Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts

Mehran Mahmoudi
University of Alberta
Edmonton, AB, Canada
mehran@ualberta.ca

Sarah Nadi
University of Alberta
Edmonton, AB, Canada
nadi@ualberta.ca

Nikolaos Tsantalis
Concordia University
Montreal, QC, Canada
tsantalis@cse.concordia.ca

***Abstract*—With the rise of distributed software development, branching has become a popular approach that facilitates collaboration between software developers. One of the biggest challenges that developers face when using multiple development branches is dealing with merge conflicts. Conflicts occur when inconsistent changes happen to the code. Resolving these conflicts can be a cumbersome task as it requires prior knowledge about the changes in each of the development branches. A type of change that could potentially lead to complex conflicts is code refactoring. Previous studies have proposed techniques for facilitating conflict resolution in the presence of refactorings. However, the magnitude of the impact that refactorings have on merge conflicts has never been empirically evaluated. In this paper, we perform an empirical study on almost 3,000 well-engineered open-source Java software repositories and investigate the relation between merge conflicts and 15 popular refactoring types. Our results show that refactoring operations are involved in 22% of merge conflicts, which is remarkable taking into account that we investigated a relatively small subset of all possible refactoring types. Furthermore, certain refactoring types, such as EXTRACT METHOD, tend to be more problematic for merge conflicts. Our results also suggest that conflicts that involve refactored code are usually more complex, compared to conflicts with no refactoring changes.**

***Index Terms*—refactoring, git, merge conflict, software evolution**

I. INTRODUCTION

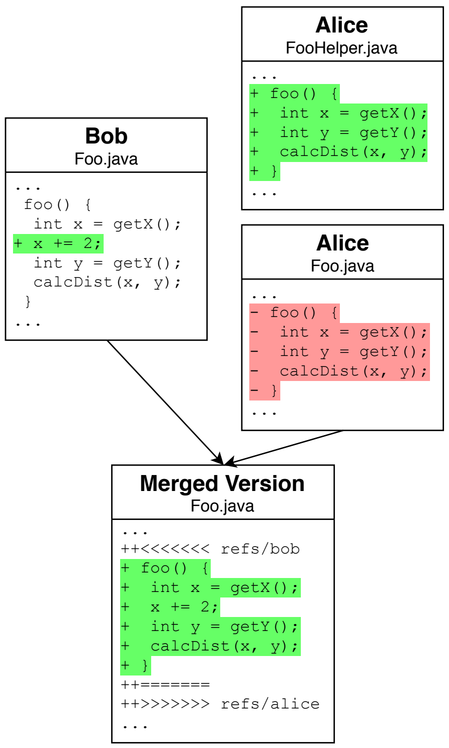
Version control systems (VCSs), which keep track of the software development history, have become an essential component of modern software development. With the increase of distributed software development [1], additional coordination tools and processes to facilitate collaboration between team members who may be working on different tasks have been introduced. For example, large software systems commonly make use of *branching* in distributed version control systems. Developers typically follow a *branch-based development approach*, where new features or bug fixes are developed in separate branches before being integrated into the master branch, or another stable branch [2].

While branching (or *forking* where the developer may make a tracked copy of the work in a separate repository [3]) has several advantages such as allowing better separation of concerns and enabling parallel development [4], it still comes at the cost of integration challenges [5]. Once a developer has completed the intended work in a given branch, they need to merge their changes with the rest of the team's work. At this point, *merge conflicts* may arise, because of inconsistent changes to the code. Previous studies have shown that up to 16% of merge scenarios lead to conflicts [6]. Developers have to resolve such conflicts before proceeding, which wastes their time and distracts them from their main tasks [7].

There are several types of conflicts and various reasons why a conflict can occur [7]. *Textual conflicts* are those that occur when simultaneous changes occur to the same lines in a file, and are the type of conflicts that popular VCSs such as GIT detect. For example, one developer may have added a new variable declaration `foo` at line 10 of a given file, while the other developer has added another variable declaration `bar` at the same line. When GIT tries to merge both changes, it cannot decide which variable declaration should appear at that line.

Another example of why a conflict can occur is shown in Figure 1. Here, Alice moves function `foo()` from `Foo.java` to `FooHelper.java`, while Bob adds the line `x += 2;` to `foo`'s implementation in its original place in `Foo.java`. The figure shows the resulting conflict in `Foo.java`, when Bob tries to merge his code with Alice. As shown, the resulting conflict in `Foo.java` shows that the whole function is deleted in one branch, but modified in the other; the number of conflicting lines reported is also large (the size of the whole function `foo`). Given that Bob is not aware that he needs to look at `FooHelper.java` to understand what happened, he would mistakenly think that this is a complex conflict that would take him lots of time to understand and resolve. In reality, the conflict is actually simple: Alice moved the function (a refactoring operation) while Bob added an extra piece of code to it. A simple resolution would be to add the extra piece of code to the new location of the function.

The above example demonstrates how refactorings may complicate the merging process. There have been a few studies that investigated how to deal with refactorings during merging. For example, Dig et al. [8] previously argued that since refactorings cut across module boundaries and affect many parts of the system, they make it harder for VCSs to merge the changed code. They proposed *refactoring-aware merging*, with the argument that if a merging tool understands the refactorings that took place, it may be able to automatically resolve the conflict and save the developer's time. In the



```
...
+ foo() {
+   int x = getX();
+   int y = getY();
+   calcDist(x, y);
+ }
...
```

```
...
 foo() {
   int x = getX();
+ x += 2;
   int y = getY();
   calcDist(x, y);
 }
...
```

```
...
- foo() {
-   int x = getX();
-   int y = getY();
-   calcDist(x, y);
- }
...
```

```
...
++<<<<<<< refs/bob
+ foo() {
+   int x = getX();
+   x += 2;
+   int y = getY();
+   calcDist(x, y);
+ }
++=======
++>>>>>>> refs/alice
...
```

Fig. 1. A sample merge conflict caused by a refactoring operation

example above, their proposed approach would "unapply" the refactoring (i.e., keep `foo` in its old place), apply the new changes to it (i.e., add the new code), and then as a last step, re-apply the refactoring. In a previous study [9], we proposed similar strategies for various types of refactorings we found while studying the simultaneous evolution of several versions of the Android operating system, but we did not develop corresponding tools. Other researchers focused on improving code matching and resolution precision in software merging by considering specific types of refactorings, such as renamings [10], [11].

While the above studies propose techniques to deal with refactorings during merging, there have not been any large-scale empirical studies investigating the relationship between refactorings and merge conflicts in the first place. Researchers agree that refactorings may potentially complicate a merge scenario. However, how often does this occur in practice? Do refactorings actually result in more complex conflicts? For example, Dig et al.'s refactoring-aware merging [8] was never evaluated on a large scale. Their technique cannot handle common refactorings such as EXTRACT METHOD. This is because contrary to refactorings such as RENAME METHOD, EXTRACT METHOD refactorings touch method bodies as well as signatures, hence making it difficult to "unapply" them. If, for example, EXTRACT METHOD refactorings are often involved in merge conflicts, then more effort should be invested into improving and extending such refactoring-aware tools.

Understanding the relationship between refactoring and merge conflicts on a large-scale is important to drive researchers' efforts in the right direction. This paper presents the first large-scale empirical study on the relationship between refactorings and merge conflicts. As opposed to related work that looked at a small number of repositories [8], [9] or at a couple of refactoring operations [10], [11], our study analyzes close to 3,000 GitHub repositories and uses the state-of-the-art refactoring detection tool, *RefactoringMiner* [12], which is able to precisely detect 15 types of refactorings. In order to understand the relationship between refactoring and merge conflicts, we break down our investigation into the following research questions.

RQ1 *How often do merge conflicts involve refactored code?* Understanding the extent of the impact that refactorings have on merge conflicts would determine the practicality of tools and techniques that assist developers in resolving conflicts that involve refactorings.

RQ2 *Are conflicts that involve refactoring more difficult to resolve?* In order to understand how problematic refactorings are, knowing how often they are involved in conflicts is not enough per se. A comparison between conflicts that involve refactorings and those that do not will help us better understand differences in complexity.

RQ3 *What types of refactoring are more commonly involved in conflicts?* Refactoring-aware merging techniques that rely on "unapplying" refactoring operations would be rendered inefficient if refactoring types that cannot be easily "unapplied", such as all the extract operations, happen to be involved in conflicts frequently.

We find that 22% of merge conflicts involve refactoring, which is remarkable taking into account that we investigated only 15 refactoring types while refactoring books describe more than 70 different types [13]. This shows that refactoring changes end up being involved in a considerable portion of merge conflicts, and suggests useful potential for refactoring-aware merging techniques.

Furthermore, we find that conflicts that involve refactorings are more complex than conflicts with no refactoring. This reaffirms the necessity of tools and techniques that can assist developers in the merging process in the presence of refactorings.

Our results also show that when adjusted for their overall frequency, refactoring types affect conflicts at different rates. EXTRACT METHOD refactorings are involved in more conflicts than their typical overall frequency, while the majority of refactoring types are involved in conflicts less frequently. This is bad news for current refactoring-aware merging techniques and calls for more sophisticated approaches.

In summary, the contributions of this paper are as follows:

- An extensive empirical study on almost 3,000 open-source Java repositories to investigate the role of refactoring operations in merge conflicts.
- A methodology for detecting refactorings in evolutionary changes that lead to merge conflicts.
- Open-source implementation of our methodology, to facilitate verification and replication efforts.

II. Background

In this section, we describe how merging works in Git, as well as the related terms we use throughout the paper. We also provide related background about refactoring and the refactoring detection tool we use in this work, *RefactoringMiner* [12].

A. Software Merging

Using multiple branches in a source control system is a common practice in software development that serves a variety of purposes [4], [14]. At one point, developers need to integrate the changes from the different branches, and this is done by merging the corresponding branches. We call such situation a *merge scenario*.

1) Merging in Git: Almost all merge tools that are currently available, including the one utilized by Git, employ three-way merging techniques [7]. In three-way merging, two versions of a software artifact are merged by making use of an additional third version, which is often called the *base version*.

Figure 2 illustrates a typical merge scenario. When merging two branches, Git attempts to merge the most recent commit in each branch. We call these commits *merge parents*. As a base version in a merge scenario, Git uses the most recent commit that both merge parents can be derived from, referred to as the *common ancestor*. The result of the merge is stored in a *merge commit*. A merge commit can be identified from the Git history, since it has two or more parents (namely *P1* and *P2* in the example of Figure 2), unlike typical commits that have one parent.

2) Merge Conflict: Based on the nature of the merge scenario, a textual three-way merge tool, such as the one used by Git, might not be able to automatically merge the two versions of a file. For a given conflicting merge scenario, Git can report conflicts across multiple files. Git categorizes conflicts into 6 types:

- `add/add`: When both merge parents add a new file with same name, but with different contents.
- `content`: When both parents apply different changes to the same file, in the same location.
- `modify/delete`: When *P1* modifies a file, while *P2* deletes it.
- `rename/add`: When *P1* renames a file, and *P2* adds a new file with the same name.
- `rename/delete`: When *P1* renames a file, and *P2* deletes it.
- `rename/rename`: When both parents rename a file to different names.

The first two types are at the content level, while the other four are at the file level. Content level conflicts could be caused by more than one location in the conflicting file. Git reports these conflicting locations by annotating them with <<<, ===, and >>> markers as shown in Figure 1. We call each of these annotated locations a *conflicting region*.

B. Refactoring

Fowler et al. [13] define *refactoring* as "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*". Refactoring is used to enhance the software with regards to reusability, modularity, extensibility, maintainability, etc. [15]. It is also utilized in software reengineering [16], which involves the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

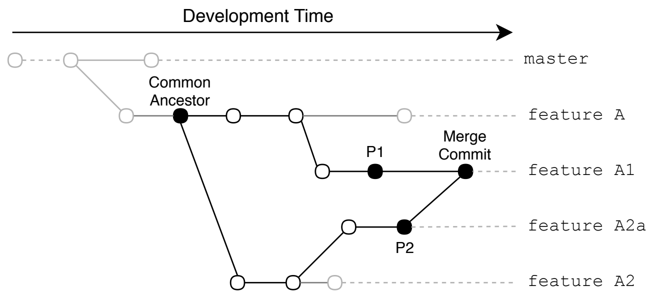


Fig. 2. An overview of a merge scenario. Each labeled line represents a branch, and the black-dotted commits constitute the merge scenario

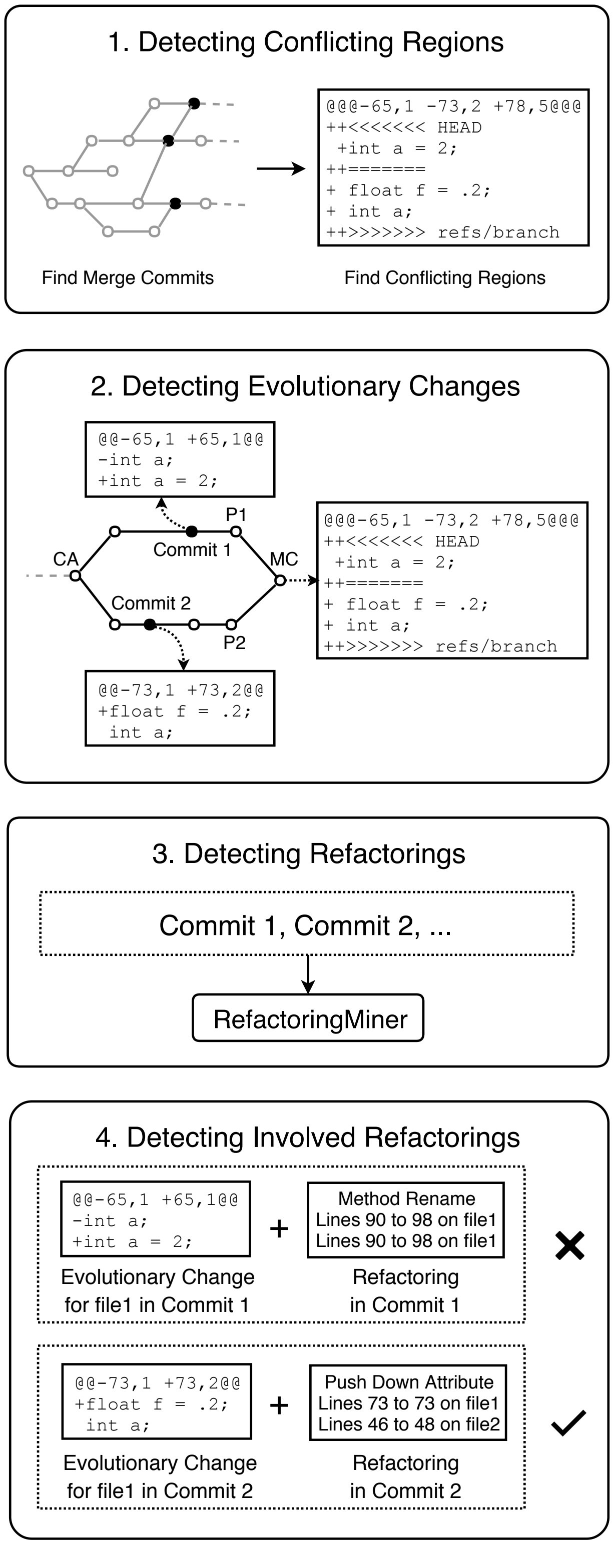
Tsantalis et al. [12] recently proposed a refactoring detection tool, named *RefactoringMiner*, which we use in this work. It detects a variety of refactoring types at multiple granularity levels: Package, Type, Method, and Field. It has two modes of operation: (1) comparing two given files or folders to detect refactorings in them or (2) detecting refactorings in a given Git commit. We use the latter mode in our work. At the time of running our study, *RefactoringMiner* could detect 15 refactoring types.

III. Methodology

As stated in Section I, our goal is to determine whether refactoring changes are involved in conflicts that occur in Java files. Investigating the relationship between refactorings and conflicts on the commit level or file level may be misleading, since the presence of a refactoring may not be related to the resulting conflict in that commit or file. Therefore, in our work, we investigate the relationship between refactorings and merge conflicts on the *conflicting region* level, because it provides more accurate results than other coarse-grained analysis approaches. The remainder of this section explains this approach in detail.

A. Overview

Figure 3 shows an overview of the steps we follow in our methodology for analyzing a given repository. After identifying merge scenarios with conflicting Java files, we detect all conflicting regions for each scenario. Since we are looking for refactorings that were involved in a conflicting region, we first find all commits after the common ancestor that touched that region, for each merge parent. Next, we use *RefactoringMiner* to detect all refactorings that happened in those commits. Using the location information reported by *RefactoringMiner* for refactoring operations and by Git for conflicting regions, we then determine whether a given refactoring was involved in the historical evolution of the conflicting region. All the information we gather in this process is stored in a MySQL database. Our approach is implemented as a Java tool, which is publicly available [17].



```
@@@-65,1 -73,2 +78,5@@@
++<<<<<<< HEAD
 +int a = 2;
++=======
+ float f = .2;
+ int a;
++>>>>>>> refs/branch
```

```
@@-65,1 +65,1@@
-int a;
+int a = 2;
```

```
@@-73,1 +73,2@@
+float f = .2;
 int a;
```

Fig. 3. An overview of our methodology

B. Step 1: Detecting Conflicting Regions

We identify all merge scenarios by finding merge commits in the Git history. Merge commits are commits that have multiple parents. In this work, we focus on merge commits that have only two parents, and record them in our database. We then *replay* the merge scenario as follows. We first detect the merge parents for each merge commit from the Git history. We then checkout *P1*, and use the `git merge` command (with default parameters) to merge *P2* into it.

```
git checkout P1
git merge P2
```

By doing so, we can learn (i) whether a given merge scenario is conflicting, as well as (ii) the conflict details, in case it is a conflicting merge scenario. This step is essential because the Git history does not contain such information.

If a merge scenario is conflicting, the `git merge` command will report the list of conflicting files, as well as their conflict type (See Section II-A2). Using this list, we record the conflicting Java files and their conflict type to the database, if any. For Java files with `content` conflict type, we detect all conflicting regions by using the `git diff` command. When in a conflicting state, this command will report all conflicting regions, along with the corresponding location of each region in both merge parents. Because this command reports a few lines before and after the conflicting region, we use the `-U0` parameter to remove these extra lines. We record this information in the database.

```
git diff -U0
```

Step 1 in Figure 3 shows an example of the conflicting region produced by running the above `git diff` command. The three pairs of numbers between the `@@@` symbols denote the conflicting region. The first pair of numbers corresponds to the region in *P1*, while the second pair corresponds to *P2*. The third pair of numbers is the conflicting region in the conflicting merged file with the markers. In each pair, the first number is the line number where the region begins and the number after comma is the length of that region. Because we are interested in the location of the conflicting region in each merge parent, we only record the first two pairs of numbers for each conflicting region.

C. Step 2: Detecting Evolutionary Changes

In the next step of our methodology, we track the historical evolution of a given conflicting region between the common ancestor and each merge parent. Using this information, we can determine if a refactoring was involved in any of these evolutionary changes which later led to a conflict. We use the `git log` command to perform this task. `git log` is a useful and versatile command thanks to the different parameters it accepts[1]. Using the `-L` parameter along with a revision range, it will report all commits in the revision range that have touched the given file in the specified location. For a given conflicting region, we run `git log` once for each

[1]https://git-scm.com/docs/git-log

TABLE I
STORED CODE RANGES FOR EACH REFACTORING TYPE

Code Element	Refactoring Type	Stored Code Range Corresponding to	
		Old Commit	**New Commit**
Package	CHANGE PACKAGE	type declarations in old package	type declarations in new package
Type	EXTRACT SUPERCLASS/INTERFACE	source type declaration(s)	extracted type declaration
	MOVE CLASS, RENAME CLASS	refactored type declaration	refactored type declaration
Method	EXTRACT METHOD, EXTRACT & MOVE METHOD	source method declaration	source and extracted method declarations
	INLINE METHOD	target and inlined method declarations	target method declaration
	PULL UP METHOD, PUSH DOWN METHOD, RENAME METHOD, MOVE METHOD	refactored method declaration	refactored method declaration
Field	PULL UP FIELD, PUSH DOWN FIELD, MOVE FIELD	refactored field declaration	refactored field declaration

merge parent, with the `-L` parameter set to the corresponding location of the conflicting region in that parent (extracted in Step 1), and the revision range set between that merge parent and the common ancestor. For example, the revision range *P2..P1* includes all commits that are reachable from *P1* and not reachable from *P2*, which is equivalent to the commits between *P1* and the common ancestor of *P1* and *P2*.

```
git log -L start_P1,end_P1:file P2..P1
git log -L start_P2,end_P2:file P1..P2
```

This command outputs all commits that have touched the specified location as well as the corresponding location information for each commit. In our example in Figure 3 (Step 2), running the above commands returns the black-dotted commits. We call these commits *evolutionary commits* since they are involved in the evolution of the conflicting region. The rectangles connected to these commits contain the reported location information. The two number pairs between the `@@` symbols correspond to the location of the conflicting region before and after that commit, respectively. For the top commit, for example, the conflicting region can be found at line number 65 before and after this commit. We save this information in the database.

D. Step 3: Detecting Refactorings

In this step, we use *RefactoringMiner*[2] to detect the refactoring operations taking place in the commits that were involved in the evolution of conflicting regions (i.e., in the evolutionary commits identified in Step 2). In addition to the refactoring type, *RefactoringMiner* reports the files and the exact code ranges (with line numbers) that were touched by a refactoring operation. We store at least two code ranges for a refactoring change: one code range corresponds to the refactored code element before refactoring, and the other corresponds to the element after refactoring. Table I provides a summary of the code ranges we store in the database for different refactoring types.

E. Step 4: Detecting Involved Refactorings

In the final step of our methodology, we identify the refactorings that have affected the evolution of conflicting regions. In other words, we are trying to determine if an evolutionary change that later lead to a conflict contains a refactoring operation. Using the code range information that we have for both refactorings (Step 3) and evolutionary changes to conflicting regions (Step 2), we determine if there is an overlap between them. We consider a refactoring and evolutionary change as *overlapping* if they have at least one line in common, either in their old-commit code ranges or in their new-commit code ranges. We call such refactorings that have overlapping code ranges with an evolutionary change *involved refactorings*, since they are involved in the changes that are related to the conflicting region. In the example of Figure 3, Step 4 shows that the refactoring in commit #1 would *not* be considered as an involved refactoring, while the refactoring in commit #2 would be considered so.

IV. EVALUATION SETUP

Repository Selection: The first step of our evaluation setup is to determine the set of GIT repositories we will run our analysis on. GitHub is a source-code hosting service that contains over 85 million software repositories.[3] Many software engineering researchers use GitHub to obtain a set of repositories and analyze them for their studies [18]–[24]. However, considering the public nature of GitHub, including random repositories without employing a filtering process could lead to misleading findings. For example, many students use GitHub to upload the source code of their course works and programming assignments. Ideally, we want the conclusions of our study to be indicative of how refactorings affect merge conflicts in realistic development setups.

Munaiah et al. [25] studied GitHub repositories and proposed two classifiers (Score-based and Random Forest) that determine whether a given repository is a well-engineered software project. Based on their results, the Random Forest classifier has a higher precision rate. Using their dataset of 1,857,423 repositories, we filter out projects that are not labeled as well-engineered by the Random Forest classifier. Additionally, given the focus of our work, we only consider repositories that are implemented in Java. In our final filtering step, to further ensure the quality of the repositories we pick,

[2]https://github.com/tsantalis/RefactoringMiner, used as of commit `46c80ad`

[3]https://github.com/features

we only include repositories with 100 or more stars on GitHub. This leaves us with a dataset of 2,955 repositories, which we use for our study. However, we find that 30 repositories from this list are no longer accessible with the provided GitHub URL, and so we exclude them from the analysis. Thus, we run our methodology on the final list of 2,925 repositories.

RefactoringMiner settings: When using *RefactoringMiner*, we find that some commits may take longer to process, sometimes leaving the process to hang. Given the scale of our study, we need to ensure that the analysis terminates in a timely manner. Accordingly, we enforce a timeout of 4 minutes on *RefactoringMiner*. If *RefactoringMiner* does not terminate within 4 minutes on a given commit, we terminate the process and skip this commit.

Running environment: For running the analysis, we used 12 threads on a machine with 16 CPU cores at 3.4GHz, 128 gigabytes of memory, a solid-state storage device, and a 1 Gbps Internet connection. Each thread runs the entire process for a repository. Analyzing all repositories took a total of 27 hours.

V. Results

In this section, we report the results of running our methodology from Section III on the 2,925 repositories described in Section IV. When running the analysis, we find that one repository (*platform_frameworks_base*[4]) took a much longer time to process, due to its unusually high number of merge scenarios: this repository has 281,251 merge scenarios, while all other projects in our dataset have 729,060 combined. We decided to skip this repository in our analysis because this uncommon irregularity might skew our results. Thus, all the results provided in this section are based on the analysis of the remaining 2,924 repositories. We first report some descriptive statistics of the data collected, and then proceed to answer each of the research questions we presented in the introduction.

A. Descriptive Statistics of Collected Data

Table II provides a summary of the collected data after running our methodology on our set of 2,924 repositories, including the mean and standard deviation for each metric. We find that 2,606 of the repositories we analyzed contained merge scenarios (a total of 729,060 merge scenarios), out of which 1,753 repositories had at least one conflicting merge scenario (a total of 63,826 conflicting merge scenarios). Out of those, 1,424 repositories had at least one conflicting merge scenario (CMS) that included a conflicting Java file (a total of 36,988 CMSs with conflicting Java files). Since not all conflict types can have conflicting regions, a fewer number of repositories, 1,403, have conflicting regions (a total of 258,956 conflicting regions). Furthermore, 1,396 of those repositories have historical evolutionary changes for their conflicting regions, with these changes occurring in a total of 657,726 commits. We explain the discrepancy between the number of repositories with conflicting regions and repositories with evolutionary

TABLE II
Statistics for merge scenarios, merge conflicts, and refactorings

	Total	# of Corresponding Repositories	Per Repository	
			Mean	SD
Merge Scenario	729,060	2,606	279.76	1,690.83
Conflicting Merge Scenario (CMS)	63,826	1,753	36.40	144.36
CMS with Java Conflicts	36,988	1,424	25.97	91.82
Conflicting Region	258,956	1,403	184.57	767.38
Evolutionary Commit	657,726	1,396	471.15	2,073.24
Refactoring in Evolutionary Commits	248,652	1,136	218.88	783.61

commits in Section VI-A. The results we present in the rest of this section are thus based on the 1,396 repositories for which we were able to extract historical evolutionary changes for their conflicting regions. Note that the last row in Table II shows the number of refactoring operations detected by *RefactoringMiner* in evolutionary commits.

As an additional data point not shown in the table, the 36,988 conflicting merge scenarios with Java conflicts we collected contain 157,422 conflicting Java files. The conflicts in these Java files can belong to any of the conflict types discussed in Section II-A2. Out of these, 99,846 files (i.e., 63%) are `content` conflicts. This suggests that `content` conflicts, which is the focus of our work, represent the majority of conflicts that developers face in practice.

For measuring the effect size, we use $r = Z/sqrt(N)$ where Z is the test statistic and N is the number of samples [26]. We interpret r as small ($\geq$ 0.1), medium ($\geq$ 0.3), and large ($\geq$ 0.5) [26].

B. RQ1. How often do merge conflicts involve refactored code?

1) Data used for RQ: We answer this question by checking whether a code change that led to a conflict involved a refactoring change. As explained in Section III, we use the term *involved refactoring* to describe a refactoring that happened in an evolutionary change and which overlaps with the conflicting region. A conflicting merge scenario can have multiple conflicting regions. If at least one of the conflicting regions in a conflicting merge scenario contains involved refactorings, we consider that merge scenario as one that contains involved refactorings.

2) Findings: We find that there are 8,155 conflicting merge scenarios that contain involved refactorings. We know from Table II that there is a total of 36,988 merge scenarios with conflicting Java files, which means that 22% of these merge scenarios involve refactorings. On the conflicting region level, we find that 28,670 (i.e., 11%) of the 258,956 conflicting regions from Table II have involved refactorings.

Answer to RQ1: 22% of merge scenarios with at least one conflicting Java file involve refactorings. More precisely, 11% of conflicting regions have at least one involved refactoring.

[4]https://github.com/android/platform_frameworks_base

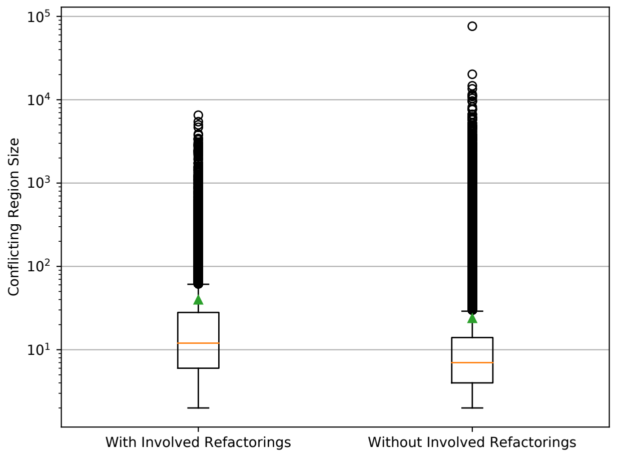


Fig. 4. Distribution of conflicting region size with and without involved refactorings

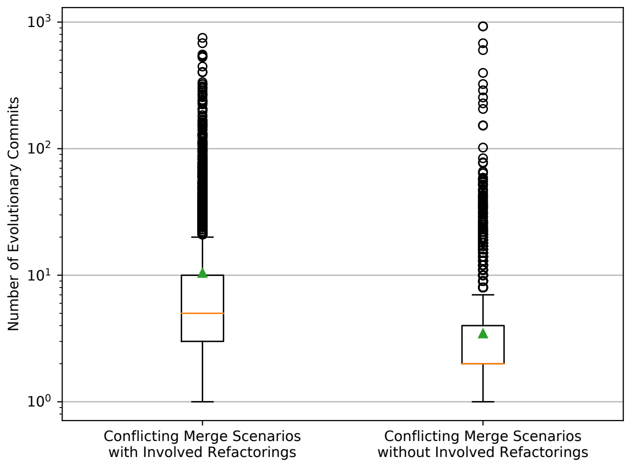


Fig. 5. Number of evolutionary commits per merge scenario with and without involved refactorings

3) Implications: Since there are no previous studies that investigated the extent of refactorings in merge conflicts, or other types of semantic code changes in merge conflicts, we have no point of reference to compare our findings to. However, previous work by McKee et al. [27] showed that when resolving merge conflicts, practitioners' top needs include: (a) understanding the code involved in the merge conflict, and (b) tools to help them explore the project history during the process of resolving conflicts. While 22% might not seem like a high number, and we cannot conclude that refactorings are involved in the majority of merge conflicts, our findings provide good news for addressing practitioners' requests from McKee et al.'s study. Since refactoring detection in commit history has now become precise and scalable, this means that, based on our results, researchers can provide developers with tools to explore the history and interpret changes in a little less than a quarter of conflicting merge scenarios, thus helping them to resolve conflicts faster. On the level of a given conflicting scenario, such tool support can be provided for approximately 11% of the conflicting regions. It should be emphasized that in our study, we considered only a subset of all possible refactoring types, because *RefactoringMiner* supports only 15 out of the 72 refactoring types described in Fowler's catalog [13]. We conjecture that the aforementioned percentages would be potentially even larger if more refactoring types were considered.

C. RQ2. Are conflicts that involve refactoring more difficult to resolve?

1) Data used in RQ: Previous work on software merging used the number of conflicting lines as a measure of the complexity of a conflict [28], [29]. Recent work also confirms that the number of conflicting lines is one of the top factors that affects the developers' perception of the difficulty of a conflict [27]. Based on the above previous work, we use the number of conflicting lines, in other words the size of conflicting region, as a proxy for describing the difficulty of a merge conflict. As an additional measure of difficulty, we also look at how refactorings can affect the number of evolutionary commits for conflicting merge scenarios. Since program comprehension is a traditionally complex and time consuming task [30], we assume that the more evolutionary changes a merge scenario has, the more complex resolving a merge conflict will be, since more changes need to be understood.

2) Findings: Figure 4 shows the size of all conflicting regions in our study. The left box plot contains conflicting regions with involved refactorings, while the right box plot contains the remaining conflicting regions. The orange lines mark the median and the green triangles show the mean. The figure shows that there are a few conflicting regions without refactorings that are larger than any conflicting region with refactoring (those with >10,000 lines). However, this does not hold on average. The mean and median for conflicting regions with involved refactorings are 39.63 and 12 respectively. The same values are lower for conflicting regions without involved refactorings, with a mean and median of 26.63 and 7, respectively. The unpaired Wilcoxon rank-sum test shows that these distributions are statistically different (*p-value* = 0.00). Additionally, a 95% confidence interval for the difference between the two population medians is between 3 and infinity, suggesting that the median of the size of conflicting regions with involved refactorings is at least 3 lines larger than the median size of the remaining conflicting regions. The effect size is 0.14 which can be interpreted as small.

Figure 5 shows the distribution of the number of evolutionary commits for each conflicting merge scenario. The left box plot contains conflicting merge scenarios with at least one involved refactoring change in their evolutionary commits. The right box plot contains the remaining conflicting merge scenarios. Similar to Figure 4, the orange line shows the median and the green triangle marks the mean. As Figure 5 suggests, conflicting merge scenarios with involved refactorings have a larger number of evolutionary commits (median:

5%, mean: 10.43%) compared to conflicting merge scenarios with no involved refactoring changes (median: 2%, mean: 3.46%). Furthermore, the Wilcoxon rank-sum test shows that this difference is significant (*p-value* = 0.00). The effect size is 0.34 which can be interpreted as medium.

Answer to RQ2: Conflicting regions that involve refactorings tend to be larger (i.e., more complex) than those without refactorings. Furthermore, conflicting merge scenarios with involved refactorings include more evolutionary changes (i.e., changes leading to conflict) than conflicting merge scenarios without involved refactorings.

3) Implications: Our findings show that conflicting regions that involve refactoring operations are indeed more complex than conflicting regions without involved refactorings. While 3 extra conflicting lines may not seem like a big difference, recall that the median size of a conflicting region is already small (7 lines), so 3 lines represents an almost 50% increase. Additionally, our results suggest that resolving merge conflicts with involved refactorings may be more difficult, since they typically involve more evolutionary changes for the developer doing the resolution to understand. Our findings provide great motivation for refactoring-aware merging tools and techniques that can help developers in the merging process.

D. RQ3. What types of refactoring are more commonly involved in conflicts?

1) Data Used for RQ: We consider 15 types of refactorings in our work. Not all of them necessarily occur with the same rate, and each refactoring type might impact conflicting regions differently. Understanding how often each refactoring type affects merge conflicts is important for any future tool support, especially for refactoring-aware merging tools and techniques [8].

When looking for differences in the distribution of each refactoring type among all involved refactorings, it is important to take into account the "typical" distribution of refactorings types as well, i.e., how often each refactoring type occurs in general. This way, we can observe if there are any discrepancies between the distribution of a given refactoring type among involved refactorings vs. in general. Specifically, it would be interesting to find the refactoring types that appear more often as involved refactorings when compared to their general distribution. Such cases indicate particularly problematic refactorings that are involved in merge conflicts. However, in our methodology, we do not collect information about *all* refactorings that happen in each repository. As described in Section III-D, we detect refactoring operations only in evolutionary commits. Since not all detected refactoring operations are involved refactorings, we use the distribution of all detected refactorings in all evolutionary commits as a proxy for the general distribution of refactorings in our data.

2) Findings: Figure 6 shows how common each refactoring type is across all projects. It provides two different distributions for each refactoring type: involved refactorings and overall refactorings. Every project has two data points in each violin plot, representing a refactoring type. The y-axis is the percentage of refactorings corresponding to the refactoring type. The width of each plot at a given percentage shows the number of projects with that percentage. For example, suppose a project has a total of 5 refactorings in its evolutionary commits, two RENAME METHOD refactorings and three MOVE CLASS refactorings. Also, assume that the two RENAME METHOD refactorings are involved in conflicting regions. This project will be represented by 30 data points in the figure, two points for each refactoring type. For the violin plots corresponding to involved refactorings, all of the points for this project will have a value of zero, except the point corresponding to RENAME METHOD which will be 100%. For the violin plots corresponding to overall refactorings, the points for RENAME METHOD and MOVE CLASS will have a value of 40% and 60%, respectively, and the points for the remaining refactoring types will be zero.

TABLE III
WILCOXON SIGNED-RANK PAIRED TEST RESULTS BETWEEN OVERALL AND INVOLVED REFACTORINGS. WHEN INVOLVED REFACTORINGS ARE MORE THAN OVERALL REFACTORINGS, THE DIRECTION OF DIFFERENCE IS $\uparrow$, AND $\downarrow$ OTHERWISE. RESULTS WITH $p < 0.05$ ARE SHOWN IN BOLD, WITH HIGHLIGHTED ROWS BEING SPECIFICALLY OF INTEREST.

Refactoring Type	**Direction of Difference**	**p-value**	**Effect Size** ($r = Z/sqrt(N)$)
CHANGE PACKAGE	$\uparrow$	0.093	0.035
EXTRACT & MOVE METHOD	$\downarrow$	0.187	0.028
EXTRACT INTERFACE	$\uparrow$	**0.000**	**0.081**
EXTRACT METHOD	$\uparrow$	**0.000**	**0.119**
EXTRACT SUPERCLASS	$\uparrow$	**0.017**	**0.054**
INLINE METHOD	$\downarrow$	0.304	0.022
MOVE & RENAME CLASS	$\downarrow$	**0.000**	**0.209**
MOVE ATTRIBUTE	$\downarrow$	**0.000**	**0.210**
MOVE CLASS	$\downarrow$	**0.000**	**0.324**
MOVE METHOD	$\downarrow$	**0.000**	**0.182**
PULL UP ATTRIBUTE	$\downarrow$	**0.000**	**0.141**
PULL UP METHOD	$\downarrow$	**0.000**	**0.131**
PUSH DOWN METHOD	$\downarrow$	**0.000**	**0.086**
RENAME CLASS	$\downarrow$	**0.000**	**0.198**
RENAME METHOD	$\downarrow$	**0.000**	**0.233**

We use a two-sided paired Wilcoxon signed-rank test to compare the distributions of overall refactorings and involved refactorings, for each refactoring type. We use a Benjamini & Hochberg (BH) p-value adjustment measure to account for multiple comparisons, and use $\alpha = 0.05$. To find the direction of the difference, we compare the means and interquartile ranges of the distributions. We show the results in Table III. The third column shows that 12 refactoring types have *p-values* lower than 0.05 (highlighted in bold), meaning that involved and overall refactorings in these types have a different distribution.

Out of these 12 types, involved refactorings have higher percentages for EXTRACT INTERFACE, EXTRACT METHOD, and EXTRACT SUPERCLASS. However, the effect size is negligible for EXTRACT SUPERCLASS and EXTRACT INTERFACE and is small for EXTRACT METHOD.

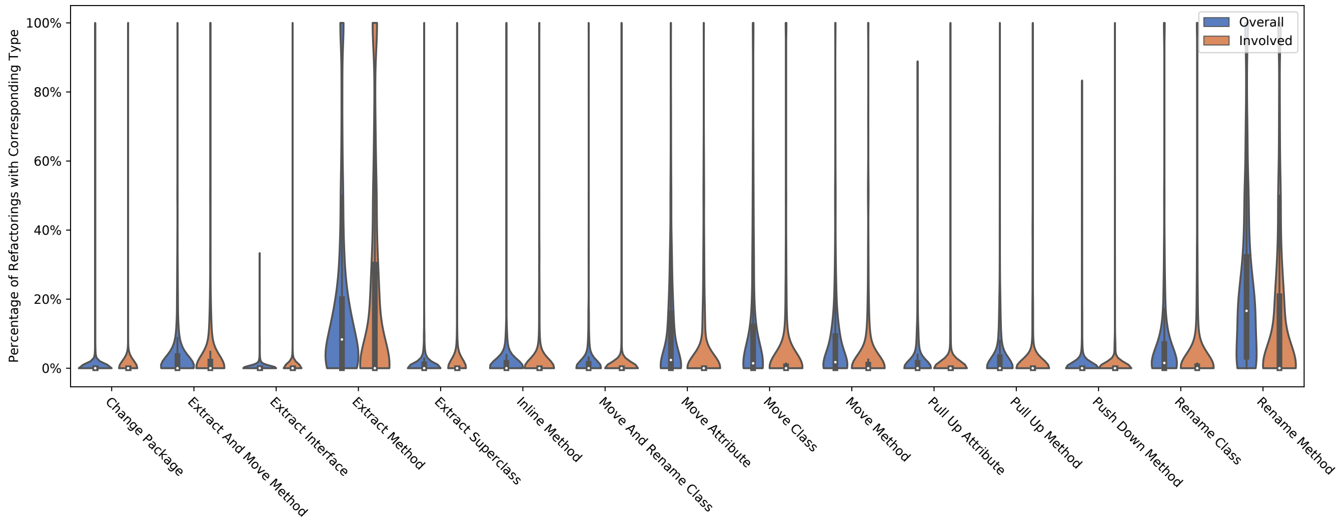


Fig. 6. Percentage of each involved refactoring's type per project

Answer to RQ3: EXTRACT METHOD is more involved in conflicts than its typical overall frequency, with a small effect size. EXTRACT INTERFACE and EXTRACT SUPERCLASS are also more involved in conflicts, but with negligible effect sizes.

3) Implications: Our results suggest bad news for existing refactoring-aware merging tools. As mentioned in the introduction, Dig et al.'s undo/redo technique [8] cannot handle extract refactorings. This calls for more sophisticated refactoring-aware merging tools that can handle such cases.

With the exception of MOVE CLASS, the effect size of the remaining refactorings in the other direction is small. The larger effect size for MOVE CLASS might have to do with the fact that GIT detects file move and rename operations. Since a Java public class is represented in a single file, the MOVE CLASS refactoring is essentially a file move operation for public Java classes. Now, if a class is moved in one branch, while its content was edited in another branch, GIT can automatically merge the content change since it is aware of the move.

VI. THREATS TO VALIDITY

A. Construct Validity

In Table II, the number of repositories with evolutionary commits is less than the number of repositories with conflicting regions. Upon further investigation, we found that about 4% (11,312 out of 258,956) of conflicting regions do not have corresponding evolutionary commits. After manual sampling, we found that this occurs for merge scenarios that contain nested merge scenarios within their evolutionary changes. Since we do not go back and examine the previous histories of these nested merge commits, it means that for some conflicting regions, we miss the changes that may have caused the conflict, if those changes were caused by a merge commit. However, since we analyze every merge scenario in a repository, the inner merge scenarios will be individually analyzed and we will collect their evolutionary commits and corresponding refactorings. Any missed involved refactorings, because of these nested merge cases simply means that our reported stats are a lower bound of the actual involvement of refactorings in merge conflicts.

When looking for merge commits, we only consider merge commits with two parents, while in reality a merge commit can have more than two parents. However, merge commits with more than two parents happen rarely in practice.

In our methodology, we looked for refactoring operations that were involved in evolutionary commits that led to conflicts. However, it is not easy to determine whether an involved refactoring indeed caused a conflict, or even more so, whether it was the sole cause of that conflict. This is because refactorings are usually interspersed with other type of changes [31]. As a result, determining whether a conflict was caused by a refactoring or other changes that were tangled with the refactoring is a difficult task. This is why we conservatively say that these refactorings were *involved* in conflicts and refrain from claiming that they directly caused the conflicts.

In Section V-C, we consider the number of conflicting lines as an indicator of resolution difficulty in a merge conflict. While previous research suggests that practitioners do perceive this as a difficulty [27], it is possible for some conflicts to not follow this trend.

B. Internal Validity

Any inaccuracies in our tooling may lead to wrong results. To mitigate that, we use the state-of-the-art refactoring tool, *RefactoringMiner*, which has high precision. Additionally, we manually reviewed samples of our results throughout our experiments. In terms of our methodology for calculating involved refactorings, we manually validated samples of our results and also publish our tooling and findings in our online artifact page [17] to facilitate reproducability.

As mentioned in Section IV, when using *RefactoringMiner* to detect refactorings in a given commit, we employ a 4-minute timeout. We keep a record of every time *RefactoringMiner* takes more than 4 minutes and the process is terminated. Out of 115,911 commits that we analyzed with *RefactoringMiner*, only 949 of them (0.81%) reached this timeout. This is a very small percentage and does not pose a serious threat to the validity of our results.

C. External Validity

Our study focuses only on Java projects. By limiting our subject systems to well-engineered software projects, we made our results more indicative of realistic development setups. However, since we were limited to open-source software systems and we did not have access to closed-source enterprise Java projects, we cannot claim that our findings can be generalized to all Java software systems. Nonetheless, the large number of subject systems we use (almost 3,000) suggest that our findings are common in open-source projects.

Finally, *RefactoringMiner* is able to detect only 15 refactoring types out of the 72 refactoring types described in Fowler's catalog [13]. However, the investigated refactoring types are among the most frequently applied types [32]. We conjecture that a study considering a larger set of refactoring types would possibly find an even stronger involvement of refactoring operations in merge conflicts.

VII. Related Work

A. Software Merging

While branch-based development is a common practice in software engineering, integration challenges for merging separate branches remain a drawback. There are multiple studies that propose new merging techniques to reduce the manual integration labor as well as to decrease the likelihood of merge conflicts. According to the seminal survey by Mens [7], software merging tools can be categorized by how they represent software artifacts.

Text-based merge tools are language-independent and consider software artifacts as text-files [33], [34]. Because of their line-based approach, these tools cannot handle simultaneous changes to the same lines. The conflicts we study in this paper, as reported by GIT, are based on text-based merge tools.

Syntactic merge tools are more advanced since they take into account the syntax of software artifacts [35], [36]. These tools can ignore unimportant conflicts such as code comments or line breaks. While these tools can ensure that the merged program is syntactically correct, they cannot prevent semantic conflicts.

Semantic-based merge tools overcome these type of conflicts by employing ASTs, dependency graphs, program slicing, and denotational semantics [28], [29], [37]–[41].

Operation-based merge tools, which are a flavor of semantic-based tools, consider changes between versions as operations. Nishimura et al. [42] proposed a tool that assists developers with merge conflicts. Their approach reduces the burden of manual inspection for developers by replaying fine-grained code changes related to conflicting class members. Dig et al. [8] proposed MOLHADOREF, a software configuration management system that is aware of refactoring operations. MOLHADOREF merges two software revisions by inverting the refactoring operations, performing a textual merge, and replaying the refactoring operations. However, they did not empirically study how often refactorings cause conflicts, and how effective their approach is on a large scale.

In addition to improving software merging algorithms themselves, some researchers proposed continuously running or merging developer changes in the background to warn developers about potential conflicts before they actually occur [6], [43], [44]. Finally, some researchers performed empirical studies to predict merge conflicts [45], [46] or to understand practitioners' views on conflicts [27].

B. Refactoring

Researchers have used refactoring detection tools to study how software evolves in the presence of refactorings. For example, how refactorings impact bugs [47], [48], software quality [49], or regression testing [50]. Other researchers studied why and how refactorings happen [31], [51], [52]. However, to the best of our knowledge, there is no study that investigates the relationship between refactorings and merge conflicts.

There are a number of refactoring detection algorithms and tools in the literature [32], [53]–[58]. However, most of these tools have low precision and/or recall, need a similarity threshold to determine if two parts of the code are related, and require two fully-built versions of the software as an input in order to utilize type-binding information from the compiler. Aiming to mitigate such problems, Tsantalis et al. [12] recently proposed a tool, named *RefactoringMiner*, which we use in this work. *RefactoringMiner* does not require predefined similarity thresholds, operates at both commit level and file level, and achieves a precision of 98% and a recall of 87%.

VIII. Conclusion

Merge conflicts are a common problem for developers in distributed software development. One possible cause of complex merge conflicts are refactoring operations. In this paper, we performed, to the best of our knowledge, the first large-scale empirical study to understand the relationship between refactorings and merge conflicts. We studied almost 3,000 well-engineered open-source Java repositories. Using *RefactoringMiner*, we detected refactoring operations that were involved in merge conflicts.

Our results show that refactoring operations are involved in 22% of merge conflicts. Moreover, we find that conflicts that involve refactorings are often more complex. Both findings suggest that merging tool support that understands refactoring can have a positive impact in practice. Such tool support can vary from helping the developer understand the changes that led to the conflict to automatically resolving the conflict for them. However, our findings also show that the EXTRACT METHOD refactoring is involved in more conflicts than its typical frequency. This means that existing tools that rely on reverting refactoring operations during merging may need improvement.

References

[1] D. C. Gumm, "Distribution dimensions in software development projects: A taxonomy," *IEEE Software*, vol. 23, pp. 45–51, 09 2006. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MS.2006.122

[2] S. Phillips, J. Sillito, and R. Walker, "Branching and merging: An investigation into current version control practices," in *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, ser. CHASE '11. New York, NY, USA: ACM, 2011, pp. 9–15. [Online]. Available: http://doi.acm.org/10.1145/1984642.1984645

[3] B. Ray and M. Kim, "A case study of cross-system porting in forked projects," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 53:1–53:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393659

[4] C. Walrad and D. Strom, "The importance of branching models in SCM," *Computer*, vol. 35, no. 9, pp. 31–38, Sept 2002.

[5] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 45:1–45:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393648

[6] 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. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025139

[7] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, May 2002.

[8] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen, "Refactoring-aware configuration management for object-oriented programs," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 427–436. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.71

[9] M. Mahmoudi and S. Nadi, "The Android update problem: An empirical study," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 220–230. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196434

[10] O. Leßenich, S. Apel, C. Kästner, G. Seibt, and J. Siegmund, "Renaming and shifted code in structured merging: Looking ahead for precision and performance," in *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 543–553.

[11] L. Angyal, L. Lengyel, and H. Charaf, "Detecting renamings in three-way merging," *Acta Polytechnica Hungarica*, vol. 4, no. 4, 2007.

[12] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180206

[13] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[14] B. Appleton, S. Berczuk, R. Cabrera, and R. Orenstein, "Streamed lines: Branching patterns for parallel software development," in *Proceedings of PloP*, vol. 98, 1998.

[15] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, Feb 2004.

[16] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-oriented Reengineering Patterns*. Square Bracket Associates, 2009. [Online]. Available: https://books.google.ca/books?id=1aUGAgAAQBAJ

[17] "Github artifact page," https://github.com/ualberta-smr/RefactoringsInMergeCommits.

[18] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining Github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 92–101. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597074

[19] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in Github," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 155–165. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635922

[20] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in Github: Transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, ser. CSCW '12. New York, NY, USA: ACM, 2012, pp. 1277–1286. [Online]. Available: http://doi.acm.org/10.1145/2145204.2145396

[21] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 345–355. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568260

[22] E. Guzman, D. Azócar, and Y. Li, "Sentiment analysis of commit comments in Github: An empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 352–355. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597118

[23] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in Github," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 805–816. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786850

[24] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in Github: What can we learn from code review and bug assignment?" *Information and Software Technology*, vol. 74, pp. 204 – 218, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916000069

[25] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating Github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, Dec 2017. [Online]. Available: https://doi.org/10.1007/s10664-017-9512-6

[26] A. Field, J. Miles, and Z. Field, *Discovering statistics using R*. Sage publications, 2012.

[27] S. McKee, N. Nelson, A. Sarma, and D. Dig, "Software practitioner perspectives on merge conflicts and resolutions," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2017, pp. 467–478.

[28] 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*. ACM, 2011, pp. 190–200.

[29] S. Apel, O. Leßenich, and C. Lengauer, "Structured merge with auto-tuning: Balancing precision and performance," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 120–129. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351694

[30] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[31] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, Jan 2012.

[32] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *ECOOP 2013 – Object-Oriented Programming*, G. Castagna, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 552–576.

[33] B. Berliner *et al.*, "CVS ii: Parallelizing software development," in *Proceedings of the USENIX Winter 1990 Technical Conference*, vol. 341, 1990, p. 352.

[34] D. Lubkin, "Heterogeneous configuration management with DSEE," in *Proceedings of the 3rd International Workshop on Software Configuration Management*, ser. SCM '91. New York, NY, USA: ACM, 1991, pp. 153–160. [Online]. Available: http://doi.acm.org/10.1145/111062.111082

[35] J. Buffenbarger, "Syntactic software merging," in *Software Configuration Management*, J. Estublier, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 153–172.

[36] N. Niu, S. Easterbrook, and M. Sabetzadeh, "A category-theoretic approach to syntactic software merging," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Sept 2005, pp. 197–206.

[37] B. Westfechtel, "Structure-oriented merging of revisions of software documents," in *Proceedings of the 3rd International Workshop on Software Configuration Management*, ser. SCM '91. New York, NY, USA: ACM, 1991, pp. 68–79. [Online]. Available: http://doi.acm.org/10.1145/111062.111071

[38] V. Berzins, "Software merge: Semantics of combining changes to programs," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1875–1903, Nov. 1994. [Online]. Available: http://doi.acm.org/10.1145/197320.197403

[39] Jackson and Ladd, "Semantic diff: a tool for summarizing the effects of modifications," in *Proceedings 1994 International Conference on Software Maintenance*, Sept 1994, pp. 243–252.

[40] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 59:1–59:27, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133883

[41] D. Binkley, S. Horwitz, and T. Reps, "Program integration for languages with procedure calls," *ACM Trans. Softw. Eng. Methodol.*, vol. 4, no. 1, pp. 3–35, Jan. 1995. [Online]. Available: http://doi.acm.org.login.ezproxy.library.ualberta.ca/10.1145/201055.201056

[42] 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.

[43] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337264

[44] P. Dewan and R. Hegde, "Semi-synchronous conflict detection and resolution in asynchronous software development," in *ECSCW 2007*, L. J. Bannon, I. Wagner, C. Gutwin, R. H. R. Harper, and K. Schmidt, Eds. London: Springer London, 2007, pp. 159–178.

[45] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen, "Indicators for merge conflicts in the wild: survey and empirical study," *Automated Software Engineering*, vol. 25, no. 2, pp. 279–313, 2018.

[46] P. Accioly, P. Borba, L. Silva, and G. Cavalcanti, "Analyzing conflict predictors in open-source java projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 2018, pp. 576–586.

[47] G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, Sept 2012, pp. 104–113.

[48] P. Weißgerber and S. Diehl, "Are refactorings less error-prone than other changes?" in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 112–118. [Online]. Available: http://doi.acm.org/10.1145/1137983.1138011

[49] G. Bavota, A. D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1 – 14, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215001053

[50] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2012, pp. 357–366.

[51] F. Palomba, A. Zaidman, R. Oliveto, and A. D. Lucia, "An exploratory study on the relationship between changes and refactoring," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, May 2017, pp. 176–185.

[52] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of API-level refactorings during software evolution," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 151–160. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985815

[53] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '00. New York, NY, USA: ACM, 2000, pp. 166–177. [Online]. Available: http://doi.acm.org/10.1145/353171.353183

[54] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *Proceedings of the 20th European Conference on Object-Oriented Programming*, ser. ECOOP'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 404–428. [Online]. Available: http://dx.doi.org/10.1007/11785477_24

[55] S. R. Foster, W. G. Griswold, and S. Lerner, "Witchdoctor: Ide support for real-time auto-completion of refactorings," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 222–232.

[56] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 211–221. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337249

[57] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: A refactoring reconstruction tool based on logic query templates," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 371–372. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882353

[58] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *2010 IEEE International Conference on Software Maintenance*, Sept 2010, pp. 1–10.