IntelliMerge: A Refactoring-Aware Software Merging Technique

BO SHEN, Peking University, China
WEI ZHANG, Peking University, China
HAIYAN ZHAO, Peking University, China
GUANGTAI LIANG, Huawei Technologies Co. Ltd, China
ZHI JIN, Peking University, China
QIANXIAANG WANG, Huawei Technologies Co. Ltd, China

In modern software development, developers rely on version control systems like Git to collaborate in the branch-based development workflow. One downside of this workflow is the conflicts occurred when merging contributions from different developers: these conflicts are tedious and error-prone to be correctly resolved, reducing the efficiency of collaboration and introducing potential bugs. The situation becomes even worse, with the popularity of refactorings in software development and evolution, because current merging tools (usually based on the text or tree structures of source code) are unaware of refactorings. In this paper, we present IntelliMerge, a graph-based refactoring-aware merging algorithm for Java programs. We explicitly enhance this algorithm’s ability in detecting and resolving refactoring-related conflicts. Through the evaluation on 1,070 merge scenarios from 10 popular open-source Java projects, we show that IntelliMerge reduces the number of merge conflicts by 58.90% comparing with GitMerge (the prevalent unstructured merging tool) and 11.84% comparing with jFSTMerge (the state-of-the-art semi-structured merging tool) without sacrificing the auto-merging precision (88.48%) and recall (90.22%). Besides, the evaluation of performance shows that IntelliMerge takes 539 milliseconds to process one merge scenario on the median, which indicates its feasibility in real-world applications.

CCS Concepts: · Software and its engineering → Software configuration management and version control systems;

Additional Key Words and Phrases: Revision control system, software merging, conflicts resolution, refactoring

ACM Reference Format:
Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. 2019. IntelliMerge: A Refactoring-Aware Software Merging Technique. Proc. ACM Program. Lang. 3, OOPSLA, Article 170 (October 2019), 28 pages. https://doi.org/10.1145/3360596

1 INTRODUCTION

Modern software products are usually developed and maintained by a team of developers, either from a company or from an open-source community. In the development process, the version control system (VCS) plays an indispensable role by providing supports that facilitate the collaborative

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART170
https://doi.org/10.1145/3360596
editing and revising of software artifacts among developers [Zhu and He 2018]. For example, an open-source project hosted on Github usually follows a branch-based development workflow, where new features or bug fixes are developed in separate branches forked from a master branch and then different branches are merged into the master branch. While facilitating collaborative software development by allowing separation of concerns and parallel development [Walrad and Strom 2002], the branch-based workflow also brings side-effects that slow down the effective collaboration among developers. One of the key side-effects is the conflicts occurred when merging contributions from different developers, or when building and testing the merged artifacts. The developers have to spend a large part of their time to resolve or repair these conflicts. Previous studies [Bird and Zimmermann 2012; Brindescu 2018; Brun et al. 2011; Le Nguyen and Ignat 2018] have shown that understanding and resolving conflicts are both demanding and tedious tasks, which distract developers from their regular development work. What’s even worse, not all conflicts between different contributions can be detected by the existing automatic merging tools; those undetected conflicts then escape to the subsequent phases of software life-cycle, even to the released software products, causing unexpected behaviors [Ahmed et al. 2017].

Conflicts will be reported when the VCS fails in merging two revisions (called the left and right version, respectively) to the same source code (called the base version) into a consistent new source code (called the merged version). The commonly-used VCS in the industry like Subversion and Git typically treats the source code in a file as a sequence of lines in plain text, and merges two files from the left and right versions line by line, reporting many fake conflicts but neglecting many real conflicts. For example, as the most widely-used VCS currently, Git merges two files in the following manner: when a line in the base file is changed only by one of the two files, such a change will be reserved in the merged file without conflicts; when a line in the base file is changed by both of the two files but in different ways, a conflict block with the conflicting code lines will be inserted into the merged file, indicating the occurrence of a conflict. However, no overlapping changes do not necessarily indicate no conflicts in the program logic (false negative), and different changes to the same line may just result from semantic-irrelevant formatting or mismatching of two lines (false positive). Text-line based merging makes the conflict resolving process not only tedious by reporting false positive conflicts, but also unreliable by neglecting false negative conflicts [Cavalcanti et al. 2017].

Researchers have proposed various approaches to overcome the drawbacks of text-line based merging methods. Some of these approaches attempt to proactively detect potential conflicts or resolve them early during parallel development, while others focus on developing better algorithms to reduce conflicts in merging. Proactive conflict detection is typically based on speculative merging [Brun et al. 2011; Guimarães and Silva 2012; Kasi and Sarma 2013], where all combinations of branches are pulled and merged in the background. This limits its application in practice, especially for large-scale open-source projects. For example, as a prevalent operating system for smartphone, the Android Open Source Project (AOSP) is mainly maintained by Google but also customized by various Android phone vendors, who need to constantly merge the latest release from Google, but merge conflicts are hard to detect in the development process. Our work is exactly inspired by this scenario, so we focus on merging. The previous merging approaches can be largely classified into three categories, namely unstructured, structured and semi-structured approaches [Apel et al. 2011]. Unstructured approaches treat the source code as plain text; structured approaches try to fully understand the semantics of source code by transforming the source code into a formal structure with explicitly defined semantics, and the semi-structured approaches aim to strike a balance between the

1https://github.com
2https://source.android.com/
generality and efficiency of automatic conflicts resolving by combining unstructured and structured approaches. These approaches can also be classified as textual, syntactic and semantic approaches, according to the extent they make use of the semantic information in the source code [Mens 2002].

However, most of the existing approaches still fail to deal with one kind of complex changes made to source code, namely, the refactorings. In general, a refactoring change to a program denotes a transformation to the program (e.g., Rename/Move Field and Extract/Inline Method) that improves the program’s internal design without changing its externally observable behavior [Fowler 2002]. Complex refactorings usually consist of a set of related changes across multiple source code files in the program. As an important kind of change in object-oriented programming (OOP) and agile development, refactorings are widespread in software development and evolution [Fowler 2002]. The automated refactoring tools also contribute to the popularity of refactorings since these tools allow programmers to apply refactoring quickly and safely [Dig et al. 2007]. However, it has been observed that refactoring is one of the main factors that make trouble to unstructured merge techniques [Dig et al. 2006a, 2007]. A recent empirical study [Mahmoudi et al. 2019] on about 3,000 open-source Java projects also concludes that (1) at least 22% of merge conflicts are related to refactoring, and (2) these conflicts are more complex and difficult to resolve comparing with those refactoring-irrelevant conflicts. On one hand, a refactoring often affect multiple parts of the program, but most existing approaches do not consider the possible dependencies between different changes in the merging process; on the other hand, refactoring-related conflicts are more difficult for developers to understand and resolve, because of the absence of the refactoring history [Tsantalis et al. 2018].

Fig. 1. An example of merge scenarios in the commit history. Each labeled line represents a branch, and each circle stands for a commit. The commits with colors constitute the merge scenario. The left commit is the HEAD commit of the current branch when merging happens, while the right commit is the HEAD commit of another branch that is merged in. Their nearest common ancestor (NCA) is called the base commit.

In this paper, we present IntelliMerge, a graph-based refactoring-aware merging algorithm for object-oriented programs. We explicitly enhance its ability in avoiding and resolving refactoring-related conflicts, without sacrificing its precision and performance, comparing with state-of-the-art merging approaches. IntelliMerge takes as input a three-way merge scenario of a Git project. A typical merge scenario (see Figure 1) consists of two branches to be merged (i.e., the left and right versions) and their nearest common ancestor (NCA) in the commit history (i.e., the base version). The output of IntelliMerge contains the merged version and a set of conflicts that cannot be automatically resolved in the merging process. IntelliMerge works in 4 sequential steps: code-to-graph, matching, merging, and graph-to-code. In the code-to-graph step, IntelliMerge transforms the three versions
to be merged into three program element graphs (PEGs), respectively, by applying a lightweight semantic analysis to the three versions, with the program elements as the vertices in PEGs, and the correlation between them as the edges. In the matching step, the three PEGs are matched with each other through a matching algorithm. A matching between two graphs is a mapping relationship between vertices in the two graphs, and the matched vertices are expected to have the same or similar semantics. The similarity of two vertices from different PEGs is calculated by combining the similarity of their inner attributes and the similarity of their context in PEGs. The inner attribute of a vertex includes a set of fine-grained components that constitute the corresponding program element, and the context of a vertex is determined by the sub-graph that consists of edges and vertices within one-hop from the vertex. In the merging step, the contents of each pair of matched vertices are merged textually to generate a merged PEG. In the graph-to-code step, the merged PEG is printed into a set of source code files, preserving the format and order of the three original versions as much as possible. The current version of IntelliMerge is implemented to work with Git and for Java programs only, but can be extended to support other VCSs and programming languages.

We collect 1,070 merge scenarios that involve refactorings from 10 real open-source Java projects on Github with high numbers of stargazers, and replay every merge scenario with three tools: IntelliJMerge, GitMerge (the most prevalent merging tool currently), and jFSTMerge [Cavalcanti et al. 2017] (the state-of-the-art semi-structured merging algorithm), in order to evaluate the effectiveness and efficiency of IntelliJMerge. The experiment results show that: (1) in terms of auto-merged results, IntelliJMerge achieves a good precision of 88.48% and a recall of 90.22% with the manually-resolved version as the ground truth, (2) in terms of merge conflicts, IntelliJMerge produces fewer conflict blocks (58.90% fewer than GitMerge and 11.84% fewer than jFSTMerge), and reduces the lines of conflicting code (by 90.98% comparing with GitMerge, and by 78.38% comparing with jFSTMerge), (3) in over 90% of examined merge scenarios, IntelliJMerge finishes the whole merging process within 5 seconds, which is acceptable to be applied in real applications.

This paper makes the following contributions:

(1) A graph-based refactoring-aware merging algorithm for Java programs.

(2) A matching algorithm that is able to match refactored program elements, by not only considering their inner attributes but also their contextual sub-graph in the program element graph.

The rest of the paper is organized as follows: Section 2 introduces a motivating example of the merge scenario that involves refactorings, which will be used later to give instances of PEGs. Section 3 describes how the proposed approach works in each step, and Section 4 introduces how we implement the approach as a tool called IntelliJMerge. In Section 5, we evaluate the effectiveness and efficiency of IntelliJMerge through experiments on 1,070 real merge scenarios from 10 popular open-source projects, and compare it with the other merging tools. Section 6 introduces and analyzes some of the related works. In Section 7 we further discuss threats-to-validity and limitations to our study. Finally, we conclude this paper with a short summary, and briefly shed light on further directions on this topic in Section 8.

2 MOTIVATING EXAMPLE

In this section, we introduce a motivating example of the merge scenario that involves refactorings, in order to illustrate the target problem. This example is extracted from the file SourceRoot.java at commit d9c990a in the master branch of the JavaParser project, and tailored for illustration.

---

4https://git-scm.com/docs/git-merge
5https://github.com/javaparser/javaparser

Proc. ACM Program. Lang., Vol. 3, No. OOPSLA, Article 170. Publication date: October 2019.
Figure 2 shows the related code snippets in this motivating example. The left and right versions both made changes to the method `saveAll()` in the base version (see Figure 2 (a)), but in different ways. From the result generated by `git diff` command, the left version (see Figure 2 (b)) modified one method invocation inside the method body of `saveAll()` by changing the argument that describes the encoding. The right version (see Figure 2 (c)) made three changes: (1) removed some statements from the method body of `saveAll()`, (2) added a new method called `save()`, and (3) added an invocation to the method `save()` inside `saveAll()`.

Actually, the newly added method `save()` was extracted from `saveAll()`, which means the right version applied an `Extract Method` refactoring. The `Extract Method` is proved to be one of the most conflict-prone refactoring types by Mahmoudi et al. [2019]. Given this merge scenario, Git merged...
the three versions textually line by line, being unaware of the refactoring change. As a result, it generated a merge conflict (see Figure 2 (d)) to indicate that some lines were modified in both of the left and right versions and could not be auto-merged, and the developer should manually resolve it. After carefully investigating the context of this conflict, the developer recognized that it was a false positive conflict, and manually resolved it by applying the refactoring change from the right version into saveAll() and the edits from the left version into save() (see Figure 2 (e)). However, assuming that the developer failed to recognize the refactoring, the inaccurately auto-merged save() would be preserved as a false negative conflict that might introduce potential bugs or severe errors.

3 APPROACH

In this section, we present a graph-based refactoring-aware merging algorithm for object-oriented programs, and demonstrate how it works with the motivating example introduced in Section 2.

3.1 Overview

Figure 3 presents an overview of our approach, which consists of 4 sequential steps in general:

1. In the code-to-graph step, the left, right, and base versions in a merge scenario are transformed into three program element graphs (PEGs), namely, the left, right, and base PEGs.
2. In the matching step, the left and right PEGs are matched with the base PEG, respectively. The purpose of matching two PEGs is to find an optimal mapping relationship between vertices in the two PEGs, in which each pair of matched vertices have the same or similar semantics.
3. In the merging step, a merged PEG is generated by merging vertices in the base PEG with their corresponding vertices in the left and/or right PEGs, and those unmatched vertices in the left and right PEGs are inserted into the merged PEG as newly added ones.
4. In the graph-to-code step, merged source files are generated from the merged PEG, with possible conflict blocks embedded in these merged files.

![Fig. 3. An overview of the merging algorithm based on program element graphs.](image)

3.2 Step 1: Code-to-Graph

In this step, for each of the left, right, and base versions in a merge scenario, we first need to locate the set of source files involved in the merging process. The three sets of source files are denoted as SRC_l, SRC_r, and SRC_b, respectively, and are decided through a two-step process. Initially, SRC_l, SRC_r, and SRC_b are assigned with \(\emptyset\). In the first step, given a source file \(f_b\) in the base version, if \(f_b\) has no corresponding file in the left or right version, or if one or both of \(f_b\)’s corresponding files
in the left and right versions are changed, then \( f_b \) and all its imported source files will be added to \( SRC_b \); given a source file \( f_i/f_r \) in the left/right version, if \( f_i/f_r \) has no corresponding file in the base version, or if \( f_i/f_r \) makes changes to its corresponding file in the base version, then \( f_i/f_r \) and all its imported source files will be added to \( SRC_l/SRC_r \). In the second step, for each file \( f \) in \( SRC_l \cup SRC_r \), if \( f \) has a corresponding file \( f_b \) in the base version, and \( f_b \notin SRC_b \), then \( f_b \) will be added to \( SRC_b \); and vice verse.

After all files involved in the merging process are collected, the source file sets \( SRC_l, SRC_r, \) and \( SRC_b \) of the three versions are transformed into three program element graphs (PEGs), respectively. Given a set of source files, the corresponding PEG is a semi-structured representation of these source files, consisting of a set of program elements (vertices) and a set of correlations between program elements (edges). Intuitively, the PEG represents the program structure above the field and method level, by encoding program elements (e.g. method and field declaration) as vertices and the correlation between them (e.g. method invocation and field access) as edges. The definition of PEG is language-dependent, and here we give our definition of PEG for Java 8:

**Definition 3.1 (Program Element Graph).** The program element graph of a set of source files is a labeled, weighted, and directed graph \( G \) of the form \((VT, EL, ET, V, E, type, wt, sig, src)\) that is defined as

1. \( VT = \{prj, pkg, cut, cls, enu, ant, itf, fld, ctr, mtd, ect, ann, ibk\} \) is the set of vertex types. Each element of \( VT \) denotes a type of element in programs. We list each vertex type and its corresponding program element for Java 8 in Table 1. These vertex types are further classified into two categories: composite and terminal types, denoted as \( VT_c \). \( VT_c = \{prj, pkg, cut, cls, enu, itf, ant\} \) and \( VT_t \). \( VT_t = \{fld, mtd, ect, ctr, ann, ibk\} \), respectively.

2. \( EL = \{contain, define, import, extend, implement, declare, read, write, call, instantiate\} \) is the set of edge labels. A subset of \( EL \) is \( EL_{tree} = \{contain, define, declare\} \), which implies the hierarchical structure between vertices.

3. \( ET \in VT \times EL \times VT \) is the set of edge types. Each element of \( ET \) is a tuple of the form \((source\ vertex\ type, edge\ label, target\ vertex\ type)\). Table 2 describes the meanings of each defined edge type.

4. \( V \) is the set of vertices of \( G \), each of which corresponds to a program element in the set of source files.

5. \( E \in V \times EL \times V \), is the set of edges of \( G \). Each element of \( E \) is a tuple of the form \((source\ vertex, edge\ label, and\ target\ vertex)\). A subset of \( E \) is \( EL_{tree} \) that consists of all the edges in \( E \) that have a label in \( EL_{tree} \). \( EL_{tree} \) formulates the tree structure, which embodies the nesting relationship between associated program elements. In the rest of the paper, we use \( tree(G) \) to denote the tree structure inside a PEG \( G \), and \( root(G) \) to denote the root vertex of this tree. For a non-root vertex \( v \) in \( tree(G) \), we use \( p(v) \) to denote the parent vertex of \( v \) in \( tree(G) \). For a non-leaf vertex \( v \) in \( tree(G) \), we use \( C(v) \) to denote the set of \( v \)’s children in \( tree(G) \).

6. \( type : (V \rightarrow VT) \cup (E \rightarrow ET) \) is a function that maps each element in \( V \cup E \) to its type. An invariant is that there is exactly one vertex of the type \( prj \) in \( V \), and the vertex of type \( prj \) is the root \( tree(G) \). A vertex of type in \( VT_c \) is called a composite vertex, and of type in \( VT_t \) a terminal vertex.

7. \( wt : E \rightarrow \mathbb{Z}^+ \) is a function that maps each edge in \( E \) to a positive integer weight, denoting the times this edge appears in source code. For example, if method \( A() \) invokes method \( B() \) in 3 different places in its method body, an edge of type \( (mtd, call, mtd) \) will be created between the two vertices of type \( mtd \), with its weight set as 3.

8. \( sig : V \rightarrow Signature \) is a function that maps each vertex in \( V \) to its signature. The signature of a vertex consists of the canonical name of the corresponding program element that uniquely identifies it in the project. In particular, for the method declaration, the signature also includes
the parameter type list since there will be multiple methods with the same name in the same class in case of overloading. Such a definition of signature ensures that no two program elements in a program have the same signature. As a result, $\text{sig}^{-1}$ is also a function that maps a signature back to a vertex in $V$; the domain of $\text{sig}^{-1}$ is $\text{sig}(V)$.

For each of the three sets of source files $\text{SRC}_l$, $\text{SRC}_r$, and $\text{SRC}_b$, the corresponding PEG is built in two steps. First, source files are parsed into a set of abstract syntax trees (ASTs). Then the program elements of concerned types are extracted from these ASTs to form the vertex set $V$ of a PEG, and the correlation between program elements are extracted to construct the edge set $E$. Edges of label `contain`, `define` and `declare` are retrieved from the hierarchical tree structure of ASTs, while edges of other labels are constructed by analyzing the logic expressed in program statements. Considering the motivating example, Figure 4 shows the simplified PEGs built upon the base and left/right versions.

![Fig. 4. The program element graphs (PEGs) built from the base and left/right version of the motivating example.](image)

### 3.3 Step 2: Matching

#### 3.3.1 Matching Algorithm

In this step, a matching algorithm (see Algorithm 1) is used to match the base PEG with the left and right PEGs, respectively. Figure 5 shows the matching process of the PEGs constructed from the motivating example. In order to find the optimal mapping relationship from vertices in the base PEG to vertices in the left and right PEGs, the matching algorithm works in two successive phases: top-down and bottom-up.

1. Top-down Phase. In this phase, the mapping relationship between vertices with the same signature in the base and left/right PEGs are built, starting from root($G$) to the leaf vertices (i.e. terminal vertices). The basic assumption is that if two vertices in two PEGs have the same signature, then they correspond to the same program element, regardless of the differences.
Table 1. Corresponding vertex type and components of program elements.

| Program Element Type | Vertex Type | Components |
|-----------------------|-------------|------------|
| Project               | prj         | {project name} |
| Package               | pkg         | {package name} |
| CompilationUnit       | cut         | {file name, relative path} |
| Class                 | cls         | {annotations, modifiers, name, type parameters} |
| Enum                  | enu         | {annotations, modifiers, name} |
| Annotation            | ant         | {annotations, modifiers, name} |
| Interface             | itf         | {annotations, modifiers, name} |
| Field                 | fld         | {annotations, modifiers, name, type, initializer} |
| Constructor           | ctr         | {annotations, modifiers, name, parameters, body} |
| Method                | mtd         | {annotations, modifiers, type, name, parameters, body} |
| EnumConstant          | ect         | {annotations, name, arguments, body} |
| AnnotationMember      | anm         | {annotations, modifiers, type, name, default value} |
| InitializerBlock      | ibk         | {modifier, body} |

Table 2. Edge types defined in the Program Element Graph.

| VTsource | EdgeLabel | VTtarget | Description |
|----------|-----------|----------|-------------|
| prj      | contain   | pkg      | Project contains package(s). |
| pkg      | contain   | pkg      | Package contains package(s). |
| cut      | import    | cls, itf, enu, ant | Compilation unit imports type(s). |
| cut      | define    | cls, itf, enu, ant | Compilation unit defines type(s). |
| cls      | extend    | cls      | Class extends another class. |
| cls      | implement | itf      | Class implements interface(s). |
| cls      | define    | fld, ibk, ctr, mtd, cls | Class defines member(s). |
| itf      | define    | fld, mtd | Interface defines field/method(s). |
| itf      | declare   | mtd      | Interface declares method(s). |
| enu      | define    | ect, fld, mtd | Enum defines constant/field/method(s). |
| ant      | define    | anm      | Annotation defines member(s). |
| fld      | read      | ect      | Field reads enum constant(s). |
| fld      | read, write | fld | Field reads/modifies the value of other field(s). |
| fld      | call      | mtd, ctr | Field calls method/constructor(s). |
| fld      | instantiate | cls | Field creates instance of class(es). |
| ctr      | read      | ect      | Constructor reads the value of enum constant(s). |
| ctr      | read, write | fld | Constructor reads/modifies the value of field(s). |
| ctr      | call      | mtd, ctr | Constructor calls method/constructor(s). |
| ctr      | instantiate | cls | Constructor creates instance of class(es). |
| mtd      | read      | ect      | Method reads the value of enum constant(s). |
| mtd      | read, write | fld | Method reads/modifies the value of field(s). |
| mtd      | call      | mtd, ctr | Method calls method/constructor(s). |
| mtd      | instantiate | cls | Method creates instance of class(es). |

that might exist within their corresponding source code [Tsantalis et al. 2018]. The output of this phase consists of 6 sets: the two sets of matched vertex pairs $M_{bl}, M_{br}$, and the 4 sets of remaining unmatched vertices $U_{bl}, U_{lb}, U_{br},$ and $U_{rb}$. 

Proc. ACM Program. Lang., Vol. 3, No. OOPSLA, Article 170. Publication date: October 2019.
(2) Bottom-up Phase. In this phase, vertices in $U_{bl}$, $U_{lb}$, $U_{br}$, and $U_{rb}$ are further matched upward from the leaf vertices according to their semantics. The bottom-up matching algorithm takes the inner attributes of vertices as well as the context of vertices into consideration, in order to evaluate the confidence of a matching relationship. Two kinds of matching are detected in this phase: 1-to-1 and m-to-n. The former matching is due to changes that affect signatures of program elements, such as refactorings like Rename/Move and other edits like parameter change of methods. The latter matching results from complex refactorings like Extract and Inline.

**Algorithm 1**: Graph Matching Algorithm

```plaintext
Input : $G_b = (VT, EL, ET, V_b, E_b, type_b, wt_b, sig_b, src_b)$, the base PEG
       $G_l = (VT, EL, ET, V_l, E_l, type_l, wt_l, sig_l, src_l)$, the left PEG
       $G_r = (VT, EL, ET, V_r, E_r, type_r, wt_r, sig_r, src_r)$, the right PEG

Output : $M_{bl}/M_{br}$: the set of tuples of matched vertices in $V_b$ and $V_l/V_r$
          $U_{bl}/U_{br}$: the set of vertices in $V_b$ not matched with any vertex in $V_l/V_r$
          $U_{lb}/U_{rb}$: the set of vertices in $V_l/V_r$ not matched with any vertex in $V_b$

1 Function match($G_b, G_l, G_r$)
2     (M_{bl}, U_{bl}, U_{lb}) ← twowayMatch($G_b, G_l$)
3     (M_{br}, U_{br}, U_{rb}) ← twowayMatch($G_b, G_r$)
4     return (M_{bl}, U_{bl}, U_{lb}, M_{br}, U_{br}, U_{rb})
5 end
6 Function twowayMatch($G_0, G_1$)
7     // the top-down phase
8     S ← sig_0(V_0) \ sig_1(V_1)
9     M ← \{(v_0, v_1) \mid s \in S, v_0 = sig_0^{-1}(s), v_1 = sig_1^{-1}(s)\}
10    U_0 ← V_0 \ sig_0^{-1}(S), U_1 ← V_1 \ sig_1^{-1}(S)
11     // the bottom-up phase
12    foreach (t ∈ VT) do
13        N_0 ← \{v \mid v \in U_0, type_0(v) = t\}, N_1 ← \{v \mid v \in U_1, type_1(v) = t\}
14        M ← M \cup 1-to-1-match(t, N_0, N_1) \cup m-to-m-match(t, N_0, N_1)
15        U_0 ← U_0 \ \{v_0 \mid (v_0, _) \in M\}, U_1 ← U_1 \ \{v_1 \mid (_, v_1) \in M\}
16    end
17    // Assert: ∀(v_0, v_1) ∈ M ∙ type_0(v_0) = type_1(v_1)
18    return (M, U_0, U_1)
19 end
```

3.3.2 Matching Refactored Program Elements. An assumption of the three-way matching algorithm is that each pair of vertices in $M_{bl} \cup M_{br}$ must have the same type. That is, matching detection in the bottom-up phase always happens between vertices of the same vertex type. Therefore, we further partition the unmatched vertex sets into subsets according to the type of each vertex, and detect matching in a divide-and-conquer manner.

For each subset of unmatched vertices (with the same type), we describe a list of heuristic rules in Table 3 to match refactored vertices. In our approach, the purpose of refactoring detection is to match changed vertices before and after refactorings, so we group refactoring changes into two categories: 1-to-1 matching and m-to-n matching, instead of explicitly identifying each refactoring type. These heuristic rules are based on the following assumptions: (1) from the base version to the left/right
version, the same program element cannot be part of multiple refactoring actions [Tsantalis et al. 2018]; and (2) refactorings with small code changes (e.g. Move Method) happen more frequently that those with big code changes (e.g. Extract Class) [Buckley et al. 2005; Negara et al. 2013]. Through this phase, the 6 sets calculated through the top-down phase are further updated: in general, the size of $U_{bl}, U_{lb}, U_{br}$ and $U_{rb}$ will decrease, and the size of $M_{bl}$ and $M_{br}$ will increase, since those matched vertices identified in this phase are moved from the former 4 sets to the latter 2 sets.

Table 3. Matching rules for supported refactoring types.

| Matching Kind | Vertex Type | Refactoring Type | Matching Rule |
|---------------|-------------|------------------|---------------|
| 1-to-1        | fld         | Rename, Move     | $\exists (fld_1, fld_2) \mid contextSimilarity(fld_1, fld_2)$ |
|               |             | Pull Up, Push Down | $+nameSimilarity(fld_1, fld_2) > \eta$ |
| m-to-n        | mtd         | Rename, Move     | $\exists (mtd_1, mtd_2) \mid contextSimilarity(mtd_1, mtd_2)$ |
|               |             | Pull Up, Push Down | $+bodySimilarity(mtd_1, mtd_2) > \eta$ |
|               | cls         | Rename, Move     | $\exists (cls_1, cls_2) \mid contextSimilarity(cls_1, cls_2) > \eta$ |
|               | pkd         | Rename           | $\exists (pkd_1, pkd_2) \mid contextSimilarity(pkd_1, pkd_2) > \eta$ |
|               | Extract     | Inline           | $\exists (mtd_1, [mtd_2, mtd_a]) \mid contextSimilarity(mtd_1, mtd_2)$ |
|               |             |                  | $\exists (mtd_1, (mtd_2 + mtd_a)) \mid contextSimilarity(mtd_1, mtd_2) > \eta$ |

(1) $v_1/v_2$ denotes one vertex from the two subsets that contain unmatched vertices with the same type
(2) $v_u$ denotes another unmatched vertex in one of the subsets
(3) $contextSimilarity(v_1, v_2)$ returns the jaccard similarity of the incoming and outgoing edges of $v_1$ and $v_2$
(4) $nameSimilarity(v_1, v_2)$ returns the string cosine similarity between the names of $v_1$ and $v_2$
(5) $bodySimilarity(v_1, v_2)$ returns the AST similarity between the bodies of terminal vertices $v_1$ and $v_2$
(6) $\eta$ denotes the confidence threshold for possible matchings, and is set to 0.618 (the golden section ratio) in our experiments

3.3.3 Best Matching Selection. For the 1-to-1 matching, if a vertex has multiple matching candidates, we use the following mechanism to select the best one. Firstly we rank the candidates in descending order according to three criteria: (1) the context similarity (i.e. given two candidate vertices, the vertex with a more similar edge set ranks higher), (2) the body-AST similarity (for terminal vertices) or the vertex-label similarity (i.e. given two candidate vertices with the same context similarity, the vertex with a more similar body or label ranks higher), and (3) the location distance from the vertex to the candidate, which is estimated according to their file paths and the start-line numbers (i.e. given two candidate vertices with the same context similarity and the same body-AST/vertex-label similarity, the vertex with closer location distance ranks higher). Since each program element in the same version of source code has an identical location, exactly one best matching candidate will be identified. Once the best matching is determined, the two vertices will be removed from the unmatched sets and added into the matched ones.

For the m-to-n matching, we use the following strategy to select the best one(s). In the current settings, the m-to-n matching mainly refers to matching the extracted method with the origin(s) and matching the inlined method with the target(s). The origin of an extracted method denotes the method from which it is extracted, and the target of an inlined method denotes the method that it is inlined into. In Java, programmers often extract the common part of multiple methods.
into a new method, so the origin can be more than one. Similarly, one method could be inlined into more than one target. If there are multiple candidate origins, we extend the context of each possible origin by combining it with the context of the extracted method, then rank the candidates by the similarity of the updated context. If there are multiple candidate targets, we update the context of each possible target by subtracting the context of the inlined method, then rank the candidates by the similarity of the updated context. We select the origin(s) and target(s) with the updated context similarity above the threshold $\eta$ as the best matching(s).

### 3.4 Step 3: Merging

In this step, a merged PEG is constructed, as well as a possible set of conflicts, based on the results of three-wayMatch($G_b, G_l, G_r$). A set $M_{lb\text{r}}$ is defined by joining elements in $M_{bl}$ and $M_{br}$.

$$M_{lb\text{r}} = \left\{ (v_l, v_b, v_r) \mid v_b \in V_b, v_l \in V_l \cup \{\phi\}, v_r \in V_r \cup \{\phi\}, (v_b, v_l) \in M_{bl}, (v_b, v_r) \in M_{br} \right\}$$

After that, the three sets of $M_{lb\text{r}}, U_{lb}$, and $U_{rb}$ are input into a three-way merging algorithm (see Algorithm 2), which consists of two procedures. First, each vertex of type cut in $M_{lb\text{r}}$ (corresponding to a Java source file) and its descendant vertices are recursively merged into a vertex tree with this vertex of type cut as the root vertex. After that, all those unmatched vertices in $U_{lb} \cup U_{rb}$ are inserted into the merged PEG at the proper position. These unmatched vertices are treated to be newly-added by either the left or right version. The position to insert newly added vertices is decided by locating its parent vertex in which it is inserted into, and its predecessor or successor sibling vertex (i.e. the neighboring vertex that appears before/after the corresponding program element in the source code) in the merged tree. If neither of the two neighbors is found, it will be appended to the end of the children vertex list.
Algorithm 2: Three-way Merging Algorithm

**Input**: \( M_{lb} \): the set of triples that represent the matchings
\( U_{lb} \): the set of vertices in \( V_l \) that are not matched with any vertex in \( V_b \)
\( U_{lr} \): the set of vertices in \( V_r \) that are not matched with any vertex in \( V_b \)

**Output**: \( CU_m \): the set of merged vertices of type cut

1. \( CU_m \leftarrow \emptyset \)
2. \( CU_{lb} \leftarrow \{ (v_l, v_b, v_r) \in M_{lb} \mid \text{type}(v_b) = \text{cut} \} \)
3. \( \text{foreach} \ (v_l, v_b, v_r) \in CU_{lb} \) do
   4. \( CU_m \leftarrow CU_m \cup \text{mergeVertex}(v_l, v_b, v_r) \)
4. \( \text{end} \)
5. \( \text{foreach} \ (v \in U_{lb} \cup U_{rb}, \ \text{type}(v) = \text{cut}) \) do
6. \( CU_m \leftarrow CU_m \cup \{ v \} \)
7. \( \text{end} \)
8. \( \text{foreach} \ (v \in U_{lb} \cup U_{rb}, \ \text{type}(v) \neq \text{cut}) \) do
   9. \( \text{parent} \leftarrow p(v) \)
10. \( \text{if} \ \text{parent} \in CU_m \) then
11. \( \text{insertVertex}(v, \text{parent}) \)
12. \( \text{end} \)
13. \( \text{return} \ CU_m \)
14. \( \text{end} \)

15. \( \text{Function} \ \text{mergeVertex}(v_l, v_b, v_r) \)
16. \( v_m \leftarrow \emptyset \)
17. \( \text{if} \ v_l \neq \emptyset \text{ and } v_r \neq \emptyset \) then
18. \( \text{if} \ \text{type}(v_b) \in VT_c \) then
19. \( v_m \leftarrow \text{mergeSignature}(v_l, v_b, v_r) \)
20. \( \text{foreach} \ (c_b \in C(v_b), (v_l, c_b, c_r) \in M_{lb}) \) do
21. \( c_m \leftarrow \text{mergeVertex}(c_l, c_b, c_r) \)
22. \( v_m, \text{appendChild}(c_m) \)
23. \( \text{end} \)
24. \( \text{else} \)
25. \( v_m \leftarrow \text{mergeComponents}(v_l, v_b, v_r) \)
26. \( \text{end} \)
27. \( \text{return} \ v_m \)

In the above process, conflicts might occur at two places: \text{mergeSignature} and \text{mergeComponents}, where the content of signature and components are merged textually in a fine-grained way. The component of a vertex is defined as the code snippets composing the corresponding program element. For example, the components of a method declaration mainly include the annotations, modifiers, return type, method name, parameter list, and the body. Table 1 describes detailed components for each vertex type.

### 3.5 Step 4: Graph-to-Code

In this step, the merged PEG is converted into a set of Java source files, with possible conflict blocks that require manual resolution. In the code-to-graph step, each file to be merged is firstly parsed.
into a Compilation Unit and then represented as a vertex of type cut in PEG. Therefore, starting from each cut vertex and traversing its outgoing edges of label in ELtree, sub-graphs that encode the content of merged files can be extracted from the merged PEG. After that, the vertices in each sub-graph are printed recursively into the code string, which will be written into a Java file with the name and path of the cut vertex. If there are conflicts, the conflicting code will be surrounded by a pair of conflicting symbols like GitMerge does, thus they can be detected by tools that facilitate developers to manually resolve conflicts.

4 IMPLEMENTATION

IntelliMerge is implemented as an open-source three-way merging tool for Java programs. It consists of 4 major modules: (1) GraphBuilder, (2) GraphMapper, (3) GraphMerger, and (4) CodeGenerator, corresponding to the 4 steps of the proposed approach. It can be used as a standalone software to merge two branches or three directories, or integrated into other applications through the well-documented API. Several open-source tools are used in the implementation: (1) jGit\(^6\) (an implementation of Git in Java) is used to compute the set of different files between the base and left/right version, including the added/deleted/modified files; (2) JavaParser\(^7\) (an AST parser for Java) is used to parse the source code into ASTs and resolve the symbols to construct the PEG; and (3) Gumtree\(^8\) (an AST diff tool for Java) is used to compute the similarity of method bodies in the form of ASTs.

Fig. 6. The GUI client of IntelliMerge to facilitate developers to review the merging results, and manually resolve the remaining conflicts.

IntelliMerge also provides a client with graphical user interface (GUI) (see Figure 6) for the developer to review the merged results and manually resolve remaining conflicts. Although IntelliMerge

\(^6\)https://www.eclipse.org/jgit
\(^7\)http://javaparser.org/
\(^8\)https://github.com/GumTreeDiff/gumtree
can significantly reduce the number of conflicts and auto-merge the code with good precision, we still think that any changes to the source code made by the merging tool should be carefully reviewed and confirmed by the developer before being formally committed. Therefore, the GUI client plays two roles: (1) marking the auto-merged results for the developer to review and confirm, (2) highlighting remaining conflict blocks and showing the corresponding code snippets in the three original versions, according to the intermediate mapping information collected in the matching procedure.

5 EVALUATION

To evaluate the feasibility of our approach in the practical environment, we conduct a series of experiments on 10 open-source projects to compare our graph-based approach with two other kinds of merging approaches, i.e. the unstructured and tree-based semi-structured approaches. For the former, we choose the default merging algorithm in Git. For the latter, we choose a state-of-the-art tree-based tool called jFSTMerge [Cavalcanti et al. 2017].

After collecting 1,070 merge scenarios with refactoring-related conflicts from open-source projects, we replay them with the three tools under the same environment to get the corresponding merged results. In accordance with the widely accepted concepts in Git, we divide the content of output files into the auto-merged part (the successfully merged code) and the conflicting part (the code inside conflict blocks). The three tools will be evaluated with the following three research questions:

(1) RQ1: What are the precision and recall of the three tools on the auto-merged code?
(2) RQ2: What are the number and size of merge conflicts reported by the three tools?
(3) RQ3: What is the performance of graph-based IntelliMerge and tree-based jFSTMerge?

5.1 Evaluation Setup

5.1.1 Data Collection. To select representative sample projects, we first search for the top 100 Java projects with high numbers of stargazers on Github, and select 10 projects according to the number of contributors and the number of merge commits. Then, those merge scenarios that involve refactoring-related conflicts in the 10 projects are collected, using the tool provided by a recent large-scale empirical study on the relationship between refactoring and merge conflicts [Mahmoudi et al. 2019]. This tool works in the following steps:

(1) For each project, analyze the commit history to find all failed merge scenarios, each of which reports a set of merge conflicts;
(2) For each failed merge scenario, find all commits that ever edited the conflicting code, between the base commit and the two parent commits;
(3) For each commit collected in the previous step, detect refactoring changes between it and the previous commit with RefactoringMiner\(^9\), a state-of-the-art refactoring detection tool with the precision of 98% and the recall of 87% [Tsantalis et al. 2018];
(4) For each failed merge scenario, if it contains at least one refactoring change that ever edited the conflicting code, add it into the experimental data set.

Since jFSTMerge does not support merging branches, in our experiments, the three merging tools all take the source files with refactoring-related conflicts as the input, and output a set of merged source code files as the results, possibly with a set of unresolved conflict blocks. All data is publicly available\(^10\).

---

\(^9\)https://github.com/tsantalis/RefactoringMiner
\(^10\)https://github.com/Symbolik/IntelliMerge-data
5.1.2 Studied Projects. Table 4 shows a statistical overview of the 10 projects studied in our experiment, which were collected on the master or main branch until April 1, 2019. The number of merge commits with conflicts is estimated by re-playing the merge scenario with `git merge` command, and whether a conflict is related with refactoring depends on if its code range overlaps with any refactoring changes detected by RefactoringMiner.

Table 4. A statistical overview of the 10 studied projects (on April 1, 2019).

| Project       | Stargazers | LOC  | Merge Commits with Conflicts | Merge Commits with Refactoring-related Conflicts |
|---------------|------------|------|------------------------------|-----------------------------------------------|
| cassandra     | 5038       | 562K | 3923                         | 587 (14.96%)                                  |
| elasticsearch | 39635      | 1906K| 568                          | 147 (25.88%)                                  |
| antlr4        | 5400       | 92K  | 345                          | 88 (25.51%)                                   |
| deeplearning4j| 10555      | 884K | 588                          | 72 (12.24%)                                   |
| gradle        | 8652       | 66K  | 710                          | 65 (9.15%)                                    |
| realm-java    | 10359      | 141K | 579                          | 56 (9.67%)                                    |
| storm         | 5618       | 398K | 258                          | 21 (8.14%)                                    |
| javaparser    | 2346       | 215K | 78                           | 18 (23.08%)                                   |
| junit4        | 7376       | 44K  | 47                           | 8 (17.02%)                                    |
| error-prone   | 4572       | 220K | 24                           | 8 (33.33%)                                    |

Figure 7 and Figure 8 present the frequency of refactoring types detected (a) in the whole commit history, and (b) in merge conflicts, respectively. In the commit history of the studied projects, the top three most frequent refactoring types are: (1) Move Field (36.7%), (2) Rename Method (12.9%) and (3) Move Class (12.25%). However, we find that over 87% (8324/9473) of Move Field refactorings are detected in the elasticsearch project, and over 92% (7668/8324) of them happened in one single commit (283fe90 (“Query Refactoring: Moving parser NAME constant to corresponding query builder”, 2015-05-13)). As a result, the frequency of Move Field refactorings is significantly increased by the outlier data. If we remove the elasticsearch project from the statistics, the top three refactoring types are: (1) Rename Method (17.74%), (2) Move Class (16.29%) and (3) Extract Method (13.55%). In refactorings that are related with merge conflicts, the top three most frequent refactoring types are: (1) Extract Method (22.56%), (2) Rename Method (16.74%), and (3) Move Field (14.55%). If we remove the elasticsearch project from the statistics, the top three refactoring types are: (1) Extract Method (25.55%), (2) Rename Method (15.53%), and (3) Rename Class (11.39%).

5.1.3 Environment. All of our experiments are conducted on a computer with Intel (R) Core (TM) i7-7700 CPU @ 3.60 GHz, 16 GB memory and Windows 10 OS.

5.2 Evaluation on Auto-merged Results (RQ1)

In this evaluation, we measure the precision and recall of the three tools by comparing the auto-merged part with the manually-committed code. In Git repositories, if the merging fails because of conflicts, the developer will be required to manually resolve them and commit the change as a `merge commit`, which has more than one parent commit. In three-way merging, the merge commit has two parent commits while the ordinary commit has only one. Therefore, we extract the source files committed at merge commits with two parents, and consider them as our ground truth. For the merged result of the three tools, we collect the auto-merged part by extracting and removing conflict blocks from the output files.
Fig. 7. The frequency of refactoring types detected in the whole commit history of the 10 studied projects. Refactoring types that account for less than 1% are ignored in the pie chart. The rows of the table are arranged in the clockwise order of the pie chart.

Fig. 8. The frequency of refactoring types involved in merge conflicts of the 10 studied projects. Refactoring types that account for less than 1% are ignored in the pie chart. The rows of the table are arranged in the clockwise order of the pie chart.

When comparing the auto-merged results with manually-committed versions, we use the `diff` tool provided by Git to calculate the number of different lines of code between them. In order to alleviate differences unrelated to the semantics of a program (such as changes in comments, format and indentation), we first remove all comments from both the auto-merged results and the manually-resolved versions, and then use `git diff` with the following parameters to compute the number of different lines:

```
$ git diff --ignore-cr-at-eol --ignore-all-space
   --ignore-blank-lines --ignore-space-change --no-index -U0
```
In the above command, the `--ignore-cr-at-eol` means to ignore carriage-return at the end of line when doing a comparison, the `--ignore-all-space` and `--ignore-space-change` mean to ignore space changes, and the `--ignore-blank-lines` is used to ignore changes whose lines are all blank. Besides, the `--no-index` allows `git diff` to compare arbitrary files that are not tracked by Git, and the `-U0` ensures entire different parts are reported without context code lines around them. More information about the parameters can be found in the official documentation\footnote{https://git-scm.com/docs/git-diff}.

We define the precision and recall of the auto-merged part as the proportion of the same lines (comparing with the manually-committed code) in auto-merged code and manually-committed code respectively. The number of same lines is computed by subtracting the number of different lines reported by the above `git diff` command from the number of auto-merged lines. Considering the motivation example, the result of GitMerge contains 23 lines of code (13 in the auto-merged part and 10 in the conflicting part), while the manually-committed result contains 14 lines of code. Comparing the GitMerge result with the manually-merged result, there are 2 different lines (line 6 and line 13). Therefore, for GitMerge in this merge scenario, the precision is about 84.62% ((13-2)/13), and the recall is about 78.57% ((13-2)/14). We use LOC to denote the number of code lines and define the precision and recall as follows:

\[
\text{precision} = \frac{\text{LOC(AutoMerged)} - \text{LOC(Diff)}}{\text{LOC(AutoMerged)}} \tag{1}
\]

\[
\text{recall} = \frac{\text{LOC(AutoMerged)} - \text{LOC(Diff)}}{\text{LOC(ManuallyMerged)}} \tag{2}
\]

Table 5. The precision and recall of the three merging tools on auto-merged code.

| Project   | IntelliMerge | jFSTMerge | GitMerge |
|-----------|--------------|-----------|----------|
|           | Precision    | Recall    | Precision| Recall    | Precision| Recall    |
| cassandra | 80.24%       | 82.25%    | 84.00%   | 72.21%    | 99.42%   | 80.06%    |
| elasticsearch | 88.86%     | 95.51%    | 89.74%   | 94.49%    | 99.73%   | 86.06%    |
| antlr4 | 79.13%       | 72.50%    | 77.25%   | 86.45%    | 99.41%   | 82.35%    |
| deeplearning4j | 87.45%     | 94.70%    | 90.99%   | 91.93%    | 99.31%   | 79.49%    |
| gradle | 89.87%       | 92.97%    | 85.98%   | 94.41%    | 99.24%   | 81.47%    |
| realm-java | 87.19%      | 93.26%    | 89.29%   | 93.60%    | 99.53%   | 82.55%    |
| storm | 88.00%       | 86.57%    | 89.77%   | 85.23%    | 99.61%   | 73.75%    |
| javaparser | 98.00%      | 97.55%    | 92.61%   | 93.47%    | 99.31%   | 81.99%    |
| junit4 | 95.84%       | 92.58%    | 88.37%   | 93.61%    | 99.24%   | 86.81%    |
| error-prone | 90.27%     | 94.35%    | 90.09%   | 87.13%    | 99.80%   | 78.27%    |
| Average | **88.48%**   | **90.22%** | 86.01%   | **87.22%** | 99.46%   | **81.28%** |

Answer to RQ1: As shown in Table 5, for auto-merged results, IntelliMerge achieves a mean precision of 88.48% and a mean recall of 90.22%. Comparing with jFSTMerge, it achieves a higher recall by merging more lines of code, without sacrificing the precision. Among the three tools, GitMerge exceeds the other tools with the precision of 99.46%. The reason is that all of the studied projects use GitMerge as their default merging tools, thus developers resolved conflicts directly based on the merged results of GitMerge. Furthermore, the auto-merged part was usually not
reviewed if the merging is done in terminal and text editors. That is to say, the manually-committed result is not the true ground truth, but a kind of biased ground truth in favor of GitMerge. This problem is further discussed in Section 7.1.

5.3 Evaluation on Reported Conflicts (RQ2)

In this evaluation, we compare the number and size of conflict blocks generated by the three tools. Although resolving all conflicts automatically without human intervention is suggested to be infeasible by [Menezes et al. 2018] through a large-scale empirical study, reducing the number of false positive merge conflicts does save developers’ effort and time in the manual resolution process. Therefore, the merge conflicts reported by the three tools are analyzed and compared. In order to do this, we firstly extract conflict blocks from all merged results, then count the total number of conflict blocks and lines of conflicting code. The results are shown in Figure 9 and Table 6, with the detailed number and the reduced percentage comparing with GitMerge for each project.

![Figure 9. The number of conflict blocks and lines of conflicting code reported for each projects by IntelliMerge, jFSTMerge and GitMerge (from left to right), in diff2 style.](image)

**Table 6. The percentage of reduced merge conflicts for each project, comparing with GitMerge**

| Project     | Number of Conflict Blocks | Lines of Conflicting Code |
|-------------|---------------------------|---------------------------|
|             | IntelliMerge | jFSTMerge | jFSTMerge | IntelliMerge | jFSTMerge |
| junit4       | 72.31%        | 46.15%    | 96.49%    | 67.24%       |
| error-prone | 62.97%        | 29.63%    | 94.03%    | 47.08%       |
| javaparser  | 81.89%        | 74.29%    | 96.18%    | 54.81%       |
| storm       | 65.35%        | 59.41%    | 92.82%    | 82.17%       |
| realm-java  | 69.27%        | 34.04%    | 88.63%    | 57.39%       |
| gradle      | 54.72%        | 54.99%    | 89.79%    | 60.42%       |
| deeplearning4j | 74.74% | 47.08%    | 91.78%    | 60.01%       |
| antlr4      | 62.90%        | 54.79%    | 88.64%    | 61.39%       |
| elasticsearch | 53.73%  | 45.24%    | 86.51%    | 56.64%       |
| cassandra   | 55.33%        | 56.86%    | 92.63%    | 57.77%       |
| Overall     | **58.90%**    | **53.38%** | **90.98%** | **58.27%**   |
**Answer to RQ2**: Comparing with the unstructured GitMerge, both semi-structured approaches (IntelliMerge and jFSTMerge) significantly reduce the number of conflicts, which is consistent with the conclusion in the previous work [Cavalcanti et al. 2017]. Comparing with GitMerge, IntelliMerge reduces the number of conflict blocks by 58.90% and the lines of conflicting code by 90.98%. Comparing with jFSTMerge, IntelliMerge further reduces the number of merge conflicts by 11.84% and the lines of conflicting code by 78.38%.

5.4 Evaluation on Performance (RQ3)

In this evaluation, we evaluate the performance of the graph-based IntelliMerge and the tree-based jFSTMerge from the perspective of runtime. For each merge scenario examined, we record the time in milliseconds that each merging tool takes to finish the merging process and output the merged results. Figure 10 shows the distribution of the execution time of IntelliMerge and jFSTMerge for examined merge scenarios, collected by executing each tool separately on the same machine.

![Fig. 10. A combination of a violin plot and a box plot (the filled part) showing the distribution of the execution time in milliseconds. The horizontal range of the colored part is the interquartile range (IQR) of the data, and the black vertical line inside it denotes the median value.](image)

**Answer to RQ3**: On the median, IntelliMerge takes 539 milliseconds to process a merge scenario, while jFSTMerge takes 1426 milliseconds. Although IntelliMerge adopts a graph-base approach, in over 90% of all merge scenarios it finishes the merging process within 5 seconds, which is acceptable for real-world applications.

6 RELATED WORK

6.1 Software Merging

In collaborative and distributed software development activities, developers rely on version control systems (VCS) to manage revisions and integrate contributions. While the branch-based workflow is widely adopted, a key drawback of it is that conflicts frequently occur when merging changes from separate branches [Mahmoudi et al. 2019], and developers have to manually resolve these conflicts. There are many studies that propose various merging techniques to reduce manual efforts in merging changes and resolving conflicts. According to the way they treat source code, software merging techniques can be largely categorized into three classes: unstructured, structured, and semi-structured merging [Mens 2002]:

6.1.1 Unstructured Merging. Unstructured merging is designed to work in a simple way: every software artifact that can be represented as text (i.e., as sequences of text lines) is processed as text. Therefore, any text-based tool is able to process all textual artifacts, regardless of the programming language [Mens 2002]. However, the downside is that when it comes to expressing differences
and handling merge conflicts, the unstructured merging is rather weak, because all structure and semantic information in source code are lost [Apel et al. 2011]. However, the most widely-used software merging tools are unstructured, such as the `diff` and `merging` tools in Unix, Subversion, and Git. The benefit is the light weight of these tools, and the cost is those false positive or negative conflicts, which have to be manually resolved or identified by developers.

### 6.1.2 Structured Merging

Structured merging aims at alleviating the problems of unstructured merging by exploiting the structure of software artifacts [Leßenich et al. 2015]. After two seminal works in the 1990s [Buffenbarger 1993; Westfechtel 1991] that pioneered in merging algorithms incorporating program structures, many proposals of structured merging techniques have been made. On one hand, there are proposals that target at modeling artifacts like diagrams and models [Jackson et al. 1994; Kolovos et al. 2006; Mehra et al. 2005]. On the other hand, there are proposals that focus on merging programs; some of them are dedicated to specific programming languages such as Java [Apel et al. 2012; Leßenich et al. 2015; Zhu and He 2018], C++ [Grass 1992] and XML [Lindholm 2004], while others try to be language-independent by representing program elements in an abstract level [Apel et al. 2011; Buffenbarger 1993; Westfechtel 1991]. However, these structured techniques are usually quite expensive and not practical in real software development projects.

### 6.1.3 Semi-structured Merging

Semi-structured merging is a trade-off between structured merging and unstructured merging, considering both the generality of unstructured merging and the precision of structured merging [Apel et al. 2011]. By representing the artifact partly as trees and partly as plain text, semi-structured merging achieves a certain degree of language-independence by merging with the generic `tree superimposition` algorithm. Previous studies [Apel et al. 2012; Cavalcanti et al. 2017] conclude that semi-structured merging is less precise than the structured merging, because only parts of the artifact are merged structurally; other parts (e.g. bodies of methods in Java/C++) are still merged in an unstructured manner. But in some cases like refactoring-involved merging, semi-structured merging can reach a balance between precision and performance, thus making it more feasible in practice than the fully structured merging.

### 6.2 Graph-Based Merging

Many researchers have found that AST tree-based structured merging approaches cannot detect all semantic conflicts or incompatible merging, because ASTs are only a kind of shallow representation of the semantics of a program [Mens 2002]. Therefore, graph-based structured merging approaches are proposed to overcome the limitations of tree-based approaches. For example, Binkley et al. proposed using call graph information to correctly merge different versions of the program [Binkley et al. 1995]. Mens used conditional graph rewriting to manage the evolution of software artifact and detect structural and behavioral inconsistencies when merging parallel evolutions [Mens 1999]. Niu et al. proposed representing programs as graphs and using graph morphisms to capture the correlation between the structural elements of the programs to be merged [Niu et al. 2005]. Despite the increased semantic information in graph-based merging approaches, these approaches are even more expensive than tree-based merging because of the in-depth program analysis and the complicated program representation. As a result, recent works mainly focus on making it work on merging software models instead of programs [Bartelt 2008; Schwägerl et al. 2015; Segura et al. 2007]. In our approach, we adopt a lightweight graph-based merging method to make it feasible in practice.

### 6.3 Refactoring Detection

As an important and common practice in agile development, refactoring is a widespread practice that helps developers improve the maintainability and readability of their code [Silva et al. 2016].
Refactoring has been well studied in the past few decades [Mens and Tourwé 2004], and one important research topic is the detection of refactoring changes in the evolution history [Choi et al. 2018]. Multiple tools are developed to detect different refactoring types defined by Martin Fowler [Fowler 2002], such as Ref-Finder [Kim et al. 2010], RefDiff [Silva and Valente 2017], RefDistiller [Alves et al. 2014] and RefactoringMiner [Tsantalis et al. 2018]. Among these tools, a state-of-the-art tool is RefactoringMiner, which achieves a precision of 98% and a recall of 87%. RefactoringMiner is used in our evaluation process to mine merge scenarios that failed to be auto-merged because of conflicts related with refactoring changes. Our objective is not to detect refactorings, but to match program elements more precisely by taking refactoring changes into consideration. Therefore, our approach does not detect explicit refactoring types but classifies refactorings with regard to how they change the program structure.

6.4 Refactoring-Aware Merging
The first refactoring-aware merging algorithm was proposed in [Dig et al. 2007, 2008, 2006b], and implemented as an Eclipse plugin named MolhadoRef. MolhadoRef takes the operation-based approach [Lippe and Van Oosterom 1992], in which one version is represented as a sequence of change operations (refactoring changes and edits) and these operations are replayed in the merging activities. In particular, MolhadoRef inverts refactoring changes first, then applies edits to refactored program entities and replays refactoring changes. However, MolhadoRef has two limitations: (1) the refactoring and edit history must be available and precise, and (2) supported refactoring types must have corresponding inverse refactoring operations. Therefore, MolhadoRef only supports a limited set of refactoring types, such as renaming and moving. The lack of support with complex refactoring types like Extract Method and Inline Method makes MolhadoRef difficult to be used in real projects. What makes our work different from previous ones is how we consider the role of refactoring-aware merging tools. Instead of merging code as much as possible, IntelliMerge focuses on avoiding fake conflicts and provide useful information for real conflicts to facilitate manual resolution.

Recently, a large-scale (3,000 projects) empirical study on the relationship between refactoring and merge conflicts [Mahmoudi et al. 2019] concludes that: (1) refactoring operations are involved in at least 22% of merge conflicts; (2) refactoring-involved conflicts are more complex both for current merging tools and developers; (3) certain refactoring types like Extract Method tend to be more possible to bring merge conflicts. Another empirical study on the motivations of Github’s contributors behind refactoring operation [Silva et al. 2016] also points out that Extract Method is the most versatile refactoring operation serving 11 different purposes, which indicates that it might be the most widespread refactoring type.

6.5 Conflicts Prevention in Real-Time Collaboration
While developers mostly rely on version control systems to collaborate currently, researchers also propose multiple approaches to prevent conflicts or identify them early by enabling developers to collaborate in real-time. Different from version control systems that use optimistic locking to enable parallel work, real-time collaborative programming typically adopts pessimistic locking to avoid competing editing, thus preventing textual conflicts from happening. In order to prevent semantic conflicts, Fan and Sun [2012] propose a dependency-based automatic locking (DAL) approach to support automatic fine-grained locking on interdependent programming segments. They further provide flexible and customizable locking policy settings to balance conflict prevention and concurrent work [Fan et al. 2017a,b]. Steimann and Kurowsky [2019] propose transactional editing by enforcing the notions of consistency and isolation from the transactional database in the context of collaborative programming. By formalizing code editing as a set of primitive edit
operations and deriving the write- and read-locks for them, they achieve fine-grained locking in collaborative programming and global consistency before committing.

7 DISCUSSION

7.1 Threat to Validity

Construct Validity. In order to evaluate the merged result, a ground truth should be firstly established that is expected to consist of the correctly merged results confirmed by developers. Initially, we assume that those files manually committed as the merge commits are a good candidate for the ground truth, because these files are generated by developers after manually resolving all the conflicts detected by Git. However, after empirically inspecting these merge commits, we find that the above assumption is not true actually. We have found many merge commits that contains code with obvious syntax errors and even unresolved conflicts. Figure 11 shows two manually-merged results with syntax errors from the studied projects. Some developers even left conflicts unresolved in explicit merge commits (one such example is given in Figure 12). More examples can be found and reproduced at https://github.com/Symbolk/IntelliMerge-data.

Fig. 11. Two examples of incorrect merging that causes syntax errors.

One possible reason behind this kind of phenomenon may be that the merging activities were usually done in the terminal mode, and the embedded text editors like Vim were used to resolve conflicts, without intelligent syntax error detection. Therefore, the developers may just resolve conflict blocks by simply accepting one side and leave auto-merged parts as they were, and then commit the modified files without compiling or testing. As a result, those syntax errors and/or unresolved conflicts have to be corrected in later commits. So we try to refine the ground truth by identifying these incorrect merged results and replacing them with the revised version in the following commit. However, we find the correction was rarely committed separately in one single commit; that is, the next commit following a merge commit usually contains not only the correction, but also other changes (like reformattting and adding new features), which are hard to distinguish from the correction. Therefore, we first identify the manually-merged files with syntax errors or unresolved conflicts, and then replace them with the corresponding version at the next commit following the merge commit. Further human-involved validation is necessary and inevitable to build a more credible ground truth.

Internal Validity. Any problems inherited from the tools used in the study may lead to inaccuracy in results. To mitigate that, we first carefully assess the role of each tool involved in our study,
then analyze the robustness of crucial ones through manual verification and explain how we try to
tackle their problems.

The tools used in the study fall into two categories:

(1) Implementation-related: this category consists of three tools integrated in our implementation:
(1) jGit, (2) JavaParser, and (3) Gumtree. As a Java implementation of Git, jGit has been widely
used both in academia and industry, and in our implementation jGit is simply used to get the
changed file set between the old and new versions based on the hash function, so the results
can be trusted. JavaParser is used to parse the Java source file into the
compilation unit (the
root AST node of the program in this file) and resolve the symbols to find corresponding
definitions. After comparing the source code with the visualized PEG, we find that JavaParser
accurately parses the source code into an AST, but it fails to infer the definition for some
symbols in the program, possibly because these symbols are defined in external jar libraries.
As a result, the matching precision will suffer from the incomplete edge set of PEG. To handle
this problem, we create dummy vertices to represent these unresolved symbols and compare
them only according to their labels. Gumtree is used to measure the similarity of method
bodies according to their ASTs. We empirically compare it with the string similarity algorithm
based on edit distance, and conclude that the result of Gumtree is more reasonable than
algorithms that treat the code as a plain text string.

(2) Evaluation-related: this category contains the tool used in the data collection, which is
provided by Mahmoudi et al. [2019]. This tool undertakes two tasks: (1) find all failed merge
cenarios from the commit history and record conflict blocks, (2) detect refactoring changes at
commits that touched the conflicting code. The first task is implemented with jGit, a
full-fledged implementation that we choose to trust. The second task is accomplished with
RefactoringMiner, which claims to achieve a precision of 98% and a recall of 87% [Tsantalis
et al. 2018]. We randomly sample 10% from the detected refactorings of each studied project,
and manually validate them by inspecting the code before and after each refactoring. The
result shows that the precision of RefactoringMiner is about 90%. The recall is hard to estimate
because we are unable to determine exactly what refactorings were applied by the original
developers in the history.

Fig. 12. An example of conflict block remaining unresolved but was committed at a merge commit.

Proc. ACM Program. Lang., Vol. 3, No. OOPSLA, Article 170. Publication date: October 2019.
Although most of the refactorings detected by RefactoringMiner can explain the change (so we consider them to be correct), they are possibly different from the actions applied by the original author. For example, since RefactoringMiner only analyzes the added/deleted/modified files, it cannot identify the indirect inheritance relationship between classes if part of the intermediate classes is unchanged. Suppose that one method is moved between two classes that have such an indirect inheritance relationship, RefactoringMiner may report a Move Method refactoring instead of a Pull Up/Push Down Method refactoring. For another example, we find that for some common and short program elements (e.g., fields like `int id/String name` and methods like `toString()` and getters/setters), RefactoringMiner tends to treat independent addition and deletion changes upon them as movements. As a result, the number of reported Move refactorings is more than actual, and the detected refactoring changes can be different from the developer’s original operations.

External Validity. Our evaluation focuses on Java open-source projects, so our results are limited in this scope. Like other works based on software analysis, the implementation of structured or semi-structured merging tool is language-specific, but the approach in general can be extended to other object-oriented program languages and applied on closed-source enterprise projects. By selecting well-engineered sample projects from various fields and with different sizes, we try our best to evaluate our approach in a wide range. Therefore, the experiment results based on these sample projects are dependable to support our conclusions.

7.2 Limitations

Non-structural Refactorings. We mainly focus on structural refactorings in this work because the previous study has proved that they bring trouble to the present merging approaches [Mahmoudi et al. 2019]. To our knowledge, previous works on refactoring detection also mainly focus on structural refactorings. For instance, RefactoringMiner only supports 25 refactoring types out of the 72 refactoring types described in Martin Fowler’s catalog, and most of the supported types are structural ones. The current implementation of the PEG only encodes the high-level structure of programs, which limits IntelliMerge’s support for non-structural refactorings like control flow refactorings. Though refactorings that change the program structure are responsible for a great number of false positive and false negative merge conflicts, refactorings that change the control flow are also important for refactoring-aware merging. Therefore, to support non-structural refactorings, we plan to extend the definition of the PEG by investigating the relevance between them and merge conflicts in our forthcoming work.

False Negative Conflicts. While IntelliMerge reports much fewer merge conflicts and less conflicting code, it cannot guarantee to reduce the number of false negative conflicts for sure. In our work, we define false negative conflicts as syntactic and semantic conflicts that are neglected by the merging tool, which can lead to latent bugs and severe errors. For example, if inconsistent changes are applied to the interdependent program elements in the left and right versions, unstructured merging tools will not report a conflict since these changes do not overlap. Some false negative conflicts that lead to compilation errors can be identified through compiling, but other false negatives that modify the program logic are hard to detect, even with building systems and test cases. As the PEG encodes the dependency relationship among program elements through static analysis, it can be used in detecting false negative conflicts that violate the well-formedness of the programming language. However, detecting other kinds of false negatives requires more in-depth analysis in the building and testing phases, as well as getting developers involved. Therefore, we plan to make further efforts in studying how to interactively assist developers in reviewing the merged result and resolving conflicts with our merging tool.

12https://refactoring.com/catalog/
8 CONCLUSION

In modern software development, merging is a vital and frequent operation in collaboration. While the industry still relies on unstructured merging techniques, previous studies have explored multiple structured and semi-structured approaches. However, when merging meets refactoring, a common and popular type of changes in OOP and agile development, current refactoring-unaware techniques often fail to match the changed code correctly, let alone merging changes in the right way. To address this issue, we propose IntelliMerge, a graph-based three-way merging technique with enhanced ability in handling refactoring changes or other changes that lead to mismatching. We implement IntelliMerge to work with Git and for Java programs, and evaluate it on 10 real-world popular Java projects. By comparing it with the prevalent unstructured merging tool GitMerge and the state-of-the-art semi-structured merging tool jFSTMerge, we show the effectiveness and practicability of IntelliMerge. We plan to explore two directions in the future work: (1) support more types of refactoring like those refactoring changes to the control flow inside a method, by extending our current definition of the PEG; (2) identify and visualize various kinds of dependencies between different conflict blocks to facilitate the developer’s manual conflict resolving activities.

ACKNOWLEDGMENTS

We would like to thank the anonymous OOPSLA’19 reviewers for their invaluable comments on an earlier version of this paper. This material is based on research sponsored by the National Natural Science Foundation of China under Grant No. 61690200 and Grant No. 61432020, and the National Basic Research Program of China (973 Program) under Grant No. 2015CB352201.

REFERENCES

Iftekhar Ahmed, Caius Brindescu, Umme Ayda Mannan, Carlos Jensen, and Anita Sarma. 2017. An empirical examination of the relationship between code smells and merge conflicts. In 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 58–67.

Everton LG Alves, Myoungkyu Song, and Miryung Kim. 2014. RefDistiller: a refactoring aware code review tool for inspecting manual refactoring edits. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 751–754.

Sven Apel, Olaf Leßenich, and Christian Lengauer. 2012. Structured merge with auto-tuning: balancing precision and performance. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ACM, 120–129.

Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured merge: rethinking merge in revision control systems. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 190–200.

Christian Bartelt. 2008. Consistence preserving model merge in collaborative development processes. In Proceedings of the 2008 international workshop on Comparison and versioning of software models. ACM, 13–18.

David Binkley, Susan Horwitz, and Thomas Reps. 1995. Program integration for languages with procedure calls. ACM Transactions on Software Engineering and Methodology (TOSEM) 4, 1 (1995), 3–35.

Christian Bird and Thomas Zimmermann. 2012. Assessing the value of branches with what-if analysis. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, 45.

Caius Brindescu. 2018. How do developers resolve merge conflicts? an investigation into the processes, tools, and improvements. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 952–955.

Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2011. Proactive detection of collaboration conflicts. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 168–178.

Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. 2005. Towards a taxonomy of software change. Journal of Software Maintenance and Evolution: Research and Practice 17, 5 (2005), 309–332.

Jim Buffenbarger. 1993. Syntactic software merging. In Software Configuration Management. Springer, 153–172.

Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and improving semistructured merge. Proceedings of the ACM on Programming Languages 1, OOPSLA (2017), 59.
Eunjong Choi, Kenji Fujiwara, Norihiro Yoshida, and Shinpei Hayashi. 2018. A survey of refactoring detection techniques based on change history analysis. arXiv preprint arXiv:1808.02320 (2018).

Danny Dig, Kashif, Tien N Nguyen, and Ralph Johnson. 2006a. Refactoring-aware software merging and configuration management. Acm Sigsoft Software Engineering Notes 31, 6 (2006), 1–2.

Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. 2007. Refactoring-Aware Configuration Management for Object-Oriented Programs. In International Conference on Software Engineering.

Danny Dig, Kashif Manzoor, Ralph E Johnson, and Tien N Nguyen. 2008. Effective software merging in the presence of object-oriented refactorings. IEEE Transactions on Software Engineering 34, 3 (2008), 321–335.

Danny Dig, Tien N Nguyen, Kashif Manzoor, and Ralph Johnson. 2006b. MolhadoRef: a refactoring-aware software configuration management tool. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. ACM, 732–733.

Hongfei Fan, Jiayao Gao, Hongming Zhu, Qin Liu, Yang Shi, and Chengzheng Sun. 2017a. Balancing Conflict Prevention and Concurrent Work in Real-Time Collaborative Programming. In Proceedings of the 12th Chinese Conference on Computer Supported Cooperative Work and Social Computing, ACM, 217–220.

Hongfei Fan and Chengzheng Sun. 2012. Dependency-based automatic locking for semantic conflict prevention in real-time collaborative programming. In Proceedings of the 27th Annual ACM Symposium on Applied Computing. ACM, 737–742.

Hongfei Fan, Hongming Zhu, Qin Liu, Yang Shi, and Chengzheng Sun. 2017b. Shared-locking for semantic conflict prevention in real-time collaborative programming. In 2017 IEEE 21st International Conference on Computer Supported Cooperative Work in Design (CSCWD). IEEE, 174–179.

Martin Fowler. 2002. Refactoring: Improving the Design of Existing Code. In Xp Universe First Agile Universe Conference on Extreme Programming Agile Methods-xp/agile Universe.

Judith E Grass. 1992. Cdiff: A syntax directed differencer for C++ programs. In C++ Conference, Portland, Oregon, 1992. 181–193.

Mário Luís Guimarães and Antônio Rito Silva. 2012. Improving early detection of software merge conflicts. In Proceedings of the 34th International Conference on Software Engineering. IEEE Press, 342–352.

Daniel Jackson, David A Ladd, et al. 1994. Semantic Diff: A Tool for Summarizing the Effects of Modifications.. In ICSM, Vol. 94, 243–252.

Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive conflict minimization through optimized task scheduling. In Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 732–741.

Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010. Ref-Finder: a refactoring reconstruction tool based on logic query templates. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. ACM, 371–372.

Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2006. Merging models with the epsilon merging language (EML). In International Conference on Model Driven Engineering Languages and Systems. Springer, 215–229.

Hoai Le Nguyen and Claudia-Lavinia Ignat. 2018. An Analysis of Merge Conflicts and Resolutions in Git-based Open Source Projects. Computer Supported Cooperative Work (CSCW) 27, 5–6 (2018), 741–765.

Olaf Leßnich, Sven Apel, and Christian Lengauer. 2015. Balancing precision and performance in structured merge. Automated Software Engineering 22, 3 (2015), 367–397.

Tancred Lindholm. 2004. A three-way merge for XML documents. In Proceedings of the 2004 ACM symposium on Document engineering. ACM, 1–10.

Ernst Lippe and Norbert Van Oosterom. 1992. Operation-based merging. In Acm Sigsoft Software Engineering Notes, Vol. 17. ACM, 78–87.

Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. 2019. Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 151–162.

Akhil Mehra, John Grundy, and John Hosking. 2005. A generic approach to supporting diagram differencing and merging for collaborative design. In Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering. ACM, 204–213.

Gleiph Ghiotto Lima Menezes, Leonardo Gresta Paulino Murta, Marcio Oliveira Barros, and Andre Van Der Hoek. 2018. On the Nature of Merge Conflicts: a Study of 2,731 Open Source Java Projects Hosted by GitHub. IEEE Transactions on Software Engineering (2018).

Tom Mens. 1999. Conditional graph rewriting as a domain-independent formalism for software evolution. In International Workshop on Applications of Graph Transformations with Industrial Relevance. Springer, 127–143.

T. Mens. 2002. A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering 28, 5 (2002), 449–462.

Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. IEEE Transactions on software engineering 30, 2 (2004), 126–139.

Proc. ACM Program. Lang., Vol. 3, No. OOPSLA, Article 170. Publication date: October 2019.
Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang

Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson, and Danny Dig. 2013. A comparative study of manual and automated refactoring. In European Conference on Object-Oriented Programming. Springer, 552–576.

Nan Niu, Steve Easterbrook, and Mehrdad Sabetzadeh. 2005. A category-theoretic approach to syntactic software merging. In 21st IEEE International Conference on Software Maintenance (ICSM’05). IEEE, 197–206.

Felix Schwägerl, Sabrina Uhrig, and Bernhard Westfechtel. 2015. A graph-based algorithm for three-way merging of ordered collections in EMF models. Science of Computer Programming 113 (2015), 51–81.

Sergio Segura, David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. 2007. Automated merging of feature models using graph transformations. In International Summer School on Generative and Transformational Techniques in Software Engineering. Springer, 489–505.

Danilo SIlva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of github contributors. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 858–870.

Danilo Silva and Marco Tulio Valente. 2017. RefDiff: detecting refactorings in version histories. In Proceedings of the 14th International Conference on Mining Software Repositories. IEEE Press, 269–279.

Friedrich Steimann and Nils Kurowsky. 2019. Transactional Editing: Giving ACID to Programmers. In Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering. ACM.

Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In Proceedings of the 40th International Conference on Software Engineering. ACM, 483–494.

Chuck Walrad and Darrel Strom. 2002. The importance of branching models in SCM. Computer 35, 9 (2002), 31–38.

Bernhard Westfechtel. 1991. Structure-oriented merging of revisions of software documents. In Software Configuration Management Workshop: Proceedings of the 3rd international workshop on Software configuration management, Vol. 12. 68–79.

Fengmin Zhu and Fei He. 2018. Conflict resolution for structured merge via version space algebra. Proceedings of the ACM on Programming Languages 2, OOPSLA (2018), 166.