Understanding Merge Conflicts and Resolutions in Git Rebases

Tao Ji Liqian Chen Xin Yi Xiaoguang Mao
Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology
Changsha 410073, China
{taoji, lqchen, yixin09, xgmao}@nudt.edu.cn

***Abstract*—Software merging is an important activity during software development. Merge conflicts may arise and degrade the software quality. Empirical studies on software merging are helpful to understand developers' needs and the challenges of detecting and resolving conflicts. Existing studies collect merges by identifying commits that have more than one parent commit. Different from these explicit merges, rebasing branches is used to merge other changes but rewrites the evolutionary history. Hence, existing studies fail to identify implicit merges performed by rebasing branches. Consequently, the results of these studies may fail to provide comprehensive insights on software merging. In our study, we leverage the recently updated APIs of GitHub to study rebase activities in the pull requests. Our study shows that rebasing is widely used in pull requests. And our results indicate that, to resolve textual conflicts, developers adopt similar strategies shown in existing studies on explicit merges. However, in 34.2% of non-conflict rebase scenarios, developers add new changes during the rebase process. And this indicates that there are some new challenges of validating rebases. Our results provide useful insights for improving the state-of-the-art techniques on resolving conflicts and validating rebases.**

***Index Terms*—software evolution, software merging, merge conflicts**

I. Introduction

Different from centralized version control systems (VCSs), distributed VCSs make it easier to create and merge branches representing different development goals [1] [2]. The pull-requests feature of GitHub makes it more convenient for other developers to contribute their code to the project. The process of incorporating other changes is well known as software merging. Security issues may arise if merged changes are not well examined [3]. And the parallel changes may lead to merge conflicts. One study [4] shows that developers sometimes avoid synchronization by merging, as merge conflicts may interrupt the development process. Therefore, much effort has been devoted to developing tools for detecting and resolving conflicts automatically.

Existing studies [4]–[9] on real-world merge conflicts and resolutions, provide insights on developing tools to assist developers in performing merges. Note that in the context of Git, besides "git merge", "git rebase" is also used to include other changes into the working branch. As shown in Fig. 1, we present the workflow of "git merge" and "git rebase". Conflicts may arise and developers' real intentions on changes may be adversely affected after rebasing [10]. However, we are not knowledgeable about the real-world processes of rebasing branches. Since rebasing branches rewrites the evolutionary history and the previous commits are missing after rebasing [11], existing studies only focus on the "git merge" cases. We wonder whether there exist the same characteristics between merging and rebasing branches. In our paper, we call merges handled by "git merge" ***explicit merges***, and call others handled by "git rebase" ***implicit merges*** or ***rebases***.

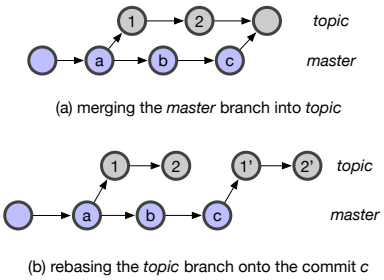

Fig. 1. Merging other changes by "git merge" and "git rebase".

Only by examining the cloned Git repository from GitHub, we fail to restore the process of rebasing branches[1]. Fortunately, the recent changes[2] of GitHub benefit us to restore rebases and relevant commits, by providing APIs on the *HeadRefForcePushedEvent*[3]. After extracting this force-pushed event that happened to the head branch of one pull request, we can have the HEAD commits (e.g., commits 2 and 2′ in Fig. 1(b)) of the previous and rebased head branches. Then, we can retrieve the missing commits from GitHub and then study the rebase.

In this paper, we collect 82 Java repositories from GitHub and identify a total of 51,183 rebase scenarios from the pull requests of these repositories. To the best of our knowledge, this is the first comprehensive and systematic study to investigate how developers rebase their working branches. Our research questions are summarized as follows:

- RQ1: How often do rebases occur in pull requests?
- RQ2: When do developers decide to rebase branches?
- RQ3: Do textual conflicts arise when rebasing branches?
- RQ4: How do developers resolve textual conflicts?

[1]https://help.github.com/en/github/committing-changes-to-your-project/commit-exists-on-github-but-not-in-my-local-clone
[2]https://github.blog/changelog/2018-11-15-force-push-timeline-event/
[3]https://developer.github.com/v4/object/headrefforcepushedevent/

2332-6549/20/$31.00 ©2020 IEEE
DOI 10.1109/ISSRE5003.2020.00016

- RQ5: Do developers add other changes when no textual conflicts arise during the rebase process?

The results show that rebases exist in the closed and merged pull requests of each repository. To sync with the updated base branch and keep the evolutionary history clean, developers choose to rebase their working branch. Our results show that textual conflicts arise in 24.3%-26.2% of rebases, and developers tend to resolve conflicts without introducing new tokens into textual files. Comparing to the results of existing studies [5] [6] [9] on explicit merges, we find that the likelihood of conflicts is not significantly different. One study on real-world explicit merges shows that program elements involved in merge conflicts have more code smells than other elements [8]. Our results indicate that rebases may have similar problems, as we find that developers deal with these conflicts by the similar way for explicit merges. Also, we find that developers often modify rebased branches when no textual conflicts arise during rebasing. Based on the above results, after comparing the workflow of "git merge" and "git rebase", we give some actionable implications for developers and researchers to resolve conflicts and validate rebases.

Overall, we make the following contributions:

- To the best of our knowledge, we are the first to study real-world rebases which also work as software merging.
- The collected rebases[4] can facilitate further studies on this topic.
- We provide evidence that developers deal with conflict rebases similarly for explicit merges.
- We provide evidence that developers need to modify the to-reapply commits during the rebase process.
- We discuss methods that should be improved to assist developers in resolving conflicts and validating rebases.

The remainder of this paper is organized as follows. Section II introduces the background and related works. Section III introduces the study procedure for collecting rebases. Section IV gives answers to each of the five research questions proposed. Section V presents threats to validity. Section VI discusses how to resolve conflicts and validate the final rebases, then Section VII concludes this paper.

II. Background and Related Work

In this section, we first introduce the Git utility "git rebase" briefly and then present related works about software merging.

A. Git Rebase

After specifying the new base commit, the "git rebase" utility individually reapplies the commits of the *to-do* list (e.g., the commit 1 and 2 of Fig. 1(b)). Once textual conflicts arise, Git would ask developers to resolve the conflicts, and then continue the rebasing process. Note that, Git also supports developers to skip this commit, or abort the process.

To facilitate developers to deal with this process, Git provides the interactive mode "git rebase -i" to help them review commits in the to-do list before starting the rebase process. Working in this mode, developers can *add*, *delete*, *modify*, and *reorder* these commits. And they also can *squash* and *fixup* commits of the to-do list. By squashing, developers can meld this commit into the previous commit. And if we fixup, changes will also be melded, but this commit's message will be discarded. The interactive mode of "git rebase" brings challenges on understanding rebases, as we fail to precisely restore the details of rebasing when we cannot match commits of the previous branch with those of the rebased branch.

Some other default settings should be noticed. For example, if one commit is a merge commit that has more than one parent commit, Git would drop it from the to-do list by default. If developers specify the option on reserving merge commits, Git would recreate the merges, while developers may need to resolve conflicts manually during the interactive process.

As introduced above, rich functions provided by "git rebase" and the missing information on relationships between commits, bring challenges on restoring the actual rebase processes.

B. Software Merging

Mens [12] introduces and concludes the early work about software merging in detail. In this section, we introduce some emerging works on software merging.

To reduce conflicts, existing works focus on task divisions [13] and conflict prediction [14]. Brun et al. [14] design and implement Crystal, which uses speculative analysis to help developers identify, manage and prevent conflicts. Guimarães and Silva [15] propose early detection that continuously merges uncommitted and committed changes to detect conflicts.

Resolving conflicts annoys developers and some works aim to relieve developers' burden on resolving conflicts. Apel et al. [16] propose the semi-structured merge to assist unstructured VCSs in automatically resolving ordering conflicts. Niu et al. [17] develop a tool scoreRec that recommends the conflict resolutions ordered by estimating the cost and benefit of resolving conflicts. Nishimura et al. [18] develop MergeHelper that exploits the fine-grained change history of Java source code to help programmers understand conflicts between class methods or fields. Zhu and He [19] propose an interactive approach for resolving conflicts of structured merges, by using version space algebra to represent the large set of candidate programs and ranking the candidate resolutions. Xing and Maruyama [20] propose to leverage the automated program repair to resolve behavioral merge conflicts.

Besides assisting developers in resolving conflicts, researchers pay attention to guaranteeing the quality of merges. Recently, inspired by the earlier work [21], Sousa et al. [22] propose the notion of semantic conflict freedom for 3-way program merges, and develop SafeMerge to verify whether one 3-way program merge violates the contract. Ji et al. [23] propose test oracles for different merges including 2-way, 3-way, and octopus merges, and develop TOM to generate tests revealing semantic conflicts.

Researchers conduct empirical studies to investigate real-world conflicts and resolutions. Zimmermann [6] studies the

[4]https://github.com/tao-ji/Rebases.

Fig. 2. The merged pull request #3299 of the repository "k9mail/k-9".

CVS history of four large projects and finds that between 22.75% and 46.62% of all integrations resulted in a conflict. Mckee et al. [24] conduct interviews of 10 software practitioners to understand their perspectives on merge conflicts and resolutions. And according to the unmet needs of software practitioners, they suggest researchers and tool builders focus on program comprehension, history exploration, etc. Accioly et al. [7] derive nine conflict patterns from semi-structured merge conflicts and find that most conflicts happen when developers edit the same lines of the same method. After conducting a survey, Leßenich et al. [4] propose seven indicators to predict whether conflicts may arise. However, all of these seven indicators are rejected by the real merge scenarios in their following empirical study. Nguyen et al. [25] analyze the collaboration process of four Git repositories at specific periods revealing that a higher integration rate of a project does not generate a higher unresolved conflict rate. And they suggest that Git should merge concurrent changes on two adjacent lines and throw a warning message instead of considering them as conflicting.

Yuzuki et al. [9] study the conflict resolutions at method-level on 779 conflict commits from 10 Java projects. Their results show that 99% (771/779) of conflicts are resolved by adopting one method directly. Menezes et al. [5] conduct a large-scale empirical study on the resolutions of 175,805 conflict chunks from more than 2,700 Java projects. They classify resolutions into different types and study the reasons why these corresponding decisions are made.

Since rebasing branches rewrites the evolutionary history, existing studies fail to study these implicit merges. Different from existing studies that focus on explicit merges, we study rebase cases. In our study, we try to find the difference between explicit merges and rebases. And our results are able to provide more insights on software merging.

III. Study Procedure

In this section, we introduce the dataset, pull requests, identifying rebases and restoring the commit history.

A. Dataset

In this study, we aim to investigate rebases among the open-source projects. GitHub provides APIs to track the force-push event in the pull requests. Hence, to study rebases, we need repositories that have a certain number of non-open pull requests. Considering the rate limits of invoking GitHub APIs, we try to collect a medium and reasonable number of popular repositories that have a large number of non-open pull requests. Hence, leveraging the advanced search[5] provided by GitHub, we search for Java repositories that have more than three thousand stars and one thousand forks. After filtering out those repositories that have less than one thousand closed and merged pull requests, we collect a total of 82 Java repositories and these repositories are popular among the open-source community.

B. Pull Requests

After forking the upstream repository and making new changes, developers can create one pull request to contribute their changes. Two basic branches (i.e., the ***base branch*** and the ***head branch***) are involved in each pull request. The base branch is where developers consider changes that should be applied, and the head branch is what developers would like to be applied. For example, as shown in Fig. 2, we present the

[5]The GitHub API v3 URL: https://api.github.com/search/repositories?q=stars:>3000+fork:>1000+language:java+is:public+archived:false+fork:false

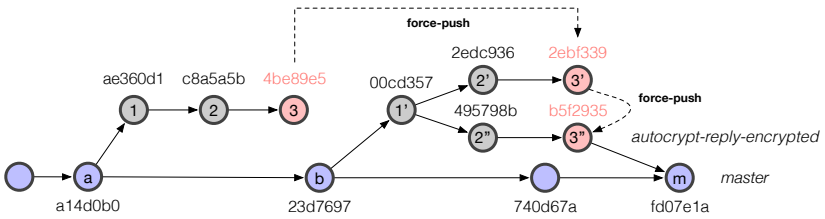

Fig. 3. The commit history of the pull request #3299 of "k9mail/k-9".

pull request #3299[6] of "k9mail/k-9" by taking a screenshot of this pull request's webpage. As we can see, the head branch "autocrypt-reply-encrypted" has been merged into the base branch "master".

During the review process, developers may need to revise their changes by resetting or rebasing the head branch. As shown in Fig. 2, GitHub tracks the changes of the head branch by showing the message "*committer* force-pushed the *head branch* from *commit* to *commit*", which is defined as the *HeadRefForcePushedEvent*. Similarly, GitHub also tracks the force-pushed events on the base branch, by providing the APIs to extract the *BaseRefForcePushedEvent*[7].

In our study, we first utilize the GitHub APIs to collect all of the closed and merged pull requests. Then, we extract BaseRefForcePushedEvents and HeadRefForcePushedEvents that happened in each pull request. We finish collecting the pull requests of 82 repositories before April 15th, 2020.

C. Identifying Rebases

For each pull request, we are able to extract the information of its base and head branches, its commits, and the relevant events. Then, we can restore the reviewing process to identify rebases that happened before this pull request is closed or merged.

For example, for the pull request #3299 of "k9mail/k-9" shown in Fig. 2, we restore the changes of the head branch, as shown in Fig. 3. And we can see that, this author force-pushed this branch twice. After the second force-push, the HEAD commit of this branch becomes the commit "b5f2935"[8], which is the latest commit of these three commits merged into the "master" branch. After cloning the repository locally, we fail to find those two commits "4be89e5" and "2ebf339" by using the Git command "git rev-list". However, fortunately, we can retrieve the information about these two commits by leveraging the GitHub APIs. For each commit, we are able to collect the information of the author, parent commits, and the patch.

Examining the first force-push, we find that the fork point between two branches changes from "a14d0b0" to "23d7697". However, as for the second force-push, two commits "2edc936" and "2ebf339" are reset and then commits "495798b" and "b5f2935" appear in the branch. Working with Git, "git reset" is used to reset the evolutionary history by canceling existing commits. The first force-push brings changes of the commit "23d7697" into the branch "autocrypt-reply-encrypted", while the second force-push does not bring any changes from the "master" branch. In our study, we need to distinguish these rebases from extracted HeadRefForcePushedEvents.

As introduced above, we identify the rebase by *comparing commits of the base branch from which the head branch originates*. In other words, we need to precisely identify the ***fork point*** of the head branch from the base branch. In our experiments, we recursively identify the parents of one commit in the head branch, until we find one ancestor appearing in the base branch but the appearance is not due to merging this pull request. We consider this ancestor as the fork point of the head branch. And to find the fork point of one branch, we just need to focus on the first parent of the commit recursively, since the other parent commits are brought by merging other branches. The remaining problem is to determine one ancestor commit from the base branch. Different from extracting the first-parent commits of the head branch, we need to examine all commits appearing in the base branch. After extracting all commits of the base branch, we are able to identify the fork point of the head branch.

Note that developers are also able to reset the evolutionary history of the base branch of one pull request, and GitHub provides APIs to track this event as the BaseRefForcePushedEvent. By examining this kind of event, we are able to determine the previous base branch. For example, the developer of the pull request #1323 of "k9mail/k-9" force-pushed the base branch. And there is one HeadRefForcePushedEvent before this BaseRefForcePushedEvent. If we miss the BaseRefForcePushedEvent and have the wrong fork point, we would fail to identify this rebase case. Looking at the BaseRefForcePushedEvent, we can tell that the base branch is force-pushed by those developers who have privileges to directly change the base branch. As a result, we need to identify the rebase by examining the head branch with the previous base branch, since the author force-pushed the head branch before knowing the changes of the base branch.

Besides force-pushing the base branch, developers are able to change the base branch to another branch. However, GitHub

[6]https://github.com/k9mail/k-9/pull/3299. The pull request can be found via the URL: https://github.com/:owner/:repo/pull/:number. We would not present the links for pull requests in the following papers.

[7]https://developer.github.com/v4/object/baserefforcepushedevent/

[8]https://github.com/k9mail/k-9/commit/b5f2935. The commit can be found via the url: https://github.com/:owner/:repo/commit/:sha. We would not present the links for commits in the following paper.

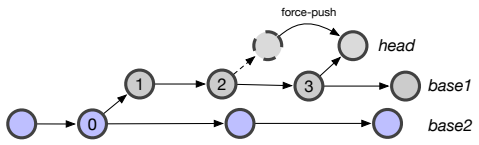

Fig. 4. A case that developers change the base branch from *base1* to *base2* after force-pushing the *head* branch.

does not provide APIs[9] to show the HEAD commits of the old and new base branches. As shown in Fig. 4, we present one case that the *head* branch is force-pushed. In this case, we can tell that developers rebase the head branch if we examine the head branch with the *base1* branch. However, if we consider the *head* branch with the *base2* branch, we fail to tell that rebasing happens to incorporate the changes from the *base2* branch. Different from the BaseRefForcePushedEvent, if the author decides to switch the base branch from *base1* to *base2*, the commits already appeared in the *base1* branch would also be merged. In other words, the author of this pull request wants to merge the commits $1, 2$, and 3 into the *base2*. In this case, it makes sense that the author resets the head branch rather than rebases this branch since this author wants to contribute the commit 3 instead of simply updating the *head* branch. Hence, to reduce the false positives on rebases, we consider the changed base branch to identify rebases. Recall that, we fail to extract the base branches from the BaseRefChangedEvent. If there is some BaseRefForcePushedEvent after the BaseRefChangedEvent, we just need to determine the changed base branch by examining the previous HEAD commit shown in this BaseRefForcePushedEvent. Otherwise, we can determine the changed base branch by examining the pull request.

Finally, we collect a total of 51,183 rebases from 82 Java repositories, and each rebase has two ordered lists of commits that start from different fork points.

D. Restoring Commit History

The locally restored commit history facilitates us to study the rebase processes, by investigating the conflicts and relevant resolutions. For each missing commit in the local repository, we can extract its patch from GitHub via the query "https://github.com/:owner/:repo/commit/:sha.diff". In our study, we focus on the changes in textual files and do not analyze binary files. Then, we can reapply these commits to the local repository to restore the missing commit history. In this procedure, due to GitHub's limitations such as returning big patches, we finally restore the commit history successfully for 50,431 out of 51,183 (98.5%) rebases. Considering the costs and benefits of fetching the entire working trees of commits, we do not restore the remaining rebases.

IV. Research Results

In each research question, we introduce the method used and then present the results.

[9]https://developer.github.com/v4/object/BaseRefChangedEvent

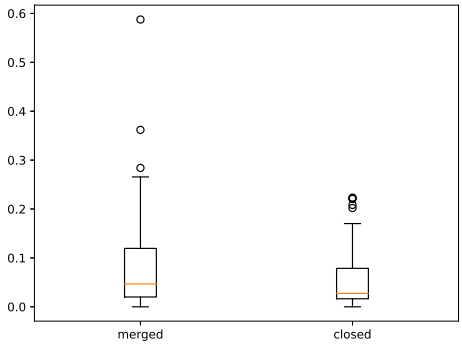

Fig. 5. The proportions of rebased pull requests among merged and closed pull requests respectively for each repository.

A. RQ1: How often do rebases occur in pull requests?

1) Method: By presenting the detailed statistics on collected rebases, we give a clear overview of rebases that happened in the pull requests of GitHub.

2) Results: As shown in Table I, we present the numbers of identified rebases for each repository. In our dataset, there are a total of 51,183 rebase scenarios identified from 82 repositories. And among these rebases, 35,676 rebases exist in the merged pull requests and 15,507 rebases exist in the closed pull requests. By examining each repository, we find that the numbers of rebases vary significantly. Note that projects that have few rebases identified from pull requests, may have other rebases performed in other development activities.

In our dataset, we have 24,778 out of 326,894 (7.6%) pull requests that have rebases. And in one pull request, developers may rebase the head branch multiple times. We find that 10,071 out of 24,778 (40.6%) rebase-involved pull requests have more than one rebase. For example, the merged pull request #15555 of "hazelcast/hazelcast" has a total of 52 rebases. This pull request survived for more than one month and its 13 commits changed 35 files. It is reasonable that developers need to rebase the head branch frequently to reduce the difference between the base branch and the head branch when they complete a complex development goal.

Naturally, we may wonder whether rebasing the head branch affects developers' decision on accepting one pull request. Examining the final status of those pull requests, we find that 7,156 out of 99,887 (7.2%) closed pull requests and 17,622 out of 227,007 (7.8%) merged pull requests have rebases. These percentages are so close that we fail to tell that rebasing branches affects the final decisions on accepting pull requests. To further investigate whether rebases happen differently in closed and merged pull requests, we calculate the percents respectively for each repository. As shown in Fig. 5, we present the proportions of rebase-involved pull requests among different types of pull requests for each repository. There are some projects in which rebasing branches has been used for a great proportion. For example, we find

TABLE I
A TOTAL OF 82 REPOSITORIES STUDIED IN THIS PAPER. THE COLUMN "#REBASE" REPRESENTS THE NUMBERS OF IDENTIFIED REBASES. THE COLUMN "CLOSED" REPRESENTS THE NUMBERS OF REBASE-INVOLVED PULL REQUESTS WITH ALL OF THE CLOSED PULL REQUESTS. THE COLUMN "MERGED" REPRESENTS THE NUMBERS OF REBASE-INVOLVED MERGED PULL REQUESTS.

Repo	#Rebase	#RebasePR		Repo	#Rebase	#RebasePR	
		#merged	#closed			#merged	#closed
MyCATApache/Mycat-Server	1	1/933	0/200	apache/incubator-shardingsphere	3	3/1,989	0/318
libgdx/libgdx	3	3/2,008	0/778	Anuken/Mindustry	6	5/777	0/352
apache/incubator-dolphinscheduler	8	3/1,289	3/180	bigbluebutton/bigbluebutton	10	7/3,980	1/334
seata/seata	10	8/909	0/227	apache/skywalking	15	8/2,056	0/338
azkaban/azkaban	16	12/1,216	1/233	vavr-io/vavr	17	14/1,375	2/99
ionic-team/capacitor	27	13/1,045	8/136	wix/react-native-navigation	31	20/1,311	5/492
swagger-api/swagger-core	37	29/896	3/294	ctripcorp/apollo	46	36/882	2/165
javaparser/javaparser	47	29/915	4/109	GoogleContainerTools/jib	52	26/1,297	1/70
SeleniumHQ/selenium	53	13/471	12/1,211	apache/dubbo	59	27/1,548	7/991
apache/pulsar	63	35/4,242	4/358	antlr/antlr4	74	41/951	4/245
apache/incubator-heron	75	45/2198	5/245	micronaut-projects/micronaut-core	80	53/969	12/300
openhab/openhab1-addons	82	48/2,643	1/466	rstudio/rstudio	83	67/2,861	3/181
arduino/Arduino	89	39/666	12/630	ReactiveX/RxJava	89	57/2,849	11/567
eclipse-vertx/vert.x	106	56/925	10/582	bisq-network/bisq	108	59/1,464	10/406
TechEmpower/FrameworkBenchmarks	108	59/3,856	12/669	dropwizard/dropwizard	115	71/1,644	9/323
Activiti/Activiti	136	79/1,142	16/421	spring-projects/spring-framework	142	30/530	47/2,069
square/okhttp	144	104/2,323	9/482	k9mail/k-9	153	72/1,223	19/643
vespa-engine/vespa	155	116/12,006	7/515	eugenp/tutorials	159	92/6,135	11/2,487
facebook/buck	168	0/8	81/1,007	apache/lucene-solr	172	58/467	42/672
apache/camel	179	58/1,389	62/2,297	jenkinsci/jenkins	183	98/3,269	27/1,268
realm/realm-java	192	138/2,298	8/431	zaproxy/zaproxy	193	132/2,177	2/90
spring-projects/spring-security	200	68/618	44/691	signalapp/Signal-Android	205	24/236	88/1,785
apache/groovy	245	127/351	63/851	keycloak/keycloak	258	125/6,023	23/864
MinecraftForge/MinecraftForge	259	90/1,511	51/1,826	eclipse/deeplearning4j	295	117/3,331	8/315
openzipkin/zipkin	325	177/1,459	9/437	naver/pinpoint	334	114/3,671	17/332
OpenAPITools/openapi-generator	347	181/2,864	21/349	apache/kafka	356	171/2,715	24/4,997
spring-projects/spring-boot	372	2/41	184/4,048	apache/zookeeper	399	0/1	191/1,188
Alluxio/alluxio	401	187/8,937	35/1,578	hibernate/hibernate-orm	423	179/674	112/2,470
bazelbuild/bazel	447	6/23	271/2,948	pentaho/pentaho-kettle	466	292/6,591	22/775
apache/hadoop	474	120/675	116/960	apache/storm	508	250/2,457	24/737
apache/hbase	519	199/974	46/380	confluentinc/ksql	565	298/2,808	12/283
apereo/cas	647	150/2,966	14/528	apache/ignite	676	13/368	289/6,591
netty/netty	787	117/1,359	314/3,724	robolectric/robolectric	996	453/2,314	126/879
Graylog2/graylog2-server	1,036	468/2,779	90/408	apache/druid	1,049	370/5,417	70/753
grpc/grpc-java	1,076	781/3,875	67/700	apache/zeppelin	1,183	3/23	514/3,618
elastic/elasticsearch	1,242	632/25,163	51/5,959	testcontainers/testcontainers-java	1,435	237/1,084	83/374
checkstyle/checkstyle	1,729	617/2,744	176/2,196	quarkusio/quarkus	1,810	675/4,090	85/518
eclipse/che	1,937	1,039/6,117	51/536	neo4j/neo4j	2,853	1,175/7,935	143/1,740
gocd/gocd	2,870	1,004/4,426	105/504	hazelcast/hazelcast	3,064	1,073/9,428	132/1,274
SonarSource/sonarqube	3,217	1,285/2,187	234/1,048	prestodb/presto	3,361	1,117/6,659	220/3,487
apache/beam	4,194	1,206/6,812	746/4,384	apache/flink	5,134	616/2,169	1,812/8,971

that "SonarSource/sonarqube" has the greatest percentage (i.e., 58.8%) of rebase-involved pull requests among merged pull requests. However, examining the distributions and considering the number of pull requests in each repository, we still do not have strong evidence that rebasing branches affect developers' decisions on accepting pull requests.

Result 1: Developers need to rebase branches often, as we find on average 7.6% of pull requests have rebases and 40.6% of them have more than one rebase.

B. RQ2: When do developers decide to rebase branches?

1) Method: It is difficult to tell when and why developers decide to rebase branches, without consulting those developers involved in identified rebases. However, we are able to infer them by examining the consequences brought by rebasing.

If the head branch of one pull request is created on one much early version of the base branch, the possibility of merge conflicts would increase. Hence, reviewers need to carefully review the difference between the head branch and the base branch. To relieve reviewers' burden on reviewing changes, the author of one pull request may rebase the head branch to reduce the difference between these two branches. Hence, we measure the size of differences between two fork points to investigate review efforts changed by rebasing. Git and GitHub present the changes in textual files at the line-level. And the numbers of changed files are explicitly shown to developers. Developers can have an intuitive understanding of the size of changes by looking at these indicators. Hence, we measure the difference between two fork points, by counting the number of changed files and the number of changed textual lines.

Different from the explicit merging, rebasing has the advan-

tage that keeping the commit history clean. Before accepting pull requests, authors may explicitly merge new changes in the base branch into the head branch and then the evolutionary history would be redundant. For example, the author of the pull request #4465 of "k9mail/k-9" commits his changes first, and then merges new changes in the base branch "master" into the head branch by using "git merge". The reviewer asks the author to rebase the head branch instead of merging commits from the base branch. After rebasing the head branch, only one commit including the author's changes is merged into the base branch. Hence, in this research question, we check explicit merge commits in head branches.

2) Results: ***Reducing Difference.*** As shown in Fig. 6, we present the numbers of rebase scenarios that have different numbers of changed files and changed lines. As we can see, more than half (61.7%) of rebase scenarios have more than 25 changed files between two fork points. Although there is no strict standard on considering the size of changes as large, we still can tell that developers may rebase the head branch when many changes are made to the base branch.

Cleaning History. In a total of 1,156 rebase scenarios, explicit merge commits appear in the previous head branch or the rebased branch. And there are 1,090 rebase scenarios in which the previous head branch includes explicit merge commits while the rebased branch does not. There are 57 rebase scenarios in which the previous head branch does not include any merge commit while the rebased branch includes. In these 57 cases, the merge commit in the rebased branch does not appear due to the rebase process of reapplying commits from the previous head branch. There are only 9 rebases from 8 pull requests whose previous and rebased head branches both include merge commits. We manually examine these 9 rebases to investigate whether developers recreate the merge commits during the rebasing process. We find that the merge commits from 7 rebased branches are missing after the following rebases occurring in pull requests. As for the merge commits of the remaining 2 rebased branches are also reset since we fail to find these merge commits in their final commits. As we can see, developers utilize "git rebase" to remove explicit merge commits in the head branches for most cases (i.e., 1090/1099).

> ***Result 2:*** When the base branch evolves with many changes or the head branch involves explicit merge commits, developers perform rebases to relieve the burden on reviewing changes in pull requests and keep the commit history clean.

C. RQ3: Do textual conflicts arise when rebasing branches?

1) Method: For each identified rebase, we use "git rebase" locally with default settings to investigate whether conflicts arise. And once conflicts arise, we abort the rebase process.

Given one previous head branch, developers may reset the latest commit and rebase this branch later. As a result, not all of the commits in the head branch are reapplied onto the new base. And as introduced in Section 2.1, developers may use the interactive mode "git rebase -i" to remove some commits from the to-do list. Hence, we need to make sure what commits in the previous head branch have been reapplied.

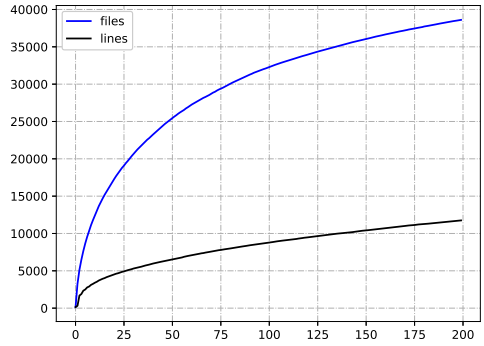

Fig. 6. The numbers of rebase scenarios (Y-axis) that have different sizes of changes (X-axis) between the fork points.

Each commit records the author date and the committer date. And during the rebase process, the author date would be reserved if developers do not specify the option "--ignore-date". In other words, if one commit c_i of the previous head branch has the same author date with one commit c'_j of the rebased branch, then we are sure that c'_j appears due to reapplying c_i. Considering this default setting on recording author dates and developers' involvement on other commit properties (e.g., messages and changes) during rebasing, we match commits by comparing the author dates in our study. This strict standard ensures matched results.

According to the matched results, we first classify the results into two main types: (1) *missing* and (2) *non-missing*. If some commit in the previous head branch cannot be matched with any commit in the rebased head branch, we consider this rebase missing. Otherwise, we consider it non-missing. As for those rebases that are non-missing, we classify them into more detailed types.

(1) *inserted*. Looking at the rebased branch, if there are other commits appearing between the matched commits, we consider that new commits are inserted. For example, in the pull request #2007 of "apache/beam", two adjacent commits "0fafd2e" and "e8405e3" are matched with "b7c28e7" and "6bc464e" respectively, while the commit "fe110a1" is inserted between "b7c28e7" and "6bc464e" in the rebased branch.

(2) *appended*. If there are some commits appearing after the matched commits, we consider that new commits are appended. For example, in the pull request #123 of "MyCatApache/Mycat-Server", the HEAD commit of the head branch changes from "98f13e7" to "a089b23". The rebased head branch has four commits among which the first three commits are matched with those three commits in the previous head branch. The HEAD commit "a089b23" of the rebased branch fails to be found in the previous branch, thus we consider it as appended.

(3) *reordered.* If the order of matched commits is changed,

we consider that developers reorder the commits. For example, in the pull request #2147 of "apache/beam", there are three commits in the previous head branch and three commits in the rebased branch. The first commit of the previous branch appears "2cfcac2" is matched with the head commit "0234b92" of the rebased branch.

Combining these three basic cases, we have eight different types. As shown in Table II, we show the numbers of rebases that have different matched types. There are 29,325 out of 51,183 (57.3%) rebase scenarios in each of which all of the commits in the previous head branch are matched with those commits of the rebased branch in the same order. Given the classification, we study real-world rebases with different levels of confidence.

TABLE II
THE NUMBERS OF REBASES THAT HAVE DIFFERENT TYPES.

Type	Num
matched	29,325
appended	6,625
inserted	569
reordered	55
inserted & appended	146
inserted & reordered	16
appended & reordered	9
inserted & appended & reordered	2

2) Results: Among 49,325 restored rebases, we find that 12,915 (26.2%) rebases have textual conflicts arose. For those "matched" and "appended" rebases, we are sure that all commits in the previous head branch have been reapplied in the same order. Then, we calculate the percentage of conflict rebases among those "matched" and "appended" rebases, and we find that 8,740 rebases out of 35,950 (24.3%) have textual conflicts. As shown in Fig. 7, for each repository, we calculate the percentages of those conflict rebases among "all" rebases and "matched+appended" rebases. Examining those few cases that the percentages vary between different types, we find that the total numbers of rebases are small. For example, the conflict percentages (i.e., 66.7% and 100.0%) of "seata/seata" vary significantly, since there is only one rebase classified as "matched" and conflicts arise in this rebase.

Case Study. Examining repositories that have more than one thousand rebases, we find that "testcontainers/testcontainers-java" has the smallest percentage (i.e., 4.0%) of conflict rebases among its all rebases. We manually examine pull requests of this repository, and we find that 1,314 out of 1,363 rebases come from those pull requests created by "Dependabot Preview"[10], which is a tool developed by GitHub to automatically keep the dependencies up to date. There are only 34 out of 1,314 rebases in which conflicts arise. Obviously, it is necessary to rebase the branches to check the effects of updating dependencies on new changes in the base branch. And it is also reasonable that conflicts do not happen often, considered that changes are made to update dependencies only. Since we fail to match author dates between commits involved in the rebases created by this tool, we find the percentage of conflict "matched+appended" rebases increases to 24.6%.

[10]https://github.com/marketplace/dependabot-preview

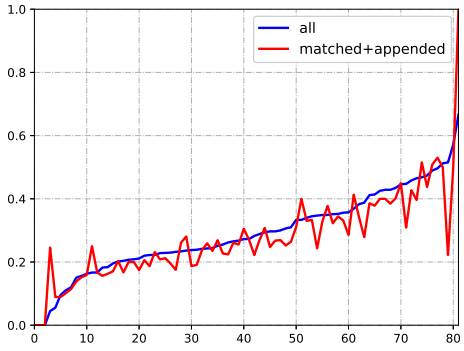

Fig. 7. The proportions of conflict rebases (Y-axis) with different types for 82 repositories (X-axis).

Result 3: Our experimental results show that textual conflicts arise in 24.3%-26.2% of rebases. And there is no significant difference between the possibilities of textual conflicts arising in rebases and explicit merges.

D. RQ4: How do developers resolve textual conflicts?

1) Method: To investigate how developers resolve textual conflicts, we need to identify what conflicts arise and what changes in the rebased branch are made to resolve conflicts.

In this question, we do not consider those rebases classified as *missing*, since we do not have strong evidence that the commits in the rebased branch contain changes resolving conflicts arose by reapplying the commits in the previous branch. Hence, in this research question, we focus on *non-missing* rebases and respectively examine these rebases that have different types.

To investigate the efforts made to resolve conflicts, we study the content of the final merge version. If all lines of the merge version can be found in the relevant versions of the rebase scenario (i.e., the *merge base*, *our* version, and *their* version), we consider this file as "line-comb". Looking into the content at the fine-grained level, if all tokens of the merge version can be found in relevant versions, we consider this file as "token-comb". Otherwise, we consider it as "new-token". If there is one file considered as "new-token", we determine the resolution type of one conflict rebase as "new-token". Otherwise, if there is one file considered as "token-comb", we determine it as "token-comb". Otherwise, we determine it as "line-comb".

2) Results: As shown in Table III, we present the numbers of rebases that have conflicts resolved by different levels of effort. As for those "inserted" rebases, most of them introduce new tokens. It makes sense that changes by introducing new tokens are committed when developers add new commits

TABLE III
THE NUMBERS OF REBASES WHOSE CONFLICTS ARE RESOLVED BY DIFFERENT METHODS.

Type	line-comb	token-comb	new-token
matched	3,705	1,372	2,118
appended	803	287	431
inserted	10	6	87
reordered	2	1	5
inserted & appended	4	2	32
inserted & reordered	0	0	2
appended & reordered	0	0	1

during the rebasing process. However, it is difficult to determine whether these new commits introduce resolutions, as developers may try to add more changes representing different intentions according to the comments in pull requests.

As for those "matched" and "appended" rebases, we are sure that all commits in the previous head branch have been reapplied, and the matched commits in the rebased branch resolve conflicts. There are 2,118 out of 7,195 (29.4%) "matched" rebases and 431 out of 1,521 (28.3%) "appended" rebases have conflicts resolved by "new-token". As we can see, the difference between these percentages is not significant. Looking at those Java files that have new tokens introduced, then we have 1,644 "matched" rebases and 299 "appended" rebases. Existing studies [9] [5] on explicit merges show that developers resolve conflicts by choosing parent versions in most cases, and developers may introduce new code to resolve conflicts in a few cases. Comparing to these studies, we also do not have strong evidence that developers have devoted more efforts to resolving conflicts when rebasing branches.

Result 4: Experimental results show that new tokens are introduced for 28.3%-29.4% of conflict rebases. And results indicate that developers adopt similar strategies used in explicit merge scenarios.

E. RQ5: Do developers add other changes when no textual conflicts arise during the rebase process?

1) Method: In this research question, we compare the branch rebased by ourselves with the collected rebased branch. As for those changed Java files, we use ChangeDistiller [26] to extract fine-grained changes. ChangeDistiller returns the change actions with the changed entities as fine-grained changes, and we combine the change action with the changed entity as the change type to study the differences.

2) Results: To our surprise, we find 7,555 out of 22,111 (34.2%) non-conflict "matched" rebases in which the commits of the to-do list have been modified after being reapplied. For example, the author of the pull request #6208 of "MinecraftForge/MinecraftForge" deleted one method by the commit "0eeb4ed". Then he rebased this branch whose head commit changes to "0f82f7e", but this commit does not delete the method, even we can delete it without any textual conflicts.

As for those "appended" rebases, we compare the constructed rebased branch with the last matched commit in the rebased branch. Then, we find that differences exist in 414 out of 5,098 (8.1%) non-conflict "appended" rebases. Looking at these two significantly different percentages (i.e., 34.2% and 8.1%), we can give reasonable explanations on the decrease. Using the interactive mode, developers are able to modify the reapplied commit. However, without the interactive mode, developers can add new commits after rebasing successfully. Hence, it is reasonable that the percentage decreases.

TABLE IV
THE PERCENTS OF FINE-GRAINED CHANGES.

Type	Percent
statement_update:method_invocation	13.18%
additional_functionality:method	11.38%
statement_update:variable_declaration_statement	10.84%
statement_insert:method_invocation	9.54%
statement_delete:method_invocation	7.76%
statement_update:return_statement	7.31%
removed_functionality:method	7.05%
statement_insert:variable_declaration_statement	6.91%
additional_object_state:field	6.14%
statement_delete:variable_declaration_statement	5.69%

In this question, we wonder what change types are prevalent among those non-conflict rebase scenarios in which the rebased branch has been modified. As shown in Table IV, we present the top-10 change types with the percentages of rebases in which the change type appears. Comparing to the statistics in the study on bug fixes [27], we find that changes made during rebasing are obviously different from bug fixes. In other words, we can tell that developers may make changes during rebasing branches to complete other different goals. For example, the change "additional_functionality:method", which intuitively requires a certain number of efforts, accounts for 11.38%. Consequently, it is difficult to automate the process of rebasing branches, as we fail to extract the exact oracles on these changes.

Result 5: Developers tend to introduce other changes when rebasing branches without textual conflicts, as we find that new changes are introduced in 34.2% of those non-conflict "matched" rebases.

V. THREATS TO VALIDITY

The main threats to the validity of our results correspond to the rebases collected. Note that, we collect rebases from pull requests only and other rebases that occur in other development activities still cannot be identified currently. Hence, although we take an important step in studying real-world rebases, our study may fail to provide the comprehensive view of rebases. Besides, projects written in other programming languages, hosted on other platforms, or developed with different methods, may have different requirements and needs when rebasing branches. The thresholds used to select popular projects are arbitrary and may have an impact on our conclusions. As we have collected a number of rebases from popular projects, we consider that our results still reflect the real situations of rebasing branches.

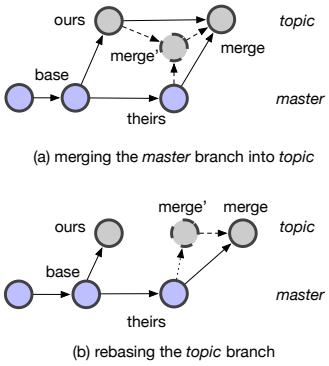

Fig. 8. Other changes are introduced when merging and rebasing branches.

VI. Discussion

A. Conflict Resolution

Similar to existing studies [9] [5] on explicit merges produced by "git merge", our results also indicate that developers tend to make use of existing sources to have the final version during rebasing. Hence, our results suggest that developers can leverage existing techniques on resolving conflicts automatically to deal with the majority of rebases.

However, please note that the processes of resolving conflicts in "git merge" and "git rebase" are different. When developers use "git merge", all conflicts would be listed once textual conflicts arise. However, when developers use "git rebase", Git individually reapplies the commits of the to-do list. Once conflicts arise during the rebase process, developers need to resolve conflicts first, and then Git continues to reapply the remaining commits. There is a chance that these resolutions would conflict with these remaining commits in the to-do list. Hence, it is also necessary to assist developers in resolving conflicts by showing the impacts of committed resolutions on the following commits. For tool-builders and researchers, we suggest that we devote more efforts to examining the impacts of resolutions and providing usable tools to present these impacts to developers, especially when developers have many to-reapply commits.

The study [8] on real-world explicit merges shows that program elements involved in conflicts have more code smells. Our results indicate that similar problems may exist in rebases, as developers resolve conflicts with similar strategies. We suggest that developers pay more attention to code smells produced during the process of resolving rebase conflicts.

B. Validating Rebase

Based on *the intuition that the merge should preserve the changed behaviors introduced in two branches*, Sousa et al. [22] propose the contract of semantic conflict freedom to verify 3-way program merges. As shown in Fig. 8, there are four versions *merge*, *ours*, *theirs* and *base* involved in explicit and implicit merges. Note that, once developers introduce other changes $\Delta(merge', merge)$ (where $merge'$ is created by Git) during merging, the contract may fail to work.

When developers use "git merge" with default settings to perform the merge without conflicts, the merge result will be committed into the repository. At the same time, there is no other intention involved in this merge. Hence, the contract can work well in most of the non-conflict explicit merges. Different from the workflow of "git merge", developers often add more changes during the rebase process, as is shown in the results of RQ5. As a result, although no textual conflicts arise, developers' other intentions (e.g., intentions on $\Delta(merge', merge)$ in the Fig. 8(b)) are introduced into the rebased branch. Consequently, it is not proper to use the notion of semantic conflict freedom to verify this implicit merge.

As shown in Fig. 8, if new introduced changes $\Delta(merge', merge)$ do not interfere with $\Delta(base, ours)$ or $\Delta(base, theirs)$, we can recreate the relevant branch, and then use the contract to investigate whether semantic conflicts arise. For example, if $\Delta(merge', merge)$ do not interfere with $\Delta(base, theirs)$, we can intuitively consider that $\Delta(base, ours)$ and $\Delta(merge', merge)$ represent the same intentions together. And then, after recreating the new version $ours'$ by applying $\Delta(merge', merge)$ on $ours$, we can detect the semantic merge conflicts on four version $(base, ours', theirs, merge)$. Otherwise, if these new changes $\Delta(merge', merge)$ do interfere with $\Delta(base, ours)$ and $\Delta(base, theirs)$, we may tell that $\Delta(merge', merge)$ are introduced to resolve conflicts during merging. Hence, we need to identify all the conflicts in the merge scenario $(base, ours, theirs, merge')$ and determine whether the undesired behavior of $merge'$ has been changed in $merge$.

As introduced above, formally describing and determining the relationship between changes, is important in the process of validating rebases. We suggest that researchers investigate these relationships by mining more information on dependencies, documents, conversations in pull requests, etc.

VII. Conclusion

In this study, we collect a total of 51,183 rebase scenarios from 82 Java repositories hosted on GitHub. Our results show that developers often rebase head branches of pull requests, and rebasing is able to relieve the burden on reviewing changes and keep the evolutionary history clean. We find that textual conflicts arise in 24.3%-26.2% of rebases, and developers resolve conflicts by introducing new tokens only for 28.3%-29.4% of conflict rebases. We find that in 34.2% of non-conflict rebases, developers add new changes during the rebase process. Based on these results, for developers and researchers, we provide actionable implications on conflict resolutions and validating rebases. In future work, we plan to explore whether potential needs can be integrated into tools that assist developers' work on software merging.

Acknowledgment

This work is supported by the National Key R&D Program of China (No. 2017YFB1001802) and the National Natural Science Foundation of China (Nos. 61672529, 61502015, and 61872445).

REFERENCES

[1] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?" in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 322–333.

[2] W. Zou, W. Zhang, X. Xia, R. Holmes, and Z. Chen, "Branch use in practice: A large-scale empirical study of 2,923 projects on github," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, 2019, pp. 306–317.

[3] S. Torres-Arias, A. K. Ammula, R. Curtmola, and J. Cappos, "On omitting commits and committing omissions: Preventing git metadata tampering that (re)introduces software vulnerabilities," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 379–395.

[4] 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*, Sep 2017.

[5] G. G. L. Menezes, L. G. P. Murta, M. O. Barros, and A. Van Der Hoek, "On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[6] T. Zimmermann, "Mining workspace updates in cvs," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, May 2007, pp. 11–11.

[7] P. Accioly, P. Borba, and G. Cavalcanti, "Understanding semi-structured merge conflict characteristics in open-source java projects," *Empirical Software Engineering*, Dec 2017.

[8] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma, "An empirical examination of the relationship between code smells and merge conflicts," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 58–67.

[9] R. Yuzuki, H. Hata, and K. Matsumoto, "How we resolve conflict: an empirical study of method-level conflict resolution," in *Proceedings of IEEE the 1st International Workshop on Software Analytics*, ser. SWAN '15, vol. 00, March 2015, pp. 21–24.

[10] S. Just, K. Herzig, J. Czerwonka, and B. Murphy, "Switching to git: The good, the bad, and the ugly," in *Proceedings of the 27th IEEE International Symposium on Software Reliability Engineering*, ser. ISSRE '16, Oct 2016, pp. 400–411.

[11] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09, May 2009, pp. 1–10.

[12] T. Mens, "A state-of-the-art survey on software merging," *IEEE transactions on software engineering*, vol. 28, no. 5, pp. 449–462, 2002.

[13] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 732–741.

[14] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 168–178.

[15] M. L. Guimarães and A. R. Silva, "Improving early detection of software merge conflicts," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 342–352.

[16] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: Rethinking merge in revision control systems," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 190–200.

[17] N. Niu, F. Yang, J.-R. C. Cheng, and S. Reddivari, "Conflict resolution support for parallel software development," *IET Software*, vol. 7, pp. 1–11(10), February 2013.

[18] Y. Nishimura and K. Maruyama, "Supporting merge conflict resolution by using fine-grained code change history," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 661–664.

[19] F. Zhu and F. He, "Conflict resolution for structured merge via version space algebra," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 166:1–166:25, Oct. 2018.

[20] X. Xing and K. Maruyama, "Automatic software merging using automated program repair," in *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*, Feb 2019, pp. 11–16.

[21] S. Horwitz, J. Prins, and T. Reps, "Integrating noninterfering versions of programs," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 3, pp. 345–387, Jul. 1989.

[22] M. Sousa, I. Dillig, and S. K. Lahiri, "Verified three-way program merge," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 165:1–165:29, Oct. 2018.

[23] T. Ji, L. Chen, X. Mao, X. Yi, and J. Jiang, "Automated regression unit test generation for program merges," *arXiv preprint arXiv:2003.00154*, 2020.

[24] S. McKee, N. Nelson, A. Sarma, and D. Dig, "Software practitioner perspectives on merge conflicts and resolutions," in *Proceedings of IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '17, Sept 2017, pp. 467–478.

[25] H. L. Nguyen and C.-L. Ignat, "An analysis of merge conflicts and resolutions in git-based open source projects," *Computer Supported Cooperative Work (CSCW)*, May 2018.

[26] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, Nov 2007.

[27] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 913–923.