How do Multiple Pull Requests Change the Same Code: A Study of Competing Pull Requests in GitHub

Xin Zhang†, Yang Chen†, Yongfeng Gu†, Weiqin Zou‡, Xiaoyuan Xie†, Xiangyang Jia†, Jifeng Xuan†*
† School of Computer Science, Wuhan University, Wuhan 430072, China
{*xinzhang_, yangchen0800, yongfenggu, xxie, jxy, jxuan*}*@whu.edu.cn*
‡ State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China
wqzou@smail.nju.edu.cn

***Abstract*—GitHub is a widely used collaborative platform for global software development. A pull request plays an important role in bridging code changes with version controlling. Developers can freely and parallelly submit pull requests to base branches and wait for the merge of their contributions. However, several developers may submit pull requests to edit the same lines of code; such pull requests result in a latent collaborative conflict. We refer such pull requests that tend to change the same lines and remain open during an overlapping time period to as *competing pull requests*.**

In this paper, we conduct a study on 9,476 competing pull requests from 60 Java repositories in GitHub. The data are collected by mining pull requests that are submitted in 2017 from top Java projects with the most forks. We explore how multiple pull requests change the same code via answering four research questions, including the distribution of competing pull requests, the involved developers, the changed lines of code, and the impact on pull request integration. Our study shows that there indeed exist competing pull requests in GitHub: in 45 out of 60 repositories, over 31% of pull requests belong to competing pull requests; 20 repositories have more than 100 groups of competing pull requests, each of which is submitted by over five developers; 42 repositories have over 10% of competing pull requests with over 10 same lines of code. Meanwhile, we observe that attributes of competing pull requests do not have strong impacts on pull request integration, comparing with other types of pull requests. Our study provides a preliminary analysis for further research that aims to detect and eliminate conflicts among competing pull requests.

***Keywords*-Pull requests, collaborative development, merge conflicts, GitHub**

I. Introduction

GitHub is a widely-used platform for collaborative development. Individual developers and companies create and contribute to software repositories without the limitation of locations and organizations. According to the official statistics,up to March 2018, there are over 80 million repositories developed by 24 million developers in GitHub. GitHub leverages pull requests to support interaction between developers and code repositories. A *pull request* consists of code commits by a developer. Once the developer aims to contribute to a repository, he/she can *fork* (i.e., clone) the repository as an own repository and submit a pull request of his/her changes to a base branch (i.e., a particular code version); then a manager decides whether the pull request can be merged into the branch or not.

Merging pull requests into repositories can cause a merge conflict when developers make different changes to the same lines in a branch. For instance, if a pull request is submitted to update a piece of code that is already changed by someone else, then a merge conflict appears [1]. It is challenging to resolve merge conflicts during the collaborative development among developers. Developers have spent much effort looking for a straightforward solution to merge conflicts [22].

To address the problem of merge conflicts, Brun et al. [5] have proposed Crystal, a leading proactive detection tool that helps developers identify and further prevent merge conflicts. Apel et al. [2] developed JDime, a structured merge technique based on the analysis of source code syntax. These methods and techniques enhance the effectiveness of resolving merge conflicts for version control systems. A recent empirical study by Accioly et al. [1] shows that making use of semi-structured information inside source code can help eliminate merge conflicts of version control systems.

GitHub has a built-in feature that can detect the merge conflict between a pull request and its base branch. The branch manager, however, cannot always instantly respond to newly-submitted pull requests as well as merge conflicts. If two or more developers submit pull requests to update the same code at the same time, a queue of pull requests appear and await to be merged. The current implementation of GitHub cannot directly recognize the inconsistence among such pull requests.

In this paper, we refer those pull requests that remain open during an overlapping time period and aim to change the same code to as *competing pull requests*.[1] Resolving competing pull requests is beyond resolving merge conflicts: a merge conflict detects the potential inconsistence between a submitted pull request and its updated base branch; for the updated branch, the edit is already integrated into the code

* Corresponding author

[1]This is named after the *competing line change merge conflict* in the document of GitHub, http://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/.

repository. In contrast, competing pull requests compare pull requests that were simultaneously submitted by different developers and decide which pull request should be merged. It is important to characterize and understand these competing pull requests and further assist the branch manager to make a decision. To the best of our knowledge, no prior work has investigated competing pull requests in GitHub.

In this paper, we conducted a study of competing pull requests on 9,476 competing pull requests from 60 most popular Java repositories in GitHub. All the data are extracted by mining pull requests that are submitted in 2017 from the top 100 Java repositories with the most forks. We collected competing pull requests by checking their previous commits that are submitted to the same files and collected the subsequent merge status through the whole life cycle of these pull requests. We explore how multiple pull requests change the same code via answering four Research Questions (RQs), including the distribution of competing pull requests, the involved developers, the changed lines of code, and the impact on pull request integration.

Our study provides a preliminary study on competing pull requests. On one hand, we found that there indeed exist competing pull requests in GitHub: in 45 out of 60 repositories, over 31% of pull requests belong to competing pull requests; 20 repositories have more than 100 groups of competing pull requests that involve over five developers; 42 repositories have over 10% of competing pull requests that aim to change over 10 same lines of code. On the other hand, we observed that there only exists a weak correlation between specific human-defined attributes of competing pull requests and their merging results (i.e., merged or not in future). We speculate that such weak correlation is caused by the complex scenario of collaboration in competing pull requests. Our study results can be used as a basis to support further research work that detect and eliminate conflicts among competing pull requests.

This paper makes the following major contributions:

1. We mined continuous data of six months from 60 most popular Java repositories in GitHub and collected 9,476 competing pull requests, i.e., pull requests that aim to simultaneously change the same code.

2. We designed a study on competing pull requests via answering four research questions, including the existence of competing pull requests, the collaborative development, the overlap of updated code, and the impact on pull request integration.

3. We empirically analyzed the differences between competing pull requests and other pull requests. The result shows that competing pull requests are more difficult to be characterized than other pull requests.

The remaining of this paper is organized as follows. Section II presents the background and motivation. Section III presents the study setup, including the data preparation and four research questions. Section IV details the results of our study. Section V discusses the threats to the validity. Section VI lists the related work and Section VII concludes this paper.

II. Background and Motivation

We introduce the background of merging pull requests and the motivation of studying competing pull requests.

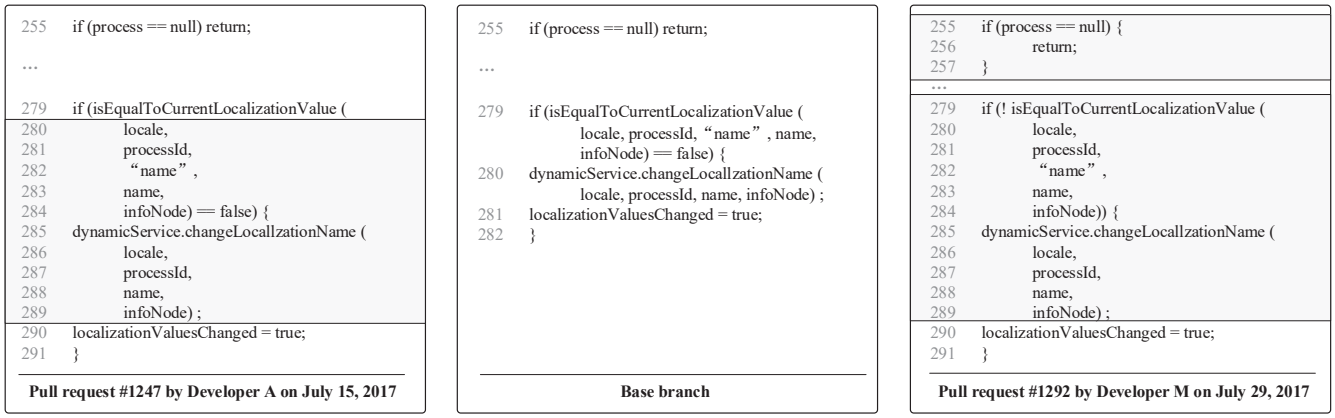
A. Background

GitHub is a collaborative service hosting platform for distributed version controlling. A manager of a project can freely deploy the project on GitHub to attract global developers for their code contributions. In the terminology of GitHub, a deployed project is called a *repository* and a parallel code version is called a *branch*. A default branch in a repository is usually called a *master* branch. To contribute to the repository, a developer can *fork* a repository on GitHub as an own repository and update source code of a chosen branch. To integrate the changed code to the repository, the developer requests to *merge* the change into the original base branch. In GitHub, the above merge is conducted via submitting a pull request.

The design of pull requests is one of core mechanisms of GitHub. A *pull request*, recording code changes in a forked repository, is submitted by a developer to the original repository [13]. A pull request consists of one or more code *commits*, each of which records the change on one or more files with a particular timestamp. Developers can freely and parallelly submit pull requests to a repository and a manager decides which pull request can be merged. In collaborative development, the original base branch may have been already updated before a developer submits a new update via a pull request; then GitHub cannot integrate the new update. This leads to a *merge conflict*. Two major reasons can cause merge conflicts: one is a pull request by someone else has been merged into the base branch; the other is the branch manager changed the branch via a commit (in this case, the branch manager does not need to submit a pull request) [12]. Then the merge conflict will be *resolved* by the manager manually. Due to the complexity of code, a developer who is responsible for addressing merge conflicts needs to spend much time understanding the changes and resolving the merge conflicts.

B. Motivation – Competing Pull Requests

Scenario of competing pull requests. Several developers may try to update the same line of code with different pull requests. Once multiple developers submit pull requests to the same code, a queue of pull requests is awaiting to be reviewed and merged. This may lead to one or more merge conflicts. Meanwhile, these submitted and unprocessed pull requests are competitive with each other: only one or zero among these pull requests can be directly integrated into the base branch. In this paper, we use *competing pull requests* to



(a) (b) (c)

Figure 1. Example of competing pull requests in the repository Activiti/Activiti, including a code excerpt in the base branch (Figure 1(b)) and two pull requests that aim to update the branch (Figures 1(a) and (c)). These two pull requests try to make changes via different edits. This leads to a scenario of deciding which pull request should be merged into the branch. Pull request 1247 was created on July 7, 2017 while Pull request 1292 was created on July 29. The branch manager has finally merged Pull request 1292 into the branch.

denote the unprocessed pull requests that aim to update the same code, then competing pull requests are grouped if they are changing the same lines of code during an overlapping time period.

All competing pull requests have to be manually checked to identify which pull request should be merged. The scenario of competing pull requests is more complex than the scenario of merge conflicts in Section II-A. The scenario of merge conflicts is to submit a pull request to change an updated branch; it explores the conflict between a new pull request and the already-updated code. The scenario of competing pull requests is to decide which pull request is the best change; it chooses a suitable change among multiple contributions by developers.

Example of competing pull requests. Figure 1 illustrates an example of two competing pull requests from the repository Activiti/Activiti. Figure 1(b) is a code excerpt of a Java class `BpmnDeployer` in a base branch; Figures. 1(a) and 1(c) are two pull requests by Developers A and M, who aim to change the base branch, respectively.[2] The pull request in Figure 1(a) updates Lines 279 and 280; the pull request in Figure 1(c) updates Lines 255, 279 and 280. Although the pull request in Figure 1(a) only focuses on the code style, both these pull requests try to change two or more lines of code. These pull requests are submitted nearly at the same time by two developers and remain open for an overlapping time period. Then the branch manager has to manually check these pull requests and choose to merge one pull request or discard both. From the perspective of the manager, these two pull requests are competing. GitHub cannot automatically detect competing pull requests unless at least one of them is merged into the repository. Accepting and merging one of these pull requests results in a merge conflict; meanwhile, different competing pull requests aim to update the same line via different ways. Therefore, even a human manager can hardly decide the most valuable pull request. A straightforward question is how many such competing pull requests exist. We conduct a study to explore competing pull requests in GitHub in this paper.

[2]Two pull requests, http://api.github.com/repos/Activiti/Activiti/pulls/1292 and http://api.github.com/repos/Activiti/Activiti/pulls/1247.

III. Study Setup

In this study, we aim to explore how multiple pull requests change the same code. This study is conducted based on the concept of competing pull requests.

A. Competing Pull Requests

We define competing pull requests based on the definition of code commits. Informally, a code commit can be viewed as a set of changes to a base version of a repository. A commit can be written as $commit = (base, tc, \{edit\})$, where $base$ is the base version that this commit wants to update, tc is the time of creating this commit and $\{edit\}$ is a set of single-file edits. A *single-file edit* is the code edit in a particular file; we define $edit = (file, diff, te)$, where $file$ indicates a file to be updated, $diff$ indicates the content difference, and te indicates the time of this edit. In practice, $diff$ can be directly obtained by differentiating the $base$ version and the newly-changed code. In GitHub, a commit (or an edit) does not contain an explicit end time; therefore, we define the end time te of an edit as the time that a next commit changes the same file, i.e., the time of creation of the next commit.

A commit consists of a set of code edits while a pull request consist of a set of commits. A pull request can be directly defined as $PR = (dev, \{commit\})$, where dev

denotes the developer who submits the pull request and $\{commit\}$ denotes a set of commits.

In this study, we study how multiple pull requests change the same code. To facilitate the statement, we refer the two or more pull requests that aim to simultaneously modify the same code to as a group of *competing pull requests*. For instance, if two developers want to change the same variable into different assignments, their changes cannot be merged automatically. At least one of their changes will be abandoned. Intuitively, in competing pull requests, different pull requests compete with each other and the branch manager has to manually decide which pull request should be merged. We formally define a group of competing pull requests as follows.

Definition 1. A group of *competing pull requests* (CPR for short) is a set of pull requests, each of which contains an edit that aim to update one or more same lines of code and these edits have an overlapping time period. We refer the same lines to as *competing lines*.

For instance, given an edit $edit_1$ belonging to a commit $commit_1$ and its pull request PR_1 and $edit_2$ belonging to $commit_2$ and PR_2, we say PR_1 and PR_2 are competing pull requests, if the base versions $base_1 = base_2$, their time $tc_1 < te_2$ and $tc_2 < te_1$ (i.e., $edit_1$ and $edit_2$ have an overlap of their time periods), the content difference $diff_1$ and $diff_2$ shares the same line, and developers $dev_1 \neq dev_2$.[3] In short, if several unmerged pull requests aim to change the same line of code, we consider them as competing pull requests.

Recall the example in Figure 1. Pull requests 1247 by Developer A and 1292 by Developer M contain two edits that aim to update the same lines in the original code of the base branch. Once the two edits share a time period, we say these two pull requests form a group of competing pull requests. From the perspective of human developers, we can briefly summarize that Pull request 1247 tries to change the code style and Pull request 1292 tries to update the expressions; from the perspective of GitHub, both pull requests are to change source code at the same lines.

Note that it is possible that more than two pull requests change one same line. In this paper, a group of competing pull requests contains all the pull requests that change the same lines, i.e., the competing lines. That is, two different groups of competing pull requests do not share the same competing lines; meanwhile, two groups of competing pull requests can share the same pull request. In practice, competing lines may cross files. Counting all possibilities of competing lines may lead to the "combinatorial explosion" problem and cannot be exhaustively enumerated [11]. In our work, we limit competing lines in each group of competing pull requests only inside one file. This file that contains competing lines is call a *competing file*. If two pull requests aim to edit two same lines from two files, we treat these pull requests as two groups competing pull requests, each group of which has competing lines in one competing file. We detail the data collection of all competing pull requests in Section III-C.

In addition, we define the concept of *pseudo-competing pull requests*. This concept is used in the study in Section IV-A for comparison.

Definition 2. A group of *pseudo-competing pull requests* (denoted by XPRs) is a set of pull requests, each of which contains an edit that aim to update the same file but no shared lines and these edits have an overlapping time period. That is, the pseudo-competing pull requests focus on editing the same file but require no same lines.

B. Research Questions

Prior work in evaluating pull request integration mainly focuses on the structured or non-structured merge conflicts [1], [2], [7]. The aim of this paper is to explore the competing pull requests and to further understand how these pull requests impact the code integration. We design four RQs and conduct a study to find out the answers.

RQ1. How do competing pull requests distribute? We give a general statistics on the competing pull requests in our dataset in RQ1.

RQ2. How many developers are involved in one group of competing pull requests? Developers submit pull requests to add their contributions to a repository. Competing pull requests reveal the divergence between developers. In RQ2, we examine how many developers are involved in one group of competing pull requests.

RQ3. How much code is modified by competing pull requests? From the definition of competing pull requests, we require pull requests update at least one identical line of code. In RQ3, we count the overlap of the code among competing pull requests.

RQ4. How does competing pull requests affect the merging of pull requests? We further analyze the impact of competing pull requests on the pull request integration. We leverage Spearman correlation coefficient to measure the impact of competing pull requests on the merge decision.

C. Data Preparation

GitHub has automatically recorded the historical data of pull requests. However, competing pull requests are not recorded directly. Conducting the dataset of competing pull requests requires extracting previous commits that are submitted to the same files and collecting subsequent merge results through the whole life cycle of pull requests. We selected 60 Java repositories with the most forks in GitHub and extracted their pull requests that are submitted in 2017.

[3]We do not consider the case that one developer submit two pull requests to change the same line.

Table I
SUMMARY OF 60 JAVA REPOSITORIES IN GITHUB IN THE STUDY

Repository statistics	Min	Median	Max	Average	St.Dev.	Total
# Java files	36	787	18120	1712	2750	102725
# Executable LoC	1416	73872	1463778	171926	270686	10315546
# Fork	2290	3724	17587	4521	2920	271256
# Releases	0	45	295	68	68	4068
# Branches	1	7	231	24	44	1432
# Pull requests	58	534	13919	1350	2222	81006
# Issues	1	90	984	141	159	8444
Groups of CPR †	2	317	60301	3792	10695	227524
Groups of XPR †	3	298	486785	18309	72495	1098534
Pull requests in CPR	2	54	1321	158	292	9476
Pull requests in XPR	1	39	1288	109	228	6565

† We use CPR and XPR to denote competing pull requests and pseudo-competing pull requests, respectively.

Table I shows the summary of our data collection from these 60 Java repositories in GitHub.[4] For sake of space, we use CPR and XPR to denote competing pull requests (Definition 1) and pseudo-competing pull requests (Definition 2), respectively. In total, 102K Java files and 10,315 KLoC are examined in our study; 9,476 pull requests are considered as competing pull requests. We collect and prepare the data based on the following steps.

Repository selection. We select the top 100 Java repositories with the most forks in GitHub.[5] We use GitHub API to access the public event timeline and extract the raw data of the whole year of 2017, i.e., pull requests that are submitted from January 1st to December 31st, 2017.

Pull request extraction. To find out competing pull requests, we need to match source code files to pull requests and compare edited files among pull requests. The GitHub API is used to exhaustively check commits in pull requests. For each pull request, we extract its commit list and the list of edited files by each commit. Given a specific branch, if several pull requests tend to update the same source code file, we record these pull requests and check whether the pull requests are derived from the same version (branch) by backtracking the base version of each pull request.

Code line analysis. We extract the time of creating each involved commit and collect the end time of each edit (see Section III-A). Then we trace back previous commits and exhaustively compare the differences of edited files by commits to check whether several pull requests tend to update the same lines. We use the same *diff* tool in GitHub to detect differences and trim extra spaces to avoid explicit duplicates. Once two or more pull requests have edited at least one same line and the edits contain an overlapping time period, we label these pull requests as competing pull requests. After this step, 61 repositories with competing pull requests are kept. All these repositories are with a large amount of forks and active users; one exception repository is jleetutorial/maven-project, which is a course-related repository for learning the usage of Maven and contains only two Java files. We removed this repository from our selection. Finally, we have collected 9,476 competing pull requests from 60 repositories.

Note that the interaction among pull requests in GitHub is complex. For instance, some pull requests may be removed from the repository by developers; in this case, we cannot collect any information of these removed pull requests.

[4]Data of competing pull requests in our study are publicly available, http://cstar.whu.edu.cn/p/cpr/.

[5]Java repositories with the most forks in GitHub, http://github.com/search?l=java&o=desc&q=""&s=forks&type=Repositories.

IV. STUDY RESULTS

A. RQ1. How do competing pull requests distribute?

Goal. RQ1 investigates the general statistics of the data scale of competing pull requests.

Figure 2 presents the groups of pull requests of competing and pseudo-competing pull requests in the log scale. The 4th repository, i.e., elastic/elasticsearch, and the 16th repository, i.e., udacity/ud851-Exercises, have the most groups of competing pull requests and pseudo-competing pull requests, respectively. During 12 months, 43 out of 60 repositories have received over 100 groups of competing pull requests; 22 repositories have received over 1,000 groups of competing pull requests.

Figure 3 presents box-plots of competing and pseudo-competing pull requests in repositories. On one hand, we show pull request groups of competing and pseudo-competing pull requests in the log scale to understand the distribution among 60 repositories. On the other hand, we show the percentage of single pull requests in all pull requests. As shown in Figure 3, a half of repositories contain over 317 groups of competing pull requests and three quarters of repositories have over 100 groups. Meanwhile, in a half of repositories, over 44% of pull requests belong to competing pull requests; in three quarters of repositories, over 31% of pull requests belong to competing pull requests.

Finding 1. There indeed exist a large number of competing pull requests in GitHub. In three quarters of repositories, 31% of pull requests belong to competing pull requests. Exploring the impacts by competing pull requests can help understand the complexity of collaborative development.

B. RQ2. How many developers are involved in one group of competing pull requests?

Goal. RQ2 is to show how many developers have contributed to competing pull requests. We leverage the distribution of developers across all repositories to present the counts of developers.

In Figure 4, we present the numbers of average contributors in competing and pseudo-competing pull requests of all repositories. The 31th repository and the 16th repository

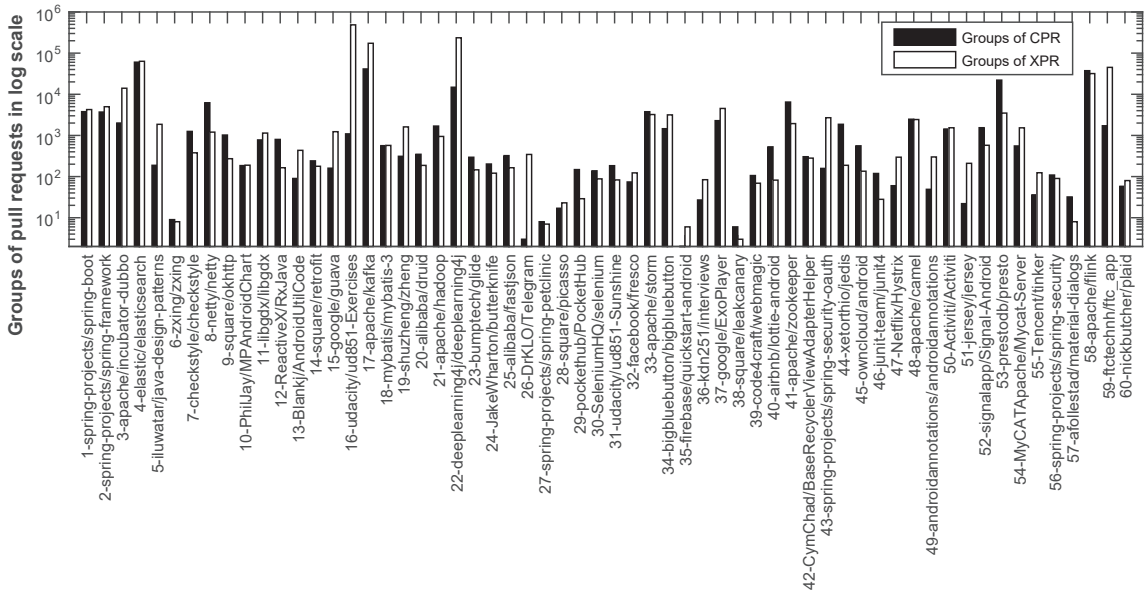


Figure 2. Groups of competing pull requests and pseudo-competing pull requests from 60 repositories (in the log scale). We list the indexes (1 – 60) and names of all repositories in details.

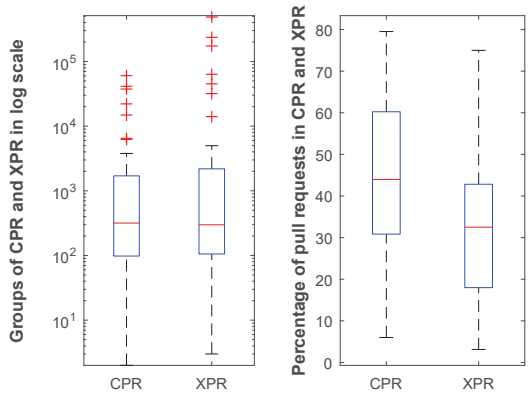


Figure 3. Box-plots of competing pull requests and pseudo-competing pull requests in repositories. The left-side subfigure shows pull request groups in the log scale; the right-side subfigure shows the percentage of single pull requests in all pull requests.

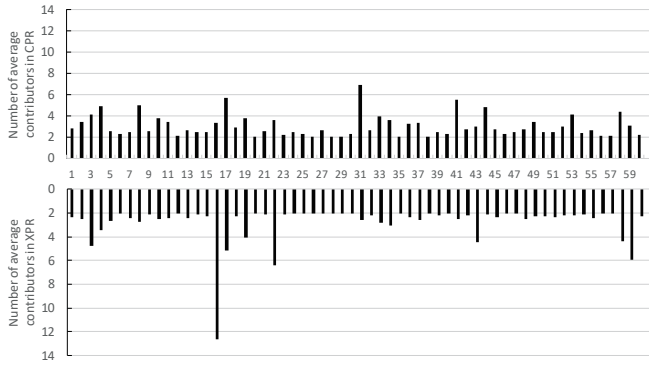


Figure 4. Number of average contributors in competing pull requests and pseudo-competing pull requests in all repositories.

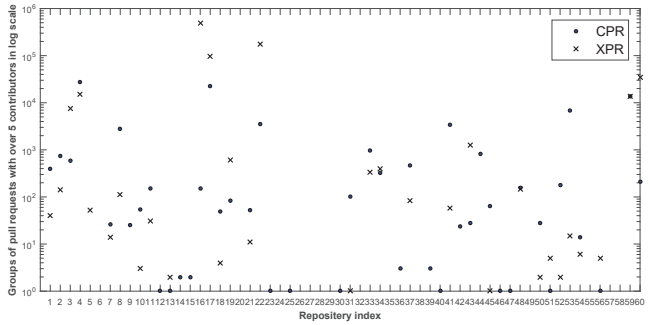


Figure 5. Groups of pull requests over five contributors for all the repositories in the log scale.

have the largest numbers of average developers for competing and pseudo-competing pull requests, respectively. By checking the average developers of pull request groups, all 60 repositories contain over two developers in average. The large number of developers in one group of competing pull requests indicates that these developers try to simultaneously update the same lines of code.

To further understand the distribution of developers, we count the groups of pull requests with over five contributors for all the repositories in Figure 5. Taking the 1st repository for an example, more than 300 groups of competing pull requests involve over five developers; that is, over five developers try to change the same code into different ways for all these groups of competing pull requests. As shown

in Figure 5, 20 out of 60 repositories have more than 100 groups of competing pull requests, each of which is submitted by over five developers.

Finding 2. In average, most repositories have competing pull requests that involve two to three developers. We observe that 20 out of 60 repositories have more than 100 groups of competing pull requests with five developers. This indicates that resolving the competition among pull requests by multiple developers can be expensive in labor cost.

C. RQ3. How much code is modified by competing pull requests?

Goal. The nature of competition among pull requests is the potential conflicts among updated code at the same lines. We employ RQ3 to explore how many lines of code are updated at the same time.

Figure 6 presents the counts of competing lines in competing pull requests for each repository. We can observe that many box-plots contain a number of outliers. As shown in Figure 6, although the median counts of competing lines are two or three, 21 out of 60 repositories have at least one group of competing pull requests with over 100 competing lines; 7 repositories even have competing pull requests with over 200 competing lines. Meanwhile, in 46 repositories, 25% groups of competing pull requests have over 5 competing lines; in 9 repositories, 25% groups of competing pull requests have over 10 competing lines; this fact indicates that there exist a large number of groups of competing pull requests that aim to update over 10 same lines of code into different code. Thus, a branch manager who tries to address the problem of competing pull requests has to spend much effort understanding the context and the semantic of these newly submitted pull requests.

To further understand competing lines, we use Figure 7 to illustrate the accumulation of groups of competing pull requests by counting the number of competing lines for each repository. Considering the groups of competing pull requests, 42 out of 60 repositories have over 10% of competing pull requests with over 10 competing lines; 21 repositories have over 20% of competing pull requests with over 10 competing lines; 3 repositories have over 30% with over 10 competing lines. Considering the number of competing lines, 16 out of 60 repositories have over 10% of competing pull requests with over 20 competing lines; 5 repositories have over 10% with over 30 competing lines. Such a large number of competing lines lead to the high complexity of choosing a suitable pull request. A possible solution to addressing these competing pull requests is to set the number of competing lines during the submission of pull requests.

Finding 3. There exist a large number of competing lines in competing pull requests. We observe that 42 out of 60 repositories have over 10% of competing pull requests with over 10 competing lines. Our study implies that a new way of resolving these competing lines in competing pull requests is in urgent need in GitHub.

D. RQ4. How does competing pull requests affect the merging of pull requests?

Goal. We leverage RQ4 to analyze the impact of competing pull requests on the merge result. Intuitively, competing pull requests may result in a more complex scenario than common pull requests. The competition among pull requests makes the integration of pull request difficult. We employ the Spearman correlation coefficient and conduct two experiments to show the impact of competing pull requests.

We extract 10 attributes of pull requests and evaluate their impacts on whether a pull request is merged. Table II lists these 10 attributes $A1$ to $A5$ and $B1$ to $B5$ and the merge result Y. We use attributes $A1$ to $A5$ to characterize single pull requests and use attributes $B1$ to $B5$ to characterize groups of pull requests. In addition, all attributes $A1$ to $A5$ can be used to describe groups of pull requests by calculating the average value of each group. For instance, a group of pull requests can be measured by the average of $A1$ of each pull request. Therefore, the aim of RQ4 is to understand which attribute has a strong correlation to the merge result of a pull request; meanwhile, we aim to understand which type of pull requests has stronger correlations.

We employ Spearman correlation coefficient to investigate the correlation between two attributes. The Spearman correlation coefficient is a non-parametric measure of rank correlations. This measurement describes the statistical dependency between the rankings of two attributes via assessing how well the relationship between two attributes using a monotonic function [8]. We briefly introduce the definition of Spearman correlation coefficient as follows. For a sample of size n, given two attributes X and Y, each of n samples X_i and Y_i ($1 \leq i \leq n$) can be converted into ranks, i.e., r_{X_i} and r_{Y_i}. The Spearman correlation coefficient $s(X, Y)$ of X and Y is computed from:

$$s(X, Y) = \frac{\text{cov}(r_X, r_Y)}{\sigma_{r_X} \sigma_{r_Y}}$$

where $\text{cov}(r_X, r_Y)$ is the covariance of the rank variables of X and Y and σ_{r_X} and σ_{r_Y} are the standard deviations of the rank variables of X and Y. The absolute value of Spearman correlation coefficient $s(X, Y)$ indicates the correlation between X and Y; the positive or negative value indicates that the correlation is positive or negative. Meanwhile, the p-value is used to identify the statistical significance.

We compare Spearman correlation coefficient among three types of single pull requests. Let $E1$ – CPR, $E2$ – XPR, and $E3$ – APR denote all single pull requests of competing

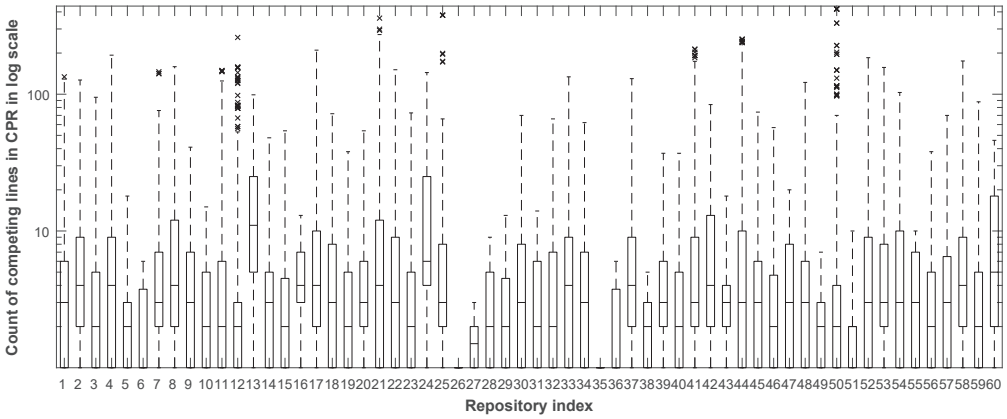


Figure 6. Count of competing lines in competing pull requests in the log scale for each repository.

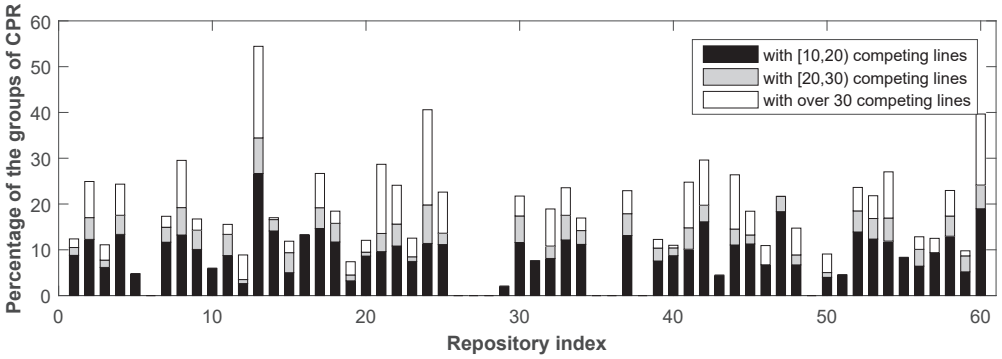


Figure 7. Accumulation of groups of competing pull requests when counting the number of competing lines.

Table II
ATTRIBUTES OF PULL REQUESTS UNDER EVALUATION

Attribute index	Attribute description
A1	Number of commits
A2	Number of edited files
A3	Number of previously merged pull requests by a developer
A4	Number of added lines in diff
A5	Number of deleted lines in diff
B1	Number of developers
B2	Number of competing lines
B3	Number of edits for the competing file
B4	LoC of comments
B5	Number of messages
Y	Merge result – a pull request is merged or not, i.e., 1 or 0.

pull requests, pseudo-competing pull requests, and any pull requests, respectively. Then for all these single pull requests, we present the correlation between attributes A_1 to A_5 and Y. Table III shows the correlation coefficient and *p*-values of three types of pull requests.

As shown in Table III, among five attributes of pull requests, $A3$, i.e., the number of previous merged pull requests by the same developer, have strongest correlation with the merge results of pull requests. In $E1$ – CPR and $E2$ – XPR, the attribute with the weakest correlation is $A4$, i.e., the number of added lines while in $E3$ – APR, the weakest one is $A5$, i.e., the number of deleted lines. The *p*-values show that the calculation of Spearman correlation coefficient for all correlations is statistically significant.

Comparing the correlation across $E1$, $E2$, and $E3$, two attributes $A2$ and $A5$ of CPR have stronger impacts on the merge results than those of XPR; three attributes $A1$, $A3$, and $A4$ of CPR have weaker impacts on the merge results than those of XPR. For APR, four attributes $A1$, $A2$, $A3$, and $A5$ win the strongest impacts on the merge results among the three types of pull requests in comparison. We can conclude that single pull requests of competing pull requests and pseudo-competing pull requests behave similar; among five attributes in the experiment, we have not observed strong impacts in CPR or XPR, comparing with APR. We can speculate that both competing pull requests and pseudo-competing pull requests are more complex than general pull requests: attributes in the study do not strongly indicate the merge results.

After comparing the impacts from single pull requests, we compare the impacts of groups of competing pull requests and pseudo-competing pull requests. Let $E4$ – groups of CPR and $E5$ – groups of XPR denote all

Table III
SPEARMAN CORRELATION COEFFICIENT BETWEEN ATTRIBUTES OF SINGLE PULL REQUESTS AND THE MERGE RESULT

Attribute	E1 – CPR		E2 – XPR		E3 – APR	
	correlation	*p*-value	correlation	*p*-value	correlation	*p*-value
A1 – Commits	0.0407	7.44E-05	0.0453	1.00E-05	**-0.1592**	4.76E-114
A2 – Edited files	0.0577	1.90E-08	0.0298	0.0037	**-0.1484**	3.38E-99
A3 – Previous merge	0.6246	0.0000	**0.6685**	0.0000	0.1913	1.02E-164
A4 – Added lines	-0.0045	0.6633	-0.0261	0.0110	**-0.1801**	4.70E-146
A5 – Deleted lines	0.0581	1.46E-08	0.0514	5.57E-07	**-0.0638**	1.37E-19

Table IV
SPEARMAN CORRELATION COEFFICIENT BETWEEN ATTRIBUTES OF GROUPS OF COMPETING PULL REQUESTS AND THE MERGE RESULT

Attribute	E4 – Groups of CPR		E5 – Groups of XPR	
	correlation	*p*-value	correlation	*p*-value
Avg. A1 – Commits †	-0.0065	0.0018	**-0.3431**	0.0000
Avg. A2 – Edited files	-0.0297	1.28E-45	**-0.5159**	0.0000
Avg. A3–Previous merge	0.7290	0.0000	**0.8090**	0.0000
Avg. A4 – Added lines	-0.0619	9.62E-192	**-0.5271**	0.0000
Avg. A5 – Deleted lines	-0.0528	6.02E-140	**-0.2993**	0.0000
B1 – Developers	0.0886	0.0000	**0.1443**	0.0000
B2 – Competing lines	0.0012	0.5624	**0.0198**	4.20E-54
B3 – Edits	**0.1524**	0.0000	-0.1308	0.0000
B4 – LoC of comments	-0.0468	1.22E-110	**-0.1321**	0.0000
B5 – Messages	0.0563	3.30E-159	**-0.4540**	0.0000

† *Avg.* is short for the average of values.

groups of competing pull requests and pseudo-competing pull requests, respectively. We present the correlation between attributes A_1 to A_5 as well as B_1 to B_5 and Y. Table IV shows the correlation coefficient and *p*-values of two types of pull request groups. For $A1$ to $A5$, we calculate the average values because a group of pull requests contains one value for each pull request. For Y of pull request groups, $Y = 1$ if at least one pull request in the group is merged, else $Y = 0$.

As shown in Table IV, among ten attributes of pull requests, $A3$, i.e., the number of previous merged pull requests by the same developers, have strongest correlation with the merge results of pull requests, in both $E4$ – groups of CPR and $E5$ – groups of XPR; meanwhile, the attribute with the weakest correlation is $B2$, i.e., the number of competing lines in a group. The *p*-values show that the calculation of Spearman correlation coefficient for all correlations is statistically significant.

Comparing the correlation across $E4$ and $E5$, we can surprisingly observe that nine out of ten attributes of $E5$ have stronger correlation values than those of $E4$. That is, the correlation of competing pull requests between attributes and the merge results is much weaker than the correlation of pseudo-competing pull requests. We speculate that attributes of competing pull requests cannot imply whether a pull request in the group can be merged, comparing with pseudo-competing pull requests. A reason for this fact is that the scenario of competing pull requests are more complex than that of pseudo-competing pull requests. Competing pull requests do not perform strong impacts to the merge results. We can further infer that addressing competing pull requests as well as their merge conflicts leads to high labor cost of developers.

Finding 4. We observe that the number of previously merged pull requests by a developer has the strongest impact on the merge result. Meanwhile, we observe that attributes of groups of competing pull requests have no strong correlations with the merge result. We speculate that this is caused by the complex scenario of competing pull requests.

V. THREATS TO VALIDITY

Conduct validity. The dataset of our study is conducted based on mining top 100 Java repositories in GitHub. After data preparation, 60 repositories with 9,476 competing pull requests for one year are kept in the study. The data scale in our study could be larger. We notice that crawling pull requests with previous commits that target the same files is time-consuming due to the storage mechanism of GitHub. We believe a study with a large dataset can reveal more findings of competing pull requests.

Internal validity. In our study, we have not considered the case of file deletion. The definition of competing pull requests is limited to updating an existing file. Submitting a pull request that aims to update a deleted file can be viewed as a kind of competing pull requests. However, as the first study on this problem, we omit the case of file deletion for the sake of clear description. Another internal validity is that in one group of competing pull requests, we limit competing lines in only one file. Therefore, if two pull requests try to update same lines from two files, we define these competing pull requests in two groups with different competing lines. It is possible to consider multiple files in one group of competing pull requests. However, the exhaustive enumeration of all the competing pull requests leads to a large amount of calculation. In our study, we define competing files in one group competing pull requests appear in only one file as a trade-off between the enumeration of possibilities and the resource of calculation.

External validity. Our study only investigates Java repositories on GitHub. We cannot guarantee that our findings can be directly generalized to repositories with other pro-

gramming languages. Another external validity is all pull requests are submitted in 2017. A large study on pull requests in several different years can explain the ability of generalization.

VI. Related Work

In this paper, we conduct a study on competing pull requests in GitHub to explore how multiple pull requests change the same code. We summarize the related work in three dimensions.

Change conflicts. Many empirical studies on conflicts have been conducted to understand the negative impacts of frequent conflicts on daily collaborative development. Perry et al. [21] have studied the collaborative development in a large-scale industrial project in 2001. They found that the high degree of parallel changes in a company has a significant correlation with the number of defects. Zimmermann et al. [27] have investigated historical data of four large-scale and open-source projects on CVS. Their study found that there are 23% to 47% commits that can cause conflicts. These conflicts have wasted decision makers a considerable time and effort on the merge of conflicts. Brun et al. [6] and Kasi & Sarma [19] analyzed open-source software on GitHub. They pointed out that conflicts are normal and harmful in daily collaborative; early detection of conflicts can improve the productivity as well as the quality of projects. Besides the work of change conflicts, existing work has explored the change-related defects. Macho et al. [20] predicted build co-changes via analyzing code changes; Jaafar et al. [17] have examined the impacts of design pattern as well as anti-patterns on changes and defects. Xuan et al. [24] have analyzed the changes of test cases via refactoring. A recent study by Huijgens et al. [16] infers the predictive ability of software metrics for continuous delivery projects via data mining on logs.

Tool support for merging. To better solve the merge conflicts, researchers have proposed variety of novel merge tools. GNU diff3 [10] is a typical unstructured line-based merge tool. The drawback of unstructured tools is *ordering conflicts*, i.e., the conflict of order changing of certain code elements, by mistake [3]. To address the issues of unstructured merge tools, JDime by Apel et al. [2] is proposed to perform the structured merge. JDime merges commits with potential conflicts according to the difference of abstract syntax trees. The structured method of merging can avoid many ordering conflicts that unstructured tools cannot correctly identify. As a trade-off between unstructured and structured tools for merging changes, Apel et al. [3] have proposed FSTMerge, a semi-structured one. FSTMerge uses program structure trees to represent programs and treat the process of merging commits as merging subtrees. A recent study by Accioly et al. [1] has examined 125 Java repositories on GitHub with the semi-structured tool, FSTMerge. This study explored patterns and frequencies of conflicts among the total 70K merges.

Factors on merging. Many studies focused on finding potential factors that have strong influences on the decision of merging changes. Dabbish et al. [9] organized an interview of 24 developers with the topic of social collaborative development on GitHub. They found that pull requests by a developer are more likely to be merged if the developer have more social contributions in coding. Gousios et al. [14] studied the pull-request model in GitHub and proposed 15 factors that can influence the acceptance of pull requests. These factors include multiple aspects, such as recent activities by developers, project information, and code quality metrics. Additionally, they further built a random forest model to predict the acceptance of pull requests. In their follow-up work, Gousios et al. [15] have conducted a survey of 749 project integrators and investigated challenges and problems in the pull-request model. Zhu et al. [25], [26] have investigated patterns and effectiveness of code changes in GitHub. Xie et al. [23] studied the impact on the human-focus factors of debugging. Jiang et al. [18] conducted a study on inactive developers to understand the status of these developers in GitHub. Recently, Beller et al. [4] have conducted an explorative analysis to investigate the impacts between continuous integration and GitHub.

VII. Conclusions

We design a study on competing pull requests to understand how multiple pull requests change the same code in GitHub. In the study, we collect 9,476 competing pull requests from 60 Java repositories with the most forks. Our result shows that the competing pull requests frequently exist in repositories in GitHub; the scenario of competing pull requests is more complex than that of other pull requests and competing pull requests have low correlation with the merge of pull requests. This is the first preliminary study that explores the complexity of competing pull requests in GitHub.

The reason for the weak correlation between human-defined attributes of competing pull requests and the merge results is unclear. We plan to design a detailed study to understand whether the weak correlation is caused by the complexity of competing pull requests. We also plan to conduct a detailed study on a large number of competing pull requests.

Acknowledgment

The work is supported by the National Key R&D Program of China under Grant No. 2018YFB1003901, the National Natural Science Foundation of China under Grant Nos. 61502345 and 61572375, the Young Elite Scientists Sponsorship Program By CAST under Grant No. 2015QNRC001, and the Technological Innovation Projects of Hubei Province under Grant No. 2017AAA125.

References

[1] P. R. G. Accioly, P. Borba, and G. Cavalcanti. Understanding semi-structured merge conflict characteristics in open-source java projects. *Empirical Software Engineering*, 23(4):2051–2085, 2018.

[2] S. Apel, O. Leßenich, and C. Lengauer. Structured merge with auto-tuning: balancing precision and performance. In *IEEE/ACM International Conference on Automated Software Engineering, ASE '12, Essen, Germany, September 3-7, 2012*, pages 120–129, 2012.

[3] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. Semistructured merge: rethinking merge in revision control systems. In *19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) and 13th European Software Engineering Conference (ESEC), Szeged, Hungary, September 5-9, 2011*, pages 190–200, 2011.

[4] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: an explorative analysis of travis CI with github. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17, Buenos Aires, Argentina, May 20-28, 2017*, pages 356–367, 2017.

[5] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 168–178, 2011.

[6] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Early detection of collaboration conflicts and risks. *IEEE Trans. Software Eng.*, 39(10):1358–1375, 2013.

[7] G. Cavalcanti, P. Borba, and P. Accioly. Evaluating and improving semistructured merge. *Proc. ACM Program. Lang.*, 1(OOPSLA):59:1–59:27, Oct. 2017.

[8] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken. *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge, 2013.

[9] L. A. Dabbish, H. C. Stuart, J. Tsay, and J. D. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Computer Supported Cooperative Work, CSCW '12, Seattle, WA, USA, February 11-15, 2012*, pages 1277–1286, 2012.

[10] F. S. Foundation. Comparing and merging files. http://www.gnu.org/software/diffutils/manual/diffutils.html, 2016.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and company, 1979.

[12] GitHub. About merge conflicts. http://help.github.com/articles/resolving-a-merge-conflict-on-github/, 2018.

[13] GitHub. About pull requests. http://help.github.com/articles/about-pull-requests/, 2018.

[14] G. Gousios, M. Pinzger, and A. van Deursen. An exploratory study of the pull-based software development model. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 345–355, 2014.

[15] G. Gousios, A. Zaidman, M. D. Storey, and A. van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *37th IEEE/ACM International Conference on Software Engineering, ICSE '15, Florence, Italy, May 16-24, 2015, Volume 1*, pages 358–368, 2015.

[16] H. Huijgens, R. Lamping, D. Stevens, H. Rothengatter, G. Gousios, and D. Romano. Strong agile metrics: mining log data to determine predictive power of software metrics for continuous delivery teams. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE '17, Paderborn, Germany, September 4-8, 2017*, pages 866–871, 2017.

[17] F. Jaafar, Y. Guéhéneuc, S. Hamel, F. Khomh, and M. Zulkernine. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. *Empirical Software Engineering*, 21(3):896–931, 2016.

[18] J. Jiang, D. Lo, X. Ma, F. Feng, and L. Zhang. Understanding inactive yet available assignees in github. *Information & Software Technology*, 91:44–55, 2017.

[19] B. K. Kasi and A. Sarma. Cassandra: proactive conflict minimization through optimized task scheduling. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 732–741, 2013.

[20] C. Macho, S. McIntosh, and M. Pinzger. Predicting build co-changes with source code change and commit categories. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER '16, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 541–551, 2016.

[21] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, 2001.

[22] Stackoverflow. Resolving a merge conflict. http://stackoverflow.com/search?q=resolving+a+merge+conflict, 2018.

[23] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu. Revisit of automatic debugging via human focus-tracking analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16, Austin, TX, USA, May 14-22, 2016*, pages 808–819, 2016.

[24] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus. B-refactoring: Automatic test code refactoring to improve dynamic analysis. *Information & Software Technology*, 76:65–80, 2016.

[25] J. Zhu, M. Zhou, and A. Mockus. Patterns of folder use and project popularity: a case study of github repositories. In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, pages 30:1–30:4, 2014.

[26] J. Zhu, M. Zhou, and A. Mockus. Effectiveness of code contribution: from patch-based to pull-request-based tools. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '16, Seattle, WA, USA, November 13-18, 2016*, pages 871–882, 2016.

[27] T. Zimmermann. Mining workspace updates in CVS. In *Fourth International Workshop on Mining Software Repositories, MSR '07 (ICSE Workshop), Minneapolis, MN, USA*, pages 1–11, 2007.