# A Systematic Comparison of Two Refactoring-aware Merging Techniques

by

Max Ellis

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

# Abstract

Dealing with merge conflicts in version control systems is a challenging task for software developers. Resolving merge conflicts is a time-consuming and error-prone process, which distracts developers from important tasks. Recent work shows that refactorings are often involved in merge conflicts and that refactoring-related conflicts tend to be larger, making them harder to resolve. In the literature, there are two refactoring-aware merging techniques that claim to automatically resolve refactoring-related conflicts, an operation-based refactoring-aware merging approach called MolhadoRef and a graph-based refactoring-aware merging approach called `IntelliMerge`. However, these two techniques have never been empirically compared. In this thesis, we present `RefMerge`, a Java re-implementation of operation-based refactoring-aware merging, but built on Git. In addition to contributing this new design and implementation of refactoring-aware operation-based merging, this thesis contains the experimental results of comparing `RefMerge` to Git and `IntelliMerge` on 2,001 merge scenarios with refactoring-related conflicts from 20 open-source projects. The results show that `RefMerge` completely resolves 143 (7%) merge scenarios while `IntelliMerge` resolves only 78 (4%). This thesis also presents a qualitative analysis of the differences between the three merging algorithms and provides insights into the strengths and weaknesses of each refactoring-aware tool.

# Preface

Chapters 3-9 of this thesis were published to arXiv (http://arxiv.org/abs/2112.10370) and submitted for publication to the Transactions on Software Engineering by Max Ellis, Sarah Nadi, and Danny Dig. The technical apparatus in chapter 3 and evaluation setup in chapter 4 were designed by myself, with the assistance of Sarah Nadi and Danny Dig. The analyses, discussion, threats to validity, and literature review in chapters 4-9 are my original work with the assistance of Sarah Nadi. This work was funded by Sarah Nadi's NSERC Discovery Grant.

# Acknowledgements

Above all, I would like to thank my supervisor, Dr. Sarah Nadi, for her guidance and support throughout my graduate studies. I will be forever grateful for her patience and guidance. This thesis would not have been possible without Sarah's support, patience, and help during my recovery from Covid-19. All of my gratitude also goes to Dr. Danny Dig from University of Colorado Boulder, whose insightful feedback and suggestions helped overcome several obstacles.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Version control systems (VCSs) play a crucial role in enabling software developers to collaborate on large projects. Whether developers are working on the same branch [15], using branch-based development [51], or using pull requests to contribute changes from their external forks [52], integration issues can arise when they push their changes to the repository. When two[1] developers try to contribute different changes to the same part of the code, a VCS reports a *merge conflict*. Developers then need to spend time and effort understanding and resolving the reported conflict. Previous work found that merge conflicts occurred more than 34% of the time and could sometimes take several days to resolve [36]. Even worse, existing merge tools cannot detect every merge conflict; such conflicts might not be discovered until building or testing and may even be released in software products, causing unexpected behavior [8].

Most modern version control systems, such as Git [1], Mercurial [2], or SVN [3], treat all stored artifacts as plain text and merge files line by line. When two different changes happen to the same line of code, a textual line-based merging tool (often referred to as an *unstructured merge tool* [14]) will report a conflict since it cannot automatically decide which change to choose. However, depending on the semantics of these code changes, automated resolution may still be possible if a tool understands the nature of the code change that occurred  [12], [44].

[1]Note that in this thesis, we focus on the common practice of merging of changes from two versions of the code, and do not consider what is often referred to as *octopus merges* when more than two branches/versions are involved.

*Refactorings* are code changes that modify the structure of the code to improve its readability or maintainability without altering its observable behavior [30]. Refactoring is commonly used to enhance the software with regards to reusability, modularity, extensibility, and more [46]. It is also utilized in software re-engineering [23], which involves examining and altering a system to reconstruct it in a new form and the following implementation of the new form. Refactorings are prevalent in Agile development processes and are readily available in the top-level menu in popular IDEs. Refactorings are one example of a code change with well-defined semantics that an automated merge-conflict resolution tool can understand and resolve [24], [44], [45], [55]. For example, if on one branch, Bob refactors method `foo` by moving it from one class to another while Alice, on another branch, adds a line of code to `foo`'s body, an unstructured merging tool will report a merge conflict because Bob and Alice changed the same line of code. However, a merge tool that is aware of the semantics behind these refactoring changes could simply add the new line of code to the new location of `foo`. Thus, understanding the semantics of refactorings could result in more precise merging results, avoiding unnecessary merge conflicts. A recent study investigated 15 of more than 70 known refactorings and found the studied refactorings are involved in 22% of merge conflicts and tend to result in larger conflicts [45]. The considerable portion of merge conflicts that refactorings complicate motivates the need for automated merging tools that can handle refactorings.

While there are several research efforts that work on understanding the structure of the underlying code to automate more merge-conflict resolutions [10]–[12], [20], [40], [54], [63], there are mainly two efforts that specifically focus on refactorings. The first is by Dig et al. [24] that proposes an *operation-based refactoring-aware merging technique*. The premise of their technique is that if the version-control history records *operations* (i.e., the types of code changes that occur instead of simple textual changes), then we can leverage refactoring operations in the history to resolve conflicts. To that end, their technique relies on developers using their operation-based version control system, Molhado. At a high level, given two branches to be merged, the technique first inverts refactorings on both branches, textually merges the refactoring-free version of the code, and then replays the refactorings on the merged code. Their evaluation, based on one project, shows a 97% reduction of merge conflicts. However, recently, Shen et al. [55] argued that such an operation-based technique has its limitations, because certain complex refactoring types, such as extract method, cannot easily be inverted. Instead, they propose `IntelliMerge`, a graph-based refactoring-aware approach. `IntelliMerge` converts code on both branches to graphs, and does a graph-based three-way merge (i.e., considering the common ancestor of both branches too) where it tries to match nodes across the three versions. This node matching is based on a set of predefined rules that are meant to capture refactoring semantics along with a similarity score threshold. The authors evaluate `IntelliMerge` on 10 projects and report 88% and 90% precision and recall, respectively, when compared to the resolution committed by developers. While the results reported by both these techniques in their respective publications is promising, they have never been directly compared to each other. Given the merit of solving merge conflicts when refactorings are involved, we believe a direct comparison will shed light on the strengths and weaknesses of these techniques. Such insights can help push the state of the art of refactoring-aware merging techniques further.

In this thesis, our goal is to compare these two techniques on real-world projects that use Git as their VCS, since it is the most popular VCS used by practitioners [16]. However, while `IntelliMerge` has a publicly available implementation, Dig et al.'s approach [24] does not. Even if such an implementation is available, the reliance on Molhado (a VCS that stores operations in its history) is a major deterrent to the application of this merging technique in practice. Thus, we first develop and present `RefMerge`, an operation-based refactoring-aware merging tool for Java programs that works with Git history. `RefMerge` is a re-imagined design and implementation of Dig et al.'s work [24]. `RefMerge` follows the same approach of reverting and replaying refactors, but has the following key novelties to enable a practical evaluation: (1) `RefMerge` directly works on top of Git and does not rely on the version control system already storing operations in its history, (2) to detect refactorings, `RefMerge` uses the state-of-the art refactoring detection tool, RefactoringMiner [59] which does not rely on similarity thresholds, (3) `RefMerge` supports merging changes with the most complex types of refactorings that challenge the idea of operation-based merging, specifically ExtractMethod and InlineMethod, and finally (4) we evaluate `RefMerge` on a large scale. In summary, this thesis makes the following contributions:

- An open-source design and implementation [4] of operation-based refactoring-aware merging for Java programs, `RefMerge`, built on top of Git and which covers 8 refactoring types, including two complex refactorings that complicate conflicts [45], *Extract Method* and *Inline Method*.

- A large-scale *quantitative* comparison of the effectiveness of operation-based refactoring implemented in `RefMerge` versus graph-based based refactoring implemented in `IntelliMerge`. Our evaluation includes 2,001 merge scenarios from 20 open-source Java projects.

- A systematic *qualitative* comparison of the strengths and weaknesses of both techniques through a manual analysis of their results across a sample of 50 merge scenarios.

- A discussion of how refactoring-aware merging can be improved based on the identified strengths and weaknesses of the two techniques.

Overall, our evaluation results show that `IntelliMerge` reduces the number of refactoring conflicts and conflicting merge scenarios that a developer needs to deal with. However, graph node matching errors and the reliance on a similarity score cause it to almost triple the number of false positives and result in more than 10 times as many false negatives than Git. On the other hand, `RefMerge` resolves almost twice as many conflicting merge scenarios as `IntelliMerge` while reporting less false positives than Git and resulting in only two false negatives. Our findings shed light on how both refactoring-aware approaches can be improved and we recommend adding support for more refactorings with operation-based merging. Our complete replication package is available online [4].

## 1.1 Thesis Organization

The thesis is organized as follows. We provide necessary background and terminology in Chapter 2. After that, we discuss related work in Chapter 3. Then, we introduce our approach in Chapter 4. Chapters 5, 6, and 7 describe the evaluation setup and results. We discuss the results in Chapter 8. We then discuss the threats to validity of the results presented in this thesis in Chapter 9. Finally, we conclude the thesis in Chapter 10.

# Chapter 2

# Background and Motivating Example

To introduce the terms we use, we briefly describe how merging works in Git and provide an example to motivate the need for refactoring-aware merging techniques. We briefly describe graph-based merging in `IntelliMerge`. We also describe operation-based merging in `MolhadoRef` to provide the background required to understand how it works and how it differs from `RefMerge` in this thesis.

## 2.1 Software Merging in Git

Figure 2.1 represents a typical merge scenario. A *merge scenario* occurs when developers using Git need to integrate changes they separately worked on in different branches. The merge tools that are commonly utilized by VCSs such as Git use three-way merging techniques [46]. In *three-way merging*, two versions of the software are merged by making use of these versions' *common ancestor*, which is the common version of the code the two versions originated from before they started diverging. When merging two branches, Git attempts to merge the most recent commit on each branch, which we refer to as the *parent commits*, using the common ancestor of these commits, which we refer to as the *base commit*. The result of the merge is stored in a *merge commit.*.

A *conflicting merge scenario* is one where a merge tool is not able to automatically merge the changes from the two versions being integrated. Git

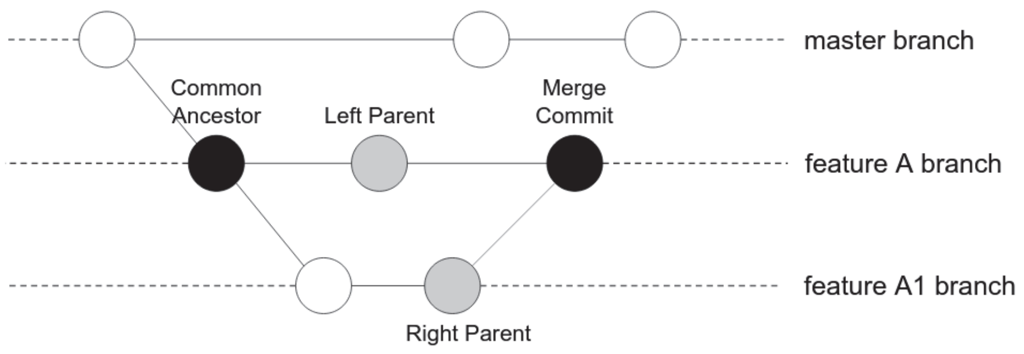


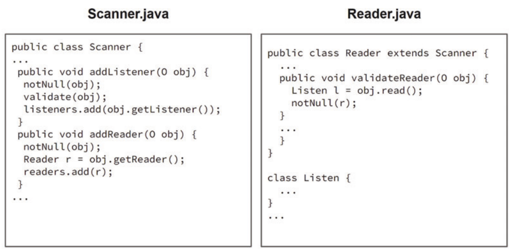
Figure 2.1: An overview of a merge scenario. The black-shaded commits and commits in-between represent a merge scenario.

reports the conflicting locations by annotating them with `<<<`, `===`, and `>>>` markers. We call these regions *conflict blocks*. When a file contains at least one conflict block, we refer to the file as a *conflicting file*. We refer to the lines within the conflict block as *conflicting lines of code* (conflicting LOC). For example, Figure 2.2d shows two conflicting files, `Scanner.java` and `Reader.java`. Each file has one conflict block. The first conflict block in `Scanner.java` has three conflicting LOC while the second conflict block in `Reader.java` has 3 conflicting LOC (Assuming we treat the whole body of the `Read` class as one line here for better visualization). We discuss the details of this example in Section 2.2.

## 2.2 Motivating Example

To understand how refactorings complicate merge scenarios, consider the example inspired by multiple real conflicts in Figure 2.2. In the left branch (Figure 2.2c), the developer renames class `Listen` to `Read` in `Reader.java` and extracts the `notNull` and `validate` calls from `addListener` to a new method, `validateObject`. In the right branch (Figure 2.2b), the other developer: (1) moves class `Listen` from being an outer class in `Reader.java` into an inner class of class `Reader` in the same file, (2) renames method `validateReader` to `validateObject`, (3) renames method `addReader` to `scanReader` in `Scanner.java`, and (4) changes the code inside `addListener`.

As shown in Figure 2.2d, Git reports a conflict in file `Reader.java` because



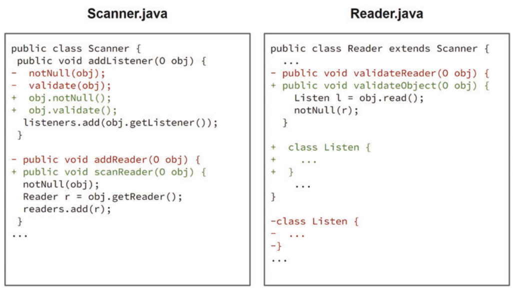
Scanner.java

```
public class Scanner {
...
 public void addListener(O obj) {
  notNull(obj);
  validate(obj);
  listeners.add(obj.getListener());
 }
 public void addReader(O obj) {
  notNull(obj);
  Reader r = obj.getReader();
  readers.add(r);
 }
...
```

Reader.java

```
public class Reader extends Scanner {
  ...
  public void validateReader(O obj) {
    Listen l = obj.read();
    notNull(r);
  }
  ...
  }
}

class Listen {
  ...
}
...
```

(a) Base commit



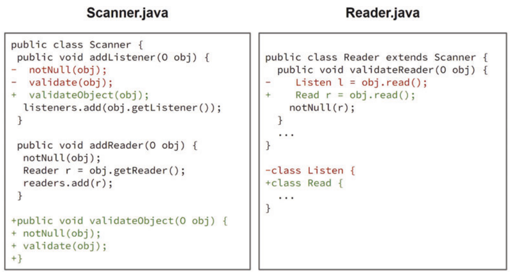
Scanner.java

```
public class Scanner {
 public void addListener(O obj) {
-  notNull(obj);
-  validate(obj);
+  obj.notNull();
+  obj.validate();
  listeners.add(obj.getListener());
 }

- public void addReader(O obj) {
+ public void scanReader(O obj) {
  notNull(obj);
  Reader r = obj.getReader();
  readers.add(r);
 }
...
```

Reader.java

```
public class Reader extends Scanner {
  ...
- public void validateReader(O obj) {
+ public void validateObject(O obj) {
    Listen l = obj.read();
    notNull(r);
  }

+  class Listen {
+    ...
+  }
    ...
}

-class Listen {
-  ...
-}
...
```

(b) Right parent



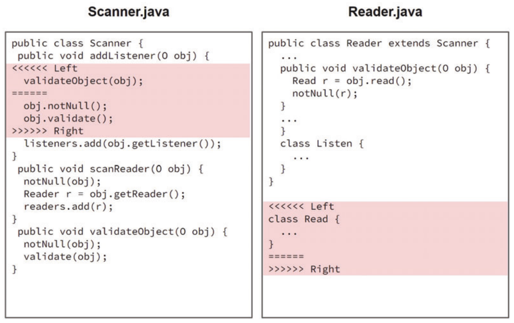
Scanner.java

```
public class Scanner {
 public void addListener(O obj) {
-  notNull(obj);
-  validate(obj);
+  validateObject(obj);
  listeners.add(obj.getListener());
 }

 public void addReader(O obj) {
  notNull(obj);
  Reader r = obj.getReader();
  readers.add(r);
 }

+public void validateObject(O obj) {
+ notNull(obj);
+ validate(obj);
+}
```

Reader.java

```
public class Reader extends Scanner {
  public void validateReader(O obj) {
-    Listen l = obj.read();
+    Read r = obj.read();
    notNull(r);
  }
  ...
}

-class Listen {
+class Read {
  ...
}
```

(c) Left parent



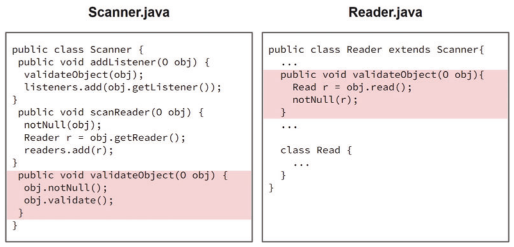
Scanner.java

```
public class Scanner {
 public void addListener(O obj) {
<<<<<< Left
  validateObject(obj);
======
  obj.notNull();
  obj.validate();
>>>>>> Right
  listeners.add(obj.getListener());
}
 public void scanReader(O obj) {
  notNull(obj);
  Reader r = obj.getReader();
  readers.add(r);
}
 public void validateObject(O obj) {
  notNull(obj);
  validate(obj);
}
```

Reader.java

```
public class Reader extends Scanner {
  ...
  public void validateObject(O obj) {
    Read r = obj.read();
    notNull(r);
  }
  ...
  }
  class Listen {
    ...
  }
}

<<<<<< Left
class Read {
  ...
}
======
>>>>>> Right
```

(d) Merge result for Git



Scanner.java

```
public class Scanner {
 public void addListener(O obj) {
  validateObject(obj);
  listeners.add(obj.getListener());
}
 public void scanReader(O obj) {
  notNull(obj);
  Reader r = obj.getReader();
  readers.add(r);
}
 public void validateObject(O obj) {
  obj.notNull();
  obj.validate();
 }
}
```

Reader.java

```
public class Reader extends Scanner{
  ...
  public void validateObject(O obj){
    Read r = obj.read();
    notNull(r);
  }
  ...

  class Read {
    ...
  }
}
```

(e) Ideal merge result

Figure 2.2: The three versions (base, left, and right) of code from Scanner.java and Reader.java, as well as the results merged by Git and an ideal merge tool.

the developers rename `Listen` on one branch and move it into class `Reader` on the other. Although both branches change the same lines of code, a smart merge tool could automatically merge these changes by considering their semantics and renaming class `Listen` to `Read`, then moving it into *Reader*, as shown in the "ideal" merge result in Figure 2.2e. We refer to the Git conflict in `Reader.java` from Figure 2.2d as a *false positive*, because it is a conflict that can be automatically resolved. If the conflict cannot be automatically resolved and required manual intervention, we refer to it as a *true positive*.

Git reports another conflict in file `Scanner.java`, where the developer on the left branch extracts code from the same region that the right branch edits the code within `addListener`. The developer now needs to compare the code inside of the extracted method with the conflict block, which may even be worse if the method was extracted to a distant location in the file, or another file altogether. However, a merge tool that considers the semantics of extract method would realize that the changes should be performed in the extracted method, rather than in `addListener` and that these changes can be merged, as shown in Figure 2.2e.

Figure 2.2e shows the ideal merge result for this scenario. This merge result avoids the unnecessary conflict in `Reader.java` by understanding the semantics of the rename and move operations. It also avoids the unnecessary conflict in `Scanner.java` by understanding the semantics of the extract method operation and applying the right changes from the right branch within the extracted method, `validateObject`. Note, however, that the ideal merge result also reports a conflict in `Reader.java` and `Scanner.java` for `validateObject`. By renaming `validateReader` to `validateObject` on the right branch and extracting a method with the same name on the left branch, the developers introduce an *accidental override*, which could introduce bugs or critical errors that may not be discovered until their software is released. Git fails to report this because the developers did not change the same lines of code. Such a case illustrates Git reporting a *false negative*, where the merge tool should report a conflict because integrating these changes requires the developer's intervention, but the tool silently merges the changes.

## 2.3 Graph-based Refactoring-aware Merging in `IntelliMerge`

Figure 2.3 shows an overview of graph-based refactoring-aware merging provided by the `IntelliMerge` paper [55]. As shown in the figure, there are four main steps in graph-based merging.

In Step 1, graph-based merging converts the base, left, and right versions of the code into graphs where the nodes are program elements and edges are their relationships, referred to as *program element graphs* (PEGs). A PEG is a labeled, weighted, and directed graph where program elements (such as classes,

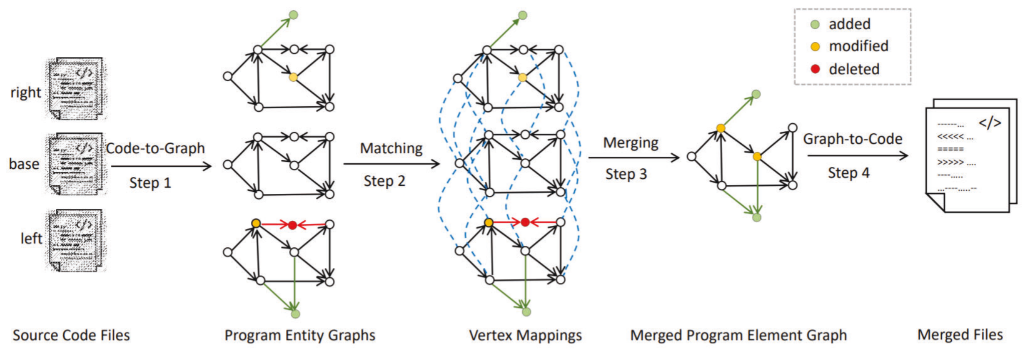


Figure 2.3: An overview of graph-based merging provided in the `IntelliMerge` paper [55].

methods, and fields) are nodes and their relationships are the edges [55]. For each of the sets of source files, the source files are first parsed into *abstract syntax trees* (ASTs). Then, the program elements are extracted from the AST to form the nodes of a PEG and the relationships between program elements are used to create the edges.

In Step 2, graph-based merging uses a matching algorithm to match the base PEG with the left and right PEGs, respectively. The matching algorithm works in two steps: *top-down* and *bottom-up*.

1. Top-down Phase. In this phase, the mapping relationship between nodes with the same signature in the base and corresponding left and right PEGs are built. This phase starts at the root of the PEG and traverses to the leaf nodes in the PEGs. The assumption here is that if two nodes in different PEGs have the same signature, then they correspond to the same program element [60].

2. Bottom-up Phase. In this phase, the PEGs are traversed from the leaf nodes back up to the root and matched based on their semantics. The bottom-up matching algorithm takes the context of the nodes into consideration into determine if there is a matching relationship. This phase detects two kinds of matching: 1-to-1 and m-to-n. The 1-to-1 matching considers changes that affects signatures of program elements, such as refactorings like rename, move, and parameter changes to methods. The m-to-n matching results from more complex refactorings like extract and

inline type refactorings.

When matching refactored program elements, the authors of `IntelliMerge` make the assumption that each pair of refactored nodes must have the same type of program element [55]. Based on this assumption, they further partition the unmatched nodes and detect refactorings in a divide and conquer manner using a list of heuristics [55]. For the 1-to-1 matching, if a node has multiple matching candidates, `IntelliMerge` ranks the candidates in descending order based on the following three criteria:

1. The context similarity (given two nodes, the node with more similar edges ranks higher).

2. The body-AST similarity for leaf nodes and the node-label similarity (When two candidate nodes have the same context similarity, the node with a body that is more similar ranks higher).

3. The location distance (determined by their file paths and start line numbers).

For m-to-n matching, `IntelliMerge` primarily refers to the extracted method with the origin(s) and inlined methods with their target(s). The *origin* of an extracted method refers to the method that it was extracted from, while the *target* of an inlined method denotes the method that it was inlined to. Programmers often extract the common part of multiple methods into a new method, so there can be more than one origin. Similarly, a method can be inlined to more than one location, so there can be more than one target. When there are multiple candidates, `IntelliMerge` extends the context of each candidate, then ranks the candidates by the similarity of the updated context.

In Step 3, the matched and unmatched sets for each pair of PEGs are input into a three-way merging algorithm. First, all matched nodes are merged recursively. After all matched nodes have been merged, each set of unmatched nodes are treated as additions and inserted into the merged PEG.

In Step 4, the merged PEG is converted into a set of source files, with possible conflict blocks that require manual intervention. If there are conflicts, `IntelliMerge` surrounds the conflicting code with the same conflict markers that Git uses.

## 2.4 Operation-based Refactoring-aware Merging in MolhadoRef

Dig et al.'s implementation of operation-based merging, MolhadoRef, works in five main steps [24].

In Step 1, MolhadoRef utilizes Molhado, the operation-based VCS, to track the code changes that happened between the base version and the left and right versions. MolhadoRef also collects refactoring logs that are collected by *Eclipse* [5]. From the refactoring logs and changes, MolhadoRef differentiates edit and refactoring operations.

In Step 2, MolhadoRef detects and solves refactoring conflicts and circular dependencies. When detecting refactoring conflicts, it uses a *conflict matrix* which was constructed by carefully considering refactoring semantics and how refactorings interact with each other. For any two kinds of supported operations, the conflict matrix gives a set of rules that determines whether the pair of operations results in a conflict. If MolhadoRef determines that two operations are conflicting, it requires manual intervention from the developer before proceeding. Then, MolhadoRef creates a dependence graph between performed operations. The dependence graph is necessary because the operations can be replayed in any order when merging unless there is a dependence between the operations. Thus, MolhadoRef maintains the order for such operations in a dependence graph. Similar to detecting conflicts, MolhadoRef utilizes a set of rules within the *dependence matrix* which were carefully developed by considering how operations could be performed such that there is no way to determine the order that the operations need to be replayed in. However, it is possible for operations with a dependence relationship to form a circular dependence. When MolhadoRef detects a circular dependence, it requires manual intervention from the practitioner before it can continue building the dependence graph.

In Step 3, MolhadoRef inverts the detected refactorings. To invert a refactoring, MolhadoRef must first create an *inverse refactoring* that it performs on the original refactoring operation to cancel out the refactoring. MolhadoRef uses the information provided by Eclipse's refactoring logs to create and perform the inverse refactoring. In the case that MolhadoRef cannot invert a refactoring, it treats the refactoring like a textual edit and does no worse than textual merging.

After all of the refactorings have been inverted, MolhadoRef performs a three-way textual merge in Step 4. By inverting all of the detected refactorings, any chance of a refactoring conflict have been removed from the textual merge.

In Step 5, MolhadoRef replays the refactorings. Refactorings that touch the same program element can has side effects on that program element. For example, if a method refactoring occurs within a class that is refactored, the refactoring engine may not be able to find the method after the class is refactored. To avoid this issue, MolhadoRef replays refactorings in a bottom-up approach. While a class refactoring can influence the method's fully qualified name, performing a method refactoring will not change the class's fully qualified name. MolhadoRef continues replaying refactorings until there are no more refactorings to replay.

# Chapter 3

# Literature Review

## 3.1 Software Merging

The proposed software merging techniques in the literature can generally be categorized into *unstructured*, *structured*, and *semi-structured* merging techniques [46].

*Unstructured merging techniques* represent any software artifact as a sequence of textual lines [14]. This gives unstructured merging techniques the strength of being able to process all textual artifacts, regardless of the programming language [46]. Due to its simplicity and versatility, modern version-control systems such as Git or mercurial still rely on such unstructured merging. The downside to this technique is that unstructured merging cannot handle multiple changes to the same lines, since it cannot consider the the syntactic and semantic meaning in software artifacts [12]. Because of that, unstructured merging cannot handle or understand refactorings, causing conflicts similar to those shown in Figure 2.2d.

*Structured merging* tries to alleviate the problems of unstructured tools by leveraging the underlying structure of software artifacts, typically through operating on an Abstract Syntax Tree (AST) instead of textual lines [10]. Westfechtel [61] and Buffenbarger [18] pioneered in proposing structured merge algorithms, proposing algorithms which incorporate context-free and context-sensitive structures. Considering the structure of software artifacts allows structured merging techniques to handle syntactic and semantic conflicts [35], [39], [62]. This comes at the cost of generally being language specific and being too expensive to be used in practice. Leßenich et al. [40] proposed *auto-tuning*, an approach that switches between structured and unstructured merging, and implemented `JDime` to demonstrate their approach. Zhu et al. [63] built on top of `JDime` by matching nodes based on an adjustable quality function. Seibt et al. [54] recently performed a large-scale empirical study with unstructured, semi-structured, and structured merge algorithms and their findings suggest that combined strategies are promising moving forward. Additionally, structured merge approaches do not specifically handle all refactoring semantics.

*Semi-structured techniques* aim to create a middle ground by considering both the language independence of unstructured merging and the precision of structured merging [12]. `FSTMerge` was proposed by Apel et al. [12] as one of the first semi-structured merging approaches. While `FSTMerge` reduces the number of merge conflicts reported compared to unstructured merge, `FSTMerge` struggles with renamings. Cavalcanti et al. [20] proposed `jFSTMerge`, building upon `FSTMerge` by adding handlers for different types of conflicts such as renaming. Cavalcanti et al. [20] go further and provide evidence that the number of unecessary conflicts is significantly reduced when using semi-structured merge. However, they do not find evidence that semi-structured merge misses fewer actual conflicts (false negatives). Similar findings are observed by Trindade et al. [58], but at a lesser extent, when investigating semi-structured merging in JavaScript systems.

By representing software artifacts partly as text and partly as trees, semi-structured merging achieves a certain level of language-independence. Cavalcanti et al. [19] performed an empirical study to compare unstructured and semi-structured merging techniques. They found that semi-structured merge can reduce the number of merge conflicts by half. We compare only against `IntelliMerge` because the `IntelliMerge` paper already shows that it already outperforms `jFSTMerge` [55]. Furthermore, we are focusing on techniques that specifically target refactorings in order to compare their strengths and weaknesses.

## 3.2 Proactive Conflict Detection & Prevention

The key idea behind this research line is that detecting conflicts as soon as they happen, even before a developer commits the changes, can lead to conflicts that are easier to resolve. Knowing what changes other developers are making is beneficial for team productivity and reducing the number of reported merge conflicts [28]. One such approach is *speculative merging* [17], [33], where all combinations of available branches are pulled and merged in the background. Owhadi-Kareshk et al. [49] designed a classifier for predicting merge conflicts with the aim of reducing the computational costs of speculative merging by filtering out merge scenarios that are unlikely to be conflicting.

`Syde` [34] and `Palantir` [53] are two tools that increase developer awareness by illustrating the code changes their team members are making. `Cassandra` [37] minimizes simultaneous edits to the same file by optimizing task scheduling. `ConE` [43] is an approach that proactively detects concurrent edits to help mitigate certain resulting problems, including merge conflicts. Silva et al. [57] proposed utilizing automated unit test creation to detect semantic conflicts that a merge tool could have missed. Fan et al. [29] proposed using dependency-based automatic locking to support fine-grained locking and avoid semantic conflicts. DeepMerge is a recent effort that defines merge conflict resolution as a machine learning problem [25]. The approach primarily leverages the fact that around 80% of merge conflict resolution only rearrange lines [32]. However, they do not explicitly consider refactoring semantics in their merge conflict resolution.

## 3.3 Refactoring Detection

Refactoring is a widespread practice that enables developers to improve the maintainability and readability of their code [56]. Refactoring has been extensively studied over the past few decades [47], with recent work focusing on the detection of refactoring changes and the relationship between refactorings and code quality [22]. Palomba et al. [50] performed a quantitative investigation on the relationship between refactorings and different types of code changes. They found that developers perform complex refactorings to improve code cohesion. Similarly, Bavota et al. [13] investigated whether common quality metrics might suggest whether refactoring operations might be necessary. `RIPE` is a technique that estimates what the impact of performing a refactoring operation will be on code quality metrics [21].

Multiple tools have been developed to detect different refactoring types, such as `Ref-Finder` [38] and `RefDistiller` [9]. We use the state-of-the-art refactoring detection tool, `RefactoringMiner`, which achieves a precision of 98% and a recall of 87% [59].

## 3.4 Operation-based & Refactoring-aware Merging

Operation-based merging is a semi-structured merging technique that models changes between versions as operations or transformations [26], [41], [42] which could be used to support refactoring-aware merging [24]. Nishimura et al. [48] proposed a tool that reduces the manual effort necessary to resolve merge conflicts by replaying fine-grained code changes related to conflicting class members. Their approach only considers edits and has problems with long edit histories and finer granularity of operations [24].

Ekmaan et al. [27] proposed a refactoring-aware versioning system designed for Eclipse IDE. Their approach keeps program elements in volatile memory, thus allowing for a short-lived history of refactored operations. However, their approach does not support software merging.

Dig et al. [24] proposed `MolhadoRef`, the first operation-based refactoring-aware merging algorithm. As described in more detail in Section 2.4, the premise behind their idea is that refactorings have well-defined semantics which can be used to treat refactorings as operations which can be undone and replayed. By undoing refactorings, it takes the refactorings out the merge, allowing a textual merge tool to perform a merge without refactorings complicating it. The well-defined refactoring semantics can further be used to check if two refactoring operations are conflicting and aid in conflict resolution. However, they did not provide coverage for any complex refactorings and they did not evaluate their approach on a large scale. In addition, `MolhadoRef` relied on a research-based version control system to keep track of operations, so it could not be applied to popular version control systems.

Meanwhile, Shen et al. [55] proposed `IntelliMerge`, a graph-based refactoring-aware merging algorithm. As described in more detail in Section 2.3, `IntelliMerge` merges two software versions by converting software artifacts to graphs. It then performs a matching step to detect refactorings and code edits, merges the graphs, and then serializes the merged graph back into text files. However, `IntelliMerge`'s evaluation did not include a qualitative analysis to check for false positives and false negatives. In addition, `IntelliMerge` relies on a similarity score which increases the chances of `IntelliMerge` missing a refactoring or incorrectly detecting a refactoring.

In our work, we make multiple design choices to re-imagine operation-based refactoring-aware merging on top of git in a scalable way. We add coverage for complex refactorings to investigate if operation-based merging can handle complex refactorings. In this thesis, we perform the first large-scale empirical evaluation on operation-based refactoring-aware merging and compare it with `IntelliMerge` for the first time. We also perform a systematic qualitative analysis to determine if either approach reports false positives or misses conflicts, resulting in false negatives or unexpected merges.

# Chapter 4

# `RefMerge`: Refactoring-aware Operation-based Merging for Java

The high level idea of operation-based refactoring-aware merging is that if we invert refactorings before merging and then replay the refactorings, there will be no refactoring related conflicts to complicate the merge. Figure 4.1 presents the following overview of our implementation of `RefMerge`, which targets Java programs and consists of the following five steps.

1. *Detect and Simplify Refactorings*: We use `RefactoringMiner`, a state-of-the-art refactoring detection tool with 99.7% precision and 94.2% recall [59] to detect refactorings in each commit between the base commit and each parent respectively. We check if each detected refactoring can be simplified and simplify the refactorings accordingly.

2. *Invert Refactorings*: We use the corresponding refactoring list from Step 1 to invert each refactoring until all supported refactorings have been inverted.

3. *Merge*: We use Git to merge the left and right parents, $P'_L$ and $P'_R$, after all their refactorings have been inverted.

4. *Detect Refactoring Conflicts*: We compare the left and right refactoring lists for potential refactoring conflicts and commutative relationships and merge them into one list.

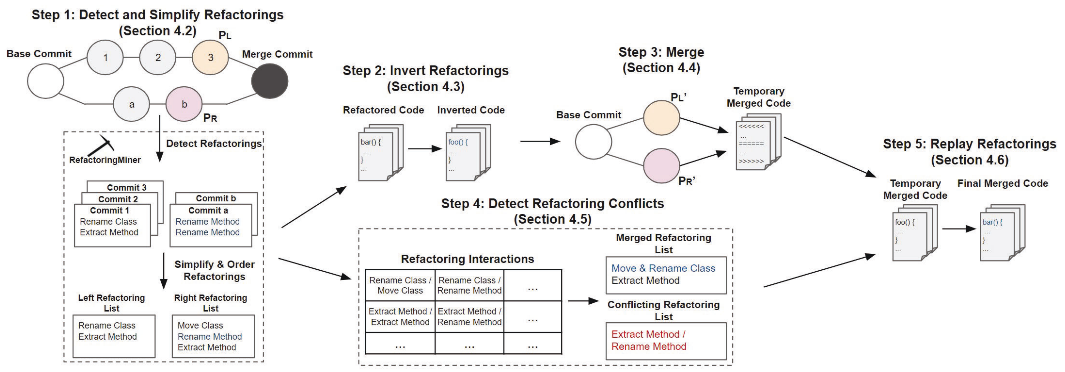


Figure 4.1: An overview of RefMerge's merging algorithm.

5. *Replay Refactorings*: We finally use the merged refactoring list to replay all non-conflicting refactorings.

In this section, we focus on our design and implementation of the operation-based approach to enable it to work on top of Git, which makes some of the details different from `MolhadoRef`.

## 4.1 Step 1: Detect and Simplify Refactorings

### 4.1.1 Refactoring Detection

We use `RefactoringMiner` to detect refactorings in each commit between the base commit and each parent commit respectively. We detect refactorings in each commit instead of only comparing the base and parent commits to ensure precise detection in longer histories. This is an important difference from `MolhadoRef` as the use of RefactoringMiner allows `RefMerge` to be implemented for Git, instead of relying on a research-based VCS.

### 4.1.2 Refactoring Simplification

`RefMerge` processes each detected refactoring one by one and keeps a list of processed refactorings for each of the left and right branches. We compare each detected refactoring to the processed refactorings in the list to determine if it is either a transitive refactoring or part of a refactoring chain (defined below).

We define *transitive refactorings* as successive related refactorings of the same refactoring type. For example, consider that method *foo* is renamed to *bar*. In the next commit, *bar* is renamed to *foobar*. In this case, the two method renamings are transitive and *foo* is eventually being renamed to *foobar*. When `RefMerge` finds that a newly detected refactoring is a transitive refactoring of an existing refactoring in the list, it updates the related transitive refactorings in the list instead of adding this new refactoring. In this example, `RefMerge` would first add *rename foo to bar* to the list. When it processes *rename bar to foobar*, it would detect that this is a transitive refactoring of an existing refactoring in the list so it will simply update that existing refactoring to *rename foo to foobar*. We carefully considered the semantics of each refactoring operation to determine transitive refactorings. The logic for detecting transitive refactorings can be found in Appendix A.3.

*Refactoring chains* consist of two or more refactorings that touch the same program element. When two refactorings touch the same program element, the details of that program element will diverge from what is stored in RefactoringMiner's refactoring object, causing the refactoring to not be found when later inverting the refactoring or detecting refactoring conflicts. For example, when a method is renamed in *class A* and *class A* is renamed to *class B* in a later commit, the first refactoring object will still associate the method with *class A*. This means that if a transitive refactoring is later performed on the same method, we will not be able to detect the transitive relationship because the methods will be associated with different classes.

When we find that a refactoring is part of a refactoring chain, we update the refactorings in the refactoring chain. For example, consider that after *A.foo* is renamed to *A.bar*, *class A* is renamed to *class B*. Then in a later commit, *B.bar* is renamed to *B.foobar*. Since method *foo* was renamed to *bar* inside of class *A*, *A.bar* and *B.bar* have different method signatures and the information that these *Rename Method*s are transitive is lost. To address this, `RefMerge` first adds the first refactoring *rename A.foo to A.bar* to the list. When it later processes the second refactoring *rename class A to B*, it adds this second refactoring to the list *and* also updates the first refactoring to *rename B.foo to*

*B.bar*. That way, when RefMerge processes the third refactoring *rename B.bar to B.foobar*, it can detect the transitive relationship and update it accordingly. All of our detailed logic for detecting transitive refactorings and refactoring chains can be found in Appendix A as well as our artifact [4].

### 4.1.3 Refactoring Order

Since we do not know the order in which developers performed refactorings within the same commit, we cannot simply invert the refactorings in the opposite order they are detected in, similar to what MolhadoRef does. Instead, we reorder the refactorings in a top-down order based on the granularity of the program element being refactored. For example, class level refactorings come before method level refactorings.

Combining transitive refactorings, updating refactoring chains and using a top-down order has three advantages. First, when inverting and replaying refactorings, all transitive refactorings are combined and can be treated as if they were detected at a coarse-grained granularity. This is an important distinction from MolhadoRef, because it reduces the number of refactorings that need to be performed and simplifies conflict detection while at the same time ensuring precise refactoring detection. Second, the combination of updating refactoring chains and using the pre-determined order removes any need to keep track of the order that the refactorings are detected in. Lastly, using a top-down order while simplifying refactorings automatically breaks any circular dependencies between refactoring operations. This is an important difference from MolhadoRef which required user intervention to help resolve circular dependencies. By automatically breaking circular dependencies, RefMerge allows the user to focus only on actual conflicts.

## 4.2 Step 2: Invert Refactorings

Once refactorings are detected, RefMerge creates a refactoring-free version of each parent commit by inverting the refactorings in the refactoring lists from Step 1. To invert a refactoring $r$, RefMerge needs to create and apply the inverse refactoring $\bar{r}$. $\bar{r}$ is an inverse of $r$ if $\bar{r}(r(E)) = E$. For example, the inverse of a refactoring that renames method *foo* to *bar* is another refactoring that renames *bar* to *foo*.

`RefMerge` uses the information provided by `RefactoringMiner` to create each inverse refactoring. Each refactoring detected by `RefactoringMiner` is represented by a data structure that contains important information about the refactoring. Among others, the data structure contains information such as the refactoring type, information about the original program element, and information about the refactored program element. From the provided information, `RefMerge` obtains the corresponding elements and executes the refactoring through a refactoring engine. Importantly, executing the inverse refactoring does not only invert the refactored program element, but it also changes any references to the program element. This includes references added at any point after the refactoring was performed. In the case that the refactored program element is deleted in a future commit, the inverse refactoring cannot be performed and `RefMerge` moves on to the next refactoring.

## 4.3 Step 3: Merge

After all refactorings are inverted on both branches, only non-refactoring changes remain in the parent commits. We refer to this version of each parent as $P'$ in Figure 4.1. In this step, we textually merge $P'_L$ and $P'_R$. Most same-line or same-block conflicts that would have been caused by refactorings are now eliminated through inverting the refactorings. However, some same-line and same-block conflicts may still exist because additional edits may have been performed to or beyond the refactored code.

For example, consider the conflict blocks in `Scanner.addListener` in Figure 2.2. If the developer adds several other lines of code to the extracted method, those lines will be inlined to the `validateObject` method invocation and reported in the conflict block. In this case, `RefMerge` will report more conflicting lines than Git because no matter how many lines of code are added to `Scanner.validateObject()`, Git's conflicting region will remain the same.

While the extra conflicting lines that `RefMerge` reports could be considered to be disadvantageous, inlining the extracted code clearly indicates what code is part of the conflict in a single location.

## 4.4 Step 4: Detect Refactoring Conflicts

Generally speaking, a pair of refactorings that touch unrelated program elements do not have any interaction. However, a pair of refactorings that touch related program elements will have interactions, which can be conflicting or commutative. For each pair of refactorings, we have to predetermine the interactions that the refactorings can result in and then use that knowledge to detect conflicts and commutative refactorings. Refactoring operations that *conflict* cannot both be replayed, while refactoring operations that are *commutative* can be replayed in either order and will result in the same code. We make the assumption that two refactoring operations cannot both conflict and have a commutative relationship. Using the semantics of refactoring operations, we carefully compute and revise the conflict and commutative logic for each refactoring combination, which we explain below and can be found in our artifact [4] as well as Appendix A. `RefMerge` uses this knowledge to compare each refactoring in the left branch with each refactoring in the right branch and detect refactoring conflicts.

**Detecting Conflicts**

`RefMerge` first checks if the two refactoring operations are conflicting. There are a series of preconditions that must be met for two refactoring operations to conflict. To illustrate, we provide an example using the conflict logic for $RenameMethod(m_1, m_2)$ and $RenameMethod(m_3, m_4)$:

$$\begin{aligned}hasConflict(RenameMethod(m_1, m_2), MoveMethod(m_3, m_4)) :=\\ ((m_1 == m_3 \land m_2 \neq m_4) \lor (m_1 \neq m_3 \land m_2 == m_4))\lor\\ (\neg overrides(m_1, m_3) \land overrides(m_2, m_4)\lor\\ (\neg overloads(m_1, m_3) \land overloads(m_2, m_4)\end{aligned} \tag{4.1}$$

These two refactorings result in a conflict if (1) the source of both refactorings is the same program element ($m_1 == m_3$) but their names differ ($m_2 \neq m_4$) or (2) the sources of both renames are different program elements ($m_1 \neq m_3$) but the renamed destinations are the same program element ($m_2 == m_4$). In other words, if the same method is renamed to two separate names or if two different methods inside of the same class are renamed to the same name with the same signature, then the refactorings conflict.

In addition, two refactoring operations can conflict without changing the same program element. We refer to this as a *semantic conflict*. There are two examples of semantic conflicts for *RenameMethod*/*RenameMethod*: (1) an accidental overload and (2) an accidental override. In the case of an accidental overload, two methods with different names are renamed to the same name in the same class but have different signatures. In the case of an accidental override, two methods within classes with an inheritance relationship are renamed to the same name with the same signature, which causes one of the methods to override the other. Semantic conflicts will not be detected by a text-based merge tool such as Git because the same line is not changed by both branches. The developer might not realize the problem until it appears in testing, or worse in production. The motivating example in Figure 2.2 showed an example of an accidental override due to the combination of *Rename Method* and *Extract Method* refactorings. The full conflict logic of the refactorings we support can be found in Appendix A.1.

### Detecting Commutative Relationships

After RefMerge checks for refactoring conflicts, it checks for a *commutative relationship* between the two refactoring operations using the corresponding predetermined commutative logic. Two refactoring operations can only be commutative if they do not conflict and if they are different types of refactorings. If the pair of refactorings meets these conditions and they both refactor the same program element, then they are commutative. For example, *Rename*

*Method* and *Rename Method* cannot be commutative because they are the same refactoring type and there is no way the same program element can be renamed on both branches to different names without conflicting. However, *Move Method* and *Rename Method* can be performed on the same program element commutatively. Similarly, the *Move Class* and *Rename Class* refactorings performed on class *Listen* in Figure 2.2 are an example of commutative refactorings.

The commutative logic for $MoveMethod(m_1, m_2)$ and $RenameMethod(m_3, m_4)$ is as follows:

$$isCommutative(MoveMethod(m_1, m_2), RenameMethod(m_3, m_4) := \\ (m_1 == m_3 \land m_2 \neq m_4)$$

These two refactorings are commutative if the source of both refactorings is the same program element ($m_1 == m_3$) and their destinations are different ($m_2 \neq m_4$). The idea is that if a *Move Method* and *Rename Method* refactoring are performed on the same program element, then we can move the program element and then rename it, or rename it and then move it. We consider the semantics of each refactoring to determine commutativity.

After all detected refactorings have been compared between branches for refactoring conflicts and commutative relationships, `RefMerge` combines the refactoring lists containing non-conflicting refactorings from each branch into one list. While `RefMerge` inverts the refactorings on each branch in a top-down order (after simplifying the refactoring lists to enable this), it orders the combined refactoring list in a bottom-up order based on the element hierarchy (for example, method before class) for replaying refactorings. Multiple refactorings might touch the same program element, such as a *Move Method* and a *Rename Class*. By renaming the class before moving the method, `RefMerge` will not be able to find the method refactoring, because the class that the method is moved from will no longer exist. Since higher-level program elements do not depend on lower level program elements, replaying refactorings bottom-up allows `RefMerge` to replay the refactorings without any additional effort. The replay refactoring list for Figure 2.2 after detecting refactoring conflicts and commutative conflicts would contain (1) *Rename Method addReader* to *scanReader* and (2) *Move And Rename Class Listen* to inner class `Reader.Read`. The conflicting refactoring list would contain *Extract Method* `validateObject` from `addListener` and *Rename Method* `validateReader` to `validateObject`. The full logic for detecting commutative refactorings can be found in Appendix A.2.

## 4.5 Step 5: Replay Refactorings

Finally, `RefMerge` replays the refactorings. For each inverted refactoring, `RefMerge` re-creates and performs the refactoring that was originally performed by the developer. Executing the refactoring includes updating all references in the program, including those added on the other branch.

## 4.6 Current Implementation

***Technologies and Tools*** We implement `RefMerge` as an IntelliJ[1] plugin for merging Java programs. It consists of four key modules corresponding to the steps of the proposed technique. We use the state-of-the-art refactoring detection tool, *RefactoringMiner* [59] to detect the refactorings and we use the *IntelliJ refactoring engine* to programatically invert and replay the refactorings.

***Supported Refactorings*** Even though the idea of operation-based refactoring-aware merging and our proposed implementation of it generally applies to all refactorings, there are more than 70 known refactoring types [30]; it is a large engineering effort to implement every refactoring. Instead of implementing every refactoring, we use a subset of eight refactorings to show the feasibility of the approach and enable the empirical comparison.

[1] https://www.jetbrains.com/idea

We first choose refactorings that commonly appear in merge scenarios since these are refactorings developers will deal with. We find the top 10 occurring refactoring types from a recent large-scale empirical study of code changes in 450,000 commits [6]. Of these top 10 refactoring types, only *Extract Method*, *Rename Method*, and *Move Class* are involved frequently in merge scenarios with refactoring-related conflicts [45]. Thus, we include these three refactoring types in our implementation.

Next, we select refactorings that can conceptually challenge the idea of operation-based merging. The IntelliMerge authors suggested that operation-based merging cannot handle *Extract Method* or *Inline Method*, because they do not have an inverse refactoring [55]. However, in theory, *Extract Method* can be inverted by performing an *Inline Method* refactoring, and vice versa. Thus, we select *Extract Method* and *Inline Method* as two refactorings that conceptually challenge operation-based merging.

Finally, we select additional refactorings from the class and method level to cover refactorings that can result in larger conflicting regions and evaluate potential problems such as accidental override. We select *Rename Class* because it is at the class granularity and renames are the most universally used refactoring in the IntelliJ IDE [31]. We select *Move Method* and *Move And Rename Method* to evaluate potential inheritance problems, and we select *Move And Rename Class* to add full coverage of *Move* and *Rename* refactorings at the method and class granularity.

When a refactoring is performed that `RefMerge` does not support or `RefMerge` fails to invert, `RefMerge` results in the same merge as Git for the program element. Thus, `RefMerge` should improve on Git for supported refactorings, but should be no worse than Git for refactorings that are not currently supported.

Although our current implementation covers only eight refactorings, our implementation is well documented and implemented in a modular way to easily allow for extension. Supporting a new refactoring involves: (1) considering how the new refactoring interacts with presently supported refactorings, (2) adding conflict logic and simplification logic for any refactoring that the new refactoring interacts with, (3) adding handlers for the conflict logic and simplification logic, (4) adding the new refactoring to the top-down order, (5) adding a method to invert the refactoring with the corresponding IntelliJ refactoring processor, and (6) add a method to replay the refactoring with the corresponding IntelliJ refactoring processor. Overall, this amounts to adding five methods in three classes as well as $n + 1$ conflict handlers where $n$ is the current number of supported refactorings, with the full guidance of our documentation.

# Chapter 5

# Evaluation Setup

We compare the effectiveness of `RefMerge`, Git, and the state-of-the-art refactoring-aware merge tool, `IntelliMerge` [55] on 2,001 merge scenarios that contain refactoring-related conflicts from 20 open-source projects. These projects include the original 10 projects `IntelliMerge` was evaluated on as well as an additional 10 projects with different distributions of conflicting merge scenarios. We answer the following research questions:

**RQ1** *How many merge conflicts do the three merge tools report?* A tool that automatically resolves more merge conflicts will reduce the time and effort developers have to spend resolving conflicts. We report conflicts at all granularity levels (scenarios, files, and conflict blocks).

**RQ2** *What are the discrepancies between the merge conflicts that RefMerge and IntelliMerge report?* While either tools may report less conflicts, which seems better at face value, we need to investigate if they correctly resolve the conflicts or if they miss reporting real conflicts. We perform a qualitative analysis on the results reported by `RefMerge` and `IntelliMerge` to understand the strengths and weaknesses of each tool.

## 5.1 Project & Merge Scenario Selection

We use the same 10 projects that the `IntelliMerge` authors use in their evaluation [55]. To select these projects, the authors searched for the top 100 Java projects with high numbers of stargazers on Github, and then selected the projects with the most merge commits and contributors [55]. The authors then ran the analysis by Mahmoudi et al. [45] on these 10 projects to identify conflicting merge scenarios that have refactoring changes involved in the conflict. In a nutshell, this analysis replays merge scenarios in the Git history to find conflicting ones, uses `RefatoringMiner` [59] to find refactorings in the history of these conflicting merge scenarios, and then compares the location of the refactorings to the location of the conflict blocks to determine if a conflict has an *involved refactoring*. At the time of the `IntelliMerge` publication, these 10 projects contained 1,070 conflicting merge scenarios with involved refactorings.

For generalizability, we expand our evaluation to cover an additional 10 projects. Mahmoudi et al. [45] shared a data set with the results of their analysis for 2,955 open-source GitHub projects. We use this data set to select the additional 10 projects for our evaluation. Our goal is to have a selection of projects with different distributions of (conflicting) merge scenarios to avoid any bias towards project-specific practices. Thus, we sort the 2,955 projects within the dataset based on the number of refactoring-related conflicts each project has. We randomly select three projects from the bottom 30% of the projects, four from the middle 40%, and three from the top 30%.

Given the 20 selected projects, we collect an up-to-date set of merge scenarios with involved refactorings by re-running Mahmoudi et al.'s analysis [45] on the latest history of each project as of September 26, 2021. Our artifact page [4] contains the exact version of each project that we consider. This means that for the 10 projects originally used by `IntelliMerge`, our data set contains the original 1,070 merge scenarios as well as any additional ones that appear in the Git history since their publication date.

Table 5.1 shows the number of merge scenarios with refactoring-related conflicts in all 20 selected projects, with the projects used in the `IntelliMerge` paper in bold. For additional context, we also show the number of stargazers of each project. Overall, we evaluate on 2,001 conflicting merge scenarios with involved refactorings.

Table 5.1: Number of conflicting merge scenarios with involved refactorings for the 20 projects we evaluate on. The 10 projects from the `IntelliMerge` paper are in bold.

| **Project** | **Stargazers** | **Merge Scenarios** |
|---|---:|---:|
| **cassandra** | 6,882 | 922 |
| **elasticsearch** | 56,665 | 178 |
| **gradle** | 12,410 | 117 |
| **antlr4** | 10,738 | 100 |
| platform_frameworks_support | 1,609 | 96 |
| **deeplearning4j** | 12,208 | 93 |
| **realm-java** | 11,206 | 92 |
| jackson-core | 1,984 | 81 |
| android | 3,161 | 81 |
| cometd | 535 | 63 |
| **storm** | 6,278 | 33 |
| ProjectE | 308 | 30 |
| **javaparser** | 3,859 | 23 |
| druid | 24,576 | 17 |
| androidannotations | 11,171 | 15 |
| **junit4** | 8,198 | 14 |
| MinecraftForge | 4,945 | 14 |
| iFixitAndroid | 143 | 13 |
| MozStumbler | 609 | 10 |
| **error-prone** | 5,717 | 9 |
| Total | | 2,001 |

## 5.2 Reproducing `IntelliMerge`

Before describing the evaluation metrics we use for comparing the merge tools, we need to ensure that we are correctly running `IntelliMerge`. Thus, we first attempt to reproduce the results found in the corresponding publication [55] using their exact setup and data, as shared in their Github repository [7]. We share the exact steps we followed as well as the details of the results of reproducing `IntelliMerge`[1].

We run `IntelliMerge` v1.0.7 on the same 1,070 merge scenarios used in the original publication, including their same post-processing steps such as remov-

[1]https://github.com/max-ellis/IntelliMerge/tree/evaluation

ing all comments from the merged files. We use the same calculation proposed by `IntelliMerge`'s authors to measure precision and recall for `IntelliMerge` and Git. They propose comparing auto-merged code with manually-merged code to measure precision and recall. They define *auto-merged code* as code that is not part of a conflict block in a tool's merge result and *manually-merged code* as the code that appears in the resolved merge commit. We use the same *diff* tool provided by Git that the `IntelliMerge` authors used to calculate the number of different lines between the auto-merged and manually-merged code. Note that `IntelliMerge` reports precision and recall based *only* on the conflicting files in each merge scenario, not on all changed files in the scenario.

We were not able to reproduce the exact numbers found in the `IntelliMerge` paper [55]. After emailing the authors, we verified that they perform post-processing steps to deal with some cases that are caused by the program elements being in a different order as well as format related diffs, such as textually moving, reordering, and cosmetic diffs. Because of these undocumented manual post-processing steps, it is impossible to reproduce the exact numbers in the `IntelliMerge` paper. Although we were not able to get the exact numbers, the precision and recall we obtained were within 10% of the numbers in their paper. For further confirmation, we explicitly shared our setup[1] with the `IntelliMerge` authors and received confirmation that our setup is correct and that the differences in results we obtained do not misrepresent `IntelliMerge`.

## 5.3 Tool Comparison Setup

After verifying with the `IntelliMerge` authors that we are correctly running their tool, we could proceed with our evaluation. Given the 2,001 merge scenarios, we identify the base commit, left parent commit, and right parent commit of each scenario. We provide each tool (Git, `IntelliMerge`, `RefMerge`) with these three commits in order to perform its three-way merge. We record the results of *all changed files* in the merge scenario, as opposed to only conflicting files (which is what the `IntelliMerge` evaluation does). Considering the result of all changed files allows us to catch cases where one of the tools introduces a conflict in a file that Git did not originally report a conflict for. Additionally, while the `IntelliMerge` authors removed comments in their evaluation, we do not post-process the results of any of the merge tools in any way to ensure that we see the same results a developer using the tool in practice would see. Overall, our goal in this evaluation is to minimize any manual pre and post processing steps such that we can compare the results of these tools in a practical setting. Note that `IntelliMerge` supports 21 refactorings, including the 15 refactorings supported by Mahmoudi et al.'s analysis [45] while `RefMerge` supports a subset of only 8 refactorings. While the scenarios we evaluate on may have refactorings that either tools do not support, we do not limit the evaluation to only supported refactorings so we can also understand how the tools handle unsupported refactorings.

We run our experiments on a quad-core computer with Intel (RJ) Core (TM) i7-7700HQ CPU @ 2.80GHz, 16 GB RAM and Ubuntu 20.04 OS. Originally, we attempted to complete our evaluation without a timeout but `IntelliMerge` took several hours to complete a merge in multiple merge scenarios. For feasibility of completing the evaluation, we re-run the evaluation from start and use a 15 minute timeout for each tool. We investigated what was causing `IntelliMerge` to take so long and found that merge scenarios where several files were changed caused the graph building step for `IntelliMerge` to take significantly longer.

## 5.4 Used Metrics and Analysis Methods

We choose not to use the same recall and precision metrics that the `IntelliMerge` authors propose, because (1) these metrics do not correctly capture the effectiveness of a merge tool and (2) auto-merged code is not a reliable way to measure false positives and false negatives. Consider the merge conflict in `Scanner.java` in Figure 2.2d. If the developer accepted the left changes but a merge tool accepted all changes, causing the auto-merged code to contain both sides of the conflict, then the auto-merged code will have

18 lines of code and the manually merged code will have 14 lines. Git diff will report 4 different lines since the auto-merged code also contains right side of the conflict, which are 4 lines that the manually merged file does not contain. In this case, the recall will be 1 because subtracting the number of lines in the diff from the number of auto-merged lines will result in the same number of lines as the manually merged code. Although the precision will suffer, it will also still be high (78%) and will not reflect the fact that the tool failed to detect the conflict. In their threats, the `IntelliMerge` authors themselves recognize that using manually committed code as the ground truth is unreliable, because manually committed files often contain mistakes.

Instead, in RQ1, we report the number of conflicts each tool detects at various granularity levels (scenarios, files, and conflict regions). Additionally, we do not only report these numbers in isolation but instead report them at a scenario level to understand the proportion of scenarios in which each tool can improve the situation for a developer. Additionally, for RQ2, we manually sample merge conflicts that differ between the merge tools to understand the quality of the merge results and how the behavior of these tools differ in handling different types of merge scenarios. A similar analysis has been used in the past by Cavalcanti et al. [20] to get a better understanding of merge results.

# Chapter 6

# RQ1: Quantitative Tool Comparison

In this RQ, we compare the effectiveness of each tool in resolving merge conflicts at all granularity levels: complete merge scenarios, conflicting files, conflict blocks, and conflicting lines of code reported by each merge tool for the merge scenarios in our data set. We first focus on comparing the number of completely resolved conflicting scenarios. Completely resolving a conflicting scenario is the best case for any tool since this completely relieves the developer from looking at this scenario. While a tool may not be able to completely resolve a scenario, it may be able to reduce the number of conflicting files or conflicting regions a developer needs to deal with, or it may also reduce the size of the reported conflicts in terms of lines of code (LOC). We report the cases in which such reduction happens. Alternatively, a tool may worsen the situation for a developer where it actually complicates the conflict by reporting more conflicting files, blocks, or lines of code.

## 6.1 Completely Resolved Merge Scenarios

Table 6.1 shows the breakdown of the merge results for each project. The *Total Scenarios* column shows the number of conflicting Git scenarios evaluated for each project. We then show the results for `IntelliMerge` and `RefMerge`, respectively. For each tool, we show the number of completely resolved merge scenarios (columns *Resolved*), the number of merge scenarios where the conflict

Table 6.1: Breakdown of merge scenario results for each tool, compare to Git. The percentage in parentheses shows the proportion from total scenarios in each project. For each project, the tool that was able to completely resolve more merge scenarios is shown in bold.

| **Project Name** | **Total Scenarios** | **IntelliMerge** | | | | **RefMerge** | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **Resolved** | **Changed** | **Unchanged** | **Timeout** | **Resolved** | **Changed** | **Unchanged** | **Timeout** |
| cassandra | 922 | 41 (5%) | 88 (9%) | 3 (0%) | 790 (86%) | **84 (9%)** | 49 (5%) | 298 (33%) | 491 (53%) |
| elasticsearch | 178 | 3 (2%) | 99 (56%) | 5 (3%) | 71 (40%) | **8 (4%)** | 57 (32%) | 96 (54%) | 17 (10%) |
| gradle | 118 | 1 (1%) | 105 (89%) | 6 (5%) | 6 (5%) | **10 (8%)** | 38 (32%) | 70 (59%) | 0 (0%) |
| antlr4 | 100 | **1 (1%)** | 95 (95%) | 1 (1%) | 3 (3%) | **1 (1%)** | 28 (28%) | 71 (71%) | 0 (0%) |
| platform_fwk_support | 95 | 5 (5%) | 55 (58%) | 3 (3%) | 32 (34%) | **7 (7%)** | 25 (26%) | 63 (66%) | 0 (0%) |
| deeplearning4j | 93 | 3 (3%) | 83 (89%) | 6 (6%) | 1 (1%) | **5 (5%)** | 19 (20%) | 69 (74%) | 0 (0%) |
| realm-java | 92 | 7 (8%) | 69 (75%) | 14 (15%) | 2 (2%) | **8 (9%)** | 20 (22%) | 64 (70%) | 0 (0%) |
| jackson-core | 81 | 0 (0%) | 80 (99%) | 1 (1%) | 0 (0%) | **3 (4%)** | 28 (35%) | 50 (62%) | 0 (0%) |
| android | 81 | 3 (4%) | 73 (90%) | 5 (6%) | 0 (0%) | **8 (10%)** | 12 (15%) | 61 (75%) | 0 (0%) |
| cometd | 63 | 2 (3%) | 55 (87%) | 5 (8%) | 1 (2%) | **5 (8%)** | 11 (17%) | 47 (75%) | 0 (0%) |
| storm | 33 | **1 (3%)** | 29 (88%) | 3 (9%) | 0 (0%) | 0 (0%) | 5 (15%) | 28 (85%) | 0 (0%) |
| ProjectE | 30 | **1 (3%)** | 26 (87%) | 2 (7%) | 1 (3%) | 0 (0%) | 12 (40%) | 18 (60%) | 0 (0%) |
| javaparser | 23 | **3 (13%)** | 13 (57%) | 7 (30%) | 0 (0%) | 0 (0%) | 9 (39%) | 14 (61%) | 0 (0%) |
| druid | 17 | **2 (12%)** | 13 (76%) | 2 (12%) | 0 (0%) | 1 (6%) | 9 (53%) | 7 (41%) | 0 (0%) |
| androidannotations | 15 | **1 (7%)** | 14 (93%) | 0 (0%) | 0 (0%) | 0 (0%) | 4 (27%) | 11 (73%) | 0 (0%) |
| junit4 | 14 | **1 (7%)** | 12 (86%) | 1 (7%) | 0 (0%) | **1 (7%)** | 5 (36%) | 8 (57%) | 0 (0%) |
| MinecraftForge | 14 | **2 (14%)** | 10 (71%) | 0 (0%) | 2 (14%) | 0 (0%) | 6 (43%) | 8 (57%) | 0 (0%) |
| iFixitAndroid | 13 | 0 (0%) | 13 (100%) | 0 (0%) | 0 (0%) | **1 (8%)** | 7 (54%) | 5 (38%) | 0 (0%) |
| MozStumbler | 10 | 0 (0%) | 8 (80%) | 2 (20%) | 0 (0%) | **1 (10%)** | 6 (60%) | 3 (30%) | 0 (0%) |
| error-prone | 9 | **1 (11%)** | 7 (78%) | 1 (11%) | 0 (0%) | 0 (0%) | 3 (33%) | 6 (67%) | 0 (0%) |
| Total | 2,001 | 78 (4%) | 947 (46%) | 67 (4%) | 909 (46%) | **143 (7%)** | 353 (18%) | 997 (50%) | 508 (25%) |

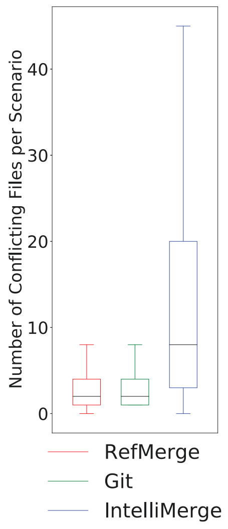
result changed from what Git reports (columns *Changed*), the number of merge scenarios where the merge conflict remains the same (columns *Unchanged*) and the number of merge scenarios where the tool times out (columns *Timeout*). Note that a change in the conflict result could mean either a decrease or increase in the number or size of the reported conflicts; we discuss the details of these changed scenarios in Section 6.1.1. Note that `RefMerge` times out on 508 merge scenarios across two different projects and `IntelliMerge` times out on 909 merge scenarios across nine different projects. The main reason `RefMerge` times out is because of long commit histories that slow `RefactoringMiner` down, often timing out before `RefMerge` begins inverting refactorings. As for `IntelliMerge`, the main contributor to timing out is larger merge scenarios where more changed files cause `IntelliMerge` to take significantly longer in the graph building step.

As the table shows across all evaluated merge scenarios, `IntelliMerge` was able to completely resolve 78 merge scenarios out of the 2,001 total scenarios (i.e., 4%) while `RefMerge` was able to completely resolve 143 (7%) scenarios. At a project level, `IntelliMerge` and `RefMerge` are able to completely resolve at least one scenario in a total of 17 (85%) and 14 (70%) of projects respectively. However, there are seven projects where `IntelliMerge` resolves more scenarios than `RefMerge`, while there are 11 projects where `RefMerge` resolves more scenarios than `IntelliMerge`.

### 6.1.1 Merge Scenarios with Differences in Conflicts

We now look at the *remaining* scenarios that the tools are not able to completely resolve, but for which the result of the conflict changed. We use Figures 6.1-6.3 to discuss these scenarios per project at the file, block, and lines of code levels respectively[1]. There are two parts to each figure. On the left-hand side, we provide a box plot of the overall distribution of reported conflicts at that granularity level for all three tools across all evaluated scenarios. On the right-hand side, we provide a table that zooms in on the conflicting scenarios from the *Changed* column of Table 6.1. For each granularity level (conflicting files, conflict blocks, and conflict size in terms of LOC), we show the number of scenarios for which a tool increased or decreased the resulting number of conflicts. For example, for the last project `error-prone` in Figure 6.1b, we can see that there are four scenarios that `IntelliMerge` reduced the number of conflicting files for, while it increased the number of conflicting files for three scenarios. The percentage shown in parentheses is the median reduction/increase per merge scenario in that project (or over all scenarios in the last row of the table). For example, if Git reports 4 conflicting files while a tool reports 2 conflicting files, then this is a $(4-2)/4 = 50\%$ reduction. In the example of `error-prone`, the median reduction of the number of conflicting files for the corresponding four scenarios is 46%. The same interpretation of the numbers can be used for all granularity levels, which we discuss in detail below. Ideally, even if a tool cannot completely resolve a scenario, it would be able to partially resolve some of the reported conflicts. For each project, we show in bold which tool achieves the most reduction and the least increase.

[1]We use platform_fwk_supp as a shortened version of platform_frameworks_support for better table sizing for readability



(a) Overall Distribution

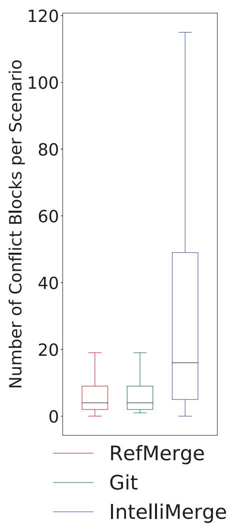
| Project | Reduced Confl. Files | | Increased Confl. Files | |
|---|---|---|---|---|
| | IntelliMerge | RefMerge | IntelliMerge | RefMerge |
| cassandra | 3 (33%) | **37 (50%)** | 68 (438%) | **0 (0%)** |
| elasticsearch | 4 (20%) | **15 (33%)** | 99 (260%) | **24 (55%)** |
| gradle | 6 (33%) | **10 (22%)** | 91 (500%) | **14 (50%)** |
| antlr4 | 2 (56%) | **6 (18%)** | 92 (445%) | **15 (17%)** |
| platform_fwk_supp | **5 (70%)** | 5 (4%) | 44 (375%) | **9 (18%)** |
| deeplearning4j | **6 (67%)** | 0 (0%) | 59 (160%) | **9 (80%)** |
| realm-java | **10 (39%)** | 4 (25%) | 46 (181%) | **5 (50%)** |
| jackson-core | 0 (0%) | **6 (33%)** | 79 (850%) | **11 (50%)** |
| android | **6 (38%)** | 2 (30%) | 55 (167%) | **1 (10%)** |
| cometd | 2 (47%) | **3 (25%)** | 48 (658%) | **3 (8%)** |
| storm | **3 (33%)** | 1 (33%) | 22 (345%) | **1 (100%)** |
| ProjectE | **4 (12%)** | 2 (45%) | 24 (79%) | **6 (7%)** |
| javaparser | **5 (43%)** | 4 (59%) | 8 (150%) | **3 (50%)** |
| druid | **8 (46%)** | 2 (33%) | **0 (0%)** | 1 (50%) |
| androidannotations | 1 (67%) | **3 (50%)** | 13 (100%) | **1 (10%)** |
| junit4 | **5 (33%)** | 0 (0%) | 7 (150%) | **5 (50%)** |
| MinecraftForge | 1 (10%) | **1 (17%)** | 9 (100%) | **2 (149%)** |
| iFixitAndroid | **5 (81%)** | 4 (93%) | 5 (78%) | **3 (1500%)** |
| MozStumbler | 1 (25%) | **2 (37%)** | 5 (50%) | **1 (20%)** |
| error-prone | **4 (46%)** | 2 (25%) | 3 (750%) | **0 (0%)** |
| Total | 81 (38%) | **109 (50%)** | 777 (333%) | **120 (33%)** |

(b) Breakdown by merge scenario

Figure 6.1: Conflicting *files* per merge scenario.

### 6.1.2 Conflicting Files

We first look at the conflicting file level in Figure 6.1. Figure 6.1a shows the distribution of the number of reported conflicting files per merge scenario. The figure shows that Git and `RefMerge` have a median number of two conflicting files while `IntelliMerge` has a median of eight. However, such a plot does not give us any indication about the developer experience on a scenario level, when it compares to what they currently experience with Git. To understand the tool's behavior on a scenario level, we look at the table in Figure 6.1b, which shows the number of scenarios for which each tool results in an increase or decrease in the number of conflicting files. Overall, the table shows that `IntelliMerge` reduces the number of reported conflicting files in 81 scenarios (4% of all evaluated scenarios) by a median 38% reduction. On the other hand, `IntelliMerge` increases the number of reported conflicting files in 777 scenarios (39%) by a median 333% increase. In other words, on average, `IntelliMerge` increases the number of conflicting files by three-fold in these scenarios. `RefMerge` reduces the number of reported conflicting files for 109



(a) Overall Distribution

| Project | Reduced Confl. Blocks | | Increased Confl. Blocks | |
|---|---|---|---|---|
| | IntelliMerge | RefMerge | IntelliMerge | RefMerge |
| cassandra | 5 (46%) | **11 (14%)** | 83 (500%) | **38 (25%)** |
| elasticsearch | 4 (28%) | **22 (33%)** | 95 (271%) | **35 (25%)** |
| gradle | 10 (33%) | **18 (19%)** | 95 (433%) | **20 (45%)** |
| antlr4 | 3 (20%) | **7 (5%)** | 92 (479%) | **21 (13%)** |
| platform_fwk_supp | 8 (51%) | **9 (25%)** | 47 (517%) | **16 (13%)** |
| deeplearning4j | **20 (42%)** | 6 (9%) | 63 (175%) | **13 (29%)** |
| realm-java | **22 (50%)** | 13 (21%) | 47 (200%) | **7 (40%)** |
| jackson-core | 0 (0%) | **11 (33%)** | 80 (1522%) | **17 (38%)** |
| android | **21 (40%)** | 3 (17%) | 52 (124%) | **9 (33%)** |
| cometd | 3 (23%) | **6 (25%)** | 52 (809%) | **5 (11%)** |
| storm | 3 (25%) | **4 (25%)** | 26 (233%) | **1 (6%)** |
| ProjectE | **4( 15%)** | **4 (20%)** | 22 (94%) | **8 (11%)** |
| javaparser | 3 (90%) | **5 (80%)** | 10 (369%) | **4 (21%)** |
| druid | **13 (67%)** | 6 (50%) | **0 (0%)** | 3 (8%) |
| androidannotations | 1 (89%) | **2 (58%)** | 13 (114%) | **2 (17%)** |
| junit4 | **5 (60%)** | 0 (0%) | 7 (233%) | **5 (25%)** |
| MinecraftForge | 1 (5%) | **3 (19%)** | 9 (100%) | **3 (33%)** |
| iFixitAndroid | **6 (57%)** | 3 (96%) | 7 (100%) | **4 (583%)** |
| MozStumbler | 3 (50%) | **4 (24%)** | 5 (86%) | **2 (21%)** |
| error-prone | 2 (91%) | **3 (33%)** | 5 (417%) | **0 (0%)** |
| Total | 137 (50%) | **140 (25%)** | 810 (367%) | **213 (25%)** |

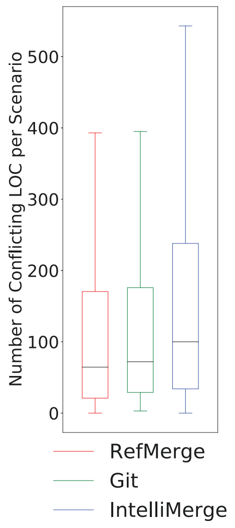
(b) Breakdown by merge scenario

Figure 6.2: Conflicting *blocks* per merge scenario.

scenarios (5%) by a median 50% reduction while it increases the number of reported conflicting files for 120 scenarios (7%) by a median 33% increase.

## 6.1.3 Conflict Blocks

We now look at the conflict block level in Figure 6.2. The number of conflict blocks indicates the number of individual conflicting regions a developer needs to deal with. Figure 6.2a shows that Git and `RefMerge` have almost the same overall distribution of number of conflicting blocks per merge scenario (with a median of 4). However, `IntelliMerge` has a much higher median number of conflicting blocks at 16. Zooming in on the breakdown of increased and reduced conflict blocks in Figure 6.2b, we find that `IntelliMerge` reduces the number of reported conflict blocks for 137 scenarios (7%) by a median 50% reduction, while it increases the number of reported conflict blocks for 810 scenarios (40%) by a median of 367%. On the other hand, `RefMerge` reduces the number of reported conflicts in 140 scenarios (7%) by a median 25% reduction and increases the number of reported conflicts for 213 scenarios



(a) Overall Distribution

| **Project** | **Reduced Confl. LOC** | | **Increased Confl. LOC** | |
|---|---|---|---|---|
| | **IntelliMerge** | **RefMerge** | **IntelliMerge** | **RefMerge** |
| cassandra | **25 (39%)** | 21 (9%) | **19 (164%)** | 37 (25%) |
| elasticsearch | 38 (45%) | **48 (17%)** | 65 (118%) | **41 (28%)** |
| gradle | **38 (45%)** | 33 (19%) | 73 (190%) | **28 (47%)** |
| antlr4 | 20 (27%) | **23 (4%)** | 75 (102%) | **39 (10%)** |
| platform_fwk_supp | **24 (58%)** | 27 (16%) | 33 (130%) | **24 (10%)** |
| deeplearning4j | **45 (45%)** | 13 (10%) | 43 (148%) | **15 (20%)** |
| realm-java | **45 (45%)** | 13 (10%) | 43 (148%) | **15 (20%)** |
| jackson-core | 8 (47%) | **20 (9%)** | 72 (430%) | **22 (42%)** |
| android | **54 (58%)** | 10 (6%) | 23 (119%) | **7 (44%)** |
| cometd | **15 (51%)** | 9 (24%) | 43 (241%) | **10 (9%)** |
| storm | **14 (39%)** | 9 (30%) | 16 (119%) | **1 (1%)** |
| ProjectE | **18 (45%)** | 13 (11%) | 10 (75%) | **9 (9%)** |
| javaparser | **12 (59%)** | 6 (40%) | **7 (275%)** | **7 (21%)** |
| druid | **15 (85%)** | 10 (7%) | **0 (0%)** | 1 (57%) |
| androidannotations | 4 (87%) | **6 (37%)** | 10 (42%) | **6 (46%)** |
| junit4 | **10 (65%)** | 3 (6%) | **3 (18%)** | 5 (33%) |
| MinecraftForge | **4 (27%)** | **4 (13%)** | **4 (150%)** | **4 (58%)** |
| iFixitAndroid | **7 (34%)** | 4 (78%) | **5 (55%)** | **5 (74%)** |
| MozStumbler | **7 (56%)** | 5 (34%) | 3 (38%) | **2 (17%)** |
| error-prone | **3 (84%)** | **3 (33%)** | 5 (187%) | **4 (65%)** |
| Total | **414 (51%)** | 287 (13%) | 584 (164%) | **274 (23%)** |

(b) Breakdown by merge scenario

Figure 6.3: Conflicting *LOC* per merge scenario.

(11%) by a median of 25% increase.

### 6.1.4 Conflicting Lines of Code

Finally, we look at the conflicting lines of code (LOC) in Figure 6.3, which measures the total number of lines in all conflict blocks/regions of a merge scenario. From Figure 6.3a, we observe similar behavior of the tools as what we observed for the conflicting files in Figure 6.1a. More closely from the table in Figure 6.3b, we find that `IntelliMerge` reduces the number of conflicting LOC in 414 scenarios (21%) by a median 51% reduction, while it increases the conflicting LOC for 584 (29%) scenarios by a median 164% increase. `RefMerge` reduces the conflicting LOC in only 287 scenarios (14%) by a median 13% reduction and increases the conflicting LOC in 274 scenarios (14%) by a median 23% increase. Note that the discrepency between `IntelliMerge`'s increase rate for conflicting regions and conflicting loc suggests that while `IntelliMerge` results in a lot more conflicting regions than Git, the size of these conflicting regions is small. To confirm this, we show the distribution of the reported

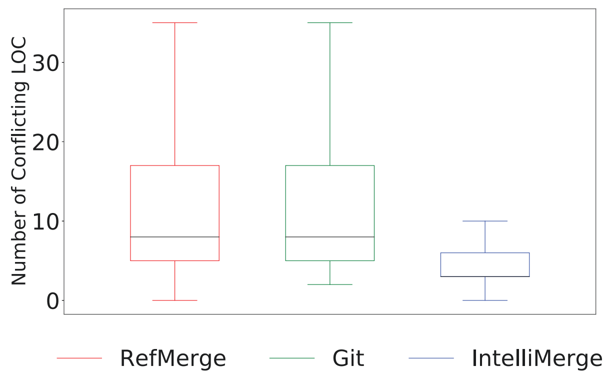


Figure 6.4: Conflicting lines of code per conflict block.

conflicting loc per block (rather than over a whole scenario) in Figure 6.4. The plot confirms that the conflict regions that `IntelliMerge` reports are indeed quite small, even if they are much more frequent than the other tools.

## 6.2 Interpretation of RQ1 Results

The above results indicate that `RefMerge` completely resolves about twice as many merge scenarios as `IntelliMerge` (143 versus 78). While `IntelliMerge` is able to reduce conflicting LOC for a higher portion of scenarios than `RefMerge` (51% versus 13%), this comes at a cost of a high increase in the reported conflicts across all granularity levels for a large portion of the merge scenarios. Additionally, `IntelliMerge` times out on a higher number of merge scenarios than `RefMerge`. Thus, it seems `IntelliMerge` works extremely well for a small proportion of scenarios where it is able to highly reduce the resulting conflicts in a scenario, but actually makes it much worse for other scenarios. Specifically, taking the total number of scenarios it can completely resolve (78 from Table 6.1) and the ones in which it can reduce the total number of conflicting LOC for (414 from Figure 6.3b), `IntelliMerge` can help the developer deal with less conflicts in 492 scenarios (25% of the overall scenarios). However, taking both timeouts (909 scenarios from Table 6.1) and worsened results in terms of overall conflicting LOC (584 scenarios from Figure 6.3b), `IntelliMerge` will not help the developer in the remaining 1,493 (75%) of the scenarios, and will in fact make it worse.

On the other hand, `RefMerge` can completely resolve or reduce the number of conflicting LOC for only 430 scenarios (22%). However, `RefMerge` worsens the situation at a much lower rate than `IntelliMerge` in only 782 (39%). Additionally, the median percentage increase for `RefMerge` in terms of conflicting LOC is much lower at 23% as opposed to 164% for `IntelliMerge`. Thus, `RefMerge` makes the situation worse for the developer both in a smaller proportion of merge scenarios *and* by a lower percentage increase. Note that the number of unchanged merge scenarios for `RefMerge` is also much higher than `IntelliMerge`, because by construction, `RefMerge` resorts to a regular Git merge when there are no supported refactorings for it to work with. Overall, our quantitative results show that each tool has its pros and cons, and it is obvious that the characteristics or difficulty of a merge scenario impact the results in some way. This is why we perform a qualitative analysis of these discrepancies in RQ2 to understand the strengths and weaknesses of each tool, as well as the characteristics of merge scenarios that cause them to fail.

*RQ1 Summary:* `IntelliMerge` completely resolves 78 (4%) of the merge scenarios while `RefMerge` completely resolves 143 (7%). For scenarios the tools cannot completely resolve, `IntelliMerge` reduces the overall conflicting LOC in 414 scenarios (21%) by a median 51% reduction while it increases it in 584 scenarios (29%) by a median 164% increase. `RefMerge` reduces the conflicting LOC in 287 scenarios (14%) by a median 13% reduction and increases it for 274 scenarios (14%) by only 23% increase.

# Chapter 7

# RQ2: Discrepancies Between the Tools

The quantitative numbers described in RQ1 are valuable for determining if a merge tool reports less conflicts. However, these numbers do not provide us information about the *quality* of the resolutions the tools provide. For example, a merge tool could report no merge conflicts in a merge scenario where conflicts should be reported. Similarly, we do not know if the reported conflicts are real conflicts or not. Thus, we perform a qualitative study for RQ2 to dig deeper into the reported results.

## 7.1 Research Method

### 7.1.1 Sampling Criteria

We manually analyze a sample of 50 merge scenarios to shed light on the strengths and weaknesses of each tool. We randomly sample the 50 merge scenarios across the following criteria: (1) `IntelliMerge` and `RefMerge` produce similar results, in terms of completely resolving the merge scenario, or equally increasing/reducing the number of Git conflicts. (2) `IntelliMerge` outperforms `RefMerge` in terms of completely resolving the scenario or reporting a lower number of conflicts at any granularity level and (3) `RefMerge` outperforms `IntelliMerge`. When sampling the merge scenarios, we also try to evenly sample across projects.

Table 7.1: Comparing the false positives and false negatives reported by each tool, across the 50 sampled scenarios.

| | **RefMerge** | **Git** | **IntelliMerge** |
|---|---|---|---|
| # Conflict Blocks Investigated | 432 | 453 | 1,243 |
| True Positives | 199 | 198 | 142 |
| False Positives | 231 | 243 | 923 |
| False Negatives | 2 | 12 | 178 |

### 7.1.2 Analysis Method

The goal of our manual analysis is to analyze the conflicts reported by all three tools across the sampled scenarios. To investigate if a merge conflict is a true positive or false positive, we look at the code region in the base commit, left commit, and right commit. We determine whether integrating the changes from both parents should result in a merge conflict, based on the semantics of the changes. If a merge conflict is expected, we label this conflict region as a *true positive*. If it should not result in a merge conflict, we label it as a *false positive*. If the other merge tools do not report the same merge conflict, we investigate the result of their merge and decide if it is a *true negative* (i.e., conflict should not be reported) or *false negative* (i.e., the tool missed the conflict). Additionally, we investigate and categorize the reasons behind false positives and false negatives for each tool. This process takes an average of 69 minutes per merge scenario.

## 7.2 Results

Table 7.1 shows the total number of conflict blocks that we analyze across the 50 sampled scenarios, as well as the number of false positives and false negatives that we find for each tool. As shown, Git reports 243 false positives and 12 false negative. `IntelliMerge` reports 923 false positives and 178 false negatives. Meanwhile, `RefMerge` reports 231 false positives and two false negatives. When compared to Git, `RefMerge` reduces the number of false positives and false negatives by 5% and 83% respectively, while `IntelliMerge` in-

Table 7.2: The reason for each false positive and false negative reported by Git, as well as the frequency for each reason.

| Git Reasons | Type | Frequency |
|---|---|---|
| No Refactoring Handling | False Positive | 157 |
| Ordering Conflict | False Positive | 75 |
| Formatting Conflict | False Positive | 11 |
| No Refactoring Handling | False Negative | 12 |
| Total | | 257 |

creases the number of false positives and false negatives by 279% and 1,383%. We also show the number of true positives reported by each tool. While Git and `RefMerge` report a total of 198 and 199 true positives respectively, `IntelliMerge` reports only 142 true positives.

## 7.2.1 Git Results

Table 7.2 shows the reasons behind the false positives and false negatives for Git. There were generally three main reasons for false positives. The most prevalent reason for Git's false positives is not being able to handle refactorings, and thus reporting conflicts that could be resolved automatically. There are 157 (65%) false positive conflicts that Git reports which involve refactorings. Given the selection of merge scenarios we use in our evaluation, it is natural to find that many of the conflicts Git reports are related to refactorings. Table 7.2 also shows that 75 (31%) of Git's reported false positives are due to ordering conflicts. An *ordering conflict* is a conflict caused by adding two program elements to the same location and the merge tool not knowing which order to put them in, when the order does not matter [12]. For example, if Bob and Alice add two new Java methods to the same location, Git will report a conflict when both methods can be added in either order. Finally, the remaining 11 (5%) of Git's false positives are *formatting conflicts* that are caused by different formatting between branches. This could be an additional white space or a new line on one branch that does not exist on the other.

Not being able to handle refactorings causes Git to miss 12 conflicts (false

Table 7.3: The reason and frequency for false positives and false negatives reported by `RefMerge`.

| `RefMerge` **Reasons** | **Type** | **Frequency** |
|---|---|---|
| Unsupported Refactoring | False Positive | 88 |
| Ordering Conflict | False Positive | 75 |
| Refactoring-related Ordering Conflict | False Positive | 26 |
| Refactoring-related Formatting Conflict | False Positive | 21 |
| Formatting Conflict | False Positive | 11 |
| IntelliJ Optimization | False Positive | 5 |
| Undetected Refactoring | False Positive | 3 |
| Fails to Invert Refactoring | False Positive | 2 |
| Fails to Replay Refactoring | False Negative | 2 |
| Total | | 233 |

negative), where a pair of `Rename Class` refactorings performed on each branch cause Git to miss several conflicting regions and result in syntax errors.

### 7.2.2 `RefMerge` Results

Table 7.3 shows the reasons behind the false positives and negatives for `RefMerge`. Similar to Git, `RefMerge` also suffers from being unable to resolve ordering and formatting conflicts, reporting the same 75 ordering conflicts and 11 formatting false positives as Git. `RefMerge` reports an additional 26 ordering conflicts and 21 formatting conflicts that arise from its refactoring handling, totalling 101 (44%) ordering conflicts and 32 (14%) formatting conflicts. The majority of these additional ordering conflicts are caused by `Move Method` and `Move Inner Class` refactorings being moved to the correct class but not being moved to the correct location within the file. Instead of reporting the original move-related refactoring conflict, `RefMerge` reports a much larger ordering conflict containing several methods and classes. The additional formatting conflicts are caused by formatting differences from inverting refactorings, also typically observed with `Move Method` and `Move Inner Class` refactorings. In these conflicts, `RefMerge` resolves the refactoring conflict but leaves behind a small conflict that usually consists of different amounts of white space.

In 88 (38%) of the false positives that `RefMerge` reports, the underlying issue is a refactoring that is not supported in the current implementation. For example, merge scenario `ea42d642` within *gradle* has an `Add Parameter` refactoring that is involved in one of the reported conflicts.

There are five (2%) false positives caused by *IntelliJ optimizations*, which are automatic optimizations done to the code after using the refactoring engine. All five of the IntelliJ optimizations were caused by inverting refactorings that were not involved in the original refactoring conflicts reported by Git. An example of this is replacing several import statements with `import package.*`, which then cause Git to detect a conflict in the merging step.

There are three (1%) false positives that are due to *undetected refactorings* that RefactoringMiner did not detect. There are several similar methods, both in structure and naming, in the classes where RefactoringMiner misses a refactoring, which likely made it difficult for RefactoringMiner to detect the refactoring. We reported the issue to the RefactoringMiner developers.

The remaining two (1%) of `RefMerge`'s false positives are refactoring conflicts that `RefMerge` fails to resolve because it could not invert a `Move Method` refactoring. After investigating, we found that an additional unsupported refactoring was performed on the method, causing `RefMerge` to be unable to provide the correct information to the refactoring engine.

Finally, `RefMerge` reports two false negatives. These false negatives are caused by successfully inverting the refactoring and resolving the conflict but failing to replay the refactoring. Although this does not result in a syntax error, this is not a result that the developer would expect. This happens for a `Move Method` refactoring and an `Extract Method` refactoring. Replaying the merges resulted in null pointer exceptions, which suggests this is caused by a bug in our implementation, which we will further investigate.

### 7.2.3 `IntelliMerge` Results

Table 7.4 shows the reasons behind the false positives and negatives for `IntelliMerge`. We start with some of the reasons we already observed for the other tools. `IntelliMerge` reports 25 false positives due to ordering conflicts and also has 74 false positives because of undetected refactorings. There are

Table 7.4: The reason and frequency of false positives and false negatives reported by `IntelliMerge`.

| `IntelliMerge` **Reasons** | **Type** | **Frequency** |
|---|---|---:|
| Matching Error | False Positive | 814 |
| Undetected Refactoring | False Positive | 74 |
| Ordering Conflict | False Positive | 25 |
| Missing Refactored Reference | False Positive | 7 |
| Incorrectly Detected Refactoring | False Positive | 2 |
| Formatting Conflict | False Positive | 1 |
| Deletes Conflict Block | False Negative | 87 |
| Matching Error | False Negative | 75 |
| Incorrectly Detected Refactoring | False Negative | 16 |
| Total | | 1,101 |

12 refactoring types where `IntelliMerge` fails to detect a refactoring with the top three being `Add Parameter` (19), `Rename Parameter` (15), and `Extract Method` (8). The undetected refactorings can be split into two groups: (1) where there are several similar program elements and a refactoring drops the program element below the similarity threshold, and (2) where several changes to a program element cause `IntelliMerge` to think that the refactored program element is an addition.

Note that, unlike Git and `RefMerge`, `IntelliMerge` reports only one false positive related to formatting conflicts. However, 814 of `IntelliMerge`'s false positives (88%) are due to matching errors. We define a *matching error* as an error caused by `IntelliMerge`'s graph node matching process. This primarily happens with comments, annotations, and imports.

There are seven false positives that are due to *missing refactoring references*, where `IntelliMerge` detects a refactored program element but fails to match added references on the other branch. Both branches added a new reference in the same location, causing `IntelliMerge` to think that a program element was refactored while an addition was made to the same location which resulted in a conflict.

The last two false positives that `IntelliMerge` reports are caused by incorrectly detecting a refactoring that was never performed. `IntelliMerge` incorrectly performs an `Add Parameter` refactoring to a method causing merge conflicts whenever the method is called.

`IntelliMerge` results in 87 (49% of all false negatives) false negatives because it *deletes conflict blocks*, incorrectly deleting code that exists in conflict blocks reported by Git and `RefMerge`. This primarily happens when `IntelliMerge` deletes the file containing a merge conflict. When a file or program element exists in the base commit and `IntelliMerge` cannot find a match for it in either parent commit, `IntelliMerge` seems to incorrectly delete it. Note that this is a lower bound for how many times `IntelliMerge` results in a false negative. `IntelliMerge` could have deleted other files or program elements that were not part of a conflict block and since we focused on the reported conflicts by each tool, we would have missed this happening in files where no merge tool reported a conflict.

We find that 75 (42% of all false negatives) of `IntelliMerge`'s false negatives are due to matching errors which lead to syntax errors. Most of the syntax errors seem to happen in classes that contain several method-level refactorings and several similar method declarations.

Finally, the 16 (9%) remaining false negatives are due `IntelliMerge` detecting refactorings that were not performed, leading to `IntelliMerge` moving methods to classes that the developers never moved them to and causing additional syntax errors. This causes `IntelliMerge` to not detect any conflicts where conflicts should have occurred.

## 7.3 Interpretation of RQ2 Results

The above results indicate that `RefMerge` reports about the same amount of false positives as Git (231 versus 243) while `IntelliMerge` increases the number of false positives by almost three-fold (923 versus 243). Unlike Git and `RefMerge`, `IntelliMerge` does well with ordering and formatting conflicts due to its graph-based approach. While `IntelliMerge` also decreases the number of refactoring conflicts a developer needs to deal with, this comes at the price of many more false positives: 814 of `IntelliMerge`'s false positives are matching errors which are typically small in size. This explains the quantitative results of RQ1 where `IntelliMerge` reports more conflicts but less conflicting LOC. Additionally, while `IntelliMerge` does not detect refactorings in 74 conflict blocks reported by Git (struggling the most with parameter level refactorings), `IntelliMerge` typically does well with the refactoring conflicts it does detect. However, `IntelliMerge` also incorrectly detects 18 refactorings (two false positives and 16 false negatives from Table 7.4) and reports a total of 178 false negatives. Thus, while the results of RQ1 suggest that `IntelliMerge` works extremely well for a small proportion of scenarios where it is able to highly reduce the resulting conflicts in a scenario, our qualitative results suggest that some of these may actually be false negatives.

On the other hand, `RefMerge` nearly eliminates false negatives, reducing them by 83%, while reducing the number of false positives reported by 5% when compared to Git. `RefMerge` worsens the situation at a much lower rate than `IntelliMerge`, reporting 52 false positives that Git does not, and 21 of them are formatting conflicts left after resolving a refactoring conflict. In general, `RefMerge` struggles most with move-related refactorings.

*RQ2 Summary:*
`RefMerge` reduces the number of false positives and false negatives by 5% and 83% respectively, while `IntelliMerge` increases them by 279% and 1,383%. `RefMerge` struggles most with `Move Method` whereas `IntelliMerge` struggles most with `Add Parameter`, `Rename Parameter`.

# Chapter 8

# Discussion

In this thesis, we compared two refactoring-aware merging approaches that have not been compared before. RQ1 results show that despite supporting less refactorings, `RefMerge` managed to resolve about twice as many conflicting merge scenarios as `IntelliMerge`. We found that while `IntelliMerge` reduced the number of conflicting LOC in more scenarios compared to `RefMerge`, `IntelliMerge` also increased the number of conflicting LOC in more scenarios. On the other hand, `RefMerge` makes the situation worse in a smaller proportion of merge scenarios and by a lower percentage increase. Additionally, our qualitative analysis shows that `IntelliMerge` reported a much higher number of false negatives whereas `RefMerge` reported only two false negatives in all 50 merge scenarios. Thus, even considering unsupported refactorings, operation-based refactoring-aware merging shows promise to help improve the developers' experience without the risk of increasing the number of false negatives. In other words, the results in this study are a lower bound of its potential.

## 8.1 Strengths and Weaknesses of`IntelliMerge`

The nature of `IntelliMerge`'s graph-based approach makes it avoid formatting and ordering conflicts. However, `IntelliMerge` seems to struggle with correctly matching graph nodes across the two versions of the code. We believe that `IntelliMerge`'s use of a similarity score for its refactoring detection is one of the main reasons for this. `IntelliMerge` often failed to detect a refactoring because the refactored program element was too similar to other existing program elements. We also found cases where a non-refactoring change caused a program element to be within the similarity threshold of other program elements, causing `IntelliMerge` to treat it as a refactoring. Although `IntelliMerge` could potentially change the used similarity threshold, the use of a similarity score will always run into these problems. `IntelliMerge` also results in several files being deleted. Upon further examination and contacting the `IntelliMerge` authors, `IntelliMerge` sometimes does not detect a match for a class between a parent commit and the base commit. This leads to it thinking that the class is deleted by one side, so it deletes the class when merging the three versions. This process results in the file containing the class to be deleted if there are no other classes within the file. However, when `IntelliMerge` successfully detects a refactoring, it handles it almost always handles it correctly and resolves the conflict.

## 8.2 Strengths and Weaknesses of `RefMerge`

Whereas `IntelliMerge`'s graph-based approach makes it avoid formatting and ordering conflicts, `RefMerge`'s operation-based approach is more prone to such conflicts. While formatting conflicts are a small price to pay considering they are typically easier to resolve than refactoring conflicts, move-related refactorings proved to be conceptually challenging when it comes to undoing/redoing them. Although `RefMerge` can move the program element to the correct class, it cannot guarantee that it is moved to the same location it was previously at. This happens because the IntelliJ refactoring engine moves the method to a location alphabetically and this is usually not the textual location it was moved from. We attempted to use information provided by RefactoringMiner to move it to the correct location textually; however other changes made on the same branch such as method additions or deletions still lead to it being moved to the wrong location. Despite this, overall, `RefMerge` resolves or simplifies more refactoring conflicts than the complications it creates, all while avoiding syntax errors.

## 8.3 Moving Forward

Driven by these findings, we propose a few paths moving forward in refactoring-aware merging. We believe that improvements in graph-based refactoring-aware merging require addressing the matching algorithm. The current merging algorithm `IntelliMerge` uses seems to work well, but the initial matching phase can be improved by avoiding the similarity score matching and instead using a refactoring detection algorithm such as that used in RefactoringMiner [60]. Additionally, further investigation and resolving the deleting files limitation is necessary to continue this line of work.

We believe that operatoin-based refactoring-aware merging shows very promising results, despite a small list of supported refactorings. Future work could go in three different directions: (1) adding support for more refactoring types, (2) handling move-related ordering conflicts using light-weight program analysis to determine where the program element was in the base commit, and (2) treating add and delete edits as operations which could help resolve the move-related ordering conflicts as well as improving conflict detection.

Finally, it could make sense to combine the two refactoring-aware approaches in some way similar to how changing strategies/auto-tuning between semi-structured and structured merge was previously proposed [11]. As the nature of graph-based merging seems to do well with ordering conflicts and formatting conflicts, this would address the weaknesses of operation-based merging. However, `IntelliMerge`'s limitations in terms of its matching algorithm would need to be addressed before this path could be considered further.

# Chapter 9

# Threats to Validity

We explain the potential threats to the validity of our results.

## 9.1 Construct Validity

By comparing implementations with different numbers of supported refactorings, the quantitative comparison could be unfair. On one hand, it could be unfair to `IntelliMerge` because by supporting more than double the refactorings compared to `RefMerge`, `IntelliMerge` has more room for error. On the other hand, it could be unfair to `RefMerge` because `RefMerge` has less potential to improve the scenarios. While this could threaten the validity of our evaluation, the evaluation is focused on a practical evaluation of the current capabilities of each tool. As each tool promises to be able to handle the refactorings they support, our evaluation looks at that and reports what their outputs are. The qualitative analysis alleviates these concerns and dives into the reasons by looking at the results more closely. All of the reasons for `IntelliMerge` seem to point to graph matching being `IntelliMerge`'s core issue, not necessarily because it handles extra refactorings poorly. Similarly, false positives for `RefMerge` actually point to unsupported refactorings so if anything, the evaluation is unfair to `RefMerge` and yet `RefMerge` showed positive results. Further analysis of only the merge scenarios that contain refactorings supported by both tools would be necessary to compare how both tools handle the same set of refactorings. Additionally, adding support for all of the refactorings that `IntelliMerge` supports to `RefMerge` and re-running the comparison would is another strategy to compare against the same set of refactorings.

In our qualitative analysis, we manually compare the results of the three tools to identify false positives and false negatives. This means we may miss false negatives that all three tools fail to report. Additionally, the analysis was done by a single author and is thus subject to their understanding of the scenario. To alleviate this as much as possible, we compare the changes in the left parent, right parent, and base commit for each merge scenario to first try to understand the developer's intentions and the expected merge result. We record a detailed description of our interpretation of the scenario and conflicts and share this in our artifact to allow further external validation. Further analysis involving investigating run time and compile time errors could also further shed further light on false negatives reported by the three approaches.

## 9.2 Internal Validity

Any problems inherited from the tools used in `RefMerge` or in our evaluation setup may lead to inaccuracies in the results. To mitigate this, we carefully consider the role of each tool used in our study and analyze its results through manual verification. While not a bug with IntelliJ per se, our qualitative analysis showed that IntelliJ's refactoring engine, which we use to invert and replay refactorings, performs optimizations that lead to unnecessary merge conflicts. This means that the reported number of conflicts in our results is an upper bound and with engineering effort and help from the IntelliJ developers to allow us to disable these optimizations, these limitations can be mitigated. Alternatively, a different refactoring engine that does not force these optimizations can be used. Any refactoring that RefactoringMiner misses will not be inverted and replayed, which will result in the same merge as Git. Any refactorings that RefactoringMiner detects which were not performed will result in RefMerge inverting and replaying a "fake" refactoring, which may lead to unnecessary merge conflicts. During our development, we came across some such occurrences and the RefactoringMiner author fixed these in the tool. In our qualitative analysis, we came across three refactorings that RefactoringMiner did not detect, which we recently reported. Overall, RefactoringMiner achieves a precision of 98% and 87% recall [59]. In general, it is important for `RefMerge` to rely on a tool with high precision to ensure we do not result in unnecessary conflicts. A lower recall simply means `RefMerge` will result in the same resolution as Git.

## 9.3 External Validity

By selecting sample projects with different sizes and refactoring histories, we try our best to have a representative evaluation. Our evaluation is limited to Java open-source projects since both tools are Java specific. That said, while our implementation of `RefMerge` is Java specific, an operation-based approach does not need to be. Our qualitative analysis is based only on a sample of 50 merge scenarios due to the time consuming nature of the process (avg. 69min/scenario). However, the 50 merge scenarios we investigated have more than 1,300 unique merge conflicts. As far as we are aware, this is the most extensive qualitative analysis performed in terms of unique merge conflicts [11], [20], [54] Naturally, investigating additional merge scenarios could reveal more for each tool.

# Chapter 10

# Conclusion

In modern software development, version control systems play a crucial role in enabling developers to collaborate on large projects. Most modern version control systems use unstructured merging techniques that do not understand code-change semantics such as refactorings. There are primarily two lines of work that specifically focus automating refactoring conflict resolution: (1) operation-based refactoring-aware merging, originally proposed by Dig et al. [24] and (2) a graph-based refactoring-aware merging called `IntelliMerge` [55]. While the evaluation results in both of their respective publications seem promising, the two refactoring-aware approaches have never been compared.

To compare the two refactoring-aware techniques, we first develop and present `RefMerge`, an operation-based refactoring-aware merging tool that works with Git history. As complex refactorings such as extract method have been proposed to be conceptually challenging for operation-based merging [55], we added support for extract method and inline method.

To evaluate the two refactoring-aware techniques, we perform a large scale quantitative comparison of the effectiveness of operation-based refactoring-aware merging implemented in `RefMerge` versus graph-based refactoring-aware merging implemented in `IntelliMerge`. Our evaluation consists of 2,001 merge scenarios from 20 open-source projects. We find that while `IntelliMerge` reduces the individual number of conflicts at a higher level than `RefMerge`, `RefMerge` resolves almost twice as many conflicting merge scenarios.

We additionally perform a systematic qualitative comparison of the strengths and weaknesses of both techniques through a manual analysis of their results across 50 merge scenarios. Our findings suggest that the nature of `IntelliMerge`'s graph-based approach makes it avoid formatting and ordering conflicts. Furthermore, when `IntelliMerge` successfully detects a refactoring, it almost always handles it correctly. However, `IntelliMerge` seems to struggle with correctly matching graph nodes across the two versions of code, resulting in almost three times as many unnecessary conflicts and more than 10 times as many false negatives. On the other hand, `RefMerge`'s operation-based approach seems to be more prone to formatting and ordering conflicts. Despite this, `RefMerge` resolves and simplifies more refactoring conflicts than the complications it creates, all while nearly eliminating false negatives.

Further work can explore adding support for more refactoring types, such as parameter and field level refactorings. Another possible direction could be to collaborate with IntelliJ developers to resolve the conflicts caused by IntelliJ optimizations. Additionally, extending the work to treat edits as operations could improve refactoring conflict resolution as well as other types of conflicts. Finally, resolving refactoring-related ordering conflicts that our implementation of operation-based refactoring-aware merging cause would greatly improve the results.

# References

[1] [Online]. Available: https://github.com/.

[2] [Online]. Available: https://www.mercurial-scm.org/.

[3] [Online]. Available: https://subversion.apache.org/.

[4] [Online]. Available: https://github.com/ualberta-smr/RefactoringAwareMerging.

[5] [Online]. Available: https://www.eclipse.org/.

[6] [Online]. Available: https://github.com/ameyaKetkar/TypeChangeMiner/blob/master/Dataset.zip.

[7] [Online]. Available: https://github.com/Symbolk/IntelliMerge.

[8] M. Ahmed-Nacer, P. Urso, and F. Charoy, "Evaluating Software Merge Quality," in *18th International Conference on Evaluation and Assessment in Software Engineering*, London, United Kingdom: ACM, May 2014, p. 9. DOI: 10.1145/2601248.2601275. [Online]. Available: https://hal.inria.fr/hal-00957168.

[9] E. L. Alves, M. Song, and M. Kim, "Refdistiller: A refactoring aware code review tool for inspecting manual refactoring edits," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 751–754.

[10] 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*, 2012, pp. 120–129.

[11] ——, "Structured merge with auto-tuning: Balancing precision and performance," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 120–129. DOI: 10.1145/2351676.2351694.

[12] 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, Szeged, Hungary: ACM, 2011, pp. 190–200, ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025141. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025141.

[13] G. Bavota, A. De Lucia, M. Di 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, ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2015.05.024`.

[14] B. Berliner *et al.*, "Cvs ii: Parallelizing software development."

[15] 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, Cary, North Carolina: ACM, 2012, 45:1–45:11, ISBN: 978-1-4503-1614-9. DOI: `10.1145/2393596.2393648`. [Online]. Available: `http://doi.acm.org/10.1145/2393596.2393648`.

[16] 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*, 2014, pp. 322–333.

[17] 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, Szeged, Hungary: ACM, 2011, pp. 168–178, ISBN: 978-1-4503-0443-6. DOI: `10.1145/2025113.2025139`. [Online]. Available: `http://doi.acm.org/10.1145/2025113.2025139`.

[18] J. Buffenbarger, "Syntactic software merging," in *Software Configuration Management*, J. Estublier, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 153–172, ISBN: 978-3-540-47768-6.

[19] G. Cavalcanti, P. Accioly, and P. Borba, "Assessing semistructured merge in version control systems: A replicated experiment," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2015, pp. 1–10. DOI: `10.1109/ESEM.2015.7321191`.

[20] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 59:1–59:27, Oct. 2017, ISSN: 2475-1421. DOI: `10.1145/3133883`. [Online]. Available: `http://doi.acm.org/10.1145/3133883`.

[21] O. Chaparro, G. Bavota, A. Marcus, and M. D. Penta, "On the impact of refactoring operations on code quality metrics," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 456–460. DOI: `10.1109/ICSME.2014.73`.

[22] E. Choi, K. Fujiwara, N. Yoshida, and S. Hayashi, "A survey of refactoring detection techniques based on change history analysis," *arXiv preprint arXiv:1808.02320*, 2018.

[23] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-oriented reengineering patterns*. Elsevier, 2002.

[24] D. Dig, T. N. Nguyen, K. Manzoor, and R. Johnson, "Molhadoref: A refactoring-aware software configuration management tool," in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06, Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 732–733, ISBN: 159593491X. DOI: 10.1145/1176617.1176698. [Online]. Available: https://doi.org/10.1145/1176617.1176698.

[25] E. Dinella, T. Mytkowicz, A. Svyatkovskiy, C. Bird, M. Naik, and S. K. Lahiri, "Deepmerge: Learning to merge programs," *arXiv preprint arXiv:2105.07569*, 2021.

[26] W. K. Edwards, "Flexible conflict detection and management in collaborative applications," in *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '97, Banff, Alberta, Canada: Association for Computing Machinery, 1997, pp. 139–148, ISBN: 0897918819. DOI: 10.1145/263407.263533. [Online]. Available: https://doi.org/10.1145/263407.263533.

[27] T. Ekman and U. Asklund, "Refactoring-aware versioning in eclipse," *Electronic Notes in Theoretical Computer Science*, vol. 107, pp. 57–69, 2004.

[28] H. C. Estler, M. Nordio, C. A. Furia, and B. Meyer, "Awareness and merge conflicts in distributed software development," in *2014 IEEE 9th International Conference on Global Software Engineering*, IEEE, 2014, pp. 26–35.

[29] H. Fan and C. Sun, "Dependency-based automatic locking for semantic conflict prevention in real-time collaborative programming," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 2012, pp. 737–742.

[30] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999, ISBN: 0-201-48567-2.

[31] Y. Golubev, Z. Kurbatova, E. A. AlOmar, T. Bryksin, and M. W. Mkaouer, "One thousand and one stories: A large-scale survey of software refactoring," *CoRR*, vol. abs/2107.07357, 2021. arXiv: 2107.07357. [Online]. Available: https://arxiv.org/abs/2107.07357.

[32] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: The contributor's perspective," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 285–296. DOI: 10.1145/2884781.2884826.

[33] M. Guimarães and A. Silva, "Improving early detection of software merge conflicts," *Proceedings - International Conference on Software Engineering*, pp. 342–352, Jun. 2012. DOI: 10.1109/ICSE.2012.6227180.

[34] L. Hattori and M. Lanza, "Syde: A tool for collaborative software development," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, 2010, pp. 235–238.

[35] D. Jackson, D. A. Ladd, *et al.*, "Semantic diff: A tool for summarizing the effects of modifications.," in *ICSM*, vol. 94, 1994, pp. 243–252.

[36] 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, San Francisco, CA, USA: IEEE Press, 2013, pp. 732–741, ISBN: 9781467330763.

[37] ——, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 732–741.

[38] 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*, 2010, pp. 371–372.

[39] 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 *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 543–553. DOI: 10.1109/ASE.2017.8115665.

[40] O. Leßenich, S. Apel, and C. Lengauer, "Balancing precision and performance in structured merge," *Automated Software Engineering*, vol. 22, no. 3, pp. 367–397, 2015.

[41] A. Lie, R. Conradi, T. M. Didriksen, and E.-A. Karlsson, "Change oriented versioning in a software engineering database," in *Proceedings of the 2nd International Workshop on Software Configuration Management*, ser. SCM '89, Princeton, New Jersey, USA: Association for Computing Machinery, 1989, pp. 56–65, ISBN: 0897913345. DOI: 10.1145/72910.73348. [Online]. Available: https://doi.org/10.1145/72910.73348.

[42] E. Lippe and N. Van Oosterom, "Operation-based merging," in *Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, 1992, pp. 78–87.

[43] C. Maddila, N. Nagappan, C. Bird, G. Gousios, and A. Deursen, *Cone: A concurrent edit detection tool for large scalesoftware development*, Jan. 2021.

[44] 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, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 220–230, ISBN: 9781450357166. DOI: `10.1145/3196398.3196434`. [Online]. Available: `https://doi.org/10.1145/3196398.3196434`.

[45] M. Mahmoudi, S. Nadi, and N. Tsantalis, “Are refactorings to blame? an empirical study of refactorings in merge conflicts,” in *Proc. of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER ’19)*, 2019.

[46] T. Mens, “A state-of-the-art survey on software merging,” *IEEE Trans. Softw. Eng.*, vol. 28, no. 5, pp. 449–462, May 2002, ISSN: 0098-5589. DOI: `10.1109/TSE.2002.1000449`. [Online]. Available: `https://doi.org/10.1109/TSE.2002.1000449`.

[47] T. Mens and T. Tourwé, “A survey of software refactoring,” *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[48] 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, 2016, pp. 661–664. DOI: `10.1109/SANER.2016.46`.

[49] M. Owhadi-Kareshk, S. Nadi, and J. Rubin, “Predicting merge conflicts in collaborative software development,” in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–11. DOI: `10.1109/ESEM.2019.8870173`.

[50] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, “An exploratory study on the relationship between changes and refactoring,” in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 176–185. DOI: `10.1109/ICPC.2017.38`.

[51] 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, Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 9–15, ISBN: 978-1-4503-0576-1. DOI: `10.1145/1984642.1984645`. [Online]. Available: `http://doi.acm.org/10.1145/1984642.1984645`.

[52] M. M. Rahman and C. K. Roy, “An insight into the pull requests of github,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 364–367, ISBN: 9781450328630. DOI: `10.1145/2597073.2597121`. [Online]. Available: `https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/2597073.2597121`.

[53] A. Sarma, D. F. Redmiles, and A. Van Der Hoek, "Palantir: Early detection of development conflicts arising from parallel code changes," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 889–908, 2011.

[54] G. Seibt, F. Heck, G. Cavalcanti, P. Borba, and S. Apel, "Leveraging structure in software merge: An empirical study," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021. DOI: `10.1109/TSE.2021.3123143`.

[55] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang, "Intellimerge: A refactoring-aware software merging technique," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 170:1–170:28, Oct. 2019, ISSN: 2475-1421. DOI: `10.1145/3360596`. [Online]. Available: `http://doi.acm.org/10.1145/3360596`.

[56] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, 2016, pp. 858–870.

[57] L. da Silva, P. Borba, W. Mahmood, T. Berger, and J. Moisakis, "Detecting semantic conflicts via automated behavior change detection," Sep. 2020, pp. 174–184. DOI: `10.1109/ICSME46990.2020.00026`.

[58] A. Trindade Tavares, P. Borba, G. Cavalcanti, and S. Soares, "Semistructured merge in javascript systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1014–1025. DOI: `10.1109/ASE.2019.00098`.

[59] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, 2020. DOI: `10.1109/TSE.2020.3007722`.

[60] 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, Gothenburg, Sweden: ACM, 2018, pp. 483–494, ISBN: 978-1-4503-5638-1. DOI: `10.1145/3180155.3180206`. [Online]. Available: `http://doi.acm.org/10.1145/3180155.3180206`.

[61] B. Westfechtel, "Structure-oriented merging of revisions of software documents," in *Proceedings of the 3rd International Workshop on Software Configuration Management*, ser. SCM '91, Trondheim, Norway: Association for Computing Machinery, 1991, pp. 68–79, ISBN: 0897914295. DOI: `10.1145/111062.111071`. [Online]. Available: `https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/111062.111071`.

[62] F. Zhu and F. He, "Conflict resolution for structured merge via version space algebra," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[63] F. Zhu, F. He, and Q. Yu, “Enhancing precision of structured merge by proper tree matching,” in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ser. ICSE ’19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 286–287. DOI: `10.1109/ICSE-Companion.2019.00117`. [Online]. Available: `https://doi.org/10.1109/ICSE-Companion.2019.00117`.

# Appendix A

# Conflict Resolution Logic

We introduce the notations used to describe refactoring logic by using the program elements in the motivating example within Figure 2.2.

- $simpleName(p)$ denotes the simple name of a program element, while $fqName(p)$ denotes the fully qualified name of the program element. For example, when $p =$ `addListener` in `Scanner.java`, $simpleName(p) =$ `addListener` and $fqName(p) =$ `Scanner.addListener`.

- $classOf(p)$ denotes the parent class that defines program element $p$. Using `addListener` as $p$, $classOf(p) =$ `Scanner`.

- $packageOf(p)$ denotes the package of program element $p$. $signature(p)$ is the signature of program element $p$, or $signature(p) =$ `void addListener(O obj)`.

- $inherits(c_i, c_j)$ is true when class $c_i$ inherits $c_j$ are in an inheritance relationship. For example, when $c_i =$ `Reader` and $c_j =$ `Scanner`, $inherits(c_i, c_j) = True$.

- $overrides(m_i, m_j)$ is true when method $m_i$ overrides method $m_j$. Similarly, $overloads(m_i, m_j)$ is true when method $m_i$ overloads method $m_j$. $codeOverlaps(m_i, m_j)$ is true when two extracted code fragments overlap within the same source method.

The following are the equality conditions for classes (c) and methods (m):
$c_i = c_j$ iff $signature(c_i) = signature(c_j) \wedge fqName(c_i) = fqName(c_j)$

$m_i = m_j$ iff $signature(m_i) = signature(m_j) \wedge fqName(m_i) = fqName(m_j)$

To remove redundant logic predicates, we combine *MoveMethod*, *RenameMethod*, and *MoveAndRenameMethod* into one logic predicate, *MoveAndRenameMethod*. We do the same for *MoveClass*, *RenameClass*, and *MoveAndRenameClass*. We can do this because *MoveMethod* is essentially a *MoveAndRenameMethod* refactoring where the method is renamed to itself. We use *sameType*() to differentiate between predicates within the same logic cell. *sameType* is false only when both refactorings are different refactoring types. For example, *sameType*() is false if one refactoring is `Rename Method` and the other is `Move Method`. However, if the refactorings are `Rename And Move Method` and `Rename Method`, then *sameType* is true because `Rename And Move Method` contains `Rename Method`.

## A.1 Conflict Predicates

If a conflict predicate returns true, then the refactoring pair is conflicting.

$$\begin{aligned} MoveAndRenameMethod(m_1, m_2)/MoveAndRenameMethod(m_3, m_4) :=& \\ (((m_1 = m_3 \wedge m_2 \neq m_4)\vee& \\ (m_1 \neq m_3 \wedge m_2 = m_4)) \wedge sameType())\vee& \\ (\neg overrides(m_1, m_3) \wedge overrides(m_2, m_4)\vee& \\ (\neg overloads(m_1, m_3) \wedge overloads(m_2, m_4)& \end{aligned} \tag{A.1}$$

$$\begin{aligned} MoveAndRenameClass(c_1, c_2)/MoveAndRenameMethod(m_1, m_2) :=& \\ false& \end{aligned} \tag{A.2}$$

$$\begin{aligned} MoveAndRenameClass(c_1, c_2)/MoveAndRenameClass(c_3, c_4) :=& \\ ((c_1 = c_2 \wedge c_3 \neq c_4)\vee& \\ (c_1 \neq c_2 \wedge c_3 = c_4)) \wedge sameType()& \end{aligned} \tag{A.3}$$

$$\begin{aligned} ExtractMethod(m_1, m_2)/MoveAndRenameMethod(m_3, m_4) :=& \\ (m_2 = m_4)\vee& \\ (overrides(m_2, m_4)\vee& \\ (overloads(m_2, m_4)& \end{aligned} \tag{A.4}$$

$$\begin{aligned} ExtractMethod(m_1, m_2)/MoveAndRenameClass(c_1, c_2) :=& \\ false& \end{aligned} \tag{A.5}$$

$$\begin{aligned} ExtractMethod(m_1, m_2)/ExtractMethod(m_3, m_4) :=& \\ (m_2 = m_4)\vee& \\ (overrides(m_2, m_4)\vee& \\ (overloads(m_2, m_4)\vee& \\ (overlappingFragments(m_1, m_3)& \end{aligned} \tag{A.6}$$

$$\begin{aligned} InlineMethod(m_1, m_2)/MoveAndRenameMethod(m_3, m_4) :=& \\ (m_1 = m_3)& \end{aligned} \tag{A.7}$$

$$\begin{aligned} InlineMethod(m_1, m_2)/MoveAndRenameClass(c_1, c_2) :=& \\ false& \end{aligned} \tag{A.8}$$

$$\begin{aligned} InlineMethod(m_1, m_2)/ExtractMethod(m_3, m_4) :=& \\ false& \end{aligned} \tag{A.9}$$

$$\begin{aligned} InlineMethod(m_1, m_2)/InlineMethod(m_3, m_4) :=& \\ false& \end{aligned} \tag{A.10}$$

## A.2 Commutative Predicates

If a commutative predicate returns true, then the refactoring pair has a commutative relationship.

$$MoveAndRenameMethod(m_1, m_2)/MoveAndRenameMethod(m_3, m_4) := (m_1 = m_3) \wedge \neg sameType() \tag{A.11}$$

$$MoveAndRenameClass(c_1, c_2)/MoveAndRenameMethod(m_1, m_2) := (classOf(m_1) = c_1) \vee (classOf(m_2) = c_1) \tag{A.12}$$

$$MoveAndRenameClass(c_1, c_2)/MoveAndRenameClass(c_3, c_4) := (c_1 = c_3) \wedge \neg sameType() \tag{A.13}$$

$$ExtractMethod(m_1, m_2)/MoveAndRenameMethod(m_3, m_4) := (m_1 = m_3) \tag{A.14}$$

$$ExtractMethod(m_1, m_2)/MoveAndRenameClass(c_1, c_2) := (classOf(m_1) = c_1) \vee (classOf(m_2) = c_1) \tag{A.15}$$

$$ExtractMethod(m_1, m_2)/ExtractMethod(m_3, m_4) := false \tag{A.16}$$

$$InlineMethod(m_1, m_2)/MoveAndRenameMethod(m_3, m_4) := (m_2 = m_3) \tag{A.17}$$

$$InlineMethod(m_1, m_2)/MoveAndRenameClass(c_1, c_2) := (classOf(m_1) = c_1) \vee (classOf(m_2) = c_1) \tag{A.18}$$

$$InlineMethod(m_1, m_2)/ExtractMethod(m_3, m_4) := false \tag{A.19}$$

$$InlineMethod(m_1, m_2)/InlineMethod(m_3, m_4) := (m_2 = m_3) \vee (m_1 = m_4) \tag{A.20}$$

## A.3 Simplification Predicates: Transitivity

If the transitivity predicates return true, then the refactorings are transitive.

$$MoveAndRenameMethod(m_1, m_2)/MoveAndRenameMethod(m_3, m_4) := (m_2 = m_3) \tag{A.21}$$

$$MoveAndRenameClass(c_1, c_2)/MoveAndRenameMethod(m_1, m_2) := false \tag{A.22}$$

$$MoveAndRenameClass(c_1, c_2)/MoveAndRenameClass(c_3, c_4) := (c_2 = c_3) \tag{A.23}$$

$$ExtractMethod(m_1, m_2)/MoveAndRenameMethod(m_3, m_4) := (m_1 = m_3) \tag{A.24}$$

$$ExtractMethod(m_1, m_2)/MoveAndRenameClass(c_1, c_2) := false \tag{A.25}$$

$$ExtractMethod(m_1, m_2)/ExtractMethod(m_3, m_4) := false \tag{A.26}$$

$$InlineMethod(m_1, m_2)/MoveAndRenameMethod(m_3, m_4) := false \tag{A.27}$$

$$InlineMethod(m_1, m_2)/MoveAndRenameClass(c_1, c_2) := false \tag{A.28}$$

$$InlineMethod(m_1, m_2)/ExtractMethod(m_3, m_4) := false \tag{A.29}$$

$$InlineMethod(m_1, m_2)/InlineMethod(m_3, m_4) := false \tag{A.30}$$

## A.4 Simplification Predicates: Refactoring Chain

If a refactoring chain predicate returns true, then the refactorings are part of a refactoring chain.

$$\begin{aligned} MoveAndRenameMethod(m_1, m_2)/MoveAndRenameMethod(m_3, m_4) := \\ false \end{aligned} \tag{A.31}$$

$$\begin{aligned} MoveAndRenameClass(c_1, c_2)/MoveAndRenameMethod(m_1, m_2) := \\ (classOf(m_1) = c_1)\lor \\ (classOf(m_2) = c_2) \end{aligned} \tag{A.32}$$

$$\begin{aligned} MoveAndRenameClass(c_1, c_2)/MoveAndRenameClass(c_3, c_4) := \\ false \end{aligned} \tag{A.33}$$

$$\begin{aligned} ExtractMethod(m_1, m_2)/MoveAndRenameMethod(m_3, m_4) := \\ (m_2 = m_3) \end{aligned} \tag{A.34}$$

$$\begin{aligned} ExtractMethod(m_1, m_2)/MoveAndRenameClass(c_1, c_2) := \\ (classOf(m_1) = c_2)\lor \\ (classOf(m_2) = c_1) \end{aligned} \tag{A.35}$$

$$\begin{aligned} ExtractMethod(m_1, m_2)/ExtractMethod(m_3, m_4) := \\ false \end{aligned} \tag{A.36}$$

$$\begin{aligned} InlineMethod(m_1, m_2)/MoveAndRenameMethod(m_3, m_4) := \\ (m_1 = m_4) \end{aligned} \tag{A.37}$$

$$\begin{aligned} InlineMethod(m_1, m_2)/MoveAndRenameClass(c_1, c_2) := \\ (classOf(m_1) = c_2)\lor \\ (classOf(m_2) = c_1) \end{aligned} \tag{A.38}$$

$$InlineMethod(m_1, m_2)/ExtractMethod(m_3, m_4) := false \tag{A.39}$$

$$InlineMethod(m_1, m_2)/InlineMethod(m_3, m_4) := false \tag{A.40}$$