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

On the Relation Between GitHub Communication Activity and Merge Conflicts

Gustavo Vale · Angelika Schmid · Alcemir Rodrigues Santos · Eduardo Santana de Almeida · Sven Apel

Received: date / Accepted: date

Abstract Version control systems assist developers in managing concurrent changes to a common code base by tracking all code contributions over time. A notorious problem is that, when integrating code contributions, merge conflicts may occur and resolving them is a time-consuming and error-prone task. There is a popular belief that communication and collaboration success are mutually dependent. So, it is believed that great communication activity helps to avoid merge conflicts. However, in practice, the role of communication activity for merge conflicts to occur or to be avoided has not been thoroughly investigated. To better understand this relation, we analyzed the history of 30 popular open-source projects involving 19 thousand merge scenarios. Methodologically, we used a bivariate (Spearman's rank correlation) and a multivariate (principal component analysis and partial correlations) analysis to quantify their correlation. In bivariate analysis, we found a weak positive correlation between GitHub communication activity and the number of merge conflicts. However, in the multivariate analysis, the positive correlation disappeared, not supporting the intuition that GitHub communication helps to avoid merge conflicts. Interestingly, we found that the strength of this relationship depends on the merge scenarios' characteristics, such as the number of lines of code changed. Puzzled by these unexpected results, we investigated each covariate, which provided justifications for our findings. The main conclusion from our study is that GitHub communication activity itself does not support the emergence or avoidance of merge conflicts even though such communication is associated only with merge scenario code changes or among developers only.

Keywords Collaborative Software Development · Version Control Systems · Developer Communication · Merge Conflicts

G. Vale* · A. Schimid* · A. R. Santos** · E. S. Almeida*** · S. Apel****
*University of Passau **State University of Piauí ***Federal University of Bahia
****Saarland University
E-mail: vale@fim.uni-passau.de, angelika.schimid@uni-passau.de,
alcemir@prp.uespi.br, esa@rise.com.br, apel@cs.uni-saarland.de

1 Introduction

Software development is a collaborative and distributed activity in which success depends on the ability to coordinate social and technical assets [23]. In this collaborative process, developers are often supported by version control systems when solving tasks (e.g., bug fixing and adding new features). Version control systems help them to manage changes to a common code base by tracking all code contributions over time. This allows a group of developers to address different tasks simultaneously without losing changes. After fulfilling their tasks, developers merge the proposed changes to the main repository.

Simultaneous contributions to a common code base may introduce problems of their own during integration, often manifesting as merge conflicts. A *merge conflict* occurs when changes to the same chunk of code are merged. Merge conflicts have been attracting researchers' and practitioners' attention for years, because resolving them is a difficult, time-consuming, and often error-prone task [34]. As merge conflicts are unexpected events, they have a negative effect on project's objectives compromising the project success, especially when arising frequently [20] [39] [28]. On the other hand, researchers found that a proper communication among contributors is fundamental for the project success [7] [19] [40] [43]. For instance, Liu et al. [32] found that GitHub communication supports a more coordinated development activity than when developers do not use GitHub communication features, such as pull requests. Despite the number of studies exploring merge conflicts [1] [3] [10] [20] [31] and communication activity [11] [16] [24] [37] [42] [46], the role of communication activity for the occurrence or avoidance of merge conflicts in practice has not been thoroughly investigated.

Our goal is to investigate and understand the relation between GitHub communication activity and merge conflicts. One of the reasons why communication is related to project success may be that keeping contributors aware of what others are doing may avoid merge conflicts. Hence, to get a more precise understanding about what kind of communication may be helpful for avoiding merge conflicts, we use different measures of communication. For instance, the communication related to the merge scenario's code changes may be more efficient for avoiding merge conflicts than the general communication in the merge scenario. Or, the communication among only *developers* may be more important for avoiding merge conflicts than the communication among *contributors*. In this setting, contributors are all GitHub users who have contributed to the project (e.g., communicating or changing the source code); a developer is a contributor who has changed the source code.

To achieve our goal, we have conducted a large empirical study analyzing the history of 30 repositories of popular software projects. In total, we considered 19 thousand merge scenarios, 325 thousand files, and 1.5 million chunks. For this purpose, we mined and linked contribution (Git) and communication (GitHub) data. Regarding code contributions, we reconstructed the merge scenarios that are present in the subject projects' histories. Regarding communication activity, we quantified the amount of GitHub com-

munication in merge scenarios by means of three alternative approaches with distinct granularity: one considering the communication of all active contributors (*awareness-based*), the second linking communication related to the merge scenario's contributions by means of pull requests and related issues (*pull-request-based*), and the third considering communication mapped to artifacts that have been changed in the merge scenario (*changed-artifact-based*). To obtain a deep understanding of the communication activity, we also distinguished between the communication related to contributors (*contributors' communication*) and developers (*developers' communication*), for each approach.

To understand the association between GitHub communication activity and the occurrence of merge conflicts (i.e., the two covariates), we performed three analyses. First, we analyzed the *bivariate* relationship between the two covariates, as is common in empirical software engineering studies. Second, we analyzed the *multivariate* relationship between the two covariates taking confounding factors into account (e.g., the number of files changed and developers involved in the merge scenario). Third, we analyzed the *moderating influence* of individual merge scenario characteristics on the strength of the relation of the two covariates (e.g., the relation may be stronger in larger merge scenarios). For these analyses, we use Spearman's rank correlation, principal component analysis, partial correlation, and moderation effects.

Summarizing our results, the bivariate analysis indicates a weak (< 0.3) but highly significant positive correlation between the number of merge conflicts and the amount of communication. The multivariate analysis indicates no relation between the two covariates, which suggests that the positive relation between the two covariates (communication activity measure and number of merge conflicts) found in the bivariate analysis is spurious under the assumption that both covariates are confounded by merge scenario characteristics. In the moderation effect analysis, we investigated the influence of three measures on the strength of the relation of the two covariates: the number of lines of code, the number of developers involved, and the number of days a merge scenario lasts. Regarding number of lines of code, we found a significant moderating influence on the strength of the relation between the two covariates for the awareness- and changed-artifact-based approaches and for both communication measures. Regarding number of developers, we also found a significant moderating influence on such relation, however, the influence lasts only for the contributors' communication. Regarding number of days, we found no significant moderating influence on the strength of the investigated relation. Note that the moderating effect analysis does not invalidate the multivariate analysis, it only presents results for the subset of "larger" merge scenarios. In practical terms, our results *contradict* the popular belief that suggests that a high communication activity helps to avoid merge conflicts.

Puzzled by our unexpected and negative results and aiming to get deeper into of all our covariables, we analyzed each of them separately, which supports the robustness and reliability of our methodology. From this further analysis, several topics for discussions arose. Most notably, (i) bivariate analysis is not enough to investigate the complex interplay of project success, communica-

tion, merge conflicts, and contextual factors, (ii) contributors and developers normally communicate independently of the emergence of merge conflicts, (iii) the size of the merge scenario's code change is not related to the number of developers involved, and (iv) speculative merge strategies (e.g., GitHub pull requests) drastically reduce the number of merge conflicts.

Overall, we make the following contributions:

- We present evidence that communication activity and merge conflicts are not related in the general case when controlling for confounding factors.
- We provide evidence that the developers' communication has a negative influence on the emergence of merge conflicts for the 10% largest merge scenarios in terms of lines of code. On the other hand, contributor's communication for the same setting has a positive influence on the emergence of merge conflicts. Therefore, developers' communication is more efficient for avoiding merge conflicts in the 10% largest merge scenarios than contributors' communication.
- We offer a rigorous methodological approach to multivariate analysis of correlation structures in socio-technical repository data analysis.
- We provide evidence by a manual analysis that merge scenarios with few developers and large changes are often related to bug fixing while merge scenarios with many developers and small changes are often related to the introduction of new features to the project.
- We provide evidence of the benefits of using speculative merge strategies (e.g., GitHub pull requests) by showing that the percentage of conflicting merge scenarios without using pull requests is 139 times greater than when using pull requests.
- We make our infrastructure publicly available to mine fine-grained information from software repositories.
- We make all data publicly available for replication and follow-up studies on a supplementary Web site [48].

2 Background and Related Work

In this section, we discuss background and related work on analyzing merge conflicts and communication among contributors and developers.

2.1 Collaborative Software Development and Merge Conflicts

Version control systems, such as Git, help developers to manage source-code changes over time by tracking all code modifications [50]. This allows developers to make concurrent contributions without losing changes. This way, multiple contributors may add new features or fix bugs simultaneously. After fulfilling their tasks, developers merge the proposed changes into the main repository. Developing software by means of merging changes into the main repository is a widely collaborative development pattern, called the *pull-based*

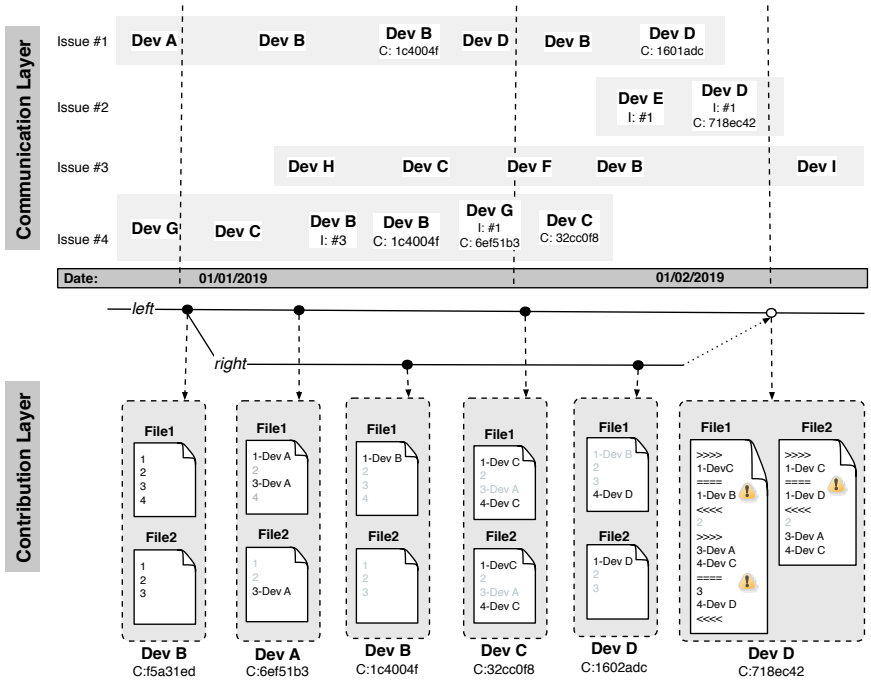

Fig. 1 Communication and contribution layers of a merge scenario. Light-gray boxes in the communication layer stand for issues and white boxes inside these light-gray boxes denote GitHub events (e.g., comments). We highlight three pieces of information: the developer who creates the event, related commits, and related issues. Commits are highlighted by "C:" followed by the commit hash, and issues by "I:" and the GitHub issue number. Regarding the contribution layer, each black dot represents a commit. We highlight four pieces of information the file name, the changed lines, the commit author, and the commit hash. Chunks that are in conflict are in evidence by the exclamation icon.

development model [16] [17]. Figure 1 exemplifies this model. A *merge scenario* in the pull-based model includes the whole timeline of creating a project branch, committing changes independently to the branch, and creating a merge commit (e.g., using a pull request). It is also called three-way merge [17] [31].

The contribution layer (bottom) in Figure 1 illustrates a merge scenario involving four developers of which developers A and C were fixing a bug while developers B and D were adding a new feature to the project. Developers changed four chunks of code of two files (File1 and File2). Three of these four chunks give rise to merge conflicts. Merge conflicts are a notorious problem in collaborative software development [34]. Resolving them and making sure that unexpected behavior or bugs have not been introduced is a time-consuming and error-prone task [31]. Hence, researchers seek to minimize the number of merge conflicts by proposing: (i) *merge strategies* (e.g., structured [2] or semi-structured [3]), (ii) *prediction strategies* (e.g., continuous integration [20] and speculative merging [10]), (iii) *awareness tools* (e.g., Col-

labVS [12], Palantír [39], Cassandra [28], and FASTDash [6]), (iv) *studies to understand their nature* (e.g., identifying the types of code changes that lead to conflicts) [1] [15] [31], and (v) *merge conflict resolution strategies* [33] [35]. Merge strategies have reduced the occurrence of merge conflicts by avoiding conflicts resulting from formating, ordering, and renaming changes [2] [3] [31]. Prediction strategies and awareness tools avoid merge conflicts by continuous integration [10] [20] or by making developers aware of changes of other developers [28] [39]. Studies to understand merge conflicts have identified common patterns and types of changes that often lead to conflicts [1] [15] [31]. Merge conflict resolution strategies, as the name suggests, investigate strategies that developers follow to remove merge conflicts from the code [33] [35]. So, previous studies have provided mechanisms to avoid, minimize, understand, and remove merge conflicts, but, they largely ignore the social dimension of the problem. Since developers create, resolve, or avoid merge conflicts, our key assumption is that, by understanding the relation of GitHub communication practices and the occurrence of merge conflicts, we will obtain insights into the communication practices that are useful to avoid merge conflicts. For instance, we investigate whether the communication among developers is more efficient for avoiding merge conflicts than the communication of all contributors in the merge scenario.

2.2 Communication Flow

As software development often requires social interaction, it is no surprise that software engineers spend a large part of their workday communicating with coworkers [5]. Numerous studies highlight the importance of communication in various software development activities. For instance, Souza et al. [43] provide evidence that communication among contributors is required for the success of software projects. Bird et al. [7], Grinter et al. [19], and Sedano et al. [40] stress that the lack of communication is a critical problem in distributed software development. So, if communication is uncertain, inaccurate, or slow, misunderstandings among developers become likely, compromising the project budget and schedule. In this sense, a proper communication culture is fundamental to support stakeholders in being aware of the project progress, to avoid of merge conflicts, for instance.

Communication channels play an essential role in supporting communication and collaboration activities within a community of practice [44]. Various researchers have investigated developer collaboration through communication channels and tools, such as mailing lists, IRC chat logs, issue trackers, and social networks (e.g., GitHub and Stack Overflow) [8] [11] [16] [21] [24] [30] [32] [37] [42] [46]. For instance, Bird et al. [8] explored the relationship between communication structure and code modularity. They found a relation between communication and code collaboration behavior for sub-communities. Guzzi et al. [21] analyzed a large sample of e-mail threads from Apache Lucene's development mailing list. They found that developers participate in less than

75% of the threads, and in only about 35% of the threads source-code details are discussed. LaToza et al. [30] interviewed eleven developers to learn about common practices in software development. They found several barriers preventing e-mail usage and they highlighted advantages of face-to-face communication and that the use of more interactive communication channels is more desirable than e-mails. Panichella et al. [37] analyzed three communication channels (mailing lists, issue trackers, and IRCs) and code changes of seven projects. They found that not all developers use all communication channels, and socio-technical relationships may change when using different communication channels and tools.

In an extensive study, Storey et al. [44] mapped different communication tools, such as e-mail lists, IRCs, SourceForge, GitHub, and Stack Overflow. They hypothesized that knowledge in software engineering is embedded in: (i) people's heads, (ii) project artifacts, (iii) community resources, such as forums, blogs, and discussion groups, and, (iv) social networks. According to their study, GitHub is the only tool able to represent (the last) three types of knowledge.

The popularity of pull-based development model and GitHub has attracted the interest of researchers. For example, Singer et al. [42] and Dabbish et al. [11] explored the value of social mechanisms in GitHub. Both studies found that transparency helps developers to connect, collaborate, create communities, share knowledge, and discover new technologies. Tsay et al. [46] analyzed the association of various technical and social measures with the likelihood of contribution acceptance. They found that pull request acceptance is related to: (i) the strength of the social connection between the submitter and the project manager, (ii) the submitter's prior interaction, (iii) the number of comments, and (iv) the current stage of the project. Gousios et al. [16] analyzed millions of pull requests to study the effectiveness and efficiency of contributors handling pull requests. They discovered that the time to merge a pull request is influenced by the developer's previous track record, the size of the project and its test coverage, and the project's openness to external contributions. Liu et al. [32] conducted a quantitative study on the specific effects of pull requests in the project. They found that pull requests help increasing the social impact, resulting in more coordinated development activity.

Considering experience from previous work [21] [30] [37] and the benefits, comprehensiveness, and popularity of GitHub, we chose to rely in our study on the GitHub platform. So, unlike related work, we use GitHub to build communication networks and use the time a merge scenario lasts to define the analysis time span (i.e., not a predefined one). Being aware of the drawbacks of using only one communication channel and considering that contributors may talk about topics not related to the code changes (e.g., usability or configuration problems), we select only projects that extensively use GitHub communication mechanism, pursue three approaches to capture communication amongst contributors, and differentiate the communication of among all contributors (*contributors' communication*) and among developers only (*developers' communication*), as we will discuss in Section 3 in more detail.

3 Building Communication Networks

In the previous section, we presented how related work have explored merge conflicts, communication activity, as well as the pull-based development model and the communication flow in the pull-based development model. In addition, we presented how communication activity may be useful for avoiding merge conflicts given the popular belief that communication and collaboration activities are mutually dependent for the project success. Aiming at providing a clearer understanding of which GitHub communication activity may be more relevant for avoiding merge conflicts, we create communication networks for each merge scenario using three approaches: *awareness-based*, *pull-request-based*, and *changed-artifact-based*, which vary in terms of granularity and coverage. As detailed in Section 4, we used mainly the number of edges of each graph to answer the research questions, however, having such detailed information supported us in illustrating the merge scenario of Figure 1, as well as, in further investigations presented in Section 6.

Communication networks are built from operational data from GitHub. Specially, we queried the GitHub API retrieving all issue events (e.g., labeling, commenting, and opening) from each issue of each subject project. A network can be formalized as a graph $G = (V, E)$, where V is a set of vertices (contributors) and E is a set of edges (communication edges), denoted by *V(G)* and *E(G)*, respectively. An edge $e \in E$ between $u \in V$ and $v \in V$ is denoted by $e = \{u, v\}$. The three communication approaches as well as their purposes are described as follows.

Awareness-based approach. This approach links communication and contribution data by means of active contributors during a merge scenario. It tries to minimize the threat of using only one channel to capture a project's communication by building a graph of all contributors that communicate during a merge scenario. There are six steps to build communication networks using this approach (see Algorithm 1). For each merge scenario, we get all GitHub events during the merge scenario time range (step 1.1) and retrieve the issues these events belong to (step 1.2). Then, we determine all events related to these issues (step 1.3) and exclude events that happened after the merge, because these events are out of scope since the issue has already been addressed (step 1.4). Finally, we retrieve the set of developers who created the events (step 1.5) and build a full graph with them (step 1.6).

Pull-request-based approach. This approach links communication and contribution data by means of pull requests and their related issues. It is motivated by the flow of communication in pull-based development model (see Section 2.2). It is meant to retrieve a refined view on communication compared to the awareness-based approach, since it considers only communication of some issues that are related to the merge scenario and not all issues opened during the merge scenario. There are six steps to build communication networks using the pull-request-based approach (see Algorithm 2). For each merge scenario, we look for a pull request with the same hash as the merge commit (step 2.1) and mine the pull request body and comments to find related issues (step 2.2).

Algorithm 1 Awareness-based approach

input : MS, I ▷ *Sets of merge scenarios and issues*
output : NET ▷ *A tuple with a graph for each approach and merge scenario*
$E \leftarrow \{\ e \mid e \in \bigcup_{i \in I}\ i.events\ \}$ ▷ *Get all events of all issues*
for each ms in MS **do**
  $E_{ms} \leftarrow \{e \mid\ e \in E \wedge e.time \in [ms.bTime,\ ms.mTime]\}$ ▷ *Step 1.1*
  $I_{ms} \leftarrow \{e.issue \mid\ e \in E_{ms}\}$ ▷ *Step 1.2*
  $E_{ms} \leftarrow E_{ms} \cup (\ \bigcup_{i \in I_{ms}}\ i.events)$ ▷ *Step 1.3*
  $E_{ms} \leftarrow E_{ms} \setminus \{\ e \mid\ e \in E_{ms} \wedge e.time > ms.mTime\}$ ▷ *Step 1.4*
  $Contrib_{ms} \leftarrow \{e.contributor \mid\ e \in E_{ms}\}$ ▷ *Step 1.5*
  $Ed_{ms} \leftarrow \{\ \{c_1,\ c_2\ \} \mid\ c_1,\ c_2 \in Contrib_{ms} \wedge c_1 \neq c_2\}$ ▷ *Step 1.6*
  $NET.add(ms,\ G(Contrib_{ms},\ Ed_{ms}))$
end for

This mining process consists of cleaning the text (removing blocks of code and external URLs because they may refer to issues of other projects) and looking for the pattern referring to other issues (`i.e., #([0-9]+), e.g., #1 or #123)`. As blocks of code are between three quotation marks (‘‘‘) or indented by four spaces and, as URLs follow the pattern “char, slash (/), and char”, we remove them. Then, we look at each remaining word of the text and check if it contains the pattern #([0-9]+). If so, we retrieve the GitHub issue. Our mining process is reliable since we followed instructions from the GitHub API documentation[1], tested, and we also checked whether the related issue exists in the repository before adding it into our analysis. After that, we get all events that happened in related issues (step 2.3), exclude the ones that happened after the merge commit (step 2.4), retrieve the set of developers that contribute to them (step 2.5), and build a full graph with these developers (step 2.6).

Algorithm 2 Pull-request-based approach

input : MS, I ▷ *Sets of merge scenarios and issues*
output : NET ▷ *A tuple with a graph for each approach and merge scenario*
for each ms in MS **do**
  $pr \leftarrow \{\ i \mid\ i \in I \wedge i.hash = ms.mergeCommitHash\ \}$ ▷ *Step 2.1*
  $RI_{ms} \leftarrow \{\ i \mid\ i \in I \wedge (\ i \in pr.relatedIssues \vee i = pr\ )\ \}$ ▷ *Step 2.2*
  $E_{ms} \leftarrow \{\ e \mid\ e \in (\bigcup_{i \in RI_{ms}}\ i.events)$ ▷ *Step 2.3*
  $E_{ms} \leftarrow E_{ms} \setminus \{\ e \mid\ e \in E_{ms} \wedge e.time > ms.mTime\}$ ▷ *Step 2.4*
  $Contrib_{ms} \leftarrow \{e.contributor \mid\ e \in E_{ms}\}$ ▷ *Step 2.5*
  $Ed_{ms} \leftarrow \{\ \{c_1,\ c_2\ \} \mid\ c_1,\ c_2 \in Contrib_{ms} \wedge c_1 \neq c_2\}$ ▷ *Step 2.6*
  $NET.add(ms,\ G(Contrib_m,\ Ed_{ms}))$
end for

[1] https://developer.github.com/v3/

Algorithm 3 Changed-artifact-based approach

input : MS, I ▷ *Sets of merge scenarios and issues*
output : NET ▷ *A tuple with a graph for each approach and merge scenario*
$E \leftarrow \{\ e \mid\ e \in \bigcup_{i \in I} i.events\ \}$ ▷ *Get all events of all issues*
for each ms in MS **do**
  $F_{ms} \leftarrow \{\ f \mid\ f \in ms.files\ \}$ ▷ *Step 3.1*
  $E_{ms} \leftarrow \{e \mid\ e \in E \wedge e.time \in [ms.bTime,\ ms.mTime]\}$ ▷ *Step 3.2*
  $I_{ms} \leftarrow \{e.issue \mid\ e \in E_{ms}\}$ ▷ *Step 3.3*
  $C_{ms} \leftarrow \{c \mid\ c \in \bigcup_{i \in I_{ms}} i.commits\ \}$ ▷ *Step 3.4*
  $C_{ms} \leftarrow \{c \mid\ c \in C_{ms} \wedge (c.files \in F_{ms} \vee c = ms.mC)\ \}$ ▷ *Step 3.5*
  $I_{ms} \leftarrow \{i \mid\ i \in I_{ms} \wedge i.commits \in C_{ms}\ \}$ ▷ *Step 3.6*
  $E_{ms} \leftarrow E_{ms} \cup (\bigcup_{i \in I_{ms}} i.events)$ ▷ *Step 3.7*
  $E_{ms} \leftarrow E_{ms} \setminus \{\ e \mid\ e \in E_{ms} \wedge e.time > ms.mTime\}$ ▷ *Step 3.8*
  $Contrib_{ms} \leftarrow \{e.contributor \mid\ e \in E_{ms}\}$ ▷ *Step 3.9*
  $Ed_{ms} \leftarrow \{\ \{c_1,\ c_2\ \} \mid\ c_1,\ c_2 \in Contrib_{ms} \wedge c_1 \neq c_2\}$ ▷ *Step 3.10*
  $NET.add(ms,\ G(Contrib_{ms},\ Ed_{ms}))$
end for

Changed-artifact-based approach. The main motivation for this approach is to obtain a finer-grained communication than the awareness-based approach with greater coverage than the pull-request-based approach. To achieve this, it links communication and contribution data by means of changed artifacts (files) referred through commits in opened issues in a merge scenario. In other words, we retrieve only communication via issues that discussed files changed in the merge scenario. So, like the pull-request-based approach, we are able to retrieve communication related to the merge scenario code changes, however, it is not pull request dependent. There are ten steps to build networks using the changed-artifact-based approach (see Algorithm 3). For each merge scenario, we determine the set of files changed in the merge scenario (step 3.1), all events that happened during the merge scenario (step 3.2), the issues each of these events belongs to (step 3.3), and all commits related to these issues (step 3.4). Commits can be related to issues in two ways: (i) contributors link the issue ID in the commit message, hence, they will be referred to in the GitHub API or (ii) contributors mention commit hashes in the issue's body or comments. After the first four steps, we refine the set of commits by keeping only the ones that have changed files modified in the merge scenario or in the merge commit (step 3.5). Then, we refine our set of issues related to the merge scenario by keeping only the ones that refer to the commits that changed files modified in the merge scenario (step 3.6). Next, we get a set of events that belong to these issues (step 3.7), exclude events after the merge (step 3.8), and get a set of contributors who created events in the remaining issues (step 3.9). Finally, we build a full graph with the remaining contributors (step 3.10).

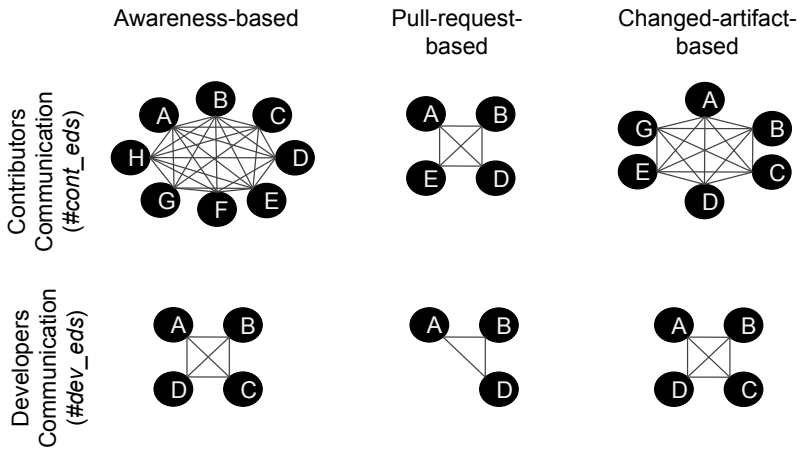

Fig. 2 Communication networks, based on the example of Figure 1.

Note that our setup for the second and third approach ensures that communication is related to the merge scenario code changes. We know all GitHub communication related to a given merge scenario. This does not mean that the developers talked about the conflict or a potential conflict, but that they communicated to make others aware of their code changes. As we consider making developers aware a key to avoid merge conflicts, we would like to know whether communication on a merge scenario (not on something else) leads to less/more merge conflicts. Looking at the amount of communication (e.g., the number of GitHub commentaries) when the conflict happened hints at the merge conflicts *resolution strategies*, which is a different story and not related to our research questions (see Section 4).

Example. The top of Figure 2 shows the three networks created from the example of Figure 1. As we can see, the *awareness-based network* contains more contributors and edges than the other networks, since it includes the communication of all four opened issues during the merge scenario. The *pull-request-based network* contains only contributors from issues #1 and #2, since issue #2 is the pull request (merge commit: 718ec42), and contributors of this issue indicate that issue #1 is related to the problem description. The *changed-artifact-based network* is between the two other networks in terms of size, since it contains developers from issues #1, #2, and #4. Even though issue #4 is not directly related to the merge scenario, two developers who contribute to the merge scenario (Dev B and Dev C) refer to commits present there. Hence, this communication may have been important to make developers aware of the merge scenario code changes.

Note that, in this example, for all communication network approaches, there is communication among non-developers. Therefore, to have an understanding about the communication among developers only, in addition to the three approaches, we distinguish between the communication of all contributors (*#cont_eds*) – *contributors' communication* – from the communication

Table 1 Statistics captured for each merge scenario

Measure	Description
	Merge conflict measure
#conflicts	Number of merge (chunk) conflicts present in the merge scenario
	Communication measures
#cont_eds	Number of pairs of contributors who communicate in a merge scenario
#dev_eds	Number of pairs of developers who modified the code and communicate in a merge scenario
	Context variables
#lines	Number of modified lines of code in the merge scenario
#chunks	Number of chunks modified in the merge scenario
#files	Number of files modified in the merge scenario
#devs	Number of distinct developers who contributed to the merge scenario
#commits	Number of commits in the merge scenario
#days	Number of days a merge scenario lasts

among developers only (*#dev_eds*) – *developers' communication*. The bottom of Figure 2 shows the networks for each approach containing only edges among developers. As we can see, the number of edges among developers is smaller for each approach since the total developers' communication is a subset of the contributors' communication.

4 Empirical Study

In this section, we present our empirical study, whose overall goal is to *investigate and provide an understanding on the role of GitHub communication activity for the occurrence of merge conflicts in the context of the pull-based development model*. First, we describe research questions and hypotheses followed by explanations of how we selected subject projects. Then, we present our approach to retrieve contribution and communication data. Finally, we explain how we answered the research questions.

4.1 Overview of the explored relation in each research question

Table 1 describes all variables we explore in this study. Our discussion of the literature has shown how painful merge conflicts are for the project objectives and how essential communication activity is for the project success (see Section 2). Nevertheless, despite the plausible connection between communication activity and merge conflicts, the role of communication activity for merge conflicts to occur or to be avoided has not been thoroughly investigated. This motivated our first research question:

RQ$_1$: *Is there a correlation between GitHub communication activity and the occurrence of merge conflicts?*

This research question addresses the *direct* relation between communication activity and merge conflicts in our subject projects. This direct correlation between merge conflicts and communication activity may be influenced due to the presence of confounding factors (i.e., merge scenarios' characteristics, such as size, number of developers, and duration), causing a spurious correlation. Only if this correlation still holds true after accounting for confounding factors, an interpretation is legitimate and we may interpret the results. This leads us to our second research question.

RQ$_2$: *How does the correlation between Github communication activity and merge conflicts change when taking confounding factors into account?*

RQ_1 and RQ_2 concentrate on the correlation between GitHub communication activity and merge conflicts for all subject merge scenarios. However, it is likely that the strength of this correlation depends on the characteristics of the merge scenario in question. For instance, it seems reasonable to expect that the correlation is different for small and large merge scenarios. *Moderation effects* arise when a situation where a third variable determines how strong a relationship between two variables is [45]. If we provide evidence that additional covariates influence the strength of the relation of communication activity and the occurrence of merge conflicts, we may find, for instance, that an intensive communication activity becomes fundamental to avoid merge conflicts only in very large merge scenarios. This motivates our third research question:

RQ$_3$: *What is the influence of merge scenario characteristics on the strength of the relation between GitHub communication activity and the occurrence of merge conflicts?*

To answer RQ_3, we formulate the following hypotheses:

H$_1$: *The larger a merge scenario is, the stronger is the relation between GitHub communication activity and merge conflicts.*

H$_2$: *The more developers are involved in a merge scenario, the stronger is the relation between GitHub communication activity and merge conflicts.*

H$_3$: *The longer the merge scenario is, the stronger is the relation between GitHub communication activity and merge conflicts.*

Figure 3 illustrates the relationships we investigate representing research questions by means of the measures presented in Table 1. So, as we use two measures of communication for each out of the three communication network approaches (see Section 3), we analyze the relation between GitHub communication activity and merge conflicts six times for each research question.

4.2 Subject Projects and Experiment Setup

Overall, we selected 30 subject projects from a variety of domains from the hosting platform GitHub. We chose to limit our analysis to Git repositories because it simplifies the identification of merge scenarios in retrospect. The

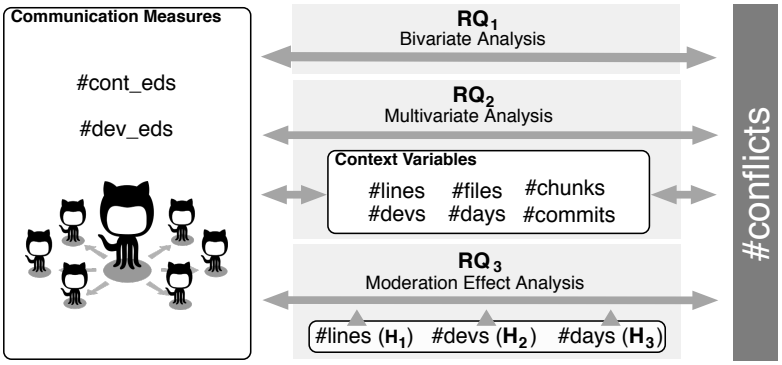

Fig. 3 Relationships investigated for each research question by means of the subject measures.

reasons why we chose GitHub are described on Section 2.2 and 3. We selected the corpus as follows. First, we retrieved the 150 most popular projects on GitHub, as determined by the number of stars [9]. Then, we applied the following five filters: (i) projects that do not have a programming language classified as the main language (i.e., the main file extension), (ii) projects with less than 50 issues and 50 pull requests, (iii) projects with less than two commits and two GitHub events per month in the last six months, (iv) projects in which it was not possible to reconstruct most of the merge scenarios, and (v) the balance of the main programming language of the subject projects.

We created these filters inspired by Kalliamvakou et al. [27]. These filters aim at selecting active projects in terms of code contributions with an active community and at increasing internal validity. For example, the second and third filters capture active community projects on GitHub and not just mirror projects, such as Linux's mirror on GitHub. The fourth filter excludes projects such as kubernetes[2] and moby[3] because we considered that these projects do not mostly use the pull-based model (i.e., do not follow the three-way merge [16]) and they could bias our analyses. Details of how we rebuild merge scenarios are provided in Section 4.3. As most of the popular projects are developed in JavaScript, in the fifth filter, we excluded less popular JavaScripts projects ordered by the number of stars until they accounted for less than half of the subject projects. After applying all filters, we arrived at 30 projects developed in 16 programming languages (i.e., a project can be developed using more than one programming language), such as JavaScript, CSS, and C++, containing around 19 thousand merge scenarios that involve 325 thousand files changed, 1.5 million chunks, 14 thousand contributors, and 134 thousand commits. Table 2 provides information and statistics of each subject project. More details, such as the subject project's URLs, are available on the supplementary Web site [48].

[2] https://github.com/kubernetes/kubernetes
[3] https://github.com/moby/moby

Table 2 Overview of the subject projects

Subject Project Name	Main Prog. Language	#Stars	#MS	#Files	#Cont.
animate.css	CSS	34 290	151	1 392	186
javascript	JavaScript	73 792	548	1 871	727
jquery	JavaScript	49 498	248	4 322	632
vue	JavaScript	108 362	160	2 258	523
html5-boilerplate	JavaScript	41 001	229	1 274	378
electron	C++	62 713	2 845	40 540	1 013
awesome-python	Python	52 886	434	469	661
reveal.js	JavaScript	41 519	612	6 617	469
Semantic-UI	JavaScript	42 171	719	51 856	421
socket.io	JavaScript	42 871	422	3 785	346
express	JavaScript	39 339	284	2 608	455
redux	JavaScript	42 784	469	6 199	883
moment	JavaScript	37 973	991	24 621	831
create-react-app	JavaScript	52 636	524	14 568	963
nw.js	C++	34 057	539	13 598	161
impress.js	JavaScript	33 747	158	584	184
Chart.js	JavaScript	33 359	669	6 603	562
flask	Python	37 517	667	7 202	730
material-design-lite	HTML	30 411	674	6 841	258
httpie	Python	36 137	76	412	132
material-design-icons	CSS	35 462	22	3 707	48
jekyll	Ruby	34 899	1 464	16 703	1 068
AFNetworking	Objective-C	31 328	701	4 213	657
thefuck	Python	36 360	354	3 705	153
normalize.css	CSS	31 763	84	353	138
requests	Python	33 652	1 144	7 365	788
RxJava	Java	34 395	1470	36 642	360
public-apis	Python	40 317	560	887	472
lantern	Go	36 312	1 616	53 472	94
awesome-machine-learning	Python	34 290	408	415	343

#Stars, #MS, #Files, and #Cont. denote the number of GitHub stars, the number of merge scenarios, the number of changed files, and the number of contributors.

We rebuilt the merge scenarios from the subject projects since their creation until July 2018. Aiming at a fairer analysis, Table 3 presents four refinements that we did in our merge scenario dataset: (i) keep only scenarios created after opening the first GitHub issue of the project, because it is not possible to recover communication from before, (ii) keep only merge scenarios from which more than one branch has been touched, because only in these cases merge conflicts may arise, (iii) keep only merge scenarios to which multiple developers were contributing, because only these scenarios need to keep developers aware of the change of others, and (iv) keep only merge scenarios that have been integrated using pull requests. The last refinement is applied only for the analysis using the pull-request-based approach (see Section 3). We discuss insights from this table in Section 6 since they are important to understand contribution activity, but not fundamental to answer our research questions.

Table 3 Overview of the refinements applied to our dataset of merge scenarios

Refinement	#MS	#MS by IN	#CMS	#CMS by #MS
Initial number (IN)	19 232	100.00%	1 079	5.61%
Possible to communicate	18 607	96.75%	1 041	5.59%
Both branches touched	7 769	40.40%	1 041	13.40%
Multiple developers	6 487	33.73%	858	13.23%
Use pull requests	3 436	17.87%	7	0.20%

#MS and #CMS denote the number of merge scenarios and the number of conflicting merge scenarios.

4.3 Data Acquisition

Given that software development is social in nature, we build socio-technical relationships to obtain an authentic representation of developers' contribution and communication.

Code Contributions. Our strategy for contribution data acquisition consists of five steps. First, we clone a subject project's repository. Second, as merge commits can be identified in Git when the number of parent commits is greater than one, we identify merge scenarios by filtering commits with multiple parent commits. Third, for each merge commit, we retrieve a common ancestor for both parent commits (i.e., the base commit). Fourth, we (re)merge the parent commits and retrieve the measures for the metrics presented in Table 1 (except communication measures) by comparing the changes that occurred since the base commit until the merge commit. Finally, we store all data and repeat steps 3 to 5 for each merge scenario found in the step 2.

Note that we have excluded merge scenarios that do not have a base commit (e.g., rebase, fast-forward, or squash integrations [26]), and we ignore binary files, because we cannot track changes from them. For the example of Figure 1, we obtain 3 merge conflicts (*#conflicts*), 19 lines of code (*#lines*), 2 files (#files) and 4 chunks (*#chunks*) changed, 4 commits (*#commits*), 4 developers (*#devs*), and the merge scenario lasts 2 days (*#days*).

Communication activity. Considering experience from related work presented in Section 2.2 and the benefits, comprehensiveness, and popularity of GitHub when compared to other communication tools and channels, we chose to rely in our study on the GitHub platform. Another benefit is that by using GitHub, projects should follow the pull-based development model, hence, we can use the time a merge scenario lasts to define the analysis time span (i.e., not a predefined one). Aware of the drawbacks of using only one communication channel and considering that contributors may talk about topics not related to the code changes (e.g., usability or configuration problems), we select only projects that extensively use GitHub communication mechanism (see filters (ii) and (iii) of Section 4.2), pursue three approaches to capture communication amongst contributors, and differentiate the communication of among all contributors (*contributors' communication*) and among developers only (*developers' communication*), as we presented in Section 3.

Framework and Data Availability. Our analysis framework (Java) and analysis scripts (R) are open-source. All data necessary for replicating this study are stored in a MySQL database and replicated on spreadsheets (.csv files). All tools, links to the subject projects, and data are available at the supplementary Web site [48].

4.4 Operationalization

We operationalize our research questions and hypotheses through several variables. In what follows, we explain how we will answer each research question.

Answering RQ$_1$. To answer RQ$_1$, we check whether there is a correlation between the number of conflicts and the amount of GitHub communication. To quantify this correlation, we compute the bivariate correlation of the number of conflicts (*#conflicts*) and communication measures (*#cont_eds* and *#dev_eds*) for each communication approach. As we have two measures for communication and three approaches, we compute the correlation six times. We use Spearman's rank correlation because it is reliable when the observed covariates are count data and highly skewed. Spearman's rank correlation is +1 in the case of a perfect monotonic correlation, -1 in the case of a perfect reverse monotonic correlation. Values around 0 imply no monotonic relation between the variables [22].

Answering RQ$_2$. To answer RQ$_2$, we perform a multivariate analysis involving principal component analysis [25] and partial correlation [29] (both based on Spearman's rank correlation). Using a principal component analysis, we reduce the number of dimensions we have (i.e., one dimension for each variable of Table 1) to the first two principal components that retain a maximum share of common variance, which simplifies the discussion of the correlation structure. We choose to use partial correlation coefficients for three main reasons: (i) it is simple and straightforward, (ii) it does not require assumptions on the distribution, as parametric models, such as regression models and structural equation models would, and (iii) it does not introduce any form of assumed causality between X and Y, like a regression model would. The partial correlation of X (i.e., the number of conflicts) and Y (i.e., the communication measure), taking into account Z (i.e., a confounding factor) is defined by

$$\rho_{XY|Z} = \frac{\rho_{XY} - \rho_{XZ} \cdot \rho_{ZY}}{\sqrt{1-\rho_{XZ}^2} \cdot \sqrt{1-\rho_{ZY}^2}}. \quad (1)$$

When there is more than one variable Z like in our case (i.e., the context variables of Table 1), the partial correlation is computed based on the residual variance of X and Y after partialling out the correlations with all the confounding factors. That is, $\rho_{XY|Z_1,\ldots Z_k}$ is equal to the correlation of the residuals of a regression of X and Y on $Z_1, \ldots, Z_k$ [29].

Answering RQ$_3$. To answer RQ$_3$, we use partial correlations, as we did in RQ$_2$, and measure the strength of the respective moderation effects by splitting the sample according to the number of code of lines changed (H$_1$), the number of developers involved (H$_2$), and the time range (H$_3$) of a merge scenario. As our data are right skewed, most merge scenarios are small, short, and have few developers involved. Hence, to get merge scenarios that are in fact large, long, and with many developers involved, we split the sample according to the 90% rule, as a previous study suggests [47]. The correlation significance test is based on the maximum likelihood algorithm-based estimation, which converges only when there is enough variation in every covariate. As result of this analysis, no matter where we split the sample (e.g., 50%, 70%, or 90%), whenever the algorithm converged, it led to the same general conclusions with similar levels of significance. Specifically, the 90% rule assures a relatively equal coverage of all projects (i.e., there is a relative homogeneity across projects) and chooses only "large" merge scenarios (with respect to the measure for each hypothesis) in all projects. The results of the splitting rule analysis, further details, and data to test it can be found on our supplementary Web site [48].

P-value correction for multiple hypothesis testing. Multiple hypothesis testing is a problem that can lead to erroneous conclusions. When answering our research questions, we conduct various significance tests, asking whether the observed effects are statistically significant. In other words, when offering multiple potential covariates to the number of merge conflicts, we augment the chance of a type-one-error, that is, we augment the chance of finding at least one significant relationship. To minimize this problem, we used stricter rules, which makes the single tests less likely to be significant and thereby reduce the probability of finding false positives. For answering RQ$_1$ and RQ$_2$, we test three potential covariates per hypothesis (that is one for each communication approach). For RQ$_3$, we test three potential partners for four different hypotheses: correlation in upper and lower quantile, and with *#cont_eds* or *#dev_eds*. Therefore, for all three research questions, the chance of at least one type-one-error when using $\alpha = 0.05$ is $1 - 0.95^3 \approx 14.3\%$. To be significant at a 5% level, we require the test p-values to be smaller than $1 - \sqrt[3]{0.95} \approx 1.7\%$ [49].

5 Results

In this section, we present the results of our empirical study structured according to our research questions.

RQ$_1$: Is there a correlation between GitHub communication activity and the occurrence of merge conflicts?

Table 4 presents the results of Spearman's rank correlation analysis for merge conflicts (*#conflicts*) and communication measures (*#cont_eds* and *#dev_eds*)

Table 4 Spearman's correlation among the subject measures

	Awareness-based		**Pull-request-based**		**Changed-artifact-based**	
	#cont_eds	*#dev_eds*	*#cont_eds*	*#dev_eds*	*#cont_eds*	*#dev_eds*
RQ_1	*0.192***	*0.221***	*−0.004*	*0.003*	*0.237***	*0.222***
RQ_2	*0.012*	*−0.006*	*−0.011*	*0.001*	*−0.010*	*−0.003*

* $p < 0.017 \cong \alpha = 0.05$ ** $p < 0.003 \cong \alpha = 0.01$ *(Correction for multiple hypothesis testing).*

for each communication approach proposed. As we can see, the estimated correlation of the number of conflicts with the *contributors' communication* is rather weak *0.192* (awareness-based), *−0.004* (pull-request-based), and *0.237* (changed-artifact-based). Regarding the correlation between the number of conflicts and the *developers' communication*, the coefficients are *0.221* (awareness-based), *0.003* (pull-request-based), and *0.222* (changed-artifact-based). Despite being weak (smaller than 0.3), the correlation coefficients for the awareness-based and changed-artifact-based approach measures are significant at a 99% confidence level, whereas in the pull-request-based approach the correlation coefficients are not significant.

Comparing the correlations for the different communication measures (*#cont_eds* and *#dev_eds*), the changed-artifact-based approach coefficients are greater than the awareness-based approach coefficients. Another point to note is that, in the case of awareness-based approach measures, the coefficient for the developers' communication is greater than contributors' communication while the opposite is true for the changed-artifact-based approach measures.

RQ_1 Summary: Overall, the bivariate correlation analysis shows a significant weak positive correlation for awareness-based and changed-artifact-based communication approaches with the number of merge conflicts. In practical terms, more GitHub communication can be observed in merge scenarios with more merge conflicts.

RQ_2: How does the correlation between GitHub communication activity and merge conflicts change when taking confounding factors into account?

Figure 4 shows the two dimensional output from the principal component analysis for each communication approach, which covers 71.9% (57.2% + 14.7%), 57.2% (44% + 13.2%), and 73.5% (58.4% + 15.1%) of the total variance for the awareness-based, pull-request-based, and changed-artifact-based communication approaches, respectively. The arrows represent the weights of each variable in the respective principal component and its color represents the square cosine (cos2). The square cosine represents the share of original variation in the variable that is retained in the dimensionality reduction. The longer the arrow, the larger is the share of a variable's variance. Arrows pointing to the same direction have a large share of common variance and can be assumed to belong to the same group.

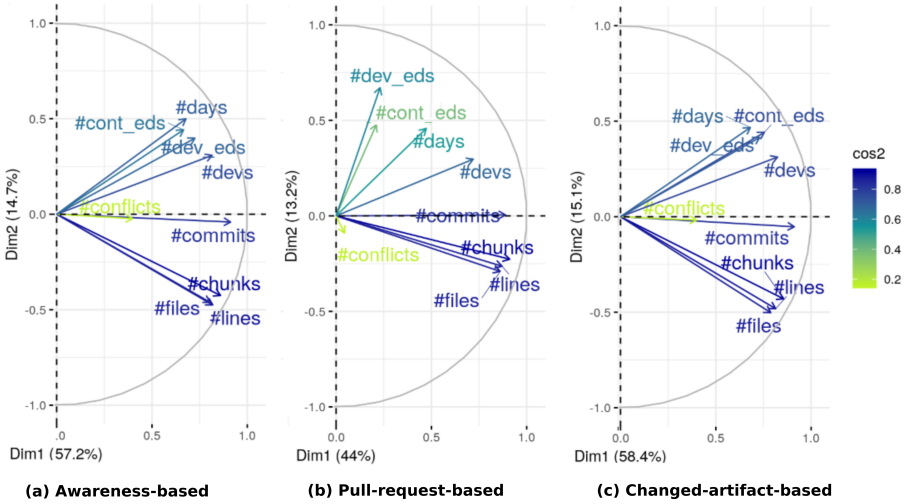

Fig. 4 Principal Component Analysis of our covariables.

Figure 4 suggests to classify the confounding variables into three groups (size, social dimension, and commit activity). The arrows representing the *#chunks*, *#files*, and *#lines* measures point in to the same direction; they represent the *size* of a merge scenario. Pointing to another direction, *#devs* and *#days* correlate strongly, and it is therefore legitimate to say that usually, merge scenarios with more developers take longer until they come to their end. We call this group the *social dimension*. The variable *#commits* is "undecided" between the two groups. This is not surprising because a large number of commits can either result from the participation of many developers or from larger merge scenarios as we will discuss in Section 6. Therefore, we keep this variable separate, in a group named *commit activity*.

Table 4 presents the results of our multivariate analysis using partial correlation for the three proposed approaches (below the answer of RQ_1). When considering the *contributors' communication*, the correlation coefficients are 0.012 (awareness-based), -0.011 (pull-request-based), -0.010 (changed-artifact-based). When considering the *developers' communication*, in the same order, the correlation coefficients are -0.006, 0.001, and -0.003. None of the six values are not significantly different from zero.

RQ_2 Summary: Accounting for confounding factors via partial correlations, the positive correlations found in RQ_1 disappear. In other words, the multivariate analysis reveals that there is no relation between the communication measures and number of merge conflicts when taking confounding factors into account. In practical terms, GitHub communication activity does not correlate with the occurrence or avoidance of merge conflicts.

Table 5 Median splits and correlations between number of conflicts and communication measures

Hyp.	Mod.	Comm.	Awareness-based		Changed-artifact-based	
			$\hat{\rho}$ lower	$\hat{\rho}$ upper	$\hat{\rho}$ lower	$\hat{\rho}$ upper
H_1	*#lines*	*#cont_eds*	*0.008*	*0.113**	*0.016*	*0.139***
		#dev_eds	*0.003*	*−0.097**	*0.010*	*−0.097**
H_2	*#devs*	*#cont_eds*	*−0.019*	*0.130***	*−0.013*	*0.216***
		#dev_eds	*−0.038**	*−0.070*	*−0.035**	*−0.025*
H_3	*#days*	*#cont_eds*	*−0.008*	*0.017*	*0.007*	*0.015*
		#dev_eds	*−0.005*	*−0.054*	*0.003*	*−0.068*

Hyp., Mod., Comm., $\hat{\rho}$ lower and $\hat{\rho}$ upper stand for hypotheses, the name of the moderator variable, the communication measure, the estimated rank correlation for the lower and upper split sample, respectively. * $p < 0.017 \cong \alpha = 0.05$, ** $p < 0.003 \cong \alpha = 0.01$.

RQ_3: What is the influence of merge scenario characteristics on the strength of the relation between GitHub communication activity and the occurrence of merge conflicts?

As illustrated in Figure 4, the size measures (i.e., *#lines*, *#chunks*, and *#files*) are highly correlated, so we choose the *#lines* measure to represent merge scenario size when answering RQ_3. Table 5 shows the results of the estimates of the partial correlation for the awareness-based and changed-artifact-based networks. We do not present the results for the pull-request-based approach, because, similar to RQ_1 and RQ_2, they are not significant for any hypothesis. We explain the reasons of these non-significant values for the pull-request-based approach in Section 6.

As our hypotheses make assumptions on the upper split (i.e., about the larger, longer, and with many developers involved in the merge scenarios), we focus our answers only on significant values of the upper split. Overall, we found significant values only for H_1 and H_2. For both approaches and hypotheses, the correlation coefficients are significant and positive when considering the *contributors' communication*. Hence, we accept H_1 and H_2 when considering the *#cont_eds* measure. For both approaches, the correlation regarding *developers' communication* is significant for H_1. Therefore, we accept H_1 as there is a stronger negative correlation between *#dev_eds* and the number of conflicts in the larger merge scenarios. Even though it may seem on first sight, our results for RQ_3 do not contradict the results for RQ_2. There is no global relationship (RQ_2), however, there is a relation for "larger" merge scenarios (RQ_3) in terms of changed lines of code (H_1) and number of developers (H_2).

RQ_3 Summary: Regarding H_1, for the awareness-based and changed-artifact-based approaches, we found a weak but statistically significant positive correlation *#cont_eds* measure. For the awareness-based approach, we further found a weak but significant negative correlation when using *#dev_eds* measure. Therefore, for these cases, we accept H_1. Regarding H_2, for both approaches, we found a weak significant positive correlation for *#cont_eds* measure. Therefore, for this measure, we accept H_2. Regarding H_3, we did not find any significant correlation, so we reject H_3 for all cases. We can conclude that merge scenarios' size and the number of developers involved influence the strength of the relation between GitHub communication activity and the occurrence of merge conflicts when investigating the "larger" merge scenarios. In practical terms, (i) in the 10% larger merge scenarios more communication activity among all contributors using the awareness- and changed-artifact-based approaches are associated with more merge conflicts, (ii) in the 10% larger merge scenarios more communication activity among among developers only using the awareness- and changed-artifact-based approaches are associated with less merge conflicts, and (iii) in the 10% merge scenarios with more developers involved, more communication among all contributors using the awareness- and changed-artifact-based approaches correlate with more merge conflicts. In all these cases, we found a weak but significant correlation.

6 Discussion

Given the popular belief that communication and collaboration success are mutually dependent (see Section 2), our results can be seen as a *negative result*, finding no indication for a global monotonic relationship between the amount of GitHub communication and the occurrence of merge conflicts for the majority of merge scenarios (answer of RQ_2). Even when considering the "larger" merge scenarios, with the moderation effects analysis (answer of RQ_3), we found only a weak correlation. As negative results are often suspected to be due to the failure of the research design [13] [31] [36] [38], we start with a discussion of potential threats to validity of our study. Subsequently, we present the implications of our study for research and practice.

6.1 Threats to Validity

External validity. External validity is threatened mainly by three factors. First, our restriction to Git and GitHub as platform as well as to the pull-based model. Generalizability to other platforms, projects, and development model is limited. This limitation of the sample was necessary to reduce the influence of confounds, increasing internal validity, though [41]. While more research is needed to generalize to other version control systems, development models, and communication platforms, we are confident that we selected and

analyzed a practically relevant platform and a substantial number of software projects from various domains, programming languages, longevity, size, and coordination practices. In addition, our filters applied during subject project selection guarantee, for instance, that we sampled projects that actively use GitHub as a communication tool and that we do not let a single programming language dominate our dataset (see Section 4.2).

Second, developers may use informal work practices, awareness-tools, or prediction strategies (e.g., continuous integration and rebase) that we are not able to measure. To minimize this threat, we manually looked at 50 issues randomly selected of each subject project searching for terms that point to such practices and tools, but we did not find any indication. One may also claim that rebased scenarios bias our analysis, however, since the commit(s) that a developer wants to integrate into another branch will be added on the top of the branch, it will make the repository's history linear avoiding merge conflicts. Hence, in all cases that we are not able to retrieve the common ancestor of two (parent) commits were excluded from our analysis. Considering that our research is only about the pull-based model (i.e., three-way merge), together with the previous actions, there is no bias.

Third, the need of triangulation through interview data. Interviewing developers could make our analyses and findings more reliable, however, as our results are counter-intuitive, we believe that asking developers could lead us to wrong conclusions. To mitigate this threat and to triangulate our data, we provided triangulation through observational data for every topic that deals with counter-intuitive findings, as we will discuss in Sections 6.2–6.3.

Internal validity. There are two major threats to the internal validity of our study. First, we may miss important communication since developers may use different communication channels (e.g., e-mail lists, IRCs, Gitter, Slack). This fact may also influence how developers communicate on the GitHub platform. We mitigate this threat by using the awareness-based approach. This communication approach retrieves the communication of all active contributors in the merge scenario even if they mostly use other channels or do not talk about merge scenario code changes in GitHub. This is still a limitation if contributors completely ignore GitHub to communicate. However, as discussed in Section 2.2, GitHub is one of the widely-used channels [37] and together with the filters presented in Section 4.2, we believe that we selected only projects that extensively use GitHub to communicate.

Second, developers may talk about other topics not related to the merge scenario code changes [4]. To mitigate this threat, we have considered two communication approaches (pull-request-based and changed-artifact-based) able to capture a focused communication (i.e., related to the merge scenario code changes). The pull-request-based approach considers only the communication of the merge scenario pull request and related issues. The changed-artifact-based approach, on the other hand, considers only the communication of opened issues during the merge scenario that contain commits that touch files changed in the merge scenario (see Section 3 for more details).

6.2 Insights and Implications for Researchers

Longer merge scenarios with more developers involve more GitHub communication, but not necessarily more merge conflicts. Figure 4 illustrates that the more time and developers are involved in a merge scenario, the more communication activity we observe. However, this does not mean more merge conflicts. To better understand it in practice we did a manual and random analysis. This analysis consisted of looking at 25 issues and 25 pull requests per subject project to understand *how* contributors interact and *what* they are talking *about*. As result, we identified two main patterns: (i) using pull requests, developers present their questions about their on-going code changes, and many contributors (including non-developers) help them with their questions, and (ii) for issues (excluding pull requests), contributors normally describe bugs, new issues, or problems (e.g., problems when configuring the tool or showing that the issue is duplicated). Therefore, we may say that most GitHub communication in issues (excluding pull requests) happens even before developers create or change a branch to implement the issue's solution. Therefore, despite being different both issue and pull-request communication are important and related to the merge scenario code changes. Looking at issues and pull requests, we saw many links among engaged conversations and pieces of code, which is inline with previous studies [44] [42] [11].

The size of the merge scenario's code change is not related to the number of developers involved. It seems obvious that the more time or developers are involved in a merge scenario, the larger the code changes will be. However, our principal component analysis reveals that this relationship is not so trivial. In other words, more developers and time do not mean larger changes. To understand the context of the changes, we manually analyzed 100 merge scenarios, split into two patterns that contradicts our first thoughts: (i) short merge scenarios, with few developers involved, and with many changes and (ii) long merge scenarios, with many developers involved, and few changes. Here, we present one example for each pattern. To represent the former pattern we chose a merge scenario from project *lantern*[4] that lasted only 4 days, with only two developers involved. However, 156 thousand lines of code of 3 380 chunks distributed in 446 files were changed. This merge scenario is related to solving a critical bug that came from the upgrade of a third tool that does not work in the Safari browser. To represent the second pattern we chose a merge scenario from project *public-apis*[5] that lasted 96 days, with 65 developers involved, however, only 130 lines of code of 39 chunks of one file were changed. This merge scenario consisted of adding new code. For short, they only updated the README.md file. Regarding GitHub communication activity, in the first example (pattern i), very few communication either among contributors or among developers only was found. In the second example (pattern ii), a high level of contributors' communication activity was found, however, few

[4] github.com/getlantern/lantern; commit 86be2a8
[5] github.com/toddmotto/public-apis; commit 0870841

developers' communication was found. This communication behavior applies to all communication approaches. These two examples represent well the analyzed set since many merge scenarios with few developers and large changes (pattern i) were related to bug fixing while many merge scenarios with many developers and small changes (pattern ii) were related to the introduction of new features to the project. In a nutshell, the type and criticality of the change may influence the characteristics of the code changes (size, time, and developers involved), as well as how contributors communicate in general. We leave the discussion regarding merge conflicts to the next paragraph.

Are large merge scenarios conflict-prone? Against our expectations, we did not find a strong correlation between the size of merge scenario code changes (in term of number of files, chunks, and lines of code involved) and the occurrence of merge conflicts. The reason may be the location of the changes (e.g., which files, branches, and architectural layers the changes happened). For instance, a merge scenario of the project *RxJava*[6] changed 642 files, 10 011 chunks, involving 44 developers and took 260 days. However, it has no merge conflicts. It is important to highlight that both branches involved complex changes, such as removal of files and semantic changes. One thing that contributed to the absence of merge conflicts is that in this example no file was changed in both branches and most of the changes happened in one branch (44 developers changed 612 files in the target branch while 4 developers changed 30 files in the source branch). Developers that changed the source branch also changed the target branch. Therefore, the size of the changes of a merge scenario is not sufficient to predict merge conflicts. In this vein, Leßenich et al. [31] tried to predict merge conflicts, but even though they have used factors that practitioners indicated to be related to the emergence of merge conflicts (e.g., scattering degree among classes, commit density, number of files), none of these factors had a strong correlation with the occurrence of merge conflicts. This shows together with our results that predicting merge conflicts is not trival and further investigation is still necessary.

Contributors and developers communicate regularly to keep others informed. By differentiating the communication of contributors and developers, we also learn whether there is a difference in the effect of communication of either of the two groups on the occurrence of merge conflicts. Unexpectedly, neither of the two communication groups correlates with the number of merge conflicts when taking confounding factors into account (RQ_2). Figure 5 compares contributors (*#cont_eds*) and developers (*#dev_eds*) communication for each merge scenario and approach. The difference in the scale of the x-axis and y-axis, in which y-axis is much greater, means that most communication happened between contributors that do not change the source code. On average, *#cont_eds* is equal to 33 019 (awareness-based), 3.25 (pull-request-based) and 2 439 (changed-artifact-based), whereas *#dev_eds* is 11.22 (awareness-based), 0.15 (pull-request-based), and 9.37 (changed-artifact-based). In addition, the average of #devs is 5.85 for the dataset used to the awareness-based

[6] github.com/ReactiveX/RxJava; commit 25ebda

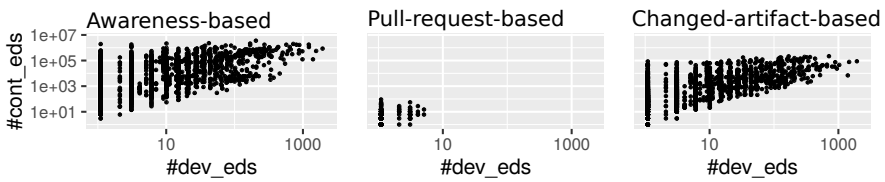

Fig. 5 Contributors' versus developers' communication.

and change-artifact-based approach analyses and 4.23 for the dataset used for the pull-request-based approach analysis. So, most of the GitHub communication takes place among contributors. Given the average of *#devs_eds* for awareness-based and changed-artifact-based, we can assume that developers also communicate, although less than non-developers. This amount of communication among contributors and also among developers only brings us to the conclusion that in general developers and non-developers communicate to keep others aware of their contribution in a merge scenario.

Developers' communication is more efficient than contributors' communication for avoiding merge conflicts. As stated by the last paragraph, contributors and developers normally communicate. Therefore, contributors normally keep others aware of their code changes. For the general case, there is no relation between GitHub communication and the occurrence of merge conflicts. However, when considering the 10% larger merge scenarios in terms of lines of code, the contributors' and the developers' communication have a different behavior. While the contributors' communication is related to an increase in the number of merge conflicts, the developers' communication is related to a decrease in the number of merge conflicts. When considering that the communication among developers will support the coordination of social and technical assets and that such coordination may minimize the number of merge conflicts, we may interpret that, for larger merge scenarios, developers' communication is normally helpful for avoiding merge conflicts being also more efficient than the contributors' communication.

The communication approach does not change the understanding on the occurrence of merge conflicts. We consider the three communication approaches used in this study (awareness-, pull-request-, and changed-artifact-based) valid to measure the communication granularity they were proposed to measure. Yet, independent of the communication approach, we did not find a different relation on the occurrence of merge conflicts in the general case (RQ_2). Therefore, we cannot pinpoint which communication approach presented in this study is best for avoiding merge conflicts.

In any event, there are three points that we can discuss: (i) as the pull-request-based approach is dependent of GitHub pull-requests and most of the merge scenarios are not integrated by means of pull-requests, this communication approach has a limited applicability, (ii) developers normally talk about the changed artifacts since the average of the amount of communication of the

awareness-based approach (general GitHub communication) and the changed-artifact-based approach (GitHub communication related to artifact changed) are similar (11.22 edges against 9.37), and (iii) regarding contributors' communication the awareness-based approach and the changed-artifact-based aproach have a very different average (33 019 edges against 2 439).

Limitations of bivariate analysis. The bivariate analysis is simple and intuitive, as it directly quantifies the correlation of interest. On the other hand, the multivariate analysis takes other factors that may influence the correlation of interest into account. Looking at the answers to RQ_1 and RQ2, our results for these two analyses are contradictory. Even though we are aware that the bivariate analysis is not enough to investigate the complex interplay of project success, communication, and contextual factors, our intention by presenting both results is to highlight the limitations of bivariate analysis. In other words, a simple and intuitive analysis may provide wrong results, hence it is necessary to reflect on the *big-picture* before determining which variables should be analyzed and modeled.

A causal perspective. Our results for RQ_2 reveal a no significant relation between GitHub communication activity and merge conflicts. Therefore, more communication activity does not imply fewer or more merge conflicts, and more or fewer merge conflicts does not result from a lack of communication. Such causal analysis of the relation between GitHub communication and merge conflicts can only be achieved with a more detailed understanding of the timely order of events (e.g., communication before and after comits), which is out of scope of this study. For that reason, our results are limited to determining whether there is correlation among these covariables or not.

6.3 Insights and Implications for Practitioners

Number of commits should be used with care. As a metric, the number of commits is often used by practitioners to obtain a feeling of the contributions of others as well as to predict conflicts [16] [31]. However, even though this metric correlates most with the number of conflicts in our study (compare arrow of *#commits* and *#conflicts* in Figure 4 (a) and (c)), the correlation is weak (≈ 0.2). Hence, this metric should be used with care, as it depends on how developers commit their code (e.g., for each function or for each feature implemented). For instance, in the project *lantern*[7] there are two commits from the same developer, but following completely different patterns. The former has changed 527 files, 3 379 chunks, 175 458 lines of code, and it is not involved in a merge conflict. The latter has changed only 1 file, chunk, and line of code, but, it was involved in a merge conflict. Two points to highlight are: (i) these commits were part of merge scenarios in which the two merged branches were changed and (ii) the target developer made substantial semantic changes (i.e., not only formatting or ordering changes).

[7] github.com/getlantern/lantern; commits:9d0bbbb and 6b6b534

Pull requests lead to fewer merge conflicts. As can be seen in the fourth row of Table 3, 6 487 merge scenarios have both branches touched and have multiple developers contributing to it. In addition, the last row of Table 3 shows that of 3 436 merge scenarios using pull requests only 7 have 9 merge conflicts (2 merge scenarios have 2 conflicts) of which 0.2% are conflicting merge scenarios. Subtracting the merge scenarios that use pull requests from the ones with both branches touched and with multiple developers, we get 3 051 merge scenarios that were merged without pull requests (e.g., by the git merge command) of which 851 conflicting merge scenarios can be found. Hence, these merge scenarios present a share of 27.89% of conflicting merge scenarios. In other words, the share of conflicting merge scenarios without using pull requests is 139 times greater than when using pull requests.

The low number of merge conflicts when integrating merge scenarios by using pull requests is likely the reason why we did not find significant correlation values when answering all research questions for the pull-request-based approach. The reason for the low number of merge conflicts comes from the fact the pull requests simulate the merge. With the simulation, developers have the chance of changing the source code before merging the branches (i.e., avoiding merge conflicts). This raises two questions: *Why are pull requests not used in most of the merge scenarios?* And, *why some merge scenarios still have merge conflicts when using pull requests?* For the first question, it is necessary to conduct interviews with developers, but we understand that, for some merge scenarios, it is simpler to merge branches locally. Regarding the second question, we manually checked each of these conflicting merge scenarios aiming at discovering why developers did not remove the conflicts before merging branches. We observed two things. First, the conflicts are in files that do not "break" the code and are normally in files not tested, such as *README.md* or *.gitignore*. Second, a commit resolving the conflict is added little time after the merge by another developer. Our assumption is that, whoever merged the code, merged because it did not fail in the test suit and another developer, maybe more experienced, chose the "best" option later on. Given the low number of merge conflicts when using pull requests, developers should adhere to such practice as well as to continuous integrations, and awareness tools in their workday tasks.

7 Final Remarks and Perspectives

Software development is a collaborative activity where success depends on the ability to coordinate social and technical assets. Software merging is a challenging and tedious task in the practice of collaborative and concurrent software development, mainly when merge conflicts arise. Since merge conflicts are unexpected events, they have a negative effect on project's objectives compromising the project budget and schedule, especially when they arise often. On the other side, empirical research has found evidence for a beneficial effect of communication on the project coordination and success. So, it is believed

that high communication activity helps to avoid merge conflicts. However, in spite of such belief the role of communication activity for merge conflicts to occur or to be avoided had not been thoroughly investigated so far.

Aiming at investigating the relation between GitHub communication and the occurrence of merge conflicts (i.e., the two covariates), we rebuilt the merge scenarios' contributions and communications of 30 subject projects. To obtain a deep understanding of communication work practices in these projects, we used three communication approaches (awareness-, pull-request-, and changed-artifact-based) and differentiate the communication among all contributors and the communication among developers only. Our investigation comprises three analyses. First, we started investigating the direct correlation between these two covariates as is common in empirical software engineering studies. As result, we found a weak but significant positive correlation when using the awareness- and changed-artifact-based approaches. While being simple and intuitive, this bivariate analysis does not consider the complex interplay of project success, communication, merge conflicts, and contextual factors, which may had led us to a misleading conclusion. Aiming at properly exploring this interplay, we performed a multivariate analysis. In this analysis, we found no significant relation between communication measures and number of merge conflicts. We conclude that the results of the bivariate analysis are spurious. Furtheremore, considering that it is likely that the strength of the relation between the two covariates depends on merge scenario characteristics, we performed a moderation effect analysis. As result of the moderation effect analysis, we found that the number of lines of code and the number of developers involved in the merge scenario influences the strength of such relation. Thus, there is no overall monotonic relation, but there is a relation for "larger" merge scenarios.

The seemingly contradictory results from our three analyses should alert the reader that overly simplistic bivariate analysis can lead to wrong conclusions. To avoid such mistakes, it is necessary to reflect on the *big-picture* to determine which variables should be analyzed and modeled when investigating complex environments such as the collaborative software development.

Our result contradict the common belief that communication per se is beneficial for avoiding merge conflicts. If this was the case, we would expect a strong negative correlation between the communication activity and the number of merge conflicts. Puzzled by our results and to ensure that they are robust, reliable, and straightforward, we provided triangulation through a manual investigation of the merge scenarios' contribution and communication activity separately.

Regarding the merge scenarios' contribution, our discussions suggest that (i) the number of developers do not influence the size of the code changes in a merge scenario, (ii) more time and developers are not accompanied by more merge conflicts, (iii) the type of the change influences the size of merge scenarios. Bug fixing normally represents short time-life scenarios, with few developers, and large code changes. The introduction of new code (features), on the other hand, represents long time-life scenarios, with many developers,

and few code changes, (iv) larger changes are not more conflicting-prone than smaller changes (which is in agreement with Leßenich et al. [31]). It suggests that the location of the changes are more related to the emergence of merge conflicts than the size of the change, (v) the number of commits should be used with care since it depends on how developers commit to the project. As exemplified, the same developer may follow different committing patterns, and (vi) the use of pull requests reduces the number of merge conflicts compared to merge scenarios integrated without pull requests in 139 times.

Regarding communication activity, our discussions suggest that (i) indeed contributors and developers normally communicate. Hence, our unexpected results did not come from a lack of communication or because we were not able to retrieve such communication, (ii) GitHub issue and pull-request communication are both important to understand merge scenario code changes. However, the former is more related to problem clarification and the latter is more related to the on going code changes, (iii) the communication approach choice (awareness-, pull-request-, and changed-artifact-based) does not change the results of the multivariate analysis which means that one is not better than the others for avoiding merge conflicts, and (iv) for the moderation effect analysis, the developers' communication was shown to be more efficient than the contributors' communication for avoiding merge conflicts.

As future work, we suggest studies as well as further investigations on the relation between communication activity and merge conflicts. Regarding communication activity, we suggest (i) the exploration of other communication channels, (ii) the impact of the use of GitHub communication when developers use other channels, and (iii) how contributors use GitHub facilities (e.g., labels, links among issues and commits). These three suggestions may overcome limitations of our study discussed in Section 6.1 and give more detailed insights into the topic. Regarding merge conflicts, we suggest explorations of (i) the lack of code ownership [14] [18], (ii) the centrality of files, and (iii) the contribution activity for each merged branch on the occurrence of merge conflicts. These three suggestions may explain why previous work [1] [15] [31] has found counter-intuitive results when predicting merge conflicts. Furthermore, we suggest three studies. First, a study using network metrics such as density, centrality, and clustering to provide an overview of how contributors organize and coordinate themselves over the project evolution [24]. It can be useful to find patterns that contributes to the emergence of merge conflicts. Second, we suggest a study that takes our data in consideration and asks developers about specific (outlier) merge scenarios. This would provide explanations for strange metric combinations, such as very large merge scenarios with hundreds of developers involved, which took only one day to be integrated. Finally, we suggest a study exploring the developer's role in the merge scenario. Previous studies have found that communication activity is required for the success of software projects [7] [19] [40] [43]. However, when considering the avoidance of merge conflicts, some developers have more responsibility on specific merge scenarios and their communication/coordination would be enough to avoid

merge conflicts. In other words, is the coordination of all developers necessary to avoid merge conflicts occurrence?

Acknowledgements This work was partially supported by CNPq (grant 290136/2015-6) and Bavarian State Ministry of Education, Science and the Arts in the framework of the Centre Digitisation.Bavaria (ZD.B).

References

1. P. Accioly, P. Borba, and G. Cavalcanti. “Understanding Semi-structured Merge Conflict Characteristics in Open-source Java Projects”. In Empirical Software Engineering, vol. 23(4), Springer, pp. 1–35, 2017.
2. S. Apel, O. Leßenich, and C. Lengauer. “Structured Merge with Autotuning: Balancing Precision and Performance”. In Proceedings of the International Conference on Automated Software Engineering (ASE). ACM, pp. 120–129, 2012.
3. S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. “Semistructured Merge: Rethinking Merge in Revision Control Systems”. In Proceedings of the Symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE). ACM, pp. 190–200, 2011.
4. J. Aranda and G. Venolia. “The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories”. In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 298–308, 2009.
5. A. Begel, Y. P. Khoo, and T. Zimmermann. “Codebook: Discovering and Exploiting Relationships in Software Repositories”. In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 125–134, 2010.
6. J. Biehl, M. Czerwinski, G. Smith and G. Robertson “FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams”. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI), ACM, pp. 1313–1322, 2007.
7. C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. ”Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista”. In Proceedings of the International Conference on Software Engineering (ICSE). IEEE, pp. 518–528, 2009.
8. C. Bird, D. Pattison, R. DâĂŹSouza, V. Filkov, and P. Devanbu. ”Latent Social Structure in Open Source Projects”. In Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). ACM, pp. 24–35, 2008.
9. H. Borges and M.T. Valente. “What’s in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform”. In Journal of Systems and Software (JSS), vol. 146 (1), pp. 112–129, 2018.
10. Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. ”Proactive Detection of Collaboration Conflicts”. In Proceedings of the European Software Engineering Conference and the Symposium on Foundations of Software Engineering (ESEC/FSE). ACM, pp. 168–178, 2011.
11. L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. “Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository”. In Proceedings of the Conference on Computer Supported Cooperative Work (CSCW). ACM, pp. 1277–1286, 2012.
12. P. Dewan and R. Hegde. “Semi-synchronous Conflict Detection and Resolution in Asynchronous Software Development”. In Proceedings of the Conference on European Computer Supported Cooperative Work (ECSCW). ACM, pp. 159–178, 2007.
13. K. Dickersin, Y. Min, and C. Meinert. “Factors Influencing Publication of Research Results: Follow-up of Applications Submitted to Two Institutional Review Boards”. Journal of the American Medical Association, vol. 267 (3), pp. 374–378, 1992.
14. M. Foucault, J.-R. Falleri and X. Blanc. “Code Ownership in Open-source Software”. In Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE), ACM, pp. 1–9, 2014.

15. G. Ghiotto, L. Murta, M. Barros, and A. van der Hoek. “On the Nature of Merge Conflicts a Study of 2,731 Open Source Java Projects Hosted by Github”. In Transactions on Software Engineering (TSE), vol. 99 (1), IEEE, pp. 1–25, 2018.
16. G. Gousios, M. Pinzger, and A. Deursen. “An Exploratory Study of the Pull-based Software Development Model”. In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 345–355, 2014.
17. G. Gousios, M.A. Storey, and A. Bacchelli. “Work Practices and Challenges in Pull-based Development: The ContributorâĂŹs Perspective”. In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 285–296, 2016.
18. M. Greiler, K. Herzig, and J. Czerwonka. “Code Ownership and Software Quality: A Replication Study”. In Proceedings of the Working Conference on Mining Software Repositories (MSR), IEEE, pp. 2–12, 2015.
19. R. E. Grinter, J. D. Herbsleb, and D. E. Perry. “The Geography of Coordination: Dealing with Distance in R & D Work”. In Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP). ACM, pp. 306–315, 1999.
20. M. L. GuimarÃčes and A. R. Silva. “Improving Early Detection of Software Merge Conflicts”. In Proceedings of the International Conference on Software Engineering (ICSE). IEEE, pp. 342–352, 2012.
21. A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. Deursen. “Communication in Open Source Software Development Mailing Lists”. In Proceedings of the Working Conference on Mining Software Repositories (MSR). IEEE, pp. 277–286, 2013.
22. H. Z. Jerrold. “Significance Testing of the Spearman Rank Correlation Coefficient”. Journal of the American Statistical Association, vol. 67 (339), Taylor & Francis, Ltd, pp. 578–580, 1972.
23. M. Joblin, S. Apel, and W. Mauerer. “Evolutionary Trends of Developer Coordination: A Network Approach”. Empirical Software Engineering, vol. 22 (4), pp. 2050–2094, 2017.
24. M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle. “From Developer Networks to Verified Communities: A Fine-grained Approach”. In Proceedings of the International Conference on Software Engineering (ICSE). IEEE, pp. 563–573, 2015.
25. I.T. Jolliffe. “Principal Component Analysis”. Springer Series in Statistics, Springer, 2nd edition, p. 487, 2002.
26. S. Just, K. Herzig, J. Czerwonka, and B. Murphy. “Switching to Git: The Good, the Bad, and the Ugly”. In Proceeding of the International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 400–411, 2016.
27. E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. German, D. Damian. “The Promises and Perfils of Mining GitHub”. In Proceedings of the Working Conference on Mining Software Repositories (MSR), ACM, 92–101, 2014.
28. B.K. Kasi and Anita Sarma. “Cassandra: Proative Conflict Minimization through Optimized Task Scheduling”. In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 732–741, 2013.
29. S. Kim “ppcor: An R Package for a Fast Calculation to Semi-partial Correlation Coefficients”. Communication for Statistical Applications and Methods, vol. 22 (6), pp. 665–674, 2015.
30. T. D. LaToza, G. Venolia, and R. DeLine. “Maintaining Mental Models: A Study of Developer Work Habits”. In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 492–501, 2006.
31. O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen. “Indicators for Merge Conflicts in the Wild: Survey and Empirical Study”. Automated Software Engineering, vol. 25 (2), Springer, pp. 1–35, 2017.
32. J. Liu, J. Li, and L. He. “A Comparative Study of the Effects of Pull Request on GitHub Projects”. In Annual Computer Software and Applications Conference (COMPSAC). IEEE, pp. 313–322, 2016.
33. S. McKee, N. Nelson, A. Sarma, and D. Dig. “Software Practitioner Perspectives on Merge Conflicts and Resolutions”. In Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 467–478, 2017.
34. T. Mens. “A State-of-the-Art Survey on Software Merging”. In IEEE Transactions on Software Engineering. IEEE, 28(5):449–462, 2002.

35. N. Nelson, C. Brindescu, S. McKee, A. Sarma, D. Dig. “The Life-Cycle of Merge Conflicts: Processes, Barriers, and Strategies”. Empirical Software Engineering, Online First, Springer, pp. 1-44, 2019.
36. C. Olson, D. Rennie, D. Cook, K. Dickersin, A. Flanagin, J. Hogan, Q. Zhu, J. Reiling, B. Pace, “Publication Bias in Editorial Decision Making”. Journal of the American Medical Association. vol 287 (21), pp. 2825–2828, 2002.
37. S. Panichella, G. Bavota, M. D. Penta, G. Canfora, and G. Antoniol. “How DevelopersâĂŹ Collaborations Identified from Different Sources Tell Us about Code Changes”. In Proceeding of the International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 251–260, 2014.
38. E. Reiter, R. Robertson, and L. Osman. “Lessons from a Failure: Generating Tailored Smoking Cessation Letters”. Artificial Intelligence. Elsevier, vol. 144(1-2), pp. 41–58, 2003.
39. A. Sarma, D.F. Redmiles, A. van der Hoek. “Palantír: Early Detection of Development Conflicts Arising from Parallel Code Changes”. IEEE Transactions on Software Engineering. IEEE, vol. 38(4), pp. 889–908, 2012.
40. T. Sedano, P. Ralph, and C. PĂlraire. “Software Development Waste”. In Proceedings of the International Conference on Software Engineering (ICSE). IEEE, pp. 130–140, 2017.
41. J. Siegmund and J. Schumann. “Confounding Parameters on Program Comprehension: A Literature Survey”. Empirical Software Engineering vol. 20 (4), pp. 1159–1192, 2015.
42. L. Singer, F. Figueira Filho, B. Cleary, C. Treude, M.A. Storey, and K. Schneider. “Mutual Assessment in the Social Programmer Ecosystem: An Empirical Investigation of Developer Profile Aggregators”. In Proceedings of the Conference on Computer Supported Cooperative Work (CSCW). ACM, pp. 103–116, 2013.
43. C. R. B. Souza, D. Redmiles, L. Cheng, D. Millen, and J. Patterson. “How a Good Software Practice Thwarts Collaboration: The Multiple Roles of APIs in Software Development”. SIGSOFT Software Engineering Notes vol. 29 (6), pp. 221–230, 2004.
44. M.A. Storey, A Zagalsky, F. Figueira Filho, L. Singer, D. M. German. “How Social and Communication Channels Shape and Challenge a Participatory Culture in Software Development”. In IEEE Transactions on Software Engineering. vol. 43 (2), pp. 185–204, 2016.
45. T. Teo. “Handbook of Quantitative Methods for Educational Research”. SensePublishers, p. 404, 2014.
46. J. Tsay, L. Dabbish, and J. Herbsleb. “Influence of Social and Technical Factors for Evaluating Contribution in GitHub”. In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 356–366, 2014.
47. G. Vale, E. Fernandes, E. Figueiredo. “On the Proposal and Evaluation of a Benchmark-based Threshold Derivation Method”. Software Quality Journal, vol. 27(1), pp.1–32, 2018.
48. G. Vale, A. Schimid, A. Santos, E. Almeida, and S. Apel, “On the Relation Between Coordination Activities and Merge Conflicts – Supplementary Web site” Available: https://sites.google.com/view/vale-emse2019, [Accessed: 07/30/2019].
49. S. P. Wright. “Adjusted P-Values for Simultaneous Inference”. Biometrics, Wiley, vol. 48(4), pp. 1005–1013, 1992.
50. T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. “Mining Version Histories to Guide Software Changes”. In Proceedings of the International Conference on Software Engineering (ICSE). IEEE, pp. 563–572, 2004.