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 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,
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. \cite{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 \cite{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 \cite{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 \cite{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 \cite{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 \cite{15}, (3) To detect refactorings in Git history, RefMerge uses the state-of-the-art refactoring detection tool, RefactoringMiner \cite{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 \cite{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 \cite{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 \cite{14} and were proposed to be difficult for operation-based merging \cite{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 \cite{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.
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 1c 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 addToListener. 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 addToListener and that these changes can be merged, as shown in Figure 1c.

Figure 1c 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 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
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 Methods 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 refactoring chains. 

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 refactoring operations 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.

### 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 I 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.
ing. 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 determine 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. 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 $\text{RenameMethod}(m_1, m_2)$ and $\text{RenameMethod}(m_3, m_4)$ in Equation 1.

$$\begin{align*}
\text{hasConflict}(\text{RenameMethod}(c_1, m_1, c_2, m_2), \text{RenameMethod}(c_3, m_3, c_4, m_4)) := \\
((c_1 \neq c_3 \land c_2 \neq m_4) \\
\lor (c_1 \neq c_3 \land c_2 \neq m_2 \neq c_4) \\
\lor \neg \text{overloads}(c_1, m_1, c_3, m_3) \land \text{overloads}(c_2, m_2, c_4, m_4)) \\
\lor ((c_1 \neq c_3 \land c_2 \neq m_2 \neq c_4) \\
\lor \neg \text{overloads}(c_1, m_1, c_3, m_3) \land \text{overloads}(c_2, m_2, c_4, m_4))
\end{align*}$$

These two refactorings result in a conflict if (1) the source of both refactorings is the same program element ($c_1 \neq c_3$) but their destinations differ ($c_2 \neq m_2 \neq c_4$) or (2) the sources of both renames are different program elements ($c_1 \neq c_3$) but the renamed destinations are the same program element ($c_2 \neq m_2 = c_4$). 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 $\text{MoveMethod}(m_1, m_2)$ and $\text{RenameMethod}(m_3, m_4)$ as an example in Equation 2.
RefactoringMiner proposed technique. We use an IntelliJ of 17 refactorings to show the feasibility of the approach and the proposed implementation of it generally applies to all refactorings and use the IntelliJ refactoring engine 3.6 Current Implementation

Executing the refactoring includes updating all references in the program, including those added on the other branch. Refactoring can be performed without any additional effort. The replay refactoring list for Figure 1 after detecting conflicts involved in the conflict. In a nutshell, this analysis by Mahmoudi et al. [14] on these 10 projects to identify conflicting merge scenarios that have refactoring changes involved in the conflict. 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.

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 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.

Note that there are 13 supported refactorings that the IntelliJMerge publication describes in its matching rules [15]. However, we find that IntelliJMerge’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 IntelliJMerge’s implementation supports.

When a refactoring is performed that RefMerge does not support or RefMerge fails to invert, RefMerge reports 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, IntelliJMerge [15] on 2,001 merge scenarios that contain refactoring-related conflicts from 20 open-source projects. These projects include the original 10 projects IntelliJMerge 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 IntelliJMerge 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 IntelliJMerge to understand the strengths and weaknesses of each tool.

4.1 Project & Merge Scenario Selection

We first include the same 10 projects that the IntelliJMerge 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
one uses RefactoringMiner \cite{shen2018} to find refactoring 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. \cite{mahmoudi2018} 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 \cite{mahmoudi2018} on the latest history of each project as of September 26, 2021. Our artifact page \cite{artifact} 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 \ref{tab:conflict} 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 \ref{tab:conflict} 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 \ref{fig:refactoring} 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 the highest 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 \cite{mahmoudi2018} using their exact setup and data, as shared in their Github repository \cite{artifact}. We share the exact steps we followed as well as the details of the results of reproducing IntelliMerge.

We run IntelliMerge v1.0.7 on the same 1,070 merge scenarios used in the original publication, including their same post-processing steps 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 \texttt{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.

\begin{table}[h]
\centering
\begin{tabular}{lcccccc}
\hline
\hline
\texttt{cassandra} & 6,882 & 10,719 & 4,509 (42\%) & 2,693 (25\%) & 922 (34\%) & 244 (27\%) \\
\texttt{elasticsearch} & 56,665 & 5,111 & 561 (11\%) & 504 (9\%) & 178 (35\%) & 30 (17\%) \\
\texttt{go4ie} & 12,410 & 7,690 & 1,127 (15\%) & 300 (3\%) & 117 (39\%) & 29 (24\%) \\
\texttt{antlr} & 10,738 & 1,935 & 398 (21\%) & 198 (10\%) & 100 (51\%) & 14 (14\%) \\
\texttt{platform\_frameworks\_support} & 1,609 & 67,584 & 3,690 (55\%) & 507 (15\%) & 96 (17\%) & 25 (26\%) \\
\texttt{deeplearning4j} & 12,208 & 6,997 & 568 (8\%) & 386 (6\%) & 93 (24\%) & 20 (22\%) \\
\texttt{realm\_java} & 11,206 & 3,405 & 683 (20\%) & 312 (9\%) & 92 (29\%) & 23 (25\%) \\
\texttt{jackson-core} & 1,984 & 584 & 318 (54\%) & 201 (34\%) & 81 (40\%) & 17 (21\%) \\
\texttt{android} & 3,161 & 1,805 & 315 (17\%) & 208 (12\%) & 81 (39\%) & 9 (11\%) \\
\texttt{cometd} & 535 & 759 & 369 (49\%) & 173 (23\%) & 63 (36\%) & 11 (18\%) \\
\texttt{storm} & 6,278 & 3,626 & 267 (7\%) & 87 (25\%) & 33 (18\%) & 12 (30\%) \\
\texttt{ProjectE} & 308 & 386 & 79 (20\%) & 73 (19\%) & 30 (41\%) & 3 (10\%) \\
\texttt{javaparser} & 3,859 & 2,427 & 94 (4\%) & 73 (3\%) & 23 (32\%) & 6 (26\%) \\
\texttt{druid} & 24,576 & 1,498 & 151 (10\%) & 138 (9\%) & 17 (12\%) & 3 (18\%) \\
\texttt{androidannotations} & 11,171 & 739 & 74 (6\%) & 67 (9\%) & 15 (21\%) & 1 (7\%) \\
\texttt{druid} & 8,198 & 400 & 46 (11\%) & 41 (10\%) & 14 (34\%) & 2 (14\%) \\
\texttt{junit4} & 4,945 & 792 & 77 (10\%) & 46 (6\%) & 14 (30\%) & 3 (21\%) \\
\texttt{MinerCraftForge} & 143 & 171 & 29 (17\%) & 24 (14\%) & 13 (14\%) & 0 (0\%) \\
\texttt{MobStumbler} & 609 & 934 & 52 (5\%) & 26 (5\%) & 10 (18\%) & 1 (10\%) \\
\texttt{error-prone} & 5,717 & 133 & 24 (18\%) & 21 (16\%) & 93 (24\%) & 20 (22\%) \\
\hline
\textbf{Total} & \textbf{2,001} & \textbf{453} & \\
\end{tabular}
\caption{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.}
\end{table}

3. \url{https://github.com/max-ellis/IntelliMerge/tree/evaluation}
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 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.
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 to discuss these scenarios per project at the file, block, and lines of code levels respectively. There are two parts to each figure. On the left-hand side, we provide a box plot of the overall
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.

<table>
<thead>
<tr>
<th>Total Scenarios</th>
<th>Project Name</th>
<th>Resolved</th>
<th>Changed</th>
<th>Unchanged</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>IntelliMerge</td>
<td>RefMerge</td>
<td>IntelliMerge</td>
<td>RefMerge</td>
</tr>
<tr>
<td>cassandra</td>
<td>922</td>
<td>33 (4%)</td>
<td>108 (12%)</td>
<td>1 (0%)</td>
<td>780 (85%)</td>
</tr>
<tr>
<td>elasticsearch</td>
<td>178</td>
<td>3 (2%)</td>
<td>103 (58%)</td>
<td>0 (0%)</td>
<td>71 (40%)</td>
</tr>
<tr>
<td>gradle</td>
<td>118</td>
<td>1 (1%)</td>
<td>106 (90%)</td>
<td>0 (0%)</td>
<td>11 (9%)</td>
</tr>
<tr>
<td>antlr4</td>
<td>100</td>
<td>1 (1%)</td>
<td>96 (96%)</td>
<td>0 (0%)</td>
<td>3 (3%)</td>
</tr>
<tr>
<td>platform_fuck_support</td>
<td>95</td>
<td>5 (5%)</td>
<td>56 (59%)</td>
<td>1 (1%)</td>
<td>33 (35%)</td>
</tr>
<tr>
<td>deeplearning4j</td>
<td>93</td>
<td>3 (3%)</td>
<td>89 (96%)</td>
<td>0 (0%)</td>
<td>1 (1%)</td>
</tr>
<tr>
<td>redblame</td>
<td>92</td>
<td>7 (8%)</td>
<td>82 (89%)</td>
<td>1 (1%)</td>
<td>2 (2%)</td>
</tr>
<tr>
<td>jackson-core</td>
<td>81</td>
<td>0 (0%)</td>
<td>81 (100%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>android</td>
<td>81</td>
<td>3 (4%)</td>
<td>78 (96%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>comedit</td>
<td>63</td>
<td>2 (3%)</td>
<td>59 (93%)</td>
<td>1 (2%)</td>
<td>1 (2%)</td>
</tr>
<tr>
<td>storm</td>
<td>33</td>
<td>1 (3%)</td>
<td>30 (91%)</td>
<td>1 (3%)</td>
<td>3 (3%)</td>
</tr>
<tr>
<td>ProjectE</td>
<td>30</td>
<td>1 (3%)</td>
<td>28 (94%)</td>
<td>0 (0%)</td>
<td>1 (3%)</td>
</tr>
<tr>
<td>javaparser</td>
<td>23</td>
<td>3 (13%)</td>
<td>19 (83%)</td>
<td>1 (4%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>druid</td>
<td>17</td>
<td>3 (18%)</td>
<td>15 (89%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>androidannotations</td>
<td>15</td>
<td>1 (7%)</td>
<td>14 (93%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>junit4</td>
<td>14</td>
<td>1 (7%)</td>
<td>13 (93%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>MinecraftForge</td>
<td>14</td>
<td>1 (7%)</td>
<td>10 (72%)</td>
<td>0 (0%)</td>
<td>3 (21%)</td>
</tr>
<tr>
<td>iFixitAndroid</td>
<td>13</td>
<td>0 (0%)</td>
<td>13 (100%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>MozStumbler</td>
<td>10</td>
<td>0 (0%)</td>
<td>10 (100%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>error-prone</td>
<td>9</td>
<td>1 (11%)</td>
<td>8 (88%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resolved</th>
<th>Changed</th>
<th>Unchanged</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,001</td>
<td>107 (0%)</td>
<td>1,017 (51%)</td>
<td>907 (45%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project</th>
<th>Reduced Conf. Files</th>
<th>Increased Conf. Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>cassandra</td>
<td>1 (33%)</td>
<td>34 (127%)</td>
</tr>
<tr>
<td>elasticsearch</td>
<td>4 (20%)</td>
<td>14 (123%)</td>
</tr>
<tr>
<td>gradle</td>
<td>7 (03%)</td>
<td>7 (135%)</td>
</tr>
<tr>
<td>antlr4</td>
<td>2 (06%)</td>
<td>15 (076%)</td>
</tr>
<tr>
<td>platform_fuck_support</td>
<td>6 (48%)</td>
<td>5 (50%)</td>
</tr>
<tr>
<td>deeplearning4j</td>
<td>6 (48%)</td>
<td>5 (50%)</td>
</tr>
<tr>
<td>redblame</td>
<td>11 (03%)</td>
<td>7 (03%)</td>
</tr>
<tr>
<td>jackson-core</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>android</td>
<td>6 (08%)</td>
<td>4 (135%)</td>
</tr>
<tr>
<td>comedit</td>
<td>2 (46%)</td>
<td>3 (225%)</td>
</tr>
<tr>
<td>druid</td>
<td>3 (03%)</td>
<td>2 (241%)</td>
</tr>
<tr>
<td>androidannotations</td>
<td>3 (03%)</td>
<td>2 (241%)</td>
</tr>
<tr>
<td>junit4</td>
<td>4 (02%)</td>
<td>3 (06%)</td>
</tr>
<tr>
<td>MinecraftForge</td>
<td>5 (04%)</td>
<td>4 (225%)</td>
</tr>
<tr>
<td>iFixitAndroid</td>
<td>7 (04%)</td>
<td>4 (225%)</td>
</tr>
<tr>
<td>MozStumbler</td>
<td>5 (04%)</td>
<td>4 (225%)</td>
</tr>
<tr>
<td>error-prone</td>
<td>1 (04%)</td>
<td>1 (04%)</td>
</tr>
</tbody>
</table>

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 IntelliJMerge 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 IntelliJMerge reduces the number of reported conflicting files in 81 scenarios (4% of all evaluated scenarios) by a median 38% reduction. On the other hand, IntelliJMerge increases the number of reported conflicting files in 810 scenarios (40%) by a median 350% increase. In other words, on average, IntelliJMerge 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.
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.

**Conflict Blocks:** We now look at the conflict block level in Figure 6. The number of conflict blocks indicates the number of individual conflicting regions a developer needs to deal with. Figure 6(b) 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 6(b), 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 for 825 scenarios (41%) by a median of 378%. On the other hand, RefMerge reduces the number of conflict blocks for 825 scenarios (41%) by a median of 378%. Inversely, RefMerge increases conflict blocks in 80 merge scenarios for project jackson-core by almost 16-fold without decreasing conflict blocks for 825 scenarios (41%) by a median of 378%. On the other hand, RefMerge reduces the number of conflict blocks for 825 scenarios (41%) by a median of 378%.

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.

In parenthesis, we show the median percentage reduction/increase per merge scenario.

**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 7, we observe similar behavior of the tools as what we observed for the conflicting files in Figure 6. More closely from the table in Figure 7, 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%.
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 7), 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 7), 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.

**R1 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.

<table>
<thead>
<tr>
<th></th>
<th>RefMerge</th>
<th>Git</th>
<th>IntelliMerge</th>
</tr>
</thead>
<tbody>
<tr>
<td># Conflict Blocks Investigated</td>
<td>379</td>
<td>433</td>
<td>866</td>
</tr>
<tr>
<td>True Positives</td>
<td>191</td>
<td>190</td>
<td>159</td>
</tr>
<tr>
<td>False Positives</td>
<td>188</td>
<td>243</td>
<td>707</td>
</tr>
<tr>
<td>False Negatives</td>
<td>0</td>
<td>5</td>
<td>71</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Git Reasons</th>
<th>Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Refactoring Handling</td>
<td>False Positive</td>
<td>140</td>
</tr>
<tr>
<td>Ordering Conflict</td>
<td>False Positive</td>
<td>61</td>
</tr>
<tr>
<td>Formatting Conflict</td>
<td>False Positive</td>
<td>41</td>
</tr>
<tr>
<td>No Refactoring Handling</td>
<td>False Negative</td>
<td>5</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>248</td>
</tr>
</tbody>
</table>

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. 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.

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’s 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.
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 11c 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, IntelliJMerge incorrectly deletes classes TableInv and TransmuteTabletContainer instead of renaming them. In IntelliJMerge’s result, files TransmuteTabletContainer.java, TransmutationContainer.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-
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 IntelliJMerge’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 12c, VERSION and INDEX are unnecessarily part of the conflict while they originally were not. While RefMerge complicates this conflict, IntelliJMerge 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 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 IntelliJMerge. 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 IntelliJMerge 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.

<table>
<thead>
<tr>
<th>IntelliMerge Reasons</th>
<th>Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matching Error</td>
<td>False Positive</td>
<td>646</td>
</tr>
<tr>
<td>Undetected Refactoring</td>
<td>False Positive</td>
<td>37</td>
</tr>
<tr>
<td>Ordering Conflict</td>
<td>False Positive</td>
<td>10</td>
</tr>
<tr>
<td>Incorrectly Detected Refactoring</td>
<td>False Positive</td>
<td>6</td>
</tr>
<tr>
<td>Unsupported Refactoring</td>
<td>False Positive</td>
<td>5</td>
</tr>
<tr>
<td>Formatting Conflict</td>
<td>False Positive</td>
<td>3</td>
</tr>
<tr>
<td>Deletes Conflict Block</td>
<td>False Negative</td>
<td>45</td>
</tr>
<tr>
<td>Matching Error</td>
<td>False Negative</td>
<td>21</td>
</tr>
<tr>
<td>Incorrectly Detected Refactoring</td>
<td>False Negative</td>
<td>5</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>778</td>
</tr>
</tbody>
</table>

devendor 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 14 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.

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 IntelliJ’s graph node matching process. This primarily happens with comments, annotations, and imports where IntelliJ 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, IntelliJMerge 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 IntelliJMerge incorrectly detecting a refactoring that was never performed. There are six cases where IntelliJMerge incorrectly matches methods in different classes, resulting in IntelliJMerge 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, boolean reAuth) to AndroidAuthenticator.java. The right branch also adds method getCurrentVelocity along with getQuery’s comment 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 IntelliJMerge. IntelliJMerge deletes

![Image](image.png)
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_mwks_support [6c371fc5]).

getCurrVelocity in ScrollerCompat.java and replaces getAuthToken() in AndroidAuthToken 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 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

Fig. 16: A syntax error (Duplicate declaration) that IntelliMerge’s merge resolution introduces versus Git’s and RefMerge’s result (deeplearning4j [8d1f15e]).

one, including six files that should contain conflicts.

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 17c, 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 == null 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 17d 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
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. 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 IntelliJMerge, 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 require addressing the matching algorithm. The current merging algorithm IntelliJMerge 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 [12].

We believe that RefMerge showed very promising results, despite supporting a subset of the refactorings IntelliJMerge 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 IntelliJMerge’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 concept. As discussed in Section 3.6, RefMerge supports the same 13 refactorings that IntelliJMerge 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 spe-
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. Seib 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. [52] 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 IntelliJMerge 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 [27] and Palantir [38] are two tools that increase developer awareness by illustrating the code changes that team members are making. Cassandra [39] minimizes simultaneous edits to the same file by optimizing task scheduling. ConB [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 MolhadRef, Ekman and Askland [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 Conclusions

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


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


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.