a line for the project-level values (i.e., across all artifact abstractions).

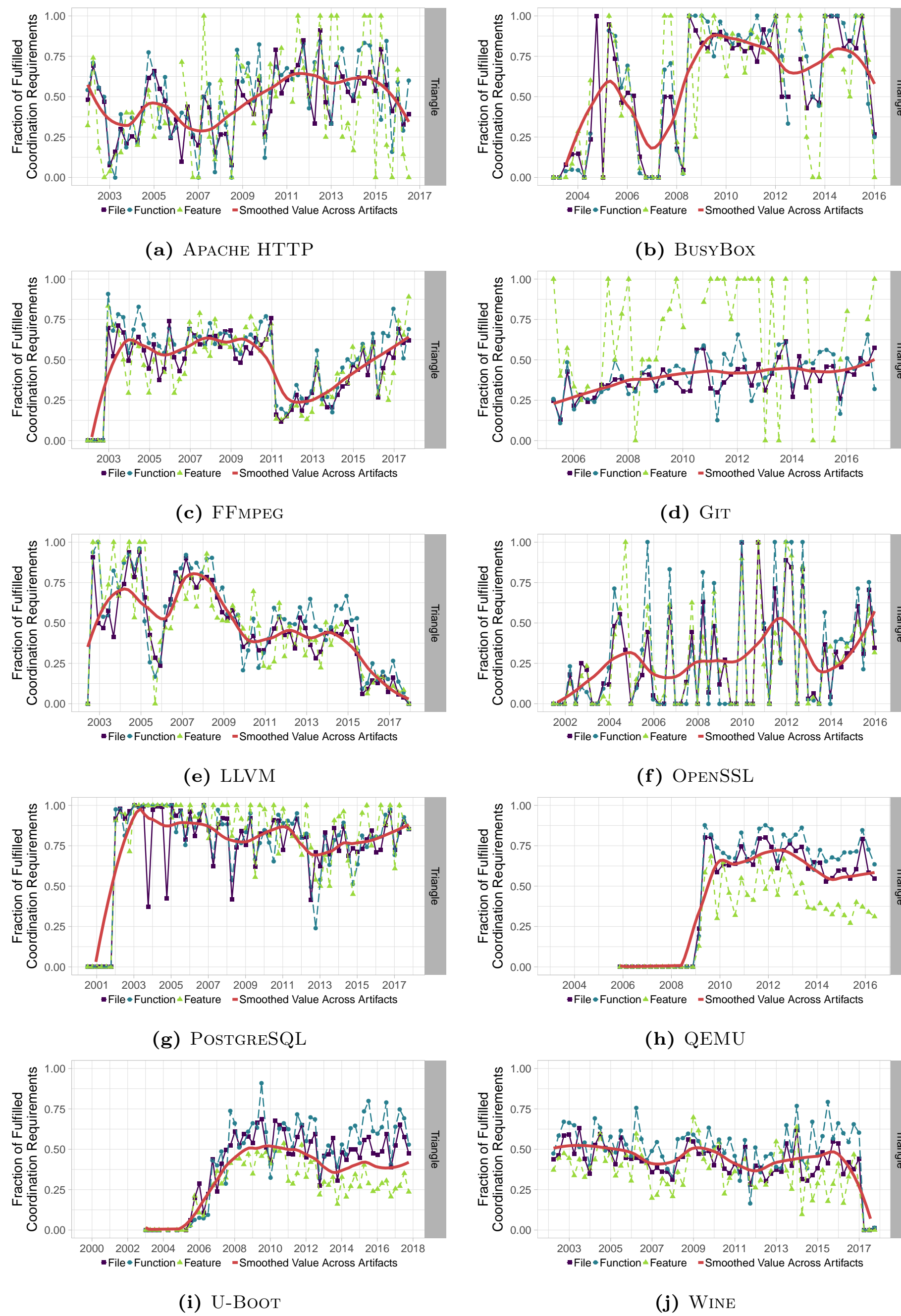
In the following, we present the lightweight evolution statistics (Section 5.5.1) before we give details on the various patterns we observed in the evolution of our subject projects (Sections 5.5.3–5.5.7).

5.5.1 Evolution Statistics

Testing Hypothesis RH_4, we rely on two lightweight statistics that support us in gaining insights into the evolution of the individual software projects: the fractal dimension of the data and the artifact-independent aggregation of $\mathit{frac}_{\mathrm{cr}}$.

The *fractal dimension* (D) is a numerical measure denoting the self-similarity of an object. It can be used to analyze the variability and complexity of time-series data over time [38], that is, in our case, the irregularity of the statistic $\mathit{frac}_{\mathrm{cr}}$ for each subject project over time: The lower the fractal dimension, the more stable the analyzed data. We use the box-count estimator, which is the method of choice for most practical fractals [38, 86]. While the estimator gives a set of estimator values for the fractal dimension, the actual fractional dimension D for a time series is then computed from the slope of the power-law estimation on those estimator values [38, 86]. For convenience, we use the R package FRACTALDIM and its function fractaldim::fd.estimate [89]. In the end, in our setting, we yield a value between 1 and 2 for further analysis, which we report for both the triangle motif $m_{\triangle}$ and the square motif $m_{\square}$. Overall, in combination with the evolutionary plots, the fractional dimension helps with the identification of software projects that are unstable in terms of their fulfillment of coordination requirements. The idea is that projects that exhibit extensively alternating values $\mathit{frac}_{\mathrm{cr}}$ over time may suffer from organizational flaws. Furthermore, we are able to identify interesting points in time for the individual subject projects that represent potentially crucial points

[12] We omit the plots for the square motif due to space restrictions. We refer to the supplementary website for all data and plots. See also Section 5.5.2.



(a) Apache HTTP (b) BusyBox (c) FFmpeg (d) Git (e) LLVM (f) OpenSSL (g) PostgreSQL (h) QEMU (i) U-Boot (j) Wine

Fig. 10 Fraction of fulfilled coordination requirements $frac_{\mathrm{cr}}(a, m_{\triangle})$ (triangle motif) for all subject projects (only revision ranges with sent e-mails shown)

in the history. We present the full list of computed fractal-dimension values in Table 7.

The *artifact-independent aggregation* of $frac_{\mathrm{cr}}$ is basically the computation of the statistic for all identified coordination requirements, independent from the artifact-abstraction level from which they arise. As the unaggregated statistic (see Section 3.2.2), the range for this aggregations is $[0, 1]$. To gain a better overview of general trends, we smooth the data with the local-smoothing approach LOESS [20] before plotting the data in Figure 10.

5.5.2 Overview

When looking at the historical data and plots (see Figure 10 and the supplementary website), we cannot find any generalizable trends across all subject projects. The projects rather exhibit individual patterns and trends that arise from their unique history and evolution. While some projects show clear ascending trends in their fulfillment of coordination requirements, others show chaotic patterns (e.g., see Figures 10c, 10d, and 10f). The evolutionary patterns are exceptionally similar for the triangle and square motifs.

5.5.3 Stable Projects

There are some projects that exhibit a quite stable evolution of fulfilled coordination requirements over time, with only few and small fluctuations—which is supported by the artifact-independent aggregation and also the fractal-dimension values. We discuss the individual projects in the following paragraphs and also refer to the respective plots shown in Figure 10.

The most stable project is QEMU (speaking in terms of fractal dimension: $D_{m_\triangle} = 1.39$, $D_{m_\square} = 1.43$). After revision range 25 (starting in February 2009), the fulfillment of coordination requirements rises from zero to a quite stable value of 0.70, approximately. Only the *feature* value is lower with 0.45, approximately, but it is also stable. We can find a similar view for the project U-Boot ($D_{m_\triangle} = 1.40$, $D_{m_\square} = 1.46$), with an approximate value of 0.55 for the $frac_{\mathrm{cr}}$-statistic across all artifact abstractions.

Table 7 Fractal-dimension values D for all subject projects and motifs, sorted by $D_{m_\triangle}$ and grouped by similar values. The groups of values are derived in combination with the plots in Figure 10.

Project	$\mathbf{D_{m_\triangle}}$	$\mathbf{D_{m_\square}}$
QEMU	1.39	1.43
U-Boot	1.40	1.46
FFmpeg	1.49	1.55
LLVM	1.51	1.54
PostgreSQL	1.51	1.54
Wine	1.57	1.64
BusyBox	1.59	1.60
Git	1.59	1.58
Apache HTTP	1.65	1.71
OpenSSL	1.67	1.69

The stable project that exhibits the highest values for $frac_{\mathrm{cr}}$ is PostgreSQL ($D_{m_\triangle} = 1.51$, $D_{m_\square} = 1.54$) for which the value for the fraction of fulfilled coordination requirements ranges between 0.75 and 0.95 and only few outliers occur over time. Interestingly, the project FFmpeg ($D_{m_\triangle} = 1.49$, $D_{m_\square} = 1.55$) exhibits a stable value before revision range 41 (starting in December 2010) where $frac_{\mathrm{cr}}$ considerably drops (see Section 5.5.6 for more details). Similarly, the $frac_{\mathrm{cr}}$ values for LLVM ($D_{m_\triangle} = 1.51$, $D_{m_\square} = 1.54$) decrease steadily after some time (see below for more details), but while decreasing, the $frac_{\mathrm{cr}}$ values do not fluctuate heavily and show a clear trend that is supported by the fractal-dimension values.

5.5.4 Unstable Projects

Unstable projects have relatively high fractal-dimension values D that reveal extensive alternating fractions of fulfilled coordination requirements among consecutive revision ranges.

The most unstable projects are OpenSSL ($D_{m_\triangle} = 1.67$, $D_{m_\square} = 1.69$) and Apache HTTP ($D_{m_\triangle} = 1.65$, $D_{m_\square} = 1.71$). Especially, OpenSSL exhibits high differences in its $frac_{\mathrm{cr}}$ values for consecutive revision ranges, even changing directly from zero to one. We can also see that there is some kind of stabilization after some point in time. For OpenSSL, this happens with the revision range 62 (starting end of March 2014) after which the metric value $frac_{\mathrm{cr}}$ steadily increases without high fluctuations (see Section 5.5.6 for more details).

5.5.5 Monotonic

On the one side, as evident in Figure 10d, Git is the only project that shows a clear and steady increase of the $frac_{\mathrm{cr}}$ over time, from about 0.25 to 0.50 (with the exception for *features*, see below for details on this). Hence, Git appears to be the only subject project to clearly support Hypothesis RH_4. On the other side, the values for LLVM clearly decrease after revision range 28 (starting in March 2008, see Figure 10e), after experiencing some disturbances in earlier stages of the inspected project history (see below for details).

5.5.6 Turning Points and Far-Reaching Events

While analyzing the software projects regarding the fulfillment of coordination requirements over time, we can see sudden changes in the values that may result from events affecting the projects' coordination positively or negatively.

For example, when inspecting the project FFmpeg closer, we observe a considerable drop of the $frac_{\mathrm{cr}}$ value after revision range 41 (i.e., after January 2011) from about 0.60 to 0.30 (see Figure 10c). Events occurring in the project during this time exactly explain both the considerable drop as well as the steady increasing afterwards:[13] Some maintainers attempted to take over the project due

[13] The upcoming description of events is based on the following list of references (all accessed 2018-11-05):
- the announcement: `https://ffmpeg.org/archive.html#return_to_freedom`, and
- further detailed information from (former) FFmpeg contributors: `https://lwn.net/Articles/424396/`, `http://blog.pkh.me/p/13-the-ffmpeg-libav-situation.html`, and `https://www.slideshare.net/SamsungOSG/ffmpeg-a-retrospective`.

to their discontent with the current leader, the development process, and, especially, the long list of open and unattended issues. One particular accusation in direction of the current leader—that is highly relevant for our setting—was that he bypassed the review process and pushed his changes directly to the repository. The designated new team took over the website and initiated an alternative source-code repository, while, in parallel, the version control system changed from SVN to Git. Overall, the situation in January 2011 was unclear, resulting in almost halted development. As a result of the incident, the project Libav emerged as a fork of FFmpeg, claiming vital parts of the former FFmpeg infrastructure. With the help of VideoLAN, FFmpeg recovered well and the fraction of fulfilled coordination requirements steadily increased again after some time to a value of about 0.50. We discuss this matter and its importance for our analysis in more detail in Section 6.3.

Similarly, the stabilization and increase in the OpenSSL data (see Figure 10f) coincides with a critical event in the project's history (revision range 62): the disclosure of the Heartbleed bug[14] in April 2014. An improvement in the fulfillment of coordination requirements is a consequence of the subsequent organizational restructuring that has been carried out to prevent such critical security bugs in the future. This result underlines the meaningfulness of our analysis and socio-technical analyses in general, as we discuss in Section 6 in more depth.

When analyzing the project BusyBox, there is also a considerable drop in the fulfillment of coordination requirements in the time from July 2006 to July 2007, which even contains several consecutive zero values (see Figure 10b). We can match this time with several far-reaching events in the project:[15] On the one side, there are, at least, two announcements of new maintainers in the course of 2006 (for the time of the release *1.1.1* and again for the release *1.2.2*), indicating major changes in the organizational structure of the project. On the other side, and connected with the maintainer switches, BusyBox was involved in one of the first major lawsuits in relation with free software licenses, which has been filed by some of the (previous) maintainers against a company, while, in parallel, conflicts with the project initiator and former maintainer arose. In the end, the project changed the license and also one of the maintainers started a related project, ToyBox. All in all, the setting-in period of the new maintainers and the legal issues of the project obviously affected the development, resulting with (almost) coordination requirements unfulfilled. After all issues had been resolved in 2007, the fraction of coordination requirements recovered and quickly stabilized with $frac_{\mathrm{cr}}$ values of about 0.80.

The initial rise of the fulfillment of coordination requirements for the project QEMU (in November 2008, see Figure 10h) coincides with the release of version *0.10.0* in early 2009. This release contained a multitude of new features and functionality and happened over one year after the previous release.[16] Further-

[14] `http://heartbleed.com/` (accessed 2018-11-05)

[15] The upcoming description of events is based on the following list of references (all accessed 2019-03-14):

- `https://www.busybox.net/oldnews.html` and `https://lwn.net/Articles/202106/`,
- `http://www.softwarefreedom.org/news/2007/sep/20/busybox/`,
- `https://lwn.net/Articles/202113/`, and
- `https://web.archive.org/web/20091220044135/perens.com/blog/2009/12/15/23/`

[16] `https://wiki.qemu.org/OlderNews` (accessed 2018-11-05)

more, the project moved to SUBVERSION as the main version-control system. We did not find further information why the coordination requirements have not been fulfilled prior to the mentioned release.

The continuing decrease in the $frac_{\mathrm{cr}}$ values of LLVM (see Figure 10e) after revision range 28 (starting in March 2008) coincides with the preparations of release *2.3*, which had been stalled due to major regressions. Apart from this, there are no external indicators that explain why the fulfillment of coordination requirements drops gradually after this release.

5.5.7 Differing Data across Levels of Abstraction

The evolution of the $frac_{\mathrm{cr}}$ values for most subject projects is quite consistent across the different artifact abstractions, however, there are some exceptions. Especially for GIT, the values for *features* differ extensively from the evolution of the other two abstractions: The values alternate heavily and there are also some revision ranges without identified coordination requirements, resulting in missing data values (see Figure 10d). Furthermore, for both QEMU and U-BOOT, also the *feature*-related $frac_{\mathrm{cr}}$ values are lower than the ones for the other artifact abstractions (see Figures 10h and 10i).

Hypothesis $\mathbf{RH_4}$: Rejected. The hypothesis that the fraction of fulfilled coordination requirements $frac_{\mathrm{cr}}(a, m)$ improves over time cannot be shown across the complete set of subject projects. Instead, we found several different patterns in the organizational evolution indicating that there are very project-specific reasons leading to fulfilled and unfulfilled coordination requirements, such as a change of maintainers or even an attempted project-takeover.

6 Discussion

In this section, we discuss the empirical and statistical results for the hypotheses (see Sections 3.3 and 5) and the related research questions (see Section 3.1).

6.1 RQ_1, RH_1, and RH_2: Coordination Requirements *not* Fulfilled by Chance

Research Question $\mathbf{RQ_1}$. *Does developer communication align with artifact-based coordination requirements in real-world open-source software projects, such that the coordination requirements are fulfilled?*

From RQ_1, we raised two hypotheses: (RH_1) coordination requirements are *not* fulfilled by chance and (RH_2) coordination requirements at the abstraction level of *features* (see Section 2.4) yield the best alignment with actual coordination effort. While we found the expected result for RH_1, we found no evidence to support RH_2. In the following, we discuss the findings for the two research hypotheses in detail.

6.1.1 RH_1: Empirical vs. Null-Model Data

We see the main reason for an intentional fulfillment of coordination requirements in the coordination process of the subject projects itself and, more importantly, the existing developer roles in the subject projects—as we hypothesized in Hypothesis RH_1. The selected subject projects have contribution guidelines including the upfront sending of patches to the mailing list for reviewing purposes (see Section 3.2.3). The fact that the statistical effect size is only moderate (with values from 0.12 to 0.48, see Table 3) is not too surprising when acknowledging that most subject projects have a small set of core developers who are responsible for most of the implementation work, reviews, and, thus, arising coordination requirements: The network structure may not change too much in the null models due to the stabilization of the degree distribution (see Section 3.2.4).

Apart from the overall result, there are interesting project-specific results: For example, for BusyBox in isolation, Hypothesis RH_1 does not hold for the square motif $m_\square$, i.e., $\mathit{frac}_{\mathrm{cr}} \not> \mathit{frac}_{\mathrm{cr}}^{\mathrm{null}}$, which is also the case for OpenSSL and the triangle motif $m_\triangle$. We see potential reasons for this in the respective project histories: Both projects struggled from organizational issues at some time in their history and, thus, a coordination breakdown causing a substantial drop in the fulfillment of coordination requirements (cf. Section 5.5.6).

Independent of the previous discussion, we note that $\mathit{frac}_{\mathrm{cr}}$ values between 0.19 and 0.80 do not illustrate consistent and reliable review processes in all the subject projects—independent of the motif. In some subject projects (e.g., PostgreSQL and QEMU), the fulfillment of coordination requirements seems to be of higher importance than in others (e.g., LLVM and Git). We see a reason for this discrepancy in the different project cultures: Each project likely handles the fulfillment of coordination requirements in its own way, somewhere in the spectrum from rigorously to even neglectfully—unless a coordination breakdown occurs and the coordination process is adjusted in consequence (cf. OpenSSL).

Furthermore, as described in Sections 3.2.4 and 5.2, we performed a sensitivity analysis following Kossinets [54] as an investigation on the stability of our results by simulating incomplete information sources and lacking coordination effort by developers. For BusyBox, Git, LLVM, and OpenSSL, we found that a removal of 10 % of all e-mail threads (i.e., coordination effort) produces a relative error of about 15 % for our metric $\mathit{frac}_{\mathrm{cr}}$. This result has several implications: (i) Any coordination effort is important to fulfill coordination requirements and to support the evolution of the projects' individual evolution (see also Section 5.5). If only a few developers do not coordinate properly, this may already have an immediate effect on the quality and speed of development. Depending on the individual developer, the effect may be even higher (cf. the "truck factor"). (ii) For our subject projects, the coordination structure is guided by the coordination requirements and is not the result of a random process. (iii) Regarding the threat to validity that developers may also rely on other coordination channels than the respective mailing lists (see also Section 7.1), the empirical results of our study are likely an underestimation of the state of coordination-requirement fulfillment in our subject projects.

6.1.2 RH_2: Triangle Motif $m_{\triangle}$ vs. Square Motif $m_{\square}$

The fact that we found no empirical and statistical evidence that more triangle motifs $m_{\triangle}$ are fulfilled than square motifs $m_{\square}$ is surprising and stands in contrast to our Hypothesis RH_2. Since the source-code artifacts from which the coordination requirements arise are a single *file*, *function*, or *feature*, it is relatively simple to identify potential interactions and conflicts based on the triangle motif (e.g., based on both the documentation and the source code, but also through failing builds and tests) and also to resolve independently by a single developer. Thus, although depending on the system modularization, developers may not need to see the necessity to discuss the changes with other developers to fulfill *all* arising coordination requirements, but are more eager to fulfill the more complex ones that occur in practice. Nevertheless, this indicates that distinguishing coordination requirements by the triangle motif $m_{\triangle}$ and the square motif $m_{\square}$ contributes to the understanding of developers' coordination in detail.

In the same vein, we also need to discuss that there may be off-list communication directly among developers that are required to coordinate.[17] Although such an action is a (not necessarily serious) violation of the development guidelines, direct communication and coordination may save time in the software-engineering process and also mitigates disruptions of other developers. Analogously, developers may see the mailing list more as a measure to discuss potential conflicts that affect more parts of the software system and, thus, have a higher impact.

Finally, other measures of the review systems may play a major role regarding the fulfillment of coordination requirements and lead to an alignment of triangle and square motifs: Gatekeepers are a measure in the review process describing module/feature owners that are responsible for revising and integrating new code for specific part of the software system. Gatekeepers or, rather, maintainers potentially do not participate in the implementation work, but have enough domain knowledge and knowledge on the source-code artifacts to perform proper reviews. They may also link the work among developers in that they reach out to the potentially affected developers who worked on other parts of the software system, serving as information brokers in the end [28, 55]. Thus, they enable the fulfillment of coordination requirements based on the square motif.

6.1.3 RQ_1: Conclusion

Overall, we argue that our analysis shows that the fulfillment of coordination requirements seems to be neglected in open-source projects, as $frac_{\text{cr}}$ values between 0.19 and 0.80 do not indicate consistent and reliable review processes in all the subject projects. Examples are the projects BusyBox and FFmpeg where the respective project leaders bypass(ed) the review process on the mailing list (e.g., see Section 5.5.6). But, we note that our analysis supports the importance of analyzing the fulfillment of coordination requirements in decentralized and globally scattered open-source software projects. Furthermore, it also emphasizes that distinguishing coordination requirements by the triangle motif $m_{\triangle}$ and the square motif $m_{\square}$ in the first place does yield valuable insights, as developers seem to focus mainly on square-based coordination requirements (regardless their inherent complexity, as

[17] We also discuss this threat to construct validity in Section 7.

discussed above). We argue that this differentiation has potentially been brushed aside too easily in previous research.

6.2 RQ_2, RH_3: The Abstraction Level Matters

Research Question RQ_2. *Does developer communication align better with feature-based coordination requirements than with function-based or file-based coordination requirements?*

Our results show that developer communication does *not* align very well with feature-based coordination requirements and clearly worse than with file-based and function-based coordination requirements for most subject systems. This indicates that developers do *not* coordinate primarily based on features, i.e., developer coordination does not happen at this semantic level.Otherwise, a greater fraction of arising coordination requirements would be fulfilled. The implications are as for all unfulfilled coordination requirements: Errors in the source code can occur, while the potential of conflicts in the development process rise. Depending on the specific development process, the consequences can be either minimal or considerable, such that a project may actually get abandoned. Either way, they result in additional work, i.e., in finding a solution to balance or combine all conflicting implementations, hindering the development process and costing time.

6.2.1 The Implementation of Features

There are two main interdependent observations regarding the implementation of features that seem to play a role in our analysis: (i) the number of touched artifacts per abstraction level (e.g., the number of *files* that are altered in a commit) and the dependent number of identified coordination requirements as well as (ii) the visibility of the artifacts of an abstraction level and their interconnectedness in the underlying software architecture. When looking at the analyzed abstraction levels of our study (i.e., *files*, *functions*, and *features*), the notion of *features* is the most distinct type of artifact: Features can be annotated in an undisciplined manner [62] and can be implemented in a cross-cutting manner, orthogonal to the file structure and the inferred architecture [49, 61, 79], causing extensive feature scattering [75, 76, 79] or even variability smells [2, 33, 34]. Overall, they may be difficult to track across the code base [32]. As a consequence, coordination requirements arising from concurrent change on *features* are also difficult to identify and, thus, even to fulfill. In contrast, the feature abstraction can also, to a great deal, coincide with the function abstraction or even the file abstraction. In our experience, however, this is not valid for, at least, some of the features [75, 79].

6.2.2 The Prominence of Features

Moreover, another reason may be the awarded prominence for features in the respective subject systems and their defined development processes in general. In particular, we note that features are not as prominently placed in the development process of most of our subject projects as assumed by the research community of

feature-oriented software development and ourselves [3]. This does not indicate that features do not play *any* role in the development process—which is definitely not the case as shown on the case of uniquely identified coordination requirements on the different abstraction levels in Section 5.4—but just not such a prominent one as expected.

6.2.3 RQ_2: Conclusion

Based on our results, we would like to raise awareness for features, to increase their prominence and importance in (open-source) software projects and communities in general as well as in the research community. First, real-world developers and their communities need to discuss whether to include features more explicitly in their daily work (implementation as well as communication level), rendering features first-class citizens in *specific* situations [73] and explicitly facilitating a shared mental model [16] to support the development process to a great extent [96]. Second, researchers should acknowledge the importance of features in the development process of their respective subject projects into their methodology more prominently, to be able to discuss their findings elaborately and also more accurately.

6.3 RQ_3, RH_4: Evolution over Time may be Coordinated Chaos

Research Question RQ_3. *Does the degree of fulfillment of coordination requirements change for different levels of abstraction during project evolution?*

Our results show that coordination requirements do not necessarily get more consistently fulfilled when software projects and communities mature, as we hypothesized in Hypothesis RH_4. Rather, we found a variety of evolutionary patterns: There are software projects that exhibit a quite stable fraction of fulfilled coordination requirements (see Section 5.5.3), while there are others that appear to be unstable to some extent (Section 5.5.4). The most interesting—and also most appreciated—result is that we can, in retrospective, identify far-reaching events and turning points in project history, which shows the potential of our analysis. It is important to note that Research Question RQ_3 is not a yes-no question, but rather a complex socio-technical object of study, involving sophisticated and complicated matters of software evolution and also community evolution.

In detail, we see several reasons for the project-specific results for RQ_3 and the depending Hypothesis RH_4 that we elaborate on in the following paragraphs.

6.3.1 Project-Dependent Dynamics

Project-dependent dynamics play a crucial role in the fulfillment of coordination requirements. Although the coordination and submission process is well-defined in most projects, developer coordination is a complex social action that depends on the persons involved, the project culture, the software system itself, its source code, implementation paradigm and architecture, and many other factors. The fact whether there is a powerful (set of) maintainer(s) or even a benevolent dictator for life can also completely shape the development environment and community (see

the subject systems BusyBox and FFmpeg, but also Linux, for example). This can be either beneficial or harmful for the projects' evolution.

Furthermore, far-reaching events can have a large impact, as we presented in Section 5.5.6. Attempted project hijacks (cf. FFmpeg) or publicly announced critical bugs (cf. Heartbleed in OpenSSL) encourage project communities to improve communication and coordination intensively. In the absence of such social events and public pressure, developers may see no significant benefit from increasing coordination effort, as we discuss below in Section 6.3.3. Overall, although our findings are project-specific, they show that socio-technical analyses are meaningful and can provide crucial insights for a sustainable project evolution.

6.3.2 Domain and Publicity of the Subject Projects

The domain and the publicity of the subject projects have a huge impact on the relevance of coordination-requirement fulfillment. In a very low-spread and non-critical user-space project (such as a video player, for example), the adherence to the submission process and development process is far less critical than for system-relevant, wide-spread, and publicly known software systems. An example for the latter systems is OpenSSL, which is a critical part of and the de-facto standard for secure communication over computer networks nowadays. The criticality of the Heartbleed bug[14], for example, posed a major threat to a large number of infrastructurally critical servers. To this end, post-hoc investigations by researchers indicate that the bug would have been found likely by a more thorough review process.[18] Although this is a basic review-centric problem and does not relate directly and completely to the fulfillment of coordination requirements, the identification of and notification on coordination requirements can be critical in the process of code reviewing. To this end, to be able to identify coordination requirements on several abstraction levels may lead to an advantage.

6.3.3 Latent Change in Coordination over Time

We see the reason for non-increasing fraction of fulfilled coordination requirements (as falsely hypothesized in Hypothesis RH_4) in the latent change in coordination over time. As people—that includes developers!—change over time, it is plausible that the coordination in software projects changes over time, too. With increasing project maturity, developers tend to know each other and the software system itself better, which, in turn, latently influences the communication and coordination effort that needs to be carried out to organize and solve development tasks. In detail, developers form a shared mental model over time that resembles a common perception of the software system on which the developers work [21, 30, 66, 82]. With such a shared mental model, developers gain a more structured understanding of the software system and, additionally, a better awareness of other developers' needs, which leads to a more accurate anticipation of requirements and resolution of potential problems as early as possible. Consequently, less coordination effort is needed [4, 31] and, thus, also less coordination requirements will be fulfilled.

[18] See the following references: `https://dwheeler.com/essays/heartbleed.html`, `https://news.ycombinator.com/item?id=7556826`, `https://www.theregister.co.uk/Print/2014/04/11/openssl_heartbleed_robin_seggelmann/` (all accessed 2019-03-15)

In other words, not every change may need to be coordinated in matured software systems as potential conflicts and problems can be solved by one experienced developer in isolation.

Researchers found that the lack of a shared mental model is not necessarily harmful, but may lead to problems [31]. Essentially, developers have to trade-off the coordination of crucial points against the reduction of a coordination overhead. We argue that a community based on this principle is likely stable until problems occur due to reduced coordination effort, triggering a potential community renewal with regard to the fulfillment of requirements. As we showed in Section 5.5.6, we found examples for this in our set of subject systems (see the related discussion in Sections 6.3.3 and 6.3.2).

However, it may be also the case that the technical structure of the system changes along with the social structure to keep or restore socio-technical congruence in a rubber-band effect, as suggested by Betz et al. [8].

6.3.4 RQ_3: Conclusion

In the end, we can conclude that, with the help of our qualitative analysis of fulfilled coordination requirements and when looking at the results especially from Section 5.5.6, we can support the need for additional and more detailed socio-technical analyses. They can bring meaningful insights, help, and support in large-scale, globally scattered open-source software projects by indicating potential social and socio-technical problems for the sake of a sustainable software evolution.

7 Threats to Validity

In this section, we discuss possible threats to the construct validity as well as the internal and external validity of our empirical study.

7.1 Construct Validity

Construct validity refers to the idea that the construct under measurement was not operationalized correctly. In our case, this affects mainly our notion of coordination requirements and their extraction (see Table 1 for the set of our variables). There are several points of discussion here:

First, while the developer–artifact and artifact–artifact relations can be operationalized in a straightforward way, we analyzed only mailing lists as proxy for developer coordination that fulfills coordination requirements (developer–developer relation). Although, developers may use further channels of communication (e.g., personal e-mails or verbal communication) [95]. To understand how developers use these further channels, we performed a sensitivity analysis and found indicators that the absence of coordination effort in terms of mailing-list threads has an immediate effect on the measured metrics (see Sections 5.2 and 6.1.1 for more details). We mitigate this threat by selecting only software projects with a well-established mailing list for development-related communication [10, 27, 81, 92]. This approach has proved to be useful in previous studies [42, 85, 94]. We have considered social coding platforms that also use other means of communication

(e.g., issues at GitHub), but these platforms may be too young for history analysis, whereas each mailing list of our subject projects dates back, at least, 10 years (see Table 2).

Second, the value of our operationalization of fulfilled coordination requirements is threatened by the possibility that coordination requirements might be fulfilled by coordination that occurs outside the configured three-month windows (see Section 3.2 for more details on the time windows). We deliberately chose three-month time windows, because they represent the sweet spot between sufficient data and avoiding overfitting [51, 52, 53, 67]. Furthermore, not all coordination requirements may be equally relevant to fulfill [13, 21, 57], while developers get overloaded with the pressure to fulfill all coordination requirements [99]. To the same end, our dependent variable, the congruence measure $frac_{\mathrm{cr}}$, does not incorporate any information on the developers' distance. Unfortunately, the application, extraction, and incorporation of additional information to assess the criticality of coordination requirements is not feasible or even possible at the scale of our study, especially when using historic project data. Therefore, we argue that consideration of all identified coordination requirements may be an overapproximation, but our results regarding the null models indicate that this decision is negligible.

Third, the notion of features that we analyzed in our study is limited to those implemented using the C preprocessor. We use cppstats to extract information on the usage of the C preprocessor in our subject systems. cppstats has been shown to work properly with cross-cutting features [49, 61] and even features that are implemented using undisciplined preprocessor annotations [62]. cppstats has been used to identify variability smells [34]. Thus, we are confident that we extract the correct information for our analysis.

7.2 Internal Validity

We see the main threat to the internal validity of our study in the potential selection bias regarding our subject systems, including that we have no knowledge about the project climates, the relationships among developers, the community demographics, and whether they had any influence on the fulfillment of coordination requirements. To mitigate this threat, we selected a wide range of software projects, conforming in important properties (see Section 3.2.3), but differing in maturity, size (both source code and number of developers), commit policies, and application domains. Selection criteria such as the evolution of the contributions are also important for our analysis, but are unfortunately not feasible to assess with reasonable effort. Nevertheless, our selection criteria allowed us to select a representable and balanced set of software projects.

Furthermore, we limited the developer–artifact data to files that are implementation-related, so header files, documentation files, and build files are not considered in our study. As a consequence, we potentially miss some important coordination requirements during the extraction process that may influence our analysis results. We deliberately chose to exclude such files, as the function-related and feature-related analyses do not work on non-implementation files, since such abstraction levels do not (usually) exist in such kinds of files. As most work in a software project happens in implementation files, we see our analysis unaffected by this choice.

Finally, our definition and identification of features threatens internal validity. We use C-preprocessor annotations (#ifdefs) as a central mechanism for implementing configurable features, which likely does not capture all notions of a feature in our subject systems. Furthermore, other means of implementing features and variability (such as run-time variability or a more sophisticated build system) may affect our conclusions regarding the abstraction level of features. However, within the implementation-related source-code files that we analyzed to extract all three levels of abstraction, the C preprocessor is a major main source of variability [49, 61] as it is widely used in open-source software projects implemented using the programming language C. Thus, we do not consider the other means of variability being the focus for our analysis.

7.3 External Validity

As discussed with internal validity, the selection of subject systems may also threaten external validity. Nevertheless, all projects are large, highly active, and widely deployed, which gives us relevant insights into the fulfillment of coordination requirements in open-source communities. Thus, our results can be carefully interpreted in the context of open-source software projects with similar properties.

The restriction to preprocessor annotations as means for locating feature-specific code may threaten external validity. Preprocessor annotations are well-established in open-source software and industrial systems to implement features (e.g., [3, 29, 49, 61]), but the findings may vary in detail for other feature implementation techniques.

8 Related Work

There has been various research on the relationship between developer communication and development, especially with respect to geographically distributed development teams. Herbsleb and Grinter [43, 44] conducted two studies on coordination in geographically distributed software projects and found that developers usually rely heavily on informal ad-hoc communication—if that is possible. They shed light on the matter that communication and coordination in a distributed environments therefore needs discipline and planning. Kwan and Damian [55] investigated the awareness of other developers' work in a locally distributed team and found that regular meetings and notifications heavily support coordination within the project. In related studies [23, 26, 46, 55, 70, 102], researchers investigated the influence of coordination in software projects on software quality, finding that the consideration of social aspects, such as developer communication and coordination, are crucial for understanding open-source software projects and for improving software quality—which is in line with our own view. For a more semantic view on developer activity, we go beyond previous studies by explicitly incorporating the link between social and technical contributions (i.e., the collaboration patterns of coordination requirements).

The idea of constructing networks to analyze software architecture and social interaction is not new: Developer networks are analyzed to foster the understanding of the software development process by incorporating views on the software

architecture and the source-code artifacts. For example, de Souza et al. [94] constructed socio-technical developer networks to uncover latent relationships among individuals and conclude that the social view on software projects is essential to their understanding. Moreover, there have been several studies to assess developer roles in open-source software systems [53, 97] and to identify developer communities based on source-code collaboration or communication effort [10, 11, 51, 69]. In the same vein, Xuan et al. [103] conducted a study on temporal motifs consisting of two developers and a source-code artifact (i.e., the triangle motif) to identify latent team structures and to assess their productivity. Finally, researchers have used developer networks to identify disadvantageous evolutionary trends in software projects, both in the social structure and in the source-code artifacts [6, 9, 52, 71]. However, as a difference to these studies, the focus of our investigation is on the synchronicity between collaboration at the code level and communication via e-mail. Further, we also incorporate the semantic level of features in contrast to other abstraction levels of source-code artifacts, which has not been considered in these studies.

Furthermore, based on work on the mirroring hypothesis and "Conway's Law" [21, 22, 64], researchers investigated directly on the fulfillment of coordination requirements and the notion of socio-technical congruence. Most importantly, Cataldo et al. [17, 18, 19] defined the notion of *socio-technical congruence* and studied coordination requirements and their fulfillment by means of qualitative studies. They used a matrix representation for modeling the socio-technical relations and triangle-based coordination requirements in their analysis, while we use the more visual notion of developer networks and network motifs. While Cataldo et al. studied in-house industrial systems and rather few observed instances in time (increasing internal validity), we analyze a substantial set of longitudinal data of different open-source software systems, summing up to over 180 years of development in total (increasing external validity), to investigate on evolutionary trends.

In 2018, Sierra et al. [91] conducted a systematic mapping study about socio-technical congruence, reviewing the state of the art in this field of research and identifying gaps and opportunities for future research. The authors categorized previous work on socio-technical congruence regarding the conducted measures and the application scenarios. Next, we discuss the most notable papers summarized also by Sierra et al. [91].

Most importantly, the study of Valetto et al. [98] is closest to our study in terms of the formal framework: The authors used socio-technical software networks to search for coordination requirements and assess their fulfillment, using the same three different relations as we did. They also apply the same measure for socio-technical congruence, although they do not distinguish between the triangle and square motif in their dependent variable (i.e., the congruence measure). Valetto et al. also rely on an algorithmic approach for the identification of coordination requirements rather than using network motifs which hide the inherent logic and, thus, are more intuitive and illustrative. We extend their set of variables by incorporating more independent variables, such as the time window for the evolutionary perspective and the network motifs as such. Furthermore, we especially focus on different abstraction levels for source-code artifacts and introduce the *feature* level.

Wagstrom et al. [100] conducted a study to investigate on the influence of individuals on socio-technical congruence at project level and proposed an individ-

ualized measure. They found support for Cataldo et al.'s results in a reproduction study and that especially the specific communication based on coordination requirements has an impact on project performance (and not an individually high amount of communication in general). Although we do not assess socio-technical congruence for each individual in our subject projects, we also found indicators that the work of project leaders is likely based on arising coordination requirements and not based on a high amount of communication in general (see Section 6.1).

In the same vein, Betz et al. [8] proposed a research design for taking an evolutionary perspective on socio-technical congruence. The authors describe a rubber-band effect describing the inevitable changes in social and technical structure to restore socio-technical congruence when there is misalignment during evolution. We take up on their proposed research questions, while we primarily focus on the existence of socio-technical congruence and on what changes may (irreversibly) destroy it (see Section 5.5, for example).

There have been several further studies analyzing different perspectives on socio-technical congruence using more sophisticated congruence measures: Kwan et al. [55, 56] proposed a weighted measure that is more suitable to detect individual coordination gaps rather than performing an analysis on project level. Li et al. [60] distinguished knowledge-dependent and resource-dependent congruence that is achieved by different means of information flow during coordination, whereas Kwan et al. [58] and Colfer and Baldwin [21] suggested that more abstract aggregation units for developers and source-code artifacts—such as teams, entire sub-projects, and components, rather than single persons and single files—potentially mitigate the mirroring effect predicted by Conway [22]. Portillo-Rodríguez et al. [77] aimed at global software engineering and introduced weights into the measurement of socio-technical congruence with respect to geographical distance among developers. In contrast to these studies, we focus on a project-level analysis using a rather simple congruence measure based on the collaboration and communication of individual developers. The analysis of individual coordination gaps is not feasible at the scale of our study,[19] which also holds for the incorporation of advanced distance information into the congruence measure.

In further work, the identification of the "best" abstraction level of source-code artifacts that give rise to coordination requirements is in the focus: López-Fernández et al. [63] analyzed modules, Jermakovics et al. [50] files, and Joblin et al. [51] functions as constructional arguments for coordination requirements. We extend on these studies by adding the abstraction level of *features* and by comparing it to previously considered levels.

Xuan and Filkov [101] as well as Gharehyazie and Filkov [36] analyzed a more dynamic view on developer collaboration and coordination. In both studies, the authors investigated either pairs or groups of developers working on the same source-code artifacts temporally close-by—while not dividing the historical project data into discrete revision ranges. In contrast to our work, they chose a more coarse-grained level of abstraction (the package level) and did not relate developers through conversation-indicating e-mail threads, but rather on a purely time-based mechanism or content analysis. Furthermore, they linked their approach with a code-growth analysis, which is not the focus of our study.

---

[19] This information could be accessed through the motif identification, which we describe in Section 2.3.

Finally, there have been studies on tools that aim at raising developers' awareness by visualizing concurrent work by other developers (tools such as PALANTÍR [83], SVNNAT [87] or TESSERACT [85]). These studies complement our analysis in that our analysis data can be easily plugged into such tools to incorporate further kinds of source-code artifacts for visualization.

9 Conclusion and Future Work

Developer coordination is crucial in large-scale open-source software projects where developers are scattered across the entire planet. It is crucial for efficient participation and contribution of developers as well as for the sustainable evolution of such software projects. A way to assess the proper alignment of communication in software projects is to measure the fulfillment of coordination requirements by communication activity. A coordination requirement arises among developers when they work on the same source-code artifacts and, thus, are required to coordinate their work, for example, to avoid duplication of work and other potential problems. It has been argued in the literature that the gap between required coordination and the actual coordination should be low, to achieve a state of socio-technical congruence to increase the chances for project success or successful evolution.

By means of an empirical study on ten large-scale open-source software projects, we shed further light on this issue. In detail, we measured the fraction of fulfilled coordination requirements over the complete history of the subject projects, altogether making up over 180 years of development history. We investigated whether coordination requirements are fulfilled on purpose, and we could support this with empirical data. Open-source developers seem to take care notably of not easily visible coordination requirements that arise from logically coupled source-code artifacts (i.e., the square motif $m_{\square}$), while they slightly neglect coordination requirements that are relatively easy to resolve and that potentially have less impact (i.e., the triangle motif $m_{\triangle}$). Furthermore, we observed that neither of the considered abstraction levels of source-code artifacts is more suitable as constructional argument for coordination requirements with respect to their fulfillment: For all abstraction levels, we found similar fractions of fulfilled coordination requirements, also when incorporating the dimension of time. Nevertheless, the set of uniquely identified coordination requirements *does* show that distinguishing abstraction levels is important. In particular, we emphasize here our finding that features are not as prominently placed in the development process of most of our subject projects as we expected and is assumed by the research community of feature-oriented software development and feature-driven development. Finally, our qualitative study on the evolution of fulfilled coordination requirements showed that far-reaching social events have a huge impact on the fulfillment of coordination requirements, both negatively and positively. For example, we noticed improvements in the coordination structure of the OPENSSL project after the discovery of the infamous bug HEARTBLEED. In summary, what one can learn from this study is that there is need to run socio-technical analyses to foster the understanding of open-source development communities and to support their evolution to be sustainable.

In future work, we plan to combine our current analyses with more sophisticated network-analytic measures. Furthermore, the analysis of further and also historically newer communication channels, such as GITHUB issues or SLACK chats,

may lead to more valuable insights on how developer communication and coordination changes over time and with the emergence of new tools and platforms.

Acknowledgements We thank Alexander Grebhahn, Angelika Schmid, Thomas Bock, and Christian Kästner for their useful comments on previous versions of this paper and their encouragement. This work was supported by the DFG (German Research Foundation, AP 206/5-1&2, AP 206/6-1&2, and AP 206/14-1). Siegmund's work is funded by the Bavarian State Ministry of Education, Science and the Arts in the framework of the Centre Digitisation.Bavaria (ZD.B) and the DFG (SI 2045/2-2).

References

1. Aljemabi MA, Wang Z (2018) Empirical Study on the Evolution of Developer Social Networks. IEEE Access 6:51049–51060
2. de Andrade HS, Almeida E, Crnkovic I (2014) Architectural Bad Smells in Software Product Lines. In: Proc. Int. Conf. Dependable and Secure Cloud Computing Architecture (DASCCA), ACM, pp 1–6
3. Apel S, Batory D, Kästner C, Saake G (2013) Feature-Oriented Software Product Lines. Springer
4. Argote L (2012) Organizational Learning. Springer
5. Arisholm E, Briand LC, Foyen A (2004) Dynamic Coupling Measurement for Object-Oriented Software. IEEE Transactions on Software Engineering (TSE) 30(8):491–506
6. 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
7. Berger T, Lettner D, Rubin J, Grünbacher P, Silva A, Becker M, Chechik M, Czarnecki K (2015) What is a Feature? In: Proc. Int. Software Product Line Conference (SPLC), ACM, pp 16–25
8. Betz S, Šmite D, Fricker S, Moss A, Afzal W, Svahnberg M, Wohlin C, Börstler J, Gorschek T (2013) An Evolutionary Perspective on Socio-Technical Congruence: The Rubber Band Effect. In: Proc. Int. Workshop on Replication in Empirical Software Engineering Research (RESER), IEEE
9. Bhattacharya P, Iliofotou M, Neamtiu I, Faloutsos M (2012) Graph-Based Analysis and Prediction for Software Evolution. In: Proc. Int. Conf. Software Engineering (ICSE), IEEE, pp 419–429
10. 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
11. 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
12. Blincoe K, Valetto G, Goggins S (2012) Proximity: A Measure to Quantify the Need for Developers' Coordination. In: Proc. Int. Conf. Computer-Supported Cooperative Work (CSCW), ACM, pp 1351–1360
13. Blincoe K, Valetto G, Damian D (2013) Do all Task Dependencies Require Coordination? The Role of Task Properties in Identifying Critical Coordination Needs in Software Projects. In: Proc. Europ. Software Engineering Conf.

and the Int. Symposium Foundations of Software Engineering (ESEC/FSE), ACM, pp 213–223
14. Brandes U, Gaertler M, Wagner D (2003) Experiments on Graph Clustering Algorithms. In: European Symposium on Algorithms (ESA), ESA 2003, Springer, pp 568–579
15. Brooks FP (1995) The Mythical Man-Month, Anniversary Edition: Essays On Software Engineering. Pearson Education
16. 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, Inc., chap 12, pp 221–246
17. 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
18. Cataldo M, Wagstrom PA, Herbsleb JD, Carley KM (2006) Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. In: Proc. Int. Conf. Computer-Supported Cooperative Work (CSCW), ACM, pp 353–362
19. 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, ACM, pp 2–11
20. Cleveland WS (1979) Robust Locally Weighted Regression and Smoothing Scatterplots. Journal of the American Statistical Association 74(368):829–836
21. Colfer LJ, Baldwin CY (2016) The Mirroring Hypothesis: Theory, Evidence, and Exceptions. Industrial and Corporate Change (ICC) 25(5):709–738
22. Conway ME (1968) How Do Committees Invent? Datamation 14(5):28–31
23. Crowston K, Howison J (2005) The Social Structure of Free and Open Source Software Development. First Monday 10(2)
24. Csardi G, Nepusz T (2006) The IGRAPH Software Package for Complex Network Research. InterJournal Complex Systems:1695, URL http://igraph.org
25. Curtis B, Krasner H, Iscoe N (1988) A Field Study of the Software Design Process for Large Systems. Communications of the ACM 31(11):1268–1287
26. Dabbish L, Stuart C, Tsay J, Herbsleb J (2012) Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In: Proc. Int. Conf. Computer-Supported Cooperative Work (CSCW), ACM, pp 1277–1286
27. 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
28. Ehrlich K, Helander M, Valetto G, Davies S, Williams C (2008) An Analysis of Congruence Gaps and Their Effect on Distributed Software Development. In: Int. Workshop on Socio-Technical Congruence (STC), available online.
29. Ernst MD, Badros GJ, Notkin D (2002) An Empirical Analysis of C Preprocessor Use. IEEE Transactions on Software Engineering (TSE) 28(12):1146–1170
30. Espinosa JA (2002) Shared Mental Models and Coordination in Large-scale, Distributed Software Development. PhD thesis, Graduate School of Industrial Administration, Carnegie Mellon University

31. Espinosa JA, Lerch FJ, Kraut RE (2002) Explicit vs. Implicit Coordination Mechanisms and Task Dependencies: One Size Does Not Fit All. In: Team Cognition: Understanding the Factors that Drive Process and Performance, American Psychological Association, pp 107–129
32. Feigenspan J, Kästner C, Apel S, Liebig J, Schulze M, Dachselt R, Papendieck M, Leich T, Saake G (2012) Do Background Colors Improve Program Comprehension in the #ifdef Hell? Empirical Software Engineering 18(4):699–745
33. Fenske W, Schulze S (2015) Code Smells Revisited: A Variability Perspective. In: Proc. Int. Workshop on Variability Modeling of Software-Intensive Systems (VaMoS), ACM, pp 3–10
34. 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 Conf. Source Code Analysis and Manipulation (SCAM), IEEE, pp 171–180
35. Gall H, Hajek K, Jazayeri M (1998) Detection of Logical Coupling Based on Product Release History. In: Proc. Int. Conf. Software Maintenance (ICSM), IEEE, pp 190–198
36. Gharehyazie M, Filkov V (2017) Tracing Distributed Collaborative Development in Apache Software Foundation Projects. Empirical Software Engineering 22(4):1795
37. Gkantsidis C, Mihail M, Zegura EW (2003) The Markov Chain Simulation Method for Generating Connected Power Law Random Graphs. In: Proc. Workshop Algorithm Engineering and Experiments (ALENEX), SIAM, pp 16–25
38. Gneiting T, Ševčíková H, Percival DB (2012) Estimators of Fractal Dimension: Assessing the Roughness of Time Series and Spatial Data. Statistical Science 27(2):247–277
39. Gobbi A, Iorio F, Albanese D, Jurman G, Saez-Rodriguez J (2017) BiRewire: High-Performing Routines for the Randomization of a Bipartite Graph (or a Binary Event Matrix), Undirected and Directed Signed Graph Preserving Degree Distribution (or Marginal Totals). URL `https://www.bioconductor.org/packages/release/bioc/html/BiRewire.html`
40. Grinter RE (1998) Recomposition: Putting It All Back Together Again. In: Proc. Int. Conf. Computer-Supported Cooperative Work (CSCW), ACM, pp 393–402
41. Grinter RE, Herbsleb JD, Perry DE (1999) The Geography of Coordination: Dealing with Distance in R&D Work. In: Proc. Int. Conf. Supporting Group Work (GROUP), ACM, pp 306–315
42. Gutwin C, Greenberg S (1999) The Effects of Workspace Awareness Support on the Usability of Real-time Distributed Groupware. ACM Transactions on Computer-Human Interaction 6(3):243–281
43. Herbsleb JD, Grinter RE (1999) Architectures, Coordination, and Distance: Conway's Law and Beyond. IEEE Software 16(5):63–70
44. Herbsleb JD, Grinter RE (1999) Splitting the Organization and Integrating the Code: Conway's Law Revisited. In: Proc. Int. Conf. Software Engineering (ICSE), ACM, pp 85–95
45. Herbsleb JD, Mockus A (2003) An Empirical Study of Speed and Communication in Globally Distributed Software Development. IEEE Transactions on Software Engineering (TSE) 29(6):481–494

46. 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
47. Herbsleb JD, Roberts JA (2006) Collaboration In Software Engineering Projects: A Theory Of Coordination. In: Proc. Int. Conf. Information Systems (ICIS), Association for Information Systems, pp 553–568
48. Hindle A, Germán DM, Holt RC (2008) What do Large Commits Tell Us?: A Taxonomical Study of Large Commits. In: Proc. Working Conf. Mining Software Repositories (MSR), pp 99–108
49. 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 21(2):449–482
50. 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
51. 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), ACM, pp 563–573
52. Joblin M, Apel S, Mauerer W (2016) Evolutionary Trends of Developer Coordination: A Network Approach. Empirical Software Engineering 22(4):2050–2094
53. Joblin M, Apel S, Hunsen C, Mauerer W (2017) 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
54. Kossinets G (2006) Effects of Missing Data in Social Networks. Social Networks 28(3):247–268
55. Kwan I, Damian D (2011) Extending Socio-Technical Congruence with Awareness Relationships. In: Proc. Int. Workshop on Social Software Engineering (SSE), ACM
56. Kwan I, Schröter A, Damian D (2009) A Weighted Congruence Measure. In: Int. Workshop on Socio-Technical Congruence (STC), available online.
57. Kwan I, Schröter 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
58. Kwan I, Cataldo M, Damian D (2012) Conway's Law Revisited: The Evidence for a Task-Based Perspective. IEEE Software 29(1):90–93
59. Levesque LL, Wilson JM, Wholey DR (2001) Cognitive Divergence and Shared Mental Models in Software Development Project Teams. Journal of Organizational Behavior 22(2):135–144
60. Li J, Carley KM, Eberlein A (2012) Assessing Team Performance from a Socio-Technical Congruence Perspective. In: Proc. Int. Conf. Software and System Process (ICSSP), IEEE, pp 160–169
61. 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
62. Liebig J, Kästner C, Apel S (2011) Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In: Proc. Int. Conf. Aspect-

Oriented Software Development (AOSD), ACM, pp 191–202
63. López-Fernández L, Robles G, González-Barahona JM (2004) Applying Social Network Analysis to the Information in CVS Repositories. In: Proc. Int. Workshop Mining Software Repositories (MSR), pp 101–105
64. MacCormack A, Baldwin C, Rusnak J (2012) Exploring the Duality Between Product and Organizational Architectures: A Test of the "Mirroring" Hypothesis. Research Policy 41(8):1309–1324
65. Malone TW, Crowston K (1990) What is Coordination Theory and How Can It Help Design Cooperative Work Systems? In: Proc. Int. Conf. Computer-Supported Cooperative Work (CSCW), ACM, pp 357–370
66. Mathieu JE, Heffner TS, Goodwin GF, Salas E, Cannon-Bowers JA (2000) The Influence of Shared Mental Models on Team Process and Performance. Journal of Applied Psychology 85(2):273–283
67. Meneely A, Williams L (2011) Socio-Technical Developer Networks: Should We Trust our Measurements? In: Proc. Int. Conf. Software Engineering (ICSE), ACM, pp 281–290
68. Milo R, Kashtan N, Itzkovitz S, Newman MEJ, Alon U (2004) On the Uniform Generation of Random Graphs with Prescribed Degree Sequences. arXiv e-prints cond-mat/0312028v2
69. Mitchell BS, Mancoridis S (2006) On the Automatic Modularization of Software Systems using the Bunch Tool. IEEE Transactions on Software Engineering 32(3):193–208
70. 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
71. Nagappan N, Murphy B, Basili V (2008) The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In: Proc. Int. Conf. Software Engineering (ICSE), ACM, pp 521–530
72. de Oliveira MC, Bonifácio R, Ramos GN, Ribeiro M (2016) Unveiling and Reasoning about Co-Change Dependencies. In: Proc. Int. Conf. Modularity (MODULARITY), ACM Press, pp 25–36
73. Palmer SR, Felsing JM (2002) A Practical Guide to Feature-Driven Development. Prentice-Hall
74. Parnas DL (1972) On the Criteria to Be Used in Decomposing Systems into Modules. Communications of the ACM 15(12):1053–1058
75. Passos L, Padilla J, Berger T, Apel S, Czarnecki K, Valente MT (2015) Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In: Proc. Int. Conf. Modularity (MODULARITY), ACM, pp 81–92
76. Passos L, Queiroz R, Mukelabai M, Berger T, Apel S, Czarnecki K, Padilla J (2018) A Study of Feature Scattering in the Linux Kernel. IEEE Transactions on Software Engineering (TSE) pp 1–16, online first
77. Portillo-Rodríguez J, Vizcaíno A, Piattini M, Beecham S (2014) Using Agents to Manage Socio-Technical Congruence in a Global Software Engineering Project. Information Sciences 264:230–259
78. Poshyvanyk D, Marcus A, Ferenc R, Gyimóthy T (2009) Using Information Retrieval Based Coupling Measures for Impact Analysis. Empirical Software Engineering 14(1):5–32
79. Queiroz R, Passos L, Valente MT, Hunsen C, Apel S, Czarnecki K (2015) The Shape of Feature Code: An Analysis of Twenty C-Preprocessor-Based

Systems. Software and Systems Modeling (SoSyM) 16(1):77–96
80. R Core Team (2016) R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, URL `https://www.r-project.org`
81. 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
82. Rouse WB, Cannon-Bowers JA, Salas E (1992) The Role of Mental Models in Team Performance in Complex Systems. IEEE Transactions on Systems, Man, and Cybernetics 22(6):1296–1308
83. Sarma A, Noroozi Z, van der Hoek A (2003) Palantir: Raising Awareness among Configuration Management Workspaces. In: Proc. Int. Conf. Software Engineering (ICSE), IEEE, pp 444–454
84. Sarma A, Herbsleb J, van der Hoek A (2008) Challenges in Measuring, Understanding, and Achieving Social-Technical Congruence. Tech. rep., Institute for Software Research, Carnegie Mellon University
85. Sarma A, Maccherone L, Wagstrom P, Herbsleb J (2009) Tesseract: Interactive Visual Exploration of Socio-Technical Relationships in Software Development. In: Proc. Int. Conf. Software Engineering (ICSE), IEEE, pp 23–33
86. Schroeder M (1992) Fractals, Chaos, Power Laws: Minutes from an Infinite Paradise. W. H. Freeman, pp. 211–215
87. Schwind M, Wegmann C (2008) SVNNAT: Measuring Collaboration in Software Development Networks. In: Proc. Int. Conf. E-Commerce Technology and Int. Conf. Enterprise Computing, E-Commerce, and E-Services (CEC/EEE), IEEE, pp 97–104
88. 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 1–10
89. Sevcikova H, Percival D, Gneiting T (2014) FRACTALDIM: Estimation of Fractal Dimensions. URL `https://CRAN.R-project.org/package=fractaldim`
90. Shen-Orr SS, Milo R, Mangan S, Alon U (2002) Network Motifs in the Transcriptional Regulation Network of Escherichia coli. Nature Genetics 31(1):64–68
91. Sierra JM, Vizcaíno A, Genero M, Piattini M (2018) A Systematic Mapping Study about Socio-Technical Congruence. Information and Software Technology (IST) 94:111–129
92. Sommerville I (2010) Software Engineering, nineth edn. Addison-Wesley
93. Sosa ME, Eppinger SD, Pich M, McKendrick DG, Stout SK (2002) Factors that Influence Technical Communication in Distributed Product Development: An Empirical Study in the Telecommunications Industry. IEEE Transactions on Engineering Management 49(1):45–58
94. 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
95. Storey MA, Singer L, Figueira Filho F, Zagalsky A, German DM (2016) How Social and Communication Channels Shape and Challenge a Participatory Culture in Software Development. IEEE Transactions on Software Engineering (TSE) 41(7):1–20

96. Stout R, Salas E (1993) The Role of Planning in Coordinated Team Decision Making: Implications for Training. Proc Human Factors and Ergonomics Society Annual Meeting 37(18):1238–1242
97. 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
98. Valetto G, Helander M, Ehrlich K, Chulani S, Wegman M, Williams C (2007) Using Software Repositories to Investigate Socio-technical Congruence in Development Projects. In: Proc. Int. Workshop Mining Software Repositories (MSR), IEEE, pp 25:1–25:4
99. Valetto G, Chulani S, Williams C (2008) Balancing the Value and Risk of Socio-Technical Congruence. In: Int. Workshop on Socio-Technical Congruence (STC), available online.
100. Wagstrom P, Herbsleb JD, Carley KM (2010) Communication, Team Performance, and the Individual: Bridging Technical Dependencies. Academy of Management Proceedings 2010(1):1–7
101. Xuan Q, Filkov V (2014) Building it Together: Synchronous Development in OSS. In: Proc. Int. Conf. Software Engineering (ICSE), ACM, pp 222–233
102. 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
103. Xuan Q, Fang H, Fu C, Filkov V (2015) Temporal Motifs Reveal Collaboration Patterns in Online Task-Oriented Networks. Physical Review E 91:052813
104. Zimmermann T, Weibgerber P, Diehl S, Zeller A (2004) Mining Version Histories to Guide Software Changes. In: Proc. Int. Conf. Software Engineering (ICSE), IEEE, pp 563–572