Operation-based Refactoring-aware Merging: An Empirical Evaluation

Max Ellis, Sarah Nadi, Danny Dig

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; however, these two techniques have never been empirically compared. In this paper, we present `RefMerge`, a rejuvenated Java-based design and implementation of the first technique, which is an operation-based refactoring-aware merging algorithm. We compare `RefMerge` to Git and the state-of-the-art graph-based refactoring-aware merging tool, `IntelliMerge`, on 2,001 merge scenarios with refactoring-related conflicts from 20 open-source projects. We find that `RefMerge` resolves or reduces conflicts in 497 (25%) merge scenarios while increasing conflicting LOC in only 214 (11%) scenarios. On the other hand, we find that `IntelliMerge` resolves or reduces conflicts in 478 (24%) merge scenarios but increases conflicting LOC in 597 (30%) merge scenarios. We additionally conduct a qualitative analysis of the differences between the three merging algorithms and provide insights of the strengths and weaknesses of each tool. We find that while `IntelliMerge` does well with ordering and formatting conflicts, it struggles with class-level refactorings and scenarios with several refactorings. On the other hand, `RefMerge` is resilient to the number of refactorings in a merge scenario, but we find that `RefMerge` introduces conflicts when inverting move-related refactorings.

Index Terms—conflict resolution, refactoring, software merging, revision control systems

1 Introduction

Version control systems (VCS) play a crucial role in enabling developers to collaborate on software projects. Whether developers are working on the same branch [1], using branch-based development [2], or using pull requests to contribute changes from their external forks [3], 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*. Based on analyzing four open-source projects, previous work found that merge conflicts occurred up to 19% of the time and could sometimes take several days to resolve [4]. As our own analysis of 20 open-source projects shows, this percentage can vary significantly per project (3% - 55%), with a median of 16% (See Table 2). 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 [4], [5]. Thus, overall, while merge conflicts are moderately frequent, they are a burden when they occur. A recent practitioner survey shows that developers spend time trying to understand and resolve conflicts and that current support tools do not meet all their conflict-resolution needs [6].

A common issue with merge conflicts is that most modern version control systems, such as Git [7], Mercurial [8], or SVN [9], 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* [10]) will report a conflict since it cannot automatically decide which change to choose. However, a tool that understands the nature of the code change that occurred may be able to resolve the conflict [11], [12]. For example, *refactorings* are code changes that modify the structure of the code to improve its readability or maintainability without altering its observable behavior [13]. Refactorings are one example of a code change with well-defined semantics that an automated merge-conflict resolution tool can understand and automatically resolve [11], [14], [15], [16]. For example, if Bob refactors method `foo` by moving it from one class to another on one branch 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 lines of code. However, a merge tool that is aware of the semantics behind these refactoring changes can resolve this conflict by adding the new line of code to `foo`'s new location. Thus, understanding the semantics of refactorings could avoid unnecessary merge conflicts and save developers' time. A recent study found that 15 of more than 70 known refactorings are involved in 22% of merge conflicts and tend to result in larger conflicts [14]. 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 underlying code to automate more merge-conflict resolutions [12], [17], [18], [19], [20], [21], [22], there are mainly two efforts that specifically focus on refactorings. The first is by Dig et al. [16] that proposes an *operation-based refactoring-aware merging technique*, MolhadoRef. At a high level, given two branches to be merged, MolhadoRef first inverts refactorings on both branches,

1. Note that in this paper, 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.

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.

The second approach by Shen et al. [15] is a graph-based refactoring-aware merging approach implemented in `IntelliMerge`. `IntelliMerge` converts code on both branches to graphs, and performs 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 both approaches show promise in their evaluation, they each have limitations. To begin, the premise of MolhadoRef 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 these refactoring operations in the history to resolve conflicts. To that end, MolhadoRef relies on developers using the researchers' operation-based version control system, Molhado, preventing the approach from being used on modern version control systems. Furthermore, MolhadoRef's implementation supported only six refactorings, none of which are complex refactorings such as *Extract Method* and *Inline Method*, which the authors of `IntelliMerge` [15] argued limit operation-based techniques due to the difficulty of inverting them. Finally, while their evaluation shows promise, it was limited to their own repository, thus MolhadoRef's feasibility in practice is unknown, especially since there is no publicly available implementation of their approach. Even if there was, MolhadoRef's reliance on Molhado makes replicating it on real-world projects impossible.

On the other hand, `IntelliMerge` relies on a similarity score for detecting refactorings, which has been argued in the literature to misidentify or miss refactorings [23]. This can lead to introducing additional conflicts, unexpected merges, or missing conflicts. In addition, `IntelliMerge` does not consider how refactorings on each branch will interact with each other. Furthermore, it has been evaluated only w.r.t. resolving more conflicts using imprecise metrics and the evaluation does not analyze whether the tool's resolutions are correct or whether there were missed conflicts that `IntelliMerge` did not detect. Finally, while criticizing operation-based merging, the authors never performed an evaluation comparing the two approaches. Given the merit of solving merge conflicts when refactorings are involved, we believe that 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.

To enable the comparison of these two techniques, this paper has two goals. Our first goal is to rejuvenate Dig et al.'s operation-based refactoring-aware merging technique [16]. This requires a re-design of the technique to enable it to work with modern VCSs, such as git. We implement our rejuvenated operation-based refactoring-aware merging technique in `RefMerge`. While `RefMerge` follows the same approach of reverting and replaying refactorings, there are several novelties that differentiate `RefMerge` from MolhadoRef: (1) Whereas MolhadoRef relies on a research-based version control system, `RefMerge` is designed to work directly on top of Git, since it is the most popular version control system used by practitioners [24], (2) `RefMerge` supports 17 refactoring types (instead of MolhadoRef's six), including *Extract Method* and *Inline Method* which were argued to not have an inverse refactoring [15], (3) To detect refactorings in Git history, `RefMerge` uses the state-of-the-art refactoring detection tool, RefactoringMiner [25], (4) `RefMerge` avoids checking for circular dependencies by simplifying and combining refactorings upon detection, and finally (5) We evaluate `RefMerge` on a large scale to determine the feasibility of operation-based refactoring-aware merging in practice.

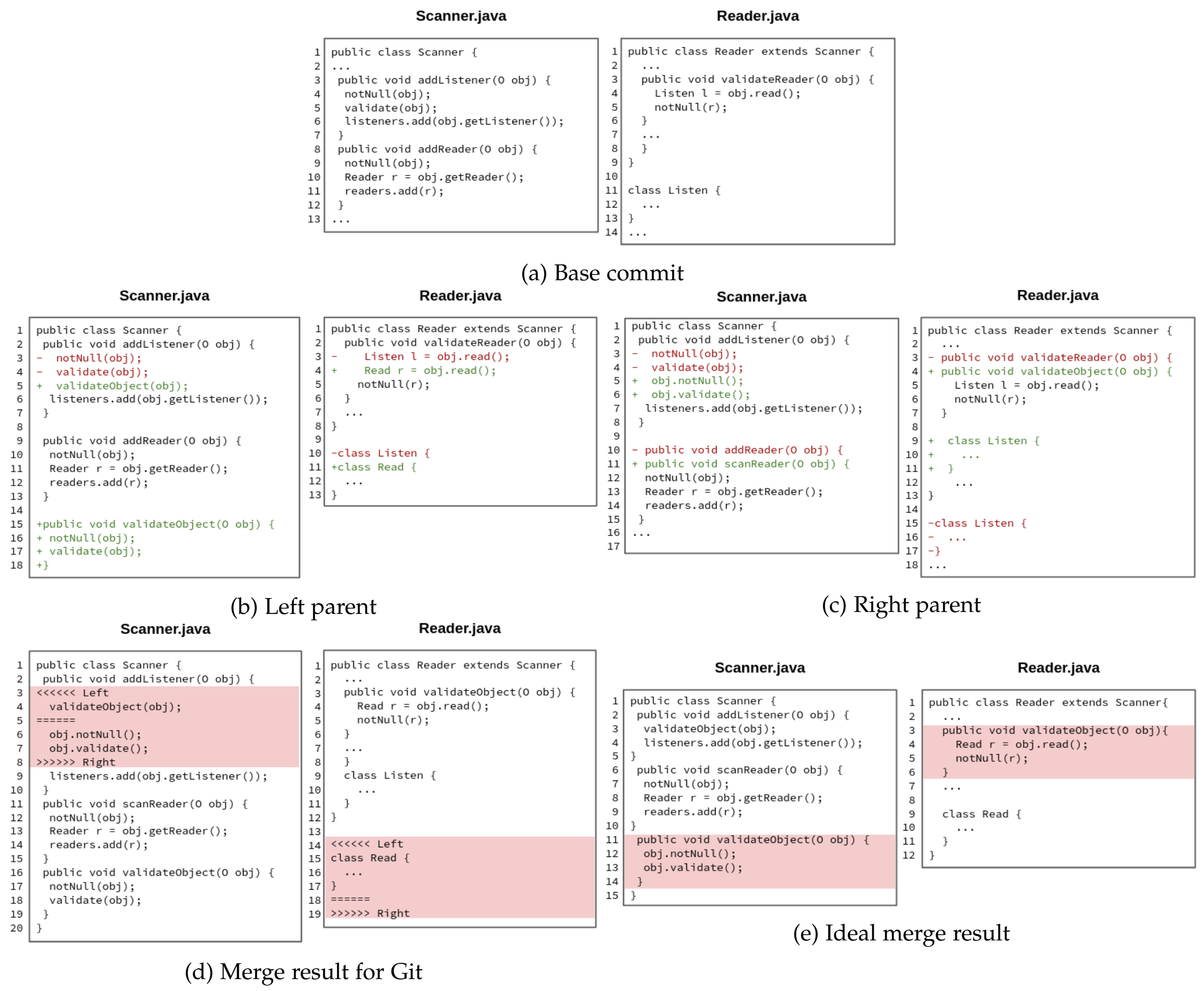
Our second goal is to compare the two refactoring-aware merging techniques on real-world projects that use Git as their version control system, since it is the most popular version control system used by practitioners [24]. To that end, we perform the first large-scale comparison of operation-based merging and graph-based merging techniques. In summary, this paper makes the following contributions:

- An open-source design and implementation [26] of operation-based refactoring-aware merging, `RefMerge`, built on top of Git and which covers 17 refactoring types, including two complex refactorings that complicate conflicts [14] and were proposed to be difficult for operation-based merging [15], *Extract Method* and *Inline Method*.
- A large-scale *quantitative* comparison of the effectiveness of operation-based refactoring implemented in `RefMerge` versus graph-based refactoring implemented in `IntelliMerge`. Our evaluation includes 2,001 merge scenarios from 20 open-source 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.

Our evaluation results show that while `IntelliMerge` reduces the number of refactoring conflicts a developer needs to deal with, graph node matching errors and the reliance on a similarity score cause `IntelliMerge` to highly increase the number of false positives and false negatives. On the other hand, `RefMerge` is able to reduce the number of false positives while eliminating false negatives. However, `RefMerge` sometimes introduces conflicts while inverting move-related refactorings. 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 [26].

2 Background and Motivating Example

To introduce the terms we use, we briefly describe how merging works in Git. We also provide an example to motivate the need for refactoring-aware merging techniques.



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);
+  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 {
  ...
}
```

(b) Left parent

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 {
-  ...
-}
...
```

(c) Right 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

Fig. 1: 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.

2.1 Software Merging in Git

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 [27]. 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*. An example of a commit history leading to a merge commit is shown at the top left corner of Figure 2.

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 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, or *conflicting LOC*. For example, Figure 1d shows two conflicting files, `Scanner.java` and `Reader.java`. Each file has one conflict block. The first conflict block in `Scanner.java` has 3 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).

2.2 Motivating Example

To understand how refactorings complicate merge scenarios, consider the example inspired by multiple real conflicts in Figure 1. In the left branch (Figure 1b), 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 1c), 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 1d, Git reports a conflict in file `Reader.java` because the developers rename `Listen` on one branch and moves 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 simply renaming the moved class `Listen` to `Read`, as shown in the "ideal" merge result in Figure 1e. We refer to the Git conflict in `Reader.java` from Figure 1d 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 would 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`. To resolve this conflict, the developer needs to compare the code inside of the extracted method `validateObject` (which is not even highlighted as part of the conflict) with the conflicting code from the right parent shown in conflict block. Such a comparison is even 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 from the right parent should be performed in the extracted method, rather than in `addListener` and that these changes can be merged, as shown in Figure 1e.

Figure 1e 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 branches changes in `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 instead Git silently merges the changes.

3 REFMERGE: REFACTORING-AWARE OPERATION-BASED MERGING

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 2 presents an overview of our implementation of `RefMerge`, which 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 [25] 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 covered 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.
5) *Replay Refactorings*: We finally use the merged refactoring list to replay all non-conflicting refactorings.

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

3.1 Step 1: Detect and Simplify Refactorings

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.

Refactoring Simplification: `RefMerge` processes each detected refactoring one by one and keeps a list of processed refactorings, *ProcessedRefList*, for the left and right branches. We compare each detected refactoring to the refactorings in *ProcessedRefList* 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 *ProcessedRefList*, it updates the related transitive refactorings in *ProcessedRefList* instead of adding a new refactoring. In this example, `RefMerge` would first add *rename `foo` to `bar`* to *ProcessedRefList*. When it processes *rename `bar` to `foobar`*, it detects that this is a transitive refactoring of an existing refactoring in *ProcessedRefList* so it will simply update the existing refactoring to *rename `foo` to `foobar`*.

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.

Therefore, 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

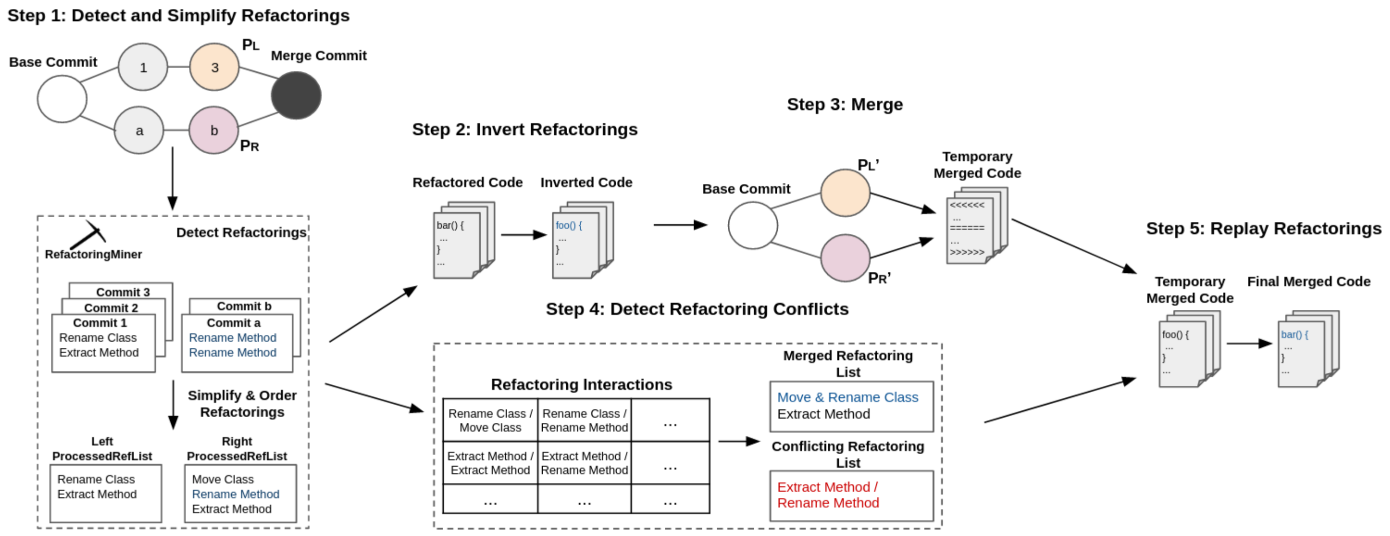


Fig. 2: An overview of RefMerge's merging algorithm

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. Our artifact contains all our detailed logic for detecting transitive refactorings and refactoring chains [26].

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. Instead, we reorder the refactorings in a pre-determined top-down order based on the granularity of the program element being refactored. For example, class level refactorings come before method level refactorings. Performing refactorings in a pre-determined order is necessary because not all refactorings are transitive. For example, refactorings that delete a program element (such as *Inline Method*) need to be inverted before refactorings that create a program element (such as *Extract Method)*. Inverting or replaying the refactoring operations in an incorrect order will lead to non-existent elements being referenced and `RefMerge` being unable to correctly perform the refactoring operations.

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 another important difference from MolhadoRef which required user intervention to help resolve circular dependencies; by automatically breaking circular dependencies, `RefMerge` allows the user to focus on the conflicts.

TABLE 1: The list of 17 refactorings supported by `RefMerge` and their corresponding inverse refactorings.

Refactoring	Inverse Refactoring
RenameMethod(m_1, m_2)	RenameMethod(m_2, m_1)
MoveMethod(c_1, c_2)	MoveMethod(c_2, c_1)
Move&RenameMethod($c_1.m_1$, $c_2.m_2$)	Move&RenameMethod($c_2.m_2$, $c_1.m_1$)
RenameClass(c_1, c_2)	RenameClass(c_2, c_1)
MoveClass(loc_1, loc_2)	MoveClass(loc_2, loc_1)
Move&RenameClass($loc_1.c_1$, $loc_2.c_2$)	Move&RenameClass($loc_2.c_2$, $loc_1.c_1$)
InlineMethod(m_1, $m_2.l_x$-l_y)	ExtractMethod($m_2.l_x$-l_y, m_1)
ExtractMethod($m_1.l_x$-l_y, m_2)	InlineMethod(m_2, $m_1.l_x$-l_y)
PullUpMethod(c_x-c_y,c_2)	PushDownMethod(c_2,c_x-c_y)
PushDownMethod(c_1,c_x-c_y)	PullUpMethod(c_x-c_y,c_1)
RenameField(f_1, f_2)	RenameField(f_2, f_1)
MoveField(c_1, c_2)	MoveField(c_2, c_1)
Move&RenameField($c_1.f_1$, $c_2.f_2$)	Move&RenameField($c_2.f_2$, $c_1.f_1$)
PullUpField(c_x-c_y,c_2)	PushDownField(c_2,c_x-c_y)
PushDownField(c_1,c_x-c_y)	PullUpField(c_x-c_y,c_1)
RenamePackage(p_1, p_2)	RenamePackage(p_2, p_1)
RenameParameter(p_1, p_2)	RenameParameter(p_2, p_1)

3.2 Step 2: Invert Refactorings

Once refactorings are detected, `RefMerge` creates a refactoring-free version of each parent commit by inverting the refactorings in the *ProcessedRefList* on each branch 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`. Table 1 provides a list of each refactoring and the refactoring operation that `RefMerge` uses to invert it.

`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.

3.3 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 2. 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 1. 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.

3.4 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. 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 [26]. `RefMerge` uses this knowledge to compare each refactoring in the left branch with each refactoring in the right branch and detect refactoring conflicts.

3.4.1 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 $RenameMethod(m_3, m_4)$ in Equation 1.

$$\begin{aligned}
&hasConflict(RenameMethod(c_1.m_1, c_2.m_2),\\
&\quad RenameMethod(c_3.m_3, c_4.m_4)) :=\\
&\quad ((c_1.m_1 == c_3.m_3 \land c_2.m_2 \neq c_4.m_4)\\
&\quad \lor(c_1.m_1 \neq c_3.m_3 \land c_2.m_2 == c_4.m_4))\\
&\lor(\neg overrides(c_1.m_1, c_3.m_3) \land overrides(c_2.m_2, c_4.m_4)\\
&\lor(\neg overloads(c_1.m_1, c_3.m_3) \land overloads(c_2.m_2, c_4.m_4)
\end{aligned} \tag{1}$$

These two refactorings result in a conflict if (1) the source of both refactorings is the same program element ($c_1.m_1 = c_3.m_3$) but their destinations differ ($c_2.m_2 \neq c_4.m_4$) or (2) the sources of both renames are different program elements ($c_1.m_1 \neq c_3.m_3$) but the renamed destinations are the same program element ($c_2.m_2 = c_4.m_4$). 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 *Rename Method* and *Extract Method* refactorings are an example of conflicting refactorings that can cause an accidental override, such as the motivating example in Figure 1.

3.4.2 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 1 are an example of commutative refactorings.

We present the commutative logic for $MoveMethod(m_1, m_2)$ and $RenameMethod(m_3, m_4)$ as an example in Equation 2.

$$isCommutative(MoveMethod(c_1.m_1, c_2.m_2), RenameMethod(c_3.m_3, c_4.m_4) := (c_1.m_1 == c_3.m_3 \land c_2.m_2 \neq c_4.m_4) \quad (2)$$

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.

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 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 1 after detecting refactoring conflicts and commutative conflicts would contain *Rename Method* `addReader` to `scanReader` and *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`.

3.5 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.

3.6 Current Implementation

Technologies and Tools: We implement `RefMerge` as an IntelliJ[2] plugin for merging Java programs. It consists of four key modules corresponding to the steps of the proposed technique. We use *RefactoringMiner* [25] to detect the refactorings and 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 [13]; it is a large engineering effort to implement every refactoring. Instead of implementing every refactoring, we use a subset of 17 refactorings to show the feasibility of the approach and enable the empirical comparison.

2. https://www.jetbrains.com/idea

Note that there are 13 supported refactorings that the `IntelliMerge` publication describes in its matching rules [15]. However, we find that `IntelliMerge`'s implementation potentially supports an additional eight refactorings. When deciding which refactorings to support in `RefMerge`, we prioritized the 13 refactoring types described in the publication. To cover more refactorings from different granularity levels, we also added support for *Rename Parameter*, *Rename And Move Field*, *Rename And Move Method*, and *Rename And Move Class* from the eight additional refactorings. Overall, `RefMerge` supports a subset of 17 out of the 21 refactorings `IntelliMerge`'s implementation supports.

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. It is worth noting that our open-source implementation of `RefMerge` is designed in a modular way to easily allow for extension. In general, adding a new refactoring requires adding a handler to invert and replay the refactoring and adding logic for how it interacts with the existing refactoring. Our artifact [26] contains a step-by-step guide for adding a new refactoring.

4 Evaluation Setup

We compare the effectiveness of `RefMerge`, Git, and the state-of-the-art refactoring-aware merge tool, `IntelliMerge` [15] 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.

4.1 Project & Merge Scenario Selection

We first include the same 10 projects that the `IntelliMerge` authors use in their evaluation [15]. 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 [15]. The authors then ran the analysis by Mahmoudi et al. [14] 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

TABLE 2: 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. Our evaluation is based on the merge scenarios in Column c.

Project	Stargazers	Merge Scenarios	a. Conflicting Merge Scenarios	b. Conflicting Java Merge Scenarios (% from Col. a)	c. Conflicting Java Merge Scenarios w/ Involved Refactorings (% from Col. b)	d. Conflicting Java Merge Scenarios w/ *Only* Involved Refactorings (% from Col. c)
cassandra	6,882	10,719	4,509 (42%)	2,693 (25%)	922 (34%)	244 (27%)
elasticsearch	56,665	5,111	561 (11%)	504 (9%)	178 (35%)	30 (17%)
gradle	12,410	7,690	1,127 (15%)	300 (3%)	117 (39%)	28 (24%)
antlr4	10,738	1,935	398 (21%)	198 (10%)	100 (51%)	14 (14%)
platform_frameworks_support	1,609	67,584	3,690 (55%)	570 (15%)	96 (17%)	25 (26%)
deeplearning4j	12,208	6,997	566 (8%)	386 (6%)	93 (24%)	20 (22%)
realm-java	11,206	3,405	683 (20%)	312 (9%)	92 (29%)	23 (25%)
jackson-core	1,984	584	318 (54%)	201 (34%)	81 (40%)	17 (21%)
android	3,161	1,805	315 (17%)	208 (12%)	81 (39%)	9 (11%)
cometd	535	759	369 (49%)	173 (23%)	63 (36%)	11 (18%)
storm	6,278	3,626	267 (7%)	87 (2%)	33 (38%)	12 (36%)
ProjectE	308	386	79 (20%)	73 (19%)	30 (41%)	3 (10%)
javaparser	3,859	2,427	94 (4%)	73 (3%)	23 (32%)	6 (26%)
druid	24,576	1,498	151 (10%)	138 (9%)	17 (12%)	3 (18%)
androidannotations	11,171	739	73 (10%)	71 (10%)	15 (21%)	1 (7%)
junit4	8,198	400	46 (11%)	41 (10%)	14 (34%)	2 (14%)
MinecraftForge	4,945	792	77 (10%)	46 (6%)	14 (30%)	3 (21%)
iFixitAndroid	143	171	29 (17%)	24 (14%)	13 (54%)	0 (0%)
MozStumbler	609	934	32 (3%)	26 (3%)	10 (38%)	1 (10%)
error-prone	5,717	133	24 (18%)	21 (16%)	9 (43%)	1 (11%)
Total					2,001	453

ones, uses RefactoringMiner [25] 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. [14] 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 [14] on the latest history of each project as of September 26, 2021. Our artifact page [26] 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 2 shows the number of merge scenarios in each project, the number of conflicting merge scenarios, the number of conflicting merge scenario with conflicting Java files, the number of merge scenarios with refactoring-related Java conflicts, and the number of merge scenarios with *only* refactoring-related Java conflicts. The table shows that the frequency of merging and frequency of conflicting merges varies between projects. Thus, our evaluation covers projects with frequent merge scenarios (e.g., cassandra and platform_frameworks_support) as well as those with infrequent merges in their history (e.g., error-prone or iFixitAndroid). We also cover both projects that are conflict prone (e.g., cassandra and jackson-core) as well as those with infrequent conflicts, regardless of frequency of merging (e.g., storm and javaparser). Table 2 also shows the projects used in the IntelliMerge paper in bold and provide the number of stargazers of each project. Overall, we evaluate on 2,001 conflicting merge scenarios with involved refactorings. Figure 3 shows the distribution of refactorings per merge scenario that we evaluate on in each project. As shown, the median number of refactorings per scenario varies widely among the projects, with *druid* having the lowest median and *cassandra* having the highest.

4.2 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 [15] using their exact setup and data, as shared in their Github repository [28]. We share the exact steps we followed as well as the details of the results of reproducing `IntelliMerge`[3].

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 removing 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 the auto-merged code with the 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 in the git history. 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.

3. https://github.com/max-ellis/IntelliMerge/tree/evaluation

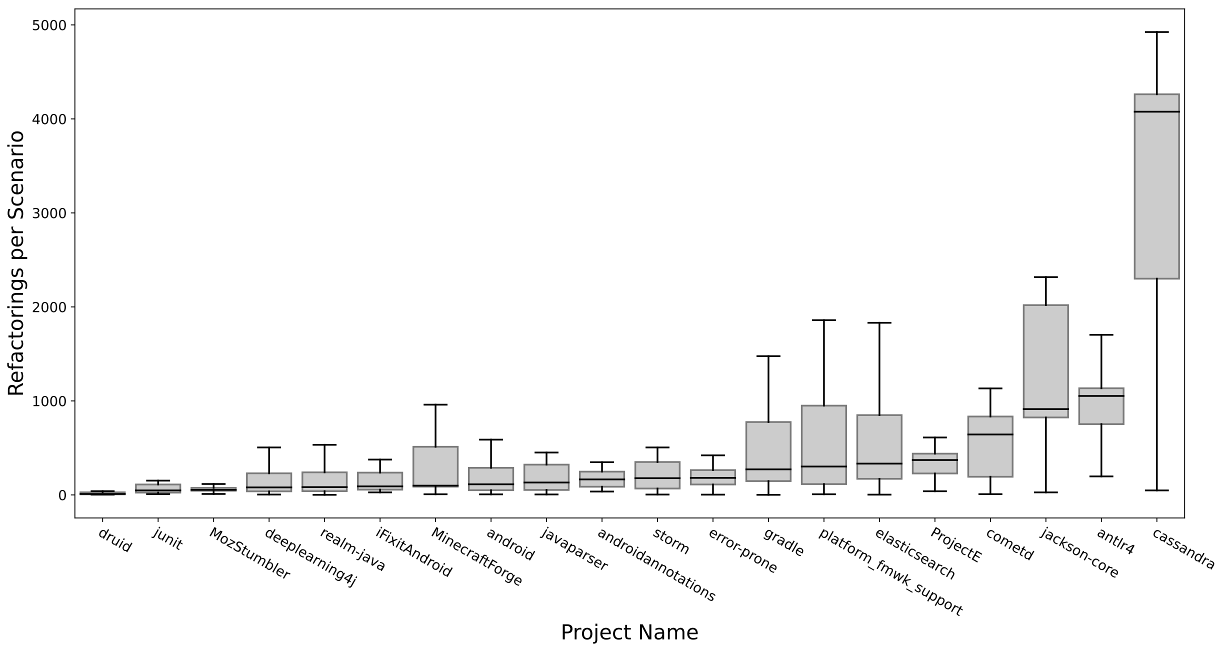


Fig. 3: Distribution of refactorings per merge scenario that we evaluate on in each project

We were not able to reproduce the exact numbers found in the IntelliMerge paper [15]. After emailing the authors, we verified that they perform manual 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[3] with the IntelliMerge authors and received confirmation that our setup is correct and that the differences in results we obtained do not misrepresent IntelliMerge.

4.3 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 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 (R) Core (TM) i5-12600K CPU @ 3.70GHz, 32 GB RAM and Ubuntu 20.04 OS. For feasibility of completing the evaluation, we use a 15 min timeout for each tool.

4.4 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 1d. If the developer chose to merge the changes from the left branch in the manual merge, the manually merged code will have 15 lines of code. Meanwhile, if a merge tool always accepts changes from both branches, then the auto-merged code will have 17 lines of code. In this case, Git diff will report 2 different lines since the auto-merged code also contains changes from the right branch while the manually merged code does not. In this case, the recall will be 1 and while the precision will not be 1, it will still be high (88%) 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.

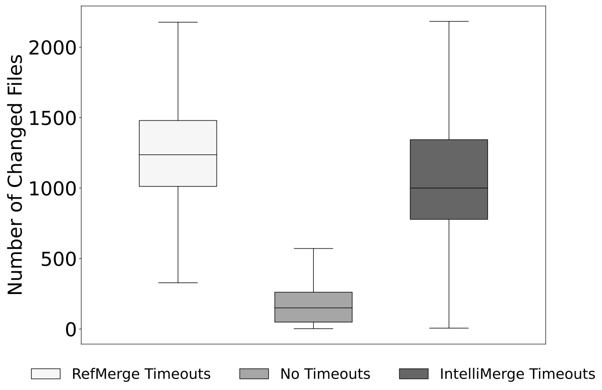


Fig. 4: Distribution of changed files per merge scenario where timeouts occur in each tool.

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. [18] to get a better understanding of merge results.

5 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 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 complicates the conflict by reporting more conflicting files, blocks, or lines of code. We first report detailed results of the evaluation in Sections 5.1-5.2 and then provide an interpretation of these results in Section 5.3.

5.1 Completely Resolved Merge Scenarios

Table 3 shows the breakdown of the merge results for each project. The *Total Scenarios* column shows the number of conflicting Git scenarios with involved refactorings evaluated for each project (same as Column c from Table 2). We then show the results for `IntelliMerge` and `RefMerge`, respectively. For each tool, we show the number of completely resolved merge scenarios (column *Resolved*), the number of merge scenarios where the conflict result changed from what Git reports (column *Changed*), the number of merge scenarios where the merge conflict remains the same as Git (column *Unchanged*), and the number of merge scenarios where the tool times out (column *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 5.2.

As Table 3 shows, across all evaluated merge scenarios, `IntelliMerge` was able to completely resolve 70 merge scenarios out of the 2,001 total scenarios (i.e., 3%) while `RefMerge` was able to completely resolve 122 (6%) scenarios. The number of merge scenarios each tool completely resolves indicates which tool can fully resolve more scenarios. However, not all scenarios have the potential to be fully resolved. The strengths of refactoring-aware merge tools is their ability to deal with refactoring conflicts. Thus, merge scenarios with *only* refactoring related conflicts have more potential to be fully resolved by a refactoring-aware merge tool. Of the 2,001 total conflicting scenarios, 453 (23%) scenarios contain *only* refactoring related conflicts. All 122 of `RefMerge`'s resolved scenarios are a subset of these 453 scenarios, whereas only 35 of `IntelliMerge`'s are. This means that `RefMerge` resolves 27% of such merge scenarios, whereas `IntelliMerge` resolves only 8%, in addition to 35 merge scenarios with other types of conflicts.

Looking at Table 3 from a project perspective, `IntelliMerge` and `RefMerge` are each able to completely resolve at least one scenario in 17 (85%) projects. Although both tools resolve merge scenarios in 17 projects, there are four projects where `IntelliMerge` resolves more merge scenarios than `RefMerge`, while there are 11 projects where `RefMerge` resolves more scenarios than `IntelliMerge`. This suggests that the characteristics of the scenarios in each project play a role in the tools' capabilities in resolving them.

We note that `RefMerge` times out on 382 merge scenarios across three different projects and `IntelliMerge` times out on 870 merge scenarios across 11 projects. To investigate the characteristics of the merge scenarios with time outs, Figure 4 shows the distribution of changed files per merge scenario where `RefMerge` times out, `IntelliMerge` times out, or neither tool times out. The median number of changed files in a merge scenario where `RefMerge` times out and `IntelliMerge` times out are 1,236 and 1,000, respectively. In all other merge scenarios where neither tool times out, the median number of changed files is 150 files. Thus, it seems that merge scenarios with a large number of changed files causes both tools to time out. This makes sense because as more files are changed, `IntelliMerge` needs to build more, and potentially larger, graphs and then match them, which leads to it taking more time in these steps. As for `RefMerge`, more changed files adds more work for RefactoringMiner which causes `RefMerge` to take more time in refactoring detection. Overall, `RefMerge` and `IntelliMerge` respectively time out on merge scenarios with eight times and seven times more changed files than those they do not time out on.

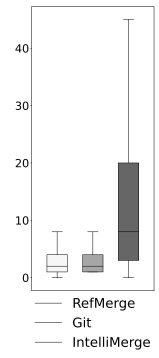
5.2 Merge Scenarios with Differences in Conflicts

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 5-7 to discuss these scenarios per project at the file, block, and lines of code levels respectively[4]. There are two parts to each figure. On the left-hand side, we provide a box plot of the overall

4. We use platform_fwk_supp as a shortened version of platform_frameworks_support for better table sizing for readability

TABLE 3: Merging results for each tool, compared to Git. Number 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	33 (4%)	108 (12%)	1 (0%)	780 (85%)	**54 (6%)**	204 (22%)	322 (35%)	342 (37%)
elasticsearch	178	3 (2%)	103 (58%)	0 (0%)	71 (40%)	**9 (5%)**	45 (25%)	85 (48%)	39 (22%)
gradle	118	1 (1%)	106 (90%)	0 (0%)	11 (9%)	**9 (8%)**	33 (28%)	75 (64%)	1 (1%)
antlr4	100	1 (1%)	96 (96%)	0 (0%)	3 (3%)	**4 (4%)**	39 (39%)	57 (57%)	0 (0%)
platform_fwk_support	95	5 (5%)	56 (59%)	1 (1%)	33 (35%)	**9 (10%)**	40 (42%)	46 (48%)	0 (0%)
deeplearning4j	93	3 (3%)	89 (96%)	0 (0%)	1 (1%)	**5 (5%)**	31 (33%)	57 (61%)	0 (0%)
realm-java	92	**7 (8%)**	82 (89%)	1 (1%)	2 (2%)	**7 (8%)**	32 (35%)	53 (58%)	0 (0%)
jackson-core	81	0 (0%)	81 (100%)	0 (0%)	0 (0%)	**3 (4%)**	25 (31%)	52 (64%)	0 (0%)
android	81	3 (4%)	78 (96%)	0 (0%)	0 (0%)	**8 (10%)**	28 (35%)	45 (56%)	0 (0%)
cometd	63	2 (3%)	59 (93%)	1 (2%)	1 (2%)	**4 (6%)**	20 (32%)	39 (62%)	0 (0%)
storm	33	**1 (3%)**	30 (91%)	1 (3%)	1 (3%)	**1 (3%)**	13 (39%)	19 (58%)	0 (0%)
ProjectE	30	**1 (3%)**	28 (94%)	0 (0%)	1 (3%)	0 (0%)	13 (43 %)	17 (57%)	0 (0%)
javaparser	23	**3 (13%)**	19 (83%)	1 (4%)	0 (0%)	2 (9%)	9 (39%)	12 (52%)	0 (0%)
druid	17	**3 (18%)**	15 (89%)	0 (0%)	0 (0%)	2 (12%)	9 (53%)	6 (35%)	0 (0%)
androidannotations	15	**1 (7%)**	14 (93%)	0 (0%)	0 (0%)	0 (0%)	9 (60%)	6 (40%)	0 (0%)
junit4	14	**1 (7%)**	13 (93%)	0 (0%)	0 (0%)	**1 (7%)**	8 (57%)	4 (29%)	0 (0%)
MinecraftForge	14	1 (7%)	10 (72%)	0 (0%)	3 (21%)	**2 (14%)**	7 (50%)	5 (36%)	0 (0%)
iFixitAndroid	13	0 (0%)	13 (100%)	0 (0%)	0 (0%)	0 (0%)	10 (77%)	3 (23%)	0 (0%)
MozStumbler	10	0 (0%)	10 (100%)	0 (0%)	0 (0%)	**1 (10%)**	7 (70%)	2 (20%)	0 (0%)
error-prone	9	**1 (11%)**	8 (88%)	0 (0%)	0 (0%)	**1 (11%)**	8 (88%)	0 (0%)	0 (0%)
Total	2,001	70 (3%)	1,017 (51%)	7 (0%)	907 (45%)	**122 (6%)**	592 (30%)	905 (45%)	382 (19%)



(a) Overall Distribution

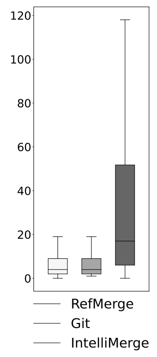
Project	**Reduced Confl. Files**		**Increased Confl. Files**	
	IntelliMerge	**RefMerge**	**IntelliMerge**	**RefMerge**
cassandra	1 (33%)	**34 (27%)**	105 (567%)	**33 (50%)**
elasticsearch	4 (20%)	**14 (21%)**	99 (260%)	**9 (67%)**
gradle	**7 (33%)**	**7 (25%)**	86 (494%)	**4 (100%)**
antlr4	2 (56%)	**11 (17%)**	92 (445%)	**4 (16%)**
platform_fwk_supp	**6 (68%)**	5 (50%)	45 (350%)	**4 (14%)**
deeplearning4j	**6 (67%)**	3 (50%)	61 (150%)	**9 (50%)**
realm-java	**11 (33%)**	7 (33%)	45 (200%)	**6 (58%)**
jackson-core	0 (0%)	**6 (29%)**	79 (850%)	**0 (0%)**
android	**6 (38%)**	4 (13%)	55 (150%)	**5 (23%)**
cometd	2 (48%)	**3 (25%)**	48 (658%)	**4 (15%)**
storm	**3 (33%)**	2 (42%)	21 (340%)	**0 (0%)**
ProjectE	**4 (12%)**	3 (8%)	24 (79%)	**4 (17%)**
javaparser	**5 (43%)**	4 (22%)	8 (150%)	**1 (100%)**
druid	**7 (43%)**	1 (29%)	**0 (0%)**	**0 (0%)**
androidannotations	1 (67%)	**2 (30%)**	13 (100%)	**0 (0%)**
junit4	**5 (33%)**	2 (42%)	7 (150%)	**4 (42%)**
MinecraftForge	1 (10%)	**1 (17%)**	9 (100%)	**1 (11%)**
iFixitAndroid	**5 (81%)**	6 (20%)	5 (78%)	**1 (33%)**
MozStumbler	1 (25%)	**4 (33%)**	5 (50%)	**0 (0%)**
error-prone	**4 (46%)**	3 (25%)	3 (750%)	**0 (0%)**
Total	81 (38%)	**122 (25%)**	810 (350%)	**89 (50%)**

(b) Breakdown by merge scenario

Fig. 5: Conflicting *files* in merge scenarios. Boxplot shows number of conflicting files per merge scenario while the table shows number of merge scenarios where a tool reduced/increased the number of conflicting files, compared to Git. In parenthesis, we show the median percentage reduction/increase per merge scenario.

distribution of reported conflicts at the discussed granularity level for all three tools across all evaluated scenarios. On the right-hand side, we show a table that provides the details of the conflicting scenarios from the *Changed* column of Table 3. 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 5b, 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.

Conflicting files: We first look at the conflicting file level in Figure 5. Figure 5a 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 5b, 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 810 scenarios (40%) by a median 350% 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 122 scenarios (6%) by a median 25% reduction while it increases the number of reported conflicting files for 89 (4%) by a median 50% increase.

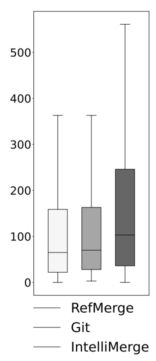


(a) Overall Distribution

Project	Reduced Confl. Blocks		Increased Confl. Blocks	
	IntelliMerge	RefMerge	IntelliMerge	RefMerge
cassandra	4 (29%)	**68 (25%)**	101 (767%)	**58 (27%)**
elasticsearch	5 (20%)	**19 (14%)**	94 (294%)	**13 (29%)**
gradle	**10 (29%)**	**10 (25%)**	91 (400%)	**14 (50%)**
antlr4	3 (20%)	**16 (8%)**	92 (479%)	**7 (9%)**
platform_fwk_supp	9 (50%)	**13 (20%)**	47 (508%)	**9 (13%)**
deeplearning4j	**18 (44%)**	13 (11%)	65 (167%)	**10 (33%)**
realm-java	**22 (50%)**	20 (28%)	47 (200%)	**5 (23%)**
jackson-core	0 (0%)	**13 (17%)**	80 (1580%)	**3 (19%)**
android	**20 (39%)**	11 (20%)	52 (119%)	**8 (32%)**
cometd	5 (26%)	**7 (33%)**	52 (790%)	**6 (21%)**
storm	3 (25%)	**4 (29%)**	25 (233%)	**0 (0%)**
ProjectE	4(15%)	**5 (8%)**	22 (94%)	**5 (11%)**
javaparser	3 (89%)	**5 (12%)**	11 (138%)	**1 (100%)**
druid	**12 (60%)**	3 (8%)	**0 (0%)**	1 (100%)
androidannotations	1 (89%)	**5 (33%)**	13 (114%)	**1 (25%)**
junit4	**5 (60%)**	4 (32%)	7 (233%)	**3 (20%)**
MinecraftForge	1 (5%)	**3 (10%)**	9 (100%)	**3 (20%)**
iFixitAndroid	6 (57%)	**8 (18%)**	7 (100%)	**1 (50%)**
MozStumbler	3 (50%)	**6 (20%)**	5 (86%)	**0 (0%)**
error-prone	2 (91%)	**5 (33%)**	5 (417%)	**0 (0%)**
Total	136 (48%)	**239 (21%)**	825 (378%)	**149 (27%)**

(b) Breakdown by merge scenario

Fig. 6: Conflicting *blocks* in merge scenarios. Boxplot shows number of conflicting blocks per merge scenario while the table shows number of merge scenarios where a tool reduced/increased the number of conflicting blocks, compared to Git. In parenthesis, we show the median percentage reduction/increase per merge scenario.



(a) Overall Distribution

Project	Reduced Confl. LOC		Increased Confl. LOC	
	IntelliMerge	RefMerge	IntelliMerge	RefMerge
cassandra	23 (36%)	**124 (26%)**	85 (291%)	**80 (14%**
elasticsearch	**39 (46%)**	28 (14%)	64 (121%)	**16 (28%)**
gradle	**37 (45%)**	20 (26%)	69 (149%)	**11 (61%)**
antlr4	19 (30%)	**21 (6%)**	75 (102%)	**18 (8%)**
platform_fwk_supp	**23 (59%)**	24 (20%)	33 (129%)	**16 (11%)**
deeplearning4j	**45 (45%)**	16 (14%)	41 (148%)	**15 (11%)**
realm-java	**53 (74%)**	25 (20%)	29 (68%)	**7 (25%)**
jackson-core	8 (48%)	**20 (14%)**	72 (420%)	**6 (10%)**
android	**54 (58%)**	18 (15%)	23 (119%)	**9 (14%)**
cometd	**15 (51%)**	9 (36%)	43 (230%)	**11 (9%)**
storm	**14 (39%)**	12 (18%)	14 (113%)	**1 (3%)**
ProjectE	**18 (45%)**	8 (3%)	10 (75%)	**5 (9%)**
javaparser	**11 (59%)**	8 (24%)	**8 (225%)**	**0 (0%)**
druid	**14 (85%)**	6 (19%)	**0 (0%)**	2 (32%)
androidannotations	4 (87%)	**8 (19%)**	10 (42%)	**1 (7%)**
junit4	**10 (65%)**	5 (20%)	**3 (18%)**	4 (8%)
MinecraftForge	**4 (27%)**	3 (11%)	**4 (150%)**	**4 (16%)**
iFixitAndroid	**7 (34%)**	5 (13%)	**5 (55%)**	**5 (21%)**
MozStumbler	**7 (60%)**	6 (31%)	3 (38%)	**1 (12%)**
error-prone	3 (84%)	**6 (47%)**	5 (187%)	**2 (25%)**
Total	**408 (51%)**	372 (21%)	597 (169%)	**214 (14%)**

(b) Breakdown by merge scenario

Fig. 7: Conflicting *LOC* in merge scenarios .Boxplot shows number of conflicting LOC per merge scenario while the table shows number of merge scenarios where a tool reduced/increased the number of conflicting LOC, compared to Git. In parenthesis, we show the median percentage reduction/increase per merge scenario.

Conflict Blocks: We now look at the conflict block level in Figure 6b. The number of conflict blocks indicates the number of individual conflicting regions a developer needs to deal with. Figure 6a 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 17. From Figure 6b, we find that `IntelliMerge` reduces the number of reported conflict blocks for 136 scenarios (7%) by a median 48% reduction, while it increases the number of reported conflict blocks for 825 scenarios (41%) by a median of 378%. On the other hand, `RefMerge` reduces the number of reported conflicts in 239 scenarios (12%) by a median 21% reduction and increases the number of reported conflicts for 197 scenarios (7%) by a median of 27% increase. Additionally, `IntelliMerge` has a high variance between projects. For example, consider project *druid*. `IntelliMerge` reduces conflict blocks in 12 merge scenarios by more than half (60%) without increasing them in any merge scenarios. Inversely, it increases conflict blocks in 80 merge scenarios for project *jackson-core* by almost 16-fold without decreasing conflict blocks in any merge scenarios. To investigate the cause of the variance, we look at Figure 3. As shown, *druid* has the lowest median number of refactorings in each scenario, while the four projects that `IntelliMerge` struggles most with (*cassandra*, *antlr4*, *jackson-core*, and *cometd*) have the highest median number of refactorings. This suggests that merge scenarios with a large number of refactorings complicates the merge resolution for `IntelliMerge`. `RefMerge` does not exhibit the same variance as `IntelliMerge`, typically reducing conflicts more often than increasing conflicts or reducing them as often as it increases them. To explore how the number of refactorings performed in a project affects `RefMerge`, we look at *MozStumbler*, *error-prone*, and *antlr4* because `RefMerge` does well on them. We additionally look at *gradle* because `RefMerge` does the worst on it. *MozStumbler* has the third lowest median number of refactorings per scenario while *antlr4* has the second highest median number of refactorings per scenario. Projects *error-prone* and *gradle* respectively have the 9th and 8th highest median number of refactorings per scenario. This suggests that `RefMerge` is not affected by the number of refactorings in a project.

Conflicting Lines of Code: Finally, we look at the conflicting lines of code (LOC) in Figure 7, which measures the total number of lines in all conflict blocks/regions of a merge scenario. From Figure 7a, we observe similar behavior of the tools as what we observed for the conflicting files in Figure 5a. More closely from the table in Figure 7b, we find that `IntelliMerge` reduces the number of conflicting LOC in 408 scenarios (20%) by a median 51% reduction, while it increases the conflicting LOC for 597 (30%) scenarios by a median 169% increase. `RefMerge` reduces the conflicting LOC in 372 scenarios (19%) by a median 21% reduction and increases the conflicting LOC in 214 scenarios (11%) by a median 14% increase. Note that the discrepancy between `IntelliMerge`'s increased 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 conflicting LOC per block (as opposed to a whole scenario) in Figure 8. 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.

5.3 Summary and Interpretation of RQ1 Results

The above results indicate that `RefMerge` completely resolves almost twice as many merge scenarios as `IntelliMerge` (122 versus 70). Of the 453 merge scenarios with *only* refactoring-related conflicts, `IntelliMerge` and `RefMerge` respectively resolve 8% and 27%.

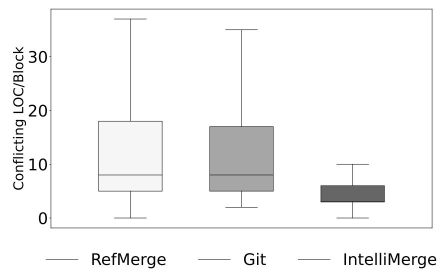


Fig. 8: Conflicting lines of code per conflict block.

While `IntelliMerge` is able to reduce conflicting LOC for a higher portion of scenarios than `RefMerge` (51% versus 21%), 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. Furthermore, there is a large variance in how `IntelliMerge` does across the projects. We find that `IntelliMerge` does well on projects whose merge scenarios have a low number of refactorings and struggles with projects that have several refactorings per merge scenario. Additionally, `IntelliMerge` times out on a higher number of merge scenarios than `RefMerge`. In our investigation into why each tool times out, we found that merge scenarios with more changed files typically cause both tools to time out. Thus, it seems `IntelliMerge` works extremely well for merge scenarios with few changes and a small number of refactorings, but actually makes it much worse for other scenarios. This makes sense because `IntelliMerge` relies on a similarity score to detect refactorings and a large number of changes make it harder to detect refactorings. Overall, taking the total number of scenarios it can completely resolve (70 from Table 3) and the ones in which it can reduce the total number of conflicting LOC for (408 from Figure 7b), `IntelliMerge` can help the developer deal with less conflicts in 478 scenarios (24% of overall scenarios). However, taking both timeouts (907 scenarios from Table 3) and worsened results in terms of overall conflicting LOC (597 scenarios from Figure 7b), `IntelliMerge` will not help the developer in the remaining 1,504 (75%) scenarios, and will in fact make it worse for them in almost a third of those.

On the other hand, `RefMerge` completely resolves or reduces the number of conflicting LOC for 497 scenarios (25% of overall scenarios). Thus, `RefMerge` helps the developer in around the same number of merge scenarios as `IntelliMerge` but it times out or worsens the situation at a much lower rate, only 596 (30%). Additionally, the median percentage increase for `RefMerge` in terms of conflicting LOC is much lower at 14% as opposed to 169% 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 while overall `RefMerge` seems to be doing better quantitatively, it is obvious that the characteristics or difficulty of a merge scenario impact the results in some way. One tool may be able to handle certain types of merge scenarios better than the other, and we do not have information about the correctness of the resolutions. 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: `RefMerge` completely resolves double the total number of merge scenarios as `IntelliMerge` (122 (6%) vs 70 (3%)). Overall, `IntelliMerge` can help the developer completely resolve conflicts or deal with less conflicts, and thus improve the situation, in 478 scenarios (24%), but at the cost of increasing conflicting LOC, thus worsening the conflicts, in 597 (30%) scenarios. In contrast, `RefMerge` improves the situation in 497 (25%) scenarios, and worsens it in only 214 (11%) scenarios by a lower LOC increase rate.

6 RQ2: Discrepancies between the tools

RQ1 quantitative results 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 conflicts in a merge scenario where conflicts should be reported. Similarly, the reported conflicts may not be real conflicts. Thus, we perform a qualitative study for RQ2 to dig deeper into the reported results.

6.1 Research Method

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 by completely resolving the merge scenario, or equally increasing/reducing the number of Git conflicts. (2) `IntelliMerge` outperforms `RefMerge` by completely resolving the scenario or reporting a lower number of conflicts at any granularity level and (3) `RefMerge` outperforms `IntelliMerge`. We also try to evenly sample across projects.

Analysis Method: Our manual analysis goal is to analyze the conflicts reported by all three tools across the sampled scenarios. To investigate if a merge conflict is a true/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 because it requires developer intervention, we label this conflict region as a *true positive*. If it should not result in a merge conflict (i.e., a tool should be able to automatically resolve it), we label it as a *false positive*. If the other merge tools do not report the same 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 other tool(s) missed the conflict). We also investigate and categorize the reasons behind false positives and false negatives for each tool. This process takes an average of 63 minutes per merge scenario.

TABLE 4: Comparing the false positives and false negatives reported by each tool, across the 50 sampled scenarios.

	RefMerge	Git	IntelliMerge
# Conflict Blocks Investigated	379	433	866
True Positives	191	190	159
False Positives	188	243	707
False Negatives	0	5	71

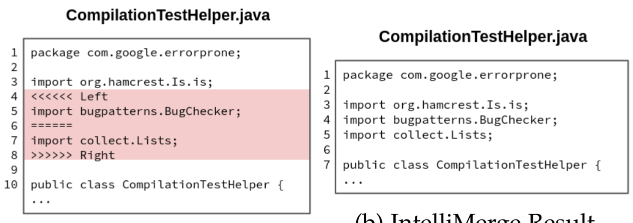
TABLE 5: The reason for each false positive and false negative reported by Git, as well as the frequency for each reason.

Git Reasons	Type	Frequency
No Refactoring Handling	False Positive	140
Ordering Conflict	False Positive	61
Formatting Conflict	False Positive	41
No Refactoring Handling	False Negative	5
Total		248

6.2 Results

Table 4 shows the total number of conflict blocks that we manually 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 5 false negatives. `IntelliMerge` reports 707 false positives and 71 false negatives. Meanwhile, `RefMerge` reports 188 false positives and no false negatives. When compared to Git, `RefMerge` reduces the number of false positives by 23% and completely eliminates false negatives, while `IntelliMerge` increases the number of false positives and false negatives by 192% and 1,320%. We also show the number of true positives reported by each tool. While Git and `RefMerge` report a total of 190 and 191 true positives respectively, `IntelliMerge` reports only 159 true positives.

False Positives/Negatives Git Results: Table 5 shows the reasons behind the false positives and false negatives for Git. There are 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 140 (58%) false positive conflicts that Git reports that 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 5 also shows that 61 (25%) 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]. Consider Figure 9 where two import statements are added to the same location in `CompilationTestHelper.java`. Git does not know which order to put the new lines in, even though they can be put in any order. Finally, the remaining 41 (17%) of Git's false positives are *formatting conflicts* that are caused by different formatting between branches, such as additional white space or a new line on one branch that does not exist on the other. Figure 10 shows a formatting conflict where the left branch does not have any space between the parameters while the right branch added a space.



CompilationTestHelper.java

```
package com.google.errorprone;

import org.hamcrest.Is.is;
<<<<<< Left
import bugpatterns.BugChecker;
======
import collect.Lists;
>>>>>> Right

public class CompilationTestHelper {
...
```

(a) Git and RefMerge Result

CompilationTestHelper.java

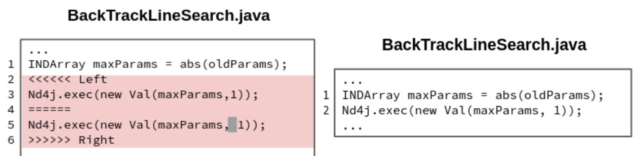
```
package com.google.errorprone;

import org.hamcrest.Is.is;
import bugpatterns.BugChecker;
import collect.Lists;

public class CompilationTestHelper {
...
```

(b) IntelliMerge Result

Fig. 9: An ordering conflict reported by Git and `RefMerge` along with the correct merge resolution by `IntelliMerge` (error-prone [`07559b47`]).



BackTrackLineSearch.java

```
...
INDArray maxParams = abs(oldParams);
<<<<<< Left
Nd4j.exec(new Val(maxParams,1));
======
Nd4j.exec(new Val(maxParams, 1));
>>>>>> Right
...
```

(a) Git and RefMerge Result

BackTrackLineSearch.java

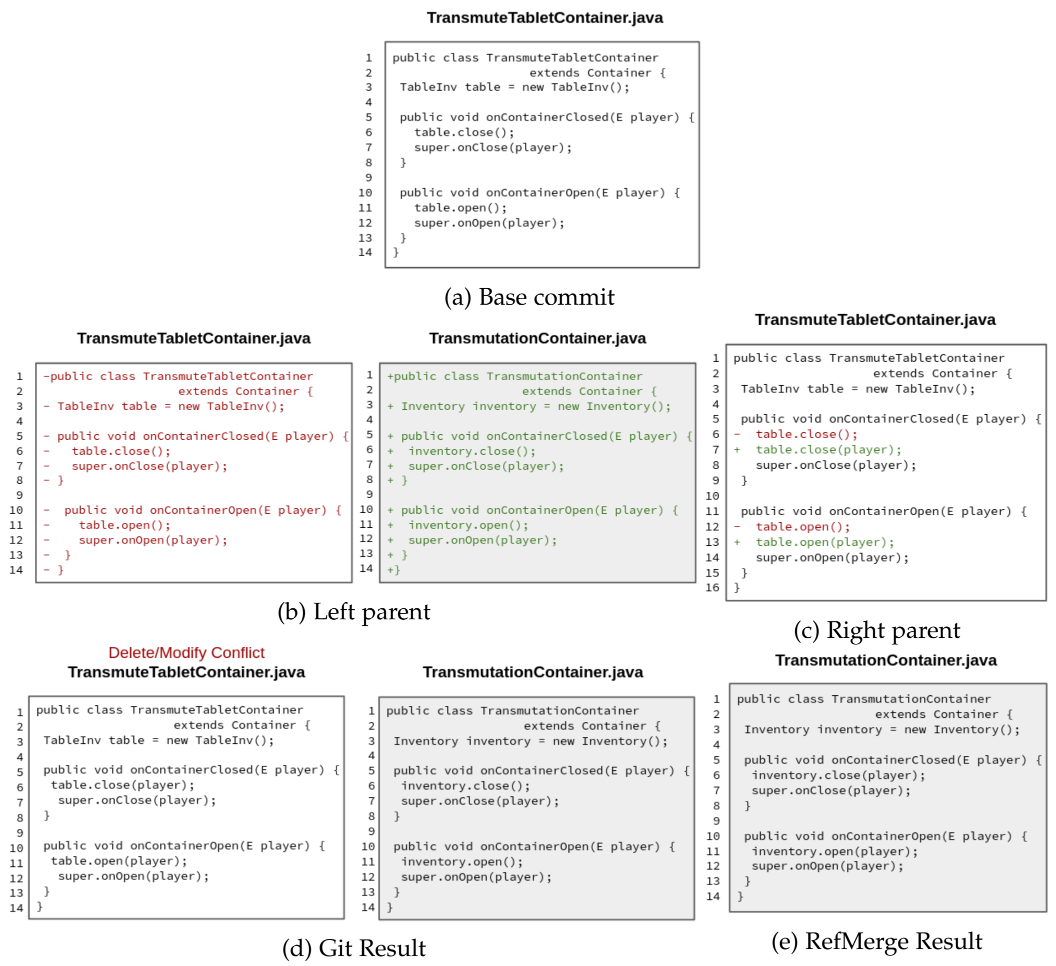
```
...
INDArray maxParams = abs(oldParams);
Nd4j.exec(new Val(maxParams, 1));
...
```

(b) IntelliMerge Result

Fig. 10: A formatting conflict reported by Git and `RefMerge` along with the expected merge resolution by `IntelliMerge` (deeplearning4j [`6c285324`]).

Not being able to handle refactorings also causes Git's results to include false negatives caused by syntax errors. Figure 11 presents a merge scenario where Git results in a false positive as well as false negatives. On the left branch, file `TransmuteTabletContainer.java` is renamed to `TransmutationContainer.java` and class `TransmuteTabletContainer` is renamed to class `TransmutationContainer`. Additionally, class `TableInv` (which is the type of field `table` on Line 3) is renamed to `Inventory` as part of a *Rename Class* refactoring. Note that `TableInv` is a class in `TableInv.java`, which we do not show in Figure 11 for better visualization. To match the type update, field `table` is also renamed to `inventory`. On the right branch, an *Add Parameter* refactoring was performed on the definitions of methods `close` and `open` in `TableInv` (not shown), which resulted in the update of the function calls on Lines 6 and 12 of Figure 11c.

We can see Git's merge result in Figure 11d where it reports a delete/modify conflict in `TransmuteTabletContainer.java`, because it mistakenly sees that the left branch deleted the whole file (i.e., does not see this as a renamed class) while the right branch changed it. This is false positive conflict, because the developer will need to deal with this reported conflict, when a tool that considers refactoring semantics can avoid this altogether. Note that when Git reports a delete/modify conflict, it does not physically delete the file because it waits for the developer's resolution. Git also does not try to detect conflicts on the internal content of the file. Moreover, in Git's resolution, it shows the newly added file `TransmutationContainer.java` as is and does not provide any context that it is related to `TransmuteTabletContainer.java`. Worse, Git's resolution of `TransmutationContainer.java` contains compilation errors, because the `close()` and `open()` calls are missing the added parameter. This is a false negative.



TransmuteTabletContainer.java

```
1  public class TransmuteTabletContainer
2                     extends Container {
3   TableInv table = new TableInv();
4
5   public void onContainerClosed(E player) {
6     table.close();
7     super.onClose(player);
8   }
9
10  public void onContainerOpen(E player) {
11    table.open();
12    super.onOpen(player);
13  }
14 }
```

(a) Base commit

TransmuteTabletContainer.java

```
1  -public class TransmuteTabletContainer
2                     extends Container {
3  - TableInv table = new TableInv();
4
5  - public void onContainerClosed(E player) {
6  -   table.close();
7  -   super.onClose(player);
8  - }
9
10 -  public void onContainerOpen(E player) {
11 -    table.open();
12 -    super.onOpen(player);
13 -  }
14 - }
```

TransmutationContainer.java

```
1  +public class TransmutationContainer
2                     extends Container {
3  + Inventory inventory = new Inventory();
4
5  + public void onContainerClosed(E player) {
6  +   inventory.close();
7  +   super.onClose(player);
8  + }
9
10 + public void onContainerOpen(E player) {
11 +   inventory.open();
12 +   super.onOpen(player);
13 + }
14 +}
```

(b) Left parent

TransmuteTabletContainer.java

```
1  public class TransmuteTabletContainer
2                     extends Container {
3   TableInv table = new TableInv();
4
5   public void onContainerClosed(E player) {
6  -  table.close();
7  +  table.close(player);
8     super.onClose(player);
9   }
10
11  public void onContainerOpen(E player) {
12 -  table.open();
13 +  table.open(player);
14    super.onOpen(player);
15  }
16 }
```

(c) Right parent

Delete/Modify Conflict
TransmuteTabletContainer.java

```
1  public class TransmuteTabletContainer
2                     extends Container {
3   TableInv table = new TableInv();
4
5   public void onContainerClosed(E player) {
6    table.close(player);
7     super.onClose(player);
8   }
9
10  public void onContainerOpen(E player) {
11   table.open(player);
12    super.onOpen(player);
13  }
14 }
```

TransmutationContainer.java

```
1  public class TransmutationContainer
2                     extends Container {
3   Inventory inventory = new Inventory();
4
5   public void onContainerClosed(E player) {
6    inventory.close();
7    super.onClose(player);
8   }
9
10  public void onContainerOpen(E player) {
11   inventory.open();
12   super.onOpen(player);
13  }
14 }
```

(d) Git Result

TransmutationContainer.java

```
1  public class TransmutationContainer
2                     extends Container {
3   Inventory inventory = new Inventory();
4
5   public void onContainerClosed(E player) {
6    inventory.close(player);
7    super.onClose(player);
8   }
9
10  public void onContainerOpen(E player) {
11   inventory.open(player);
12   super.onOpen(player);
13  }
14 }
```

(e) RefMerge Result

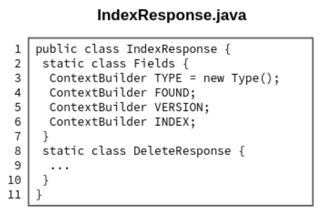
Fig. 11: The three versions (base, left, and right) of code involved in two of Git's false negatives, as well as the results merged by Git and `RefMerge`'s result (ProjectE [`12b0545e`]) .

Figure 11e shows `RefMerge`'s resolution, which illustrates the strengths of a refactoring-aware merging tool. We can see that no modify/delete conflict is reported and that `TransmuteTabletContainer.java` is correctly deleted. In addition, field `table` is renamed to `inventory`, along with the rename of its type from `TableInv` to `Inventory`. Although `RefMerge` does not support *Add Parameter*, the added parameters are the only changes after `RefMerge` inverts *Rename Class* and *Rename Field*, resulting in the correct method calls on lines 6 and 11. We note that in this scenario, `IntelliMerge` incorrectly deletes classes `TableInv` and `TransmuteTabletContainer` instead of renaming them. In `IntelliMerge`'s result, files `TransmuteTabletContainer.java`, `TransmutationContainer.java`, `TableInv.java`, and `Inventory.java` do not exist, resulting in an incorrect merge.

False positive/negative `RefMerge` Results: Table 6 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 61 ordering conflicts and 41 formatting false positives as Git. `RefMerge` reports an additional 5 ordering conflicts and 16 formatting conflicts that arise from its refactoring handling, totalling 66 (35%) ordering conflicts and 55 (29%) formatting conflicts. All of the additional ordering conflicts are caused by move-related refactorings such as `Move Inner Class` or `Pull Up Method` being moved to the correct class but not being moved to the correct location within the file. For example, Figure 12 shows a merge scenario where `RefMerge` introduces a refactoring-related ordering conflict. In this example, the left branch changes the field declaration `ContextBuilder Type = new Type()` to `ContextBuilder TYPE = null`. The right branch moves field `TYPE` from inner class `IndexResponse.Fields` to inner class `IndexResponse.DeleteResponse`. This results in Git reporting the conflict in Figure 12d be-

TABLE 6: The reason and frequency for false positives and false negatives reported by `RefMerge`.

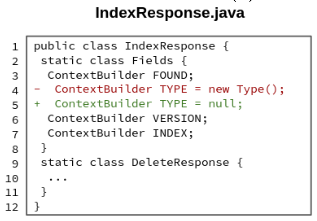
`RefMerge` **Reasons**	**Type**	**Frequency**
Ordering Conflict	False Positive	61
Formatting Conflict	False Positive	41
Unsupported Refactoring	False Positive	41
Refactoring-related Formatting Conflict	False Positive	16
Fails to Invert Refactoring	False Positive	14
IntelliJ Optimization	False Positive	5
Undetected Refactoring	False Positive	5
Refactoring-related Ordering Conflict	False Positive	5
Total		188



IndexResponse.java

```
1  public class IndexResponse {
2   static class Fields {
3    ContextBuilder TYPE = new Type();
4    ContextBuilder FOUND;
5    ContextBuilder VERSION;
6    ContextBuilder INDEX;
7   }
8   static class DeleteResponse {
9    ...
10  }
11 }
```

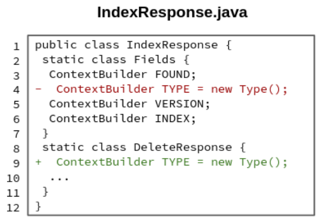
(a) Base Commit



IndexResponse.java

```
1  public class IndexResponse {
2   static class Fields {
3    ContextBuilder FOUND;
4  -  ContextBuilder TYPE = new Type();
5  +  ContextBuilder TYPE = null;
6    ContextBuilder VERSION;
7    ContextBuilder INDEX;
8   }
9   static class DeleteResponse {
10   ...
11  }
12 }
```

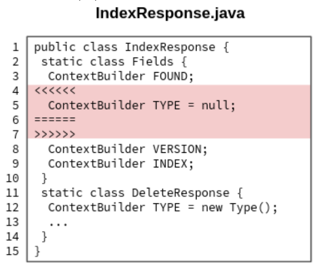
(b) Left Commit



IndexResponse.java

```
1  public class IndexResponse {
2   static class Fields {
3    ContextBuilder FOUND;
4  -  ContextBuilder TYPE = new Type();
5    ContextBuilder VERSION;
6    ContextBuilder INDEX;
7   }
8   static class DeleteResponse {
9  +  ContextBuilder TYPE = new Type();
10   ...
11  }
12 }
```

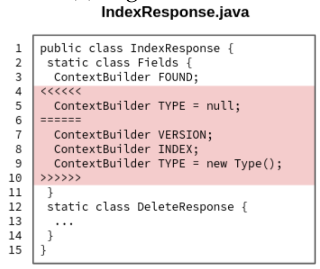
(c) Right Commit



IndexResponse.java

```
1  public class IndexResponse {
2   static class Fields {
3    ContextBuilder FOUND;
4  <<<<<<
5    ContextBuilder TYPE = null;
6  ======
7  >>>>>>
8    ContextBuilder VERSION;
9    ContextBuilder INDEX;
10  }
11  static class DeleteResponse {
12   ContextBuilder TYPE = new Type();
13   ...
14  }
15 }
```

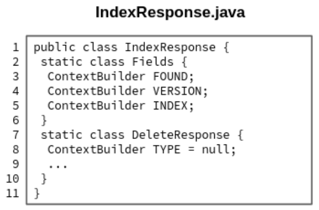
(d) Git Result



IndexResponse.java

```
1  public class IndexResponse {
2   static class Fields {
3    ContextBuilder FOUND;
4  <<<<<<
5    ContextBuilder TYPE = null;
6  ======
7    ContextBuilder VERSION;
8    ContextBuilder INDEX;
9    ContextBuilder TYPE = new Type();
10 >>>>>>
11  }
12  static class DeleteResponse {
13   ...
14  }
15 }
```

(e) RefMerge Result



IndexResponse.java

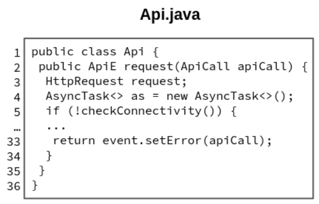
```
1  public class IndexResponse {
2   static class Fields {
3    ContextBuilder FOUND;
4    ContextBuilder VERSION;
5    ContextBuilder INDEX;
6   }
7   static class DeleteResponse {
8    ContextBuilder TYPE = null;
9    ...
10  }
11 }
```

(f) IntelliMerge Result

Fig. 12: The three versions (base, left, and right) of code involved in a refactoring-related ordering conflict that RefMerge introduces while along with Git's and IntelliMerge's results (elasticsearch [503a166b]).

cause Git sees that the left branch made a change to the line that the right branch deleted. When RefMerge inverts the *Move Field* refactoring, it correctly moves TYPE to IndexResponse.Fields but it moves it to the wrong location textually, moving it to line 9 instead of line 5. If RefMerge inverts TYPE to the correct location, RefMerge would be able to resolve the conflict. However, as shown in Figure 12e, VERSION and INDEX are unnecessarily part of the conflict while they originally were not. While RefMerge complicates this conflict, IntelliMerge resolves the conflict by moving TYPE = null to IndexResponse.DeleteResponse, as shown in Figure 12f.

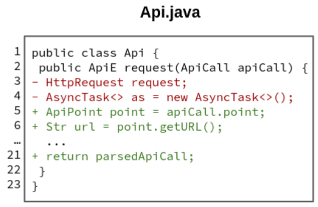
The additional formatting conflicts are caused by formatting differences from inverting refactorings, also typically Move Method and Move Inner Class refactorings. In these conflicts, RefMerge resolves the refactoring conflict but leaves a formatting conflict that usually consists of two lines with different amounts of white space. Figure 13 provides an example where RefMerge results in a larger



Api.java

```
public class Api {
 public ApiE request(ApiCall apiCall) {
  HttpRequest request;
  AsyncTask<> as = new AsyncTask<>();
  if (!checkConnectivity()) {
  ...
   return event.setError(apiCall);
  }
 }
}
```

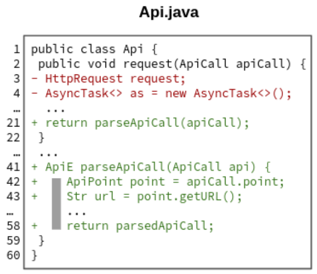
(a) Base Commit



Api.java

```
public class Api {
 public ApiE request(ApiCall apiCall) {
- HttpRequest request;
- AsyncTask<> as = new AsyncTask<>();
+ ApiPoint point = apiCall.point;
+ Str url = point.getURL();
  ...
+ return parsedApiCall;
 }
}
```

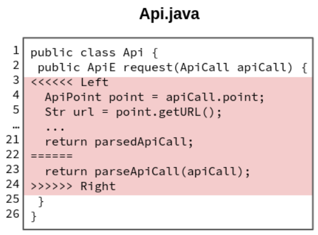
(b) Left Commit



Api.java

```
public class Api {
 public void request(ApiCall apiCall) {
- HttpRequest request;
- AsyncTask<> as = new AsyncTask<>();
  ...
+ return parseApiCall(apiCall);
 }
 ...
+ ApiE parseApiCall(ApiCall api) {
+    ApiPoint point = apiCall.point;
+    Str url = point.getURL();
     ...
+    return parsedApiCall;
 }
}
```

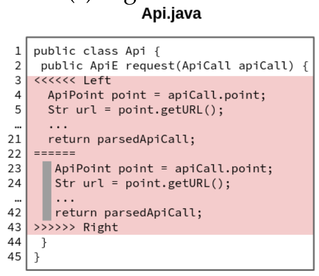
(c) Right Commit



Api.java

```
public class Api {
 public ApiE request(ApiCall apiCall) {
<<<<<< Left
  ApiPoint point = apiCall.point;
  Str url = point.getURL();
  ...
  return parsedApiCall;
======
  return parseApiCall(apiCall);
>>>>>> Right
 }
}
```

(d) Git and IntelliMerge Result



Api.java

```
public class Api {
 public ApiE request(ApiCall apiCall) {
<<<<<< Left
  ApiPoint point = apiCall.point;
  Str url = point.getURL();
  ...
  return parsedApiCall;
======
   ApiPoint point = apiCall.point;
   Str url = point.getURL();
   ...
   return parsedApiCall;
>>>>>> Right
 }
}
```

(e) RefMerge Result

Fig. 13: The three versions (base, left, and right) of code involved in a refactoring-related formatting conflict that RefMerge introduces along with Git's and IntelliMerge's results (iFixitAndroid [91932083]).

formatting conflict. In this example, the right branch makes the same changes to method request as the left branch, except the right branch also extracts the changes to method parseApiCall. As shown in Figure 13d, Git reports a conflict involving *Extract Method*, but this conflict has potential to be resolved by RefMerge and IntelliMerge. Figure 13e shows that RefMerge inverts the *Extract Method* refactoring but has more space on each line than the right branch. We mark the additional space on the right branch with gray to make the spacing difference clear. The difference in spacing results in a formatting conflict twice as large as the conflict reported by Git. It is worth noting that whenever RefMerge inverts an *Extract Method* refactoring and it cannot resolve the conflict, RefMerge usually results in a conflict twice as large. While we typically associate larger conflicting regions with a more difficult merge resolution, we believe that RefMerge's process has its own merits. For example, while Git and IntelliMerge report a smaller conflicting region than RefMerge, a developer trying to resolve the merge conflict first has to find the extracted method, then compare the changes between the two methods, and finally find the commit the method was extracted from to understand what the method looked like before the extract method refactoring. While RefMerge doubles the size of the conflict since it has the complete bodies of the two versions of the method (with and without the extracted code), the changes are shown side by side so a

TABLE 7: The reason and frequency of false positives and false negatives reported by `IntelliMerge`.

IntelliMerge Reasons	**Type**	**Frequency**
Matching Error	False Positive	646
Undetected Refactoring	False Positive	37
Ordering Conflict	False Positive	10
Incorrectly Detected Refactoring	False Positive	6
Unsupported Refactoring	False Positive	5
Formatting Conflict	False Positive	3
Deletes Conflict Block	False Negative	45
Matching Error	False Negative	21
Incorrectly Detected Refactoring	False Negative	5
Total		778

developer does not need to search for the extracted method. Additionally, it is more clear what the method body changes were without the extracted method's parameters making things less clear. Thus, `RefMerge` actually provides additional information about the refactorings originally involved within the conflicting region to provide the context needed to resolve the conflict.

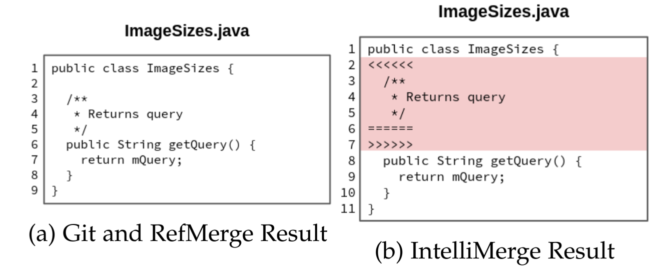
In 41 (22%) of the false positives that `RefMerge` reports, the underlying issue is a refactoring that is not supported in the current implementation. For example, a merge conflict in MinecraftForge ([`f5781488`]) contains an *Add Parameter* refactoring and while `RefMerge` does resolve the conflict, `IntelliMerge` does.

`RefMerge` reports 14 (7%) false positives involving refactoring conflicts it supports but that it fails to resolve, because it could not invert the refactoring. Nine of the refactorings that `RefMerge` fails to invert are *Rename Method* refactorings. After investigation, we found that these are typically caused by `RefMerge` being unable to find the refactored program element in IntelliJ's AST due to technical issues in our code, which we plan to fix.

There are five (3%) 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.

Finally, there are five (3%) false positives that are due to *undetected refactorings* that RefactoringMiner did not detect. In these scenarios, there are several methods that are similar, both in structure and naming, which likely made it difficult for RefactoringMiner to detect the refactoring. We reported the issue to the RefactoringMiner developers.

False positive/negative `IntelliMerge` Results: Table 7 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 10 (1%) false positives due to ordering conflicts and also has 37 (5%) because of undetected refactorings. `IntelliMerge` also sometimes fails to detect a refactoring, most commonly with parameter-level refactorings (14), *Extract Method* (9), and `Rename Class` (5). The undetected refactorings can be split into two groups: (1) the presence of several similar program elements drops the correct refactored program element below the similarity threshold, and (2) the presence of several changes to a program element cause `IntelliMerge` to think that the refactored program element is an addition. For example, `IntelliMerge` missed the *Extract Method* refactoring in Figure 13 that `RefMerge` was able to detect. In this specific scenario, several methods had `parse` or `ApiCall` in their name, resulting in an incorrect match. In addition, there are 125 refactorings that were performed in class `Api`, resulting in several changes that made it more difficult for `IntelliMerge` to detect the refactoring.


ImageSizes.java

```
public class ImageSizes {

  /**
   * Returns query
   */
  public String getQuery() {
    return mQuery;
  }
}
```

(a) Git and RefMerge Result

ImageSizes.java

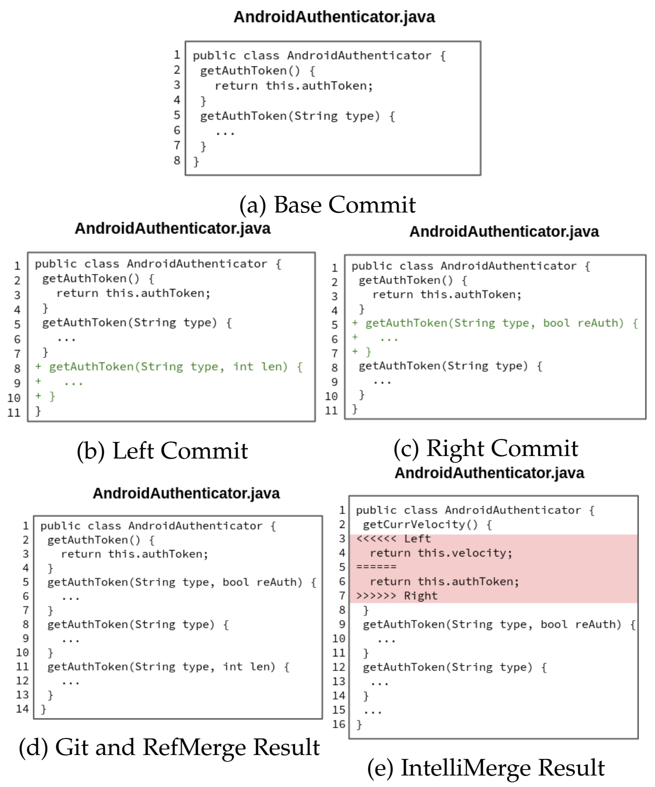
```
public class ImageSizes {
<<<<<<
  /**
   * Returns query
   */
======
>>>>>>
  public String getQuery() {
    return mQuery;
  }
}
```

(b) IntelliMerge Result

Fig. 14: A conflict reported by `IntelliMerge` caused by a matching error along with Git's result (iFixitAndroid [`91932083`]).

Note that, unlike Git and `RefMerge`, `IntelliMerge` reports only three false positives related to formatting conflicts. On the other hand, 646 of `IntelliMerge`'s false positives (91%) 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 where `IntelliMerge` cannot find matches for these nodes and assumes they were deleted. In Figure 14, Git does not report a merge conflict. As seen in Figure 14b, `IntelliMerge` incorrectly reports a conflict that indicates that the comment was changed on one branch and deleted on the other. In this example, the left branch added method `getQuery` along with `getQuery`'s comment to class `ImageSizes`. The right branch does not add any changes to class `ImageSizes`. While conflicts caused by matching errors are typically small, they happen frequently and the developer needs to spend time to investigate the conflict and decide that they can ignore it.

Six false positives are caused by `IntelliMerge` incorrectly detecting a refactoring that was never performed. There are six cases where `IntelliMerge` incorrectly matches methods in different classes, resulting in `IntelliMerge` incorrectly moving a method and reporting a conflict in its method body. In Figure 15, Git does not report a merge conflict. As shown in Figure 15b, the developers add method `getAuthToken(String type, int len)` to `AndroidAuthenticator.java`. On the right branch, the developers add method `getAuthToken(String type, bool reAuth)` to `AndroidAuthenticator.java`. The right branch also adds method `getCurrVelocity` to class `ScrollerCompat.java`, not shown in this figure. As shown in Figure 15d, both Git and `RefMerge` do not report a conflict. Figure 15e shows the result for `IntelliMerge`. `IntelliMerge` deletes



AndroidAuthenticator.java

```
public class AndroidAuthenticator {
 getAuthToken() {
   return this.authToken;
 }
 getAuthToken(String type) {
   ...
 }
}
```

(a) Base Commit

AndroidAuthenticator.java

```
public class AndroidAuthenticator {
 getAuthToken() {
   return this.authToken;
 }
 getAuthToken(String type) {
   ...
 }
+ getAuthToken(String type, int len) {
+   ...
+ }
}
```

(b) Left Commit

AndroidAuthenticator.java

```
public class AndroidAuthenticator {
 getAuthToken() {
   return this.authToken;
 }
+ getAuthToken(String type, bool reAuth) {
+   ...
+ }
 getAuthToken(String type) {
   ...
 }
}
```

(c) Right Commit

AndroidAuthenticator.java

```
public class AndroidAuthenticator {
 getAuthToken() {
   return this.authToken;
 }
 getAuthToken(String type, bool reAuth) {
   ...
 }
 getAuthToken(String type) {
   ...
 }
 getAuthToken(String type, int len) {
   ...
 }
}
```

(d) Git and RefMerge Result

AndroidAuthenticator.java

```
public class AndroidAuthenticator {
 getCurrVelocity() {
<<<<<< Left
  return this.velocity;
======
  return this.authToken;
>>>>>> Right
 }
 getAuthToken(String type, bool reAuth) {
   ...
 }
 getAuthToken(String type) {
  ...
 }
 ...
}
```

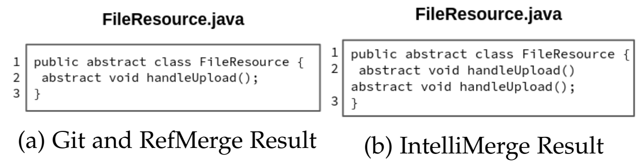
(e) IntelliMerge Result

Fig. 15: The three versions (base, left, and right) of code involved in a conflict reported by `IntelliMerge` caused by incorrectly detecting a refactoring along with the results from Git and `RefMerge` (platform_fmwk_support [`6c371fc5`]).

`getCurrVelocity` in `ScrollerCompat.java` and replaces `getAuthToken()` in `AndroidAuthenticator` with `getCurrVelocity()`. `IntelliMerge` reports a conflict in `getCurrVelocity` with the changes added to `getCurrVelocity` on the left side and the already existing method body of method `getAuthToken` on the right, resulting in a false positive.

The remaining 5 false positives are caused by unsupported refactorings. Both `IntelliMerge` and `RefMerge` do not support *Extract Superclass*.

`IntelliMerge` results in 45 (63%) false negatives because it incorrectly deletes the changes made in the left and right branches, causing it to completely delete code that should have resulted in conflict blocks. We find that `IntelliMerge` frequently incorrectly deletes complete classes that are involved in *Rename Class* or *Move Class* refactorings, such as when it deleted `TransmuteTabletContainer.java` in the example in Figure 11. This is likely caused by a matching error where if `IntelliMerge` cannot find a match for program elements in the base commit, it assumes these elements were deleted and removes them accordingly. Note that this is a lower bound for how many times `IntelliMerge` could have deleted other files or program elements that were not part of a conflict block and since we focus on the reported conflicts by each tool, we would have missed this happening in files where no merge tool reports a conflict. For example, there is a merge scenario in MinecraftForge ([`c3559b2d`]) where `IntelliMerge` deletes every file in the scenario except for one, including six files that should contain conflicts.



FileResource.java

```
1 public abstract class FileResource {
2  abstract void handleUpload();
3 }
```

(a) Git and RefMerge Result

FileResource.java

```
1 public abstract class FileResource {
2  abstract void handleUpload()
  abstract void handleUpload();
3 }
```

(b) IntelliMerge Result

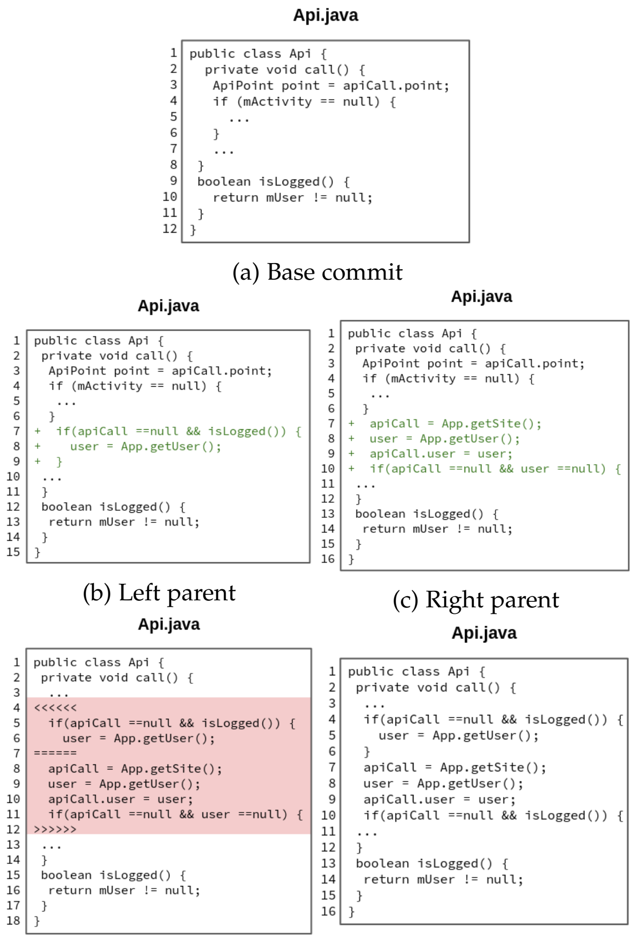
Fig. 16: A syntax error (Duplicate declaration) that `IntelliMerge`'s merge resolution introduces versus Git's and `RefMerge`'s result (deeplearning4j [`8d1ff15f`]).

We find that 21 (30%) of `IntelliMerge`'s false negatives are due to matching errors that eventually 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. Figure 16 shows a merge scenario where `IntelliMerge` results in a duplicate declaration syntax error while Git and `RefMerge` do not. In this example, neither branch changed method `handleUpload` on line 2 of `FileResource.java`. However, `IntelliMerge`'s resolution results in a syntax error where it duplicates the method declaration on line 2.

Finally, the 5 (7%) remaining false negatives are due to `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 is similar to the false positives caused by incorrectly detected refactorings. However, in this case `IntelliMerge` does not report the expected conflict block. Figure 17 provides an example where `IntelliMerge` misses a conflict. In this example, the left branch and right branch add new code to the same spot in method `call`. As shown in Figure 17d, Git and `RefMerge` report a conflict with the conflicting region containing the additions from each branch. This is a necessary conflict because the left branch sets `user` in the if statement, while the right branch sets `user` before the if statement. Additionally, method `isLogged` checks if `mUser != null` instead of `user == null`. In Figure 17e, `IntelliMerge` incorrectly replaces `user == null` with `isLogged()` on line 10 even though method `isLogged` returns `mUser != null`. Thus resulting in a different logic check than is expected and resulting in a false negative.

6.3 Interpretation of RQ2 Results

The above results indicate that, in our sample, `RefMerge` automatically resolves some of Git's reported conflicts, which results in less false positives than Git (188 versus 243). On the other hand, `IntelliMerge` almost triples the number of false positives (707 versus 242). While `IntelliMerge` reports more false positives than Git and `RefMerge`, `IntelliMerge` does well with ordering and formatting conflicts due to its graph-based approach. `IntelliMerge` also decreases the number of refactoring conflicts a developer needs to deal with, but this comes at the price of many more false positives: 646 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



Api.java

```
public class Api {
  private void call() {
    ApiPoint point = apiCall.point;
    if (mActivity == null) {
      ...
    }
    ...
  }
  boolean isLogged() {
    return mUser != null;
  }
}
```

(a) Base commit

Api.java

```
public class Api {
 private void call() {
  ApiPoint point = apiCall.point;
  if (mActivity == null) {
   ...
  }
+  if(apiCall ==null && isLogged()) {
+    user = App.getUser();
+  }
 ...
 }
 boolean isLogged() {
  return mUser != null;
 }
}
```

(b) Left parent

Api.java

```
public class Api {
 private void call() {
  ApiPoint point = apiCall.point;
  if (mActivity == null) {
   ...
  }
+  apiCall = App.getSite();
+  user = App.getUser();
+  apiCall.user = user;
+  if(apiCall ==null && user ==null) {
 ...
 }
 boolean isLogged() {
  return mUser != null;
 }
}
```

(c) Right parent

Api.java

```
public class Api {
 private void call() {
  ...
<<<<<<
  if(apiCall ==null && isLogged()) {
    user = App.getUser();
======
  apiCall = App.getSite();
  user = App.getUser();
  apiCall.user = user;
  if(apiCall ==null && user ==null) {
>>>>>>
 ...
 }
 boolean isLogged() {
  return mUser != null;
 }
}
```

(d) Git and RefMerge Result

Api.java

```
public class Api {
 private void call() {
  ...
  if(apiCall ==null && isLogged()) {
    user = App.getUser();
  }
  apiCall = App.getSite();
  user = App.getUser();
  apiCall.user = user;
  if(apiCall ==null && isLogged()) {
 ...
 }
 boolean isLogged() {
  return mUser != null;
 }
}
```

(e) IntelliMerge Result

Fig. 17: The three versions (base, left, and right) of code involved in a necessary conflict that Git and `RefMerge` report and `IntelliMerge` misses due to incorrectly detecting a refactoring (iFixitAndroid [`91932083`]).

`IntelliMerge` does not detect refactorings in 37 conflict blocks reported by Git (struggling most with parameter level refactorings), `IntelliMerge` typically does well with the refactoring conflicts it does detect. However, `IntelliMerge` also incorrectly detects 11 refactorings (six false positives and five false negatives from Table 7) and reports a total of 71 false negatives. Thus, while the results of RQ1 show that `IntelliMerge` works 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` does not miss any conflicts that need to be reported (i.e., completely eliminates false negatives) and reduces the number of false positives reported by 23%, when compared to Git. `RefMerge` worsens the situation at a much lower rate than `IntelliMerge`, reporting 26 false positives that were not reported by Git, where 16 of these are formatting conflicts that are typically introduced after resolving a refactoring conflict. In general, `RefMerge` struggles most with move-related refactorings where it introduces ordering conflicts.

RQ2 Summary: Compared to Git, `RefMerge` reduces the number of false positives by 23% and completely eliminates false negatives while `IntelliMerge` increases them by 192% and 1,320% respectively. `RefMerge` struggles most with move-related refactorings whereas `IntelliMerge` struggles most with parameter-level and class-level refactorings.

7 Discussion

In our study, we compared two refactoring-aware merging approaches that have not been compared before. RQ1 results show that `RefMerge` manages 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 positives and false negatives whereas `RefMerge` reduced the number of reported false positives and completely eliminated false negatives in the sampled 50 merge scenarios. Thus, operation-based refactoring-aware merging shows promise to help improve the developers' experience without the risk of increasing the number of false negatives.

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 [23]. Additionally, we found that `IntelliMerge` generally struggles with *Rename Class* and *Move Class* refactorings. When class-level refactorings are performed, `IntelliMerge` frequently deletes the entire related class. However, `IntelliMerge` seemed to do well with the other refactorings it detected.

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 textual location it was previously at. Despite this, `RefMerge` resolves or simplifies more refactoring-related conflicts than the complications it introduces, all while avoiding syntax errors. Additionally, while the number of refactorings in a merge scenario can cause problems for `IntelliMerge`, `RefMerge` is resilient to the number of refactorings in a given scenario.

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 requires 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 [23].

We believe that `RefMerge` showed very promising results, despite supporting a subset of the refactorings `IntelliMerge` supports. Future work could go in three different directions: (1) adding support for more refactoring types, (2) undertaking a user study with practitioners to determine if `RefMerge`'s merge resolutions, even if partially resolved, help practitioners, and (3) using language semantics to address ordering-related conflicts when possible, such as the approach proposed by Apel et al. [12].

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 [19]. 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, addressing the weaknesses caused by `IntelliMerge`'s matching algorithm would need to happen before this path could be considered further.

8 Threats to Validity

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

Construct Validity: 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.

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 [25]. 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.

Supporting additional refactorings requires additional engineering effort. In this paper, we focused on creating a first implementation of operation-based refactoring-aware merging that allows us to perform a thorough evaluation of the the concept. As discussed in Section 3.6, `RefMerge` supports the same 13 refactorings that `IntelliMerge` reports supporting in its paper as well as an additional 4 out of 8 we found in its implementation. To alleviate any potential issues with supporting only 17/21 refactorings, we manually verified the correctness of each merge tool's resulting merge and investigated the reason for each reported merge conflict (RQ2).

We carefully considered how each pair of refactorings interact with each other and can result in a merge conflict. We describe the conflict logic for each refactoring pair in our artifact [26]. Any oversights or mistakes we could have made in determining these interactions would have appeared in our results, especially during the manual evaluation. As per our results in RQ2, we did not find any false positives or false negatives that are due to incorrect conflict detection logic.

Finally, inverting and replaying refactorings in the wrong order could result in adverse effects. Specifically, refactorings that create a new program element (such as *Extract Method*) need to be applied before refactorings related to the new program element can be performed. Performing refactorings in an incorrect order will result in `RefMerge` being unable to correctly invert and replay refactorings, resulting in the same merge as Git. We took this into account when determining the ordering for inverting and replaying refactorings (see Section 3.1). If we relied on a refactoring order that is incorrect, we would have seen `RefMerge` result in more conflicts caused by refactorings failing to invert. While we did see some refactorings that failed to invert, this is due to technical issues in `RefMerge` caused by bugs that we plan to fix, rather than the refactoring order.

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. 63min/scenario). However, the 50 merge scenarios we investigated have more than 1,000 unique merge conflicts. As far as we are aware, this is the most extensive qualitative analysis performed in terms of unique merge conflicts [18], [19], [22]. Naturally, investigating additional merge scenarios could reveal more for each tool.

9 Related Work

Software Merging: The proposed software merging techniques in the literature can generally be categorized into *unstructured*, *structured*, and *semi-structured* merging techniques [27].

Unstructured merging techniques represent any software artifact as a sequence of text lines [10]. This gives unstructured merging techniques the strength of being able to process all textual artifacts, regardless of the programming language [27]. The downside to this technique is that unstructured merging cannot handle multiple changes to the same lines, since it cannot consider the syntactic and semantic meaning in software artifacts [12]. Due to its simplicity and versatility, modern version-control systems such as Git or mercurial still rely on such unstructured merging.

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 [17]. Considering the structure of software artifacts allows structured merging techniques to handle syntactic and semantic conflicts [29], [30], [31]. This comes at the cost of generally being language specific and being too expensive to be used in practice. JDime is a structured merge tool that is capable of tuning the merging process by switching between unstructured merge and structured merge [19]. Zhu et al. [21] built on top of JDime by matching nodes based on an adjustable quality function. Leßenich et al. [20] proposed *auto-tuning*, an approach that switches between structured and unstructured merging, and implemented JDime to demonstrate their approach. Seibt et al. [22] 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.

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. [18] proposed jFSTMerge, building upon FSTMerge by adding handlers for different types of conflicts such as renaming.

By representing software artifacts partly as text and partly as trees, semi-structured merging achieves a certain level of language-independence. Cavalcanti et al. [32] 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 in their paper, they show that they outperform jFSTMerge [15]. Furthermore, we are focusing on techniques that specifically target refactorings in order to compare their strengths and weaknesses.

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 [33]. One such approach is *speculative merging* [34], [35], where all combinations of available branches are pulled and merged in the background. Owhadi-Kareshk et al. [36] 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 [37] and Palantir [38] are two tools that increase developer awareness by illustrating the code changes their team members are making. Cassandra [39] minimizes simultaneous edits to the same file by optimizing task scheduling. ConE [40] is an approach that proactively detects concurrent edits to help mitigate certain resulting problems, including merge conflicts. Dewan et al. propose CollabVS, a semi-synchronous detection and resolution tool that detects a potential conflict when a user starts editing a program element that has a dependency on another program element that has been edited but not committed by another developer [41]. Silva et al. [42] proposed utilizing automated unit test creation to detect semantic conflicts that a merge tool could have missed. Fan et al. [43] 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 [44]. The approach primarily leverages the fact that around 80% of merge conflict resolution only rearrange lines [45]. However, they do not explicitly consider refactoring semantics in their merge conflict resolution.

Refactoring Detection: Refactoring is a widespread practice that enables developers to improve the maintainability and readability of their code [46]. 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 [48], [49], [50], [51]. Multiple tools have been developed to detect different refactoring types, such as Ref-Finder [52] and RefDistiller [53]. We use the state-of-the-art refactoring detection tool, RefactoringMiner, which achieves a precision of 98% and a recall of 87% [25].

Operation-based & Refactoring-aware Merging: Operation-based merging is a semi-structured merging technique that models changes between versions as operations or transformations [54], [55], [56] which could be used to support refactoring-aware merging [16]. Nishimura et al. [57] 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 [16].

Similar to MolhadoRef, Ekman and Asklund [58] present a refactoring-aware versioning system. Their approach is more lightweight since it keeps the program elements and their IDs in volatile memory, thus allowing for a short-lived history of refactored program entities. However, their approach has only implemented rename- and move-refactorings. Thus, it remains to be seen whether the approach is generic enough for other refactorings. Additionally, `RefMerge` does not rely on short-lived history because it detects refactorings at any point in the life time of branches from the common ancestor until the merging point.

Dig et al. [16] proposed MolhadRef, an operation-based refactoring-aware merging algorithm that treats refactorings as operations and considers their semantics, and Shen et al. [15] proposed `IntelliMerge`, a graph-based refactoring-aware algorithm. Since we discuss these two approaches in detail in the paper and in our empirical comparison, we do not discuss them again here.

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. In this paper, we rejuvenate operation-based refactoring-aware merging [16] with the hope that this invigorates a fruitful line of research that has a large potential for practical impact. We design and implement the first Git-based refactoring-aware merging implementation in `RefMerge`. We add support for 17 refactoring types, including *Extract Method* and *Inline Method* which were argued to be a limitation for operation-based merging [15]. We perform the first large-scale empirical evaluation of operation-based refactoring-aware merging, implemented in `RefMerge`, and compare it to `IntelliMerge`, a graph-based refactoring-aware merging technique [15]. Our evaluation on 2,001 merge scenarios from 20 open-source projects sheds light on the strengths and weaknesses of each approach. We find that `RefMerge` is able to completely resolve or reduce the number of conflicts in more scenarios than `IntelliMerge` without creating as much extra work for the developers.

References

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

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

[3] 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. New York, NY, USA: Association for Computing Machinery, 2014, p. 364–367. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/2597073.2597121

[4] 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. IEEE Press, 2013, p. 732–741.

[5] 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. [Online]. Available: https://hal.inria.fr/hal-00957168

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

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

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

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

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

[11] M. Mahmoudi and S. Nadi, "The android update problem: An empirical study," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 220–230. [Online]. Available: https://doi.org/10.1145/3196398.3196434

[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. New York, NY, USA: ACM, 2011, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025141

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

[14] 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.

[15] 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, pp. 170:1–170:28, Oct. 2019. [Online]. Available: http://doi.acm.org/10.1145/3360596

[16] 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. New York, NY, USA: Association for Computing Machinery, 2006, p. 732–733. [Online]. Available: https://doi.org/10.1145/1176617.1176698

[17] 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.

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

[19] S. Apel, O. Leßenich, and C. Lengauer, "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.

[20] 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.

[21] 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. IEEE Press, 2019, p. 286–287. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00117

[22] 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.

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

[24] 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.

[25] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, 2020.

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

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

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 168–178. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025139

[35] M. Guimarães and A. Silva, "Improving early detection of software merge conflicts," *Proceedings - International Conference on Software Engineering*, pp. 342–352, 06 2012.

[36] 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.

[37] 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.

[38] 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.

[39] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 732–741.

[40] C. Maddila, N. Nagappan, C. Bird, G. Gousios, and A. van Deursen, "Cone: A concurrent edit detection tool for large-scale software development," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–26, 2021.

[41] P. Dewan and R. Hegde, "Semi-synchronous conflict detection and resolution in asynchronous software development." 01 2007, pp. 159–178.

[42] L. da Silva, P. Borba, W. Mahmood, T. Berger, and J. Moisakis, "Detecting semantic conflicts via automated behavior change detection," 09 2020, pp. 174–184.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[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] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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. New York, NY, USA: Association for Computing Machinery, 1997, p. 139–148. [Online]. Available: https://doi.org/10.1145/263407.263533

[55] 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. New York, NY, USA: Association for Computing Machinery, 1989, p. 56–65. [Online]. Available: https://doi.org/10.1145/72910.73348

[56] 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.

[57] 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.

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

Max Ellis received his B.Sc. degree in computer science from Washington State University, Vancouver, WA, USA, in 2019, and the M.Sc. degree in computer science from the University of Alberta, Canada, in 2022, where he focused on revitalizing refactoring-aware operation-based software merging. Presently, Max is a software engineer at Act-on Software. His research interests include refactoring, software merging, and software maintenance.

Sarah Nadi is an Associate Professor in the Department of Computing Science at the University of Alberta, and a Tier II Canada Research Chair in Software Reuse. She obtained her Master's (2010) and PhD (2014) degrees from the University of Waterloo in Canada. Before joining the University of Alberta in 2016, she spent approximately two years as a post-doctoral researcher at the Technische Universität Darmstadt in Germany. Sarah's research provides automated support tools that help software developers accomplish their tasks more efficiently. She has a long line of work on supporting variability and reuse practices. Her recent work focuses on supporting developers as they use software libraries, including the initial selection process, correctly using the library's API, and potential migration to newer alternative libraries. Sarah leads the Software Maintenance and Reuse (SMR) lab at the University of Alberta. For more information about the work we do at SMR, please visit https://sarahnadi.org/smr/.

Danny Dig is an associate professor of computer science at the University of Colorado, and an adjunct professor at University of Illinois and Oregon State. He enjoys doing research in Software Engineering, with a focus on interactive program transformations that improve programmer productivity and software quality. He successfully pioneered interactive program transformations by opening the field of refactoring in cutting-edge domains including AI/ML, mobile, concurrency and parallelism, component-based, testing, and end-user programming. He earned his Ph.D. from the University of Illinois at Urbana-Champaign where his research won the best Ph.D. dissertation award, and the First Prize at the ACM Student Research Competition Grand Finals. He did a postdoc at MIT.
He (co-)authored 60+ journal and conference papers that appeared in top places in SE/PL. His group's research was recognized with 9 paper awards at the flagship and top conferences in SE and 3 winners of the ACM Student Research Competition. He received the NSF CAREER award, the Google Faculty Research Award (twice), and the Microsoft Software Engineering Innovation Award (twice). His research group released dozens of software systems, among them the world's first open-source refactoring tool. Some of the techniques they developed are shipping with the official release of the popular Eclipse, NetBeans, Visual Studio, Android Studio development environments and are used daily by millions of software developers. He is grateful for research funding from NSF, Boeing, IBM, Intel, Google, Microsoft, NEC, and Trimble.