Understanding the Impact of Branch Edit Features for the Automatic Prediction of Merge Conflict Resolutions

Waad Aldndni
Virginia Tech
Blacksburg, VA, U.S.A.
waada@vt.edu

Francisco Servant
ITIS Software, Universidad de Málaga
Málaga, Spain
fservant@uma.es

Na Meng
Virginia Tech
Blacksburg, VA, U.S.A.
nm8247@vt.edu

ABSTRACT

Developers regularly have to resolve merge conflicts, *i.e.,* two conflicting sets of changes to the same files in different branches, which can be tedious and error-prone. To resolve conflicts, developers typically: keep the local version (KL) or the remote version (KR) of the code. They also sometimes manually edit both versions into a single one (ME). However, most existing techniques only support merging the local and remote versions (the ME strategy).

We recently proposed RPredictor, a machine learning-based approach to support developers in choosing how to resolve a conflict (by KL, KR, or ME), by predicting their resolution strategy. In its original design, RPredictor uses a set of *Evolution History Features* (*EHF*s) that capture: the magnitude of the changes in conflict, their evolution, and the experience of the developers involved.

In this paper, we proposed and evaluated a new set of *Branch Edit Features* (*BEF*s), that capture the fine-grained edits that were performed on each branch of the conflict. We learned multiple lessons. First, *BEF*s provided lower effectiveness (F-score) than the original *EHF*s. Second, combining *BEF*s with *EHF*s still did not improve the effectiveness of *EHF*s, it provided the same f-score. Third, the feature set that provided highest effectiveness in our experiments was the combination of *EHF*s with a subset of *BEF*s that captures the number of insertions performed in the local branch, but this combination only improved *EHF*s by 3 pp. f-score. Finally, our experiments also share the lesson that some feature sets provided higher C-score (*i.e.,* the safety of the technique's mistakes) as a trade-off for lower f-scores. This may be valued by developers and we believe that it should be studied in the future.

ACM Reference Format:
Waad Aldndni, Francisco Servant, and Na Meng. 2024. Understanding the Impact of Branch Edit Features for the Automatic Prediction of Merge Conflict Resolutions. In *32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3643916.3644433

1 INTRODUCTION

In collaborative software development, developers often create separate branches to handle different tasks simultaneously (e.g., add new features, fix bugs, or refactor code). When developers merge changes from various branches, edits that were made to the same line of code can conflict with each other.

The manual resolution of such conflicts is typically quite challenging and time-consuming. A previous study [47] found that 56% of developers postponed resolving merge conflicts for various reasons, most of them related to the complexity of the conflicts.

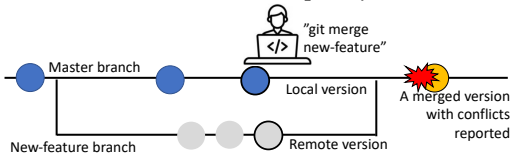

Figure 1: Developers use textual merge (e.g., git-merge) to merge branches and reveal conflicts.

The most popular merge tools are text-based (*e.g.,* git-merge [1]) and assist developers in tentatively merging the latest version of their own branch (*i.e.,* **local version (L)**) with the latest version of a specified branch (*i.e.,* **remote version (R)**), and in detecting textual conflicts in this process (see Figure 1). Because such tools treat programs as plain text, they can merge the code in ways that are syntactically or semantically incorrect, due to code mismatches between branches [15, 48, 61]. To improve over textual merge, researchers proposed tools that analyze the syntactic structures of programs to better detect and resolve conflicts [9, 10, 62, 68]. For instance, JDime [9] matches Java code based on abstract syntax trees (ASTs). It conducts tree-based merge instead of text-based merge for each matching node pair, to better align code and integrate as many edits as possible between branches.

These techniques mainly focus on merging the local and remote branches into a single version, by adapting them. However, developers resolve conflicts via three main strategies: choosing the local version while discarding the remote one (**KL**), choosing the remote version while discarding local (**KR**), or modifying edits from either or both branches for edit integration (**ME**), *e.g.,* [8, 25]. Furthermore, Yuzuki *et al.* [66] found that developers resolved 99% of conflicting methods by keeping only one of the conflicting versions (KL or KR). In a different dataset, Ghiotto *et al.* [25] found that developers resolved 56% of cases by KL or KR.

Inspired by these studies, we created RPredictor [8], a novel approach that resolves merge conflicts by considering developers' preferences. Given a merge conflict, RPredictor recommends to resolve it by KL, KR, or ME, based on what it predicts that the developer will prefer. Developers can benefit from RPredictor in two ways. First, when it predicts KL or KR, RPredictor could automatically apply the strategy and resolve the conflict, saving developers time and manual effort (that they could invest in better resolving other conflicts). Since the KL and KR strategies are the most popular ones [25, 66], this can produce very high effort savings. Second,

ICPC '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0586-1/24/04
https://doi.org/10.1145/3643916.3644433

when RPredictor predicts ME, developers would be reminded to be cautious in the resolution.

In this paper, we aim to improve RPredictor's effectiveness with an alternative set of prediction features, inspired by the research literature. RPredictor uses a Random Forest classifier, originally based on a set of *Evolution History Features* (*EHF*s) that capture: the magnitude of the conflicting changes, their evolution, and the experience of the developers involved [8]. We propose a new set of *Branch Edit Features* (*BEF*s), that capture the fine-grained edits that were performed on each conflicting branch. Past work observed that the modified code elements on each branch [25], the edit types applied to them [61], or the complexity of the changed code [65] could influence how developers resolve the conflict.

We defined a collection of 396 *BEF*s that capture the number of changes performed over 66 fine-grained code elements, for 3 edit types, on each of the two conflicting branches. Then, we selected among them the 122 *BEF*s that we found to differ with statistical significance for the three resolution strategies —- since they would be promising for prediction. We evaluated the prediction power of these 122 *BEF*s, in different ways. First, we compared the effectiveness provided by *BEF*s as features in RPredictor with that provided by its original design using *EHF*s. Then, we measured the effectivenes of combining both *BEF*s and *EHF*s. Then, we also studied in finer granularity the effectiveness provided by 6 subsets of *BEF*s, (based on each branch and edit type) as well as their combination with *EHF*s. We performed this evaluation over 15,899 resolved conflicts from 377 software projects, for both the within-project and cross-project prediction contexts.

Our experiments provided multiple findings. First, using *BEF*s for conflict resolution prediction did not improve the effectiveness of RPredictor— *BEF*s provided slightly lower effectiveness than *EHF*s. Second, combining *EHF*s and *BEF*s for prediction also did not improve the effectiveness of using only *EHF*s — both approaches provided the same f-score. Third, among all our studied feature sets, the best-performing one was *EHF*s and BEF_{LI}, but it only improved the effectiveness of *EHF*s by 3pp. f-score. We are unsure if developers would consider such an incremental improvement worth the effort of writing the code for collecting the additional BEF_{LI} features, and maintaining it over time. Finally, some of the feature sets that could not improve the effectiveness of *EHF*s were still able to improve their C-score, *e.g.,* BEF_{RD} provided 18pp. higher C-score with the trade-off of 3pp. lower f-score. This motivates future work to study what kinds of trade-offs between f-score and C-score developers would prefer.

Research Artifact Availability. Our research artifact can be accessed online [4].

2 APPROACH

We previously proposed RPredictor [8] to automatically predict the resolution strategy for developers to resolve merge conflicts, among: choosing the local version while discarding the remote one (**KL**), choosing the remote version while discarding local (**KR**), or modifying edits from either or both branches for integration (**ME**). We represent RPredictor in Figure 2.

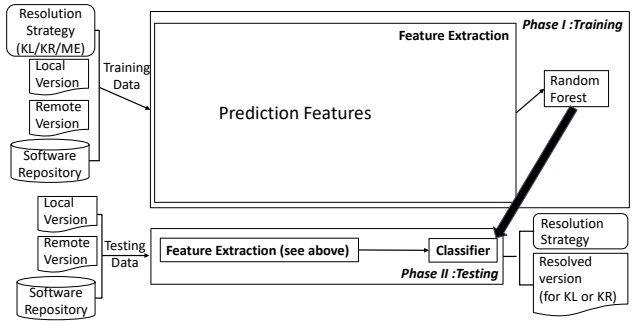

Figure 2: RPredictor has two phases: training and testing

RPredictor uses a random forest (RF) algorithm to make its predictions. First, in its training phase, it analyzes the merge conflicts that were resolved in the past (from the same and/or other software projects) to build its prediction model. Then, for a given merge conflict, it uses this model to predict whether developers will resolve it by KL, KR, or ME (as a three-class classifier). We implemented RPredictor using using scikit-learn [52].

Original: Evolution History Features (*EHF*s). Our original design of RPredictor [8] used features that aimed to capture some of the dimensions that we believed would influence developer decisions when resolving a merge conflict. They focused on the magnitude of the changes, their evolution, and the experience of the developers involved. We report them in Table 3.

New: Branch Edit Features (*BEF*s). In this paper, we propose a new set of *Branch Edit Features* (*BEF*s) and study whether they could improve RPredictor's effectiveness. *BEF*s capture the fine-grained edits on each of the branches involved in the merge conflict. Past work observed that the strategy that developers followed to resolve merge conflicts could be influenced by the code elements that were modified on each branch [25], the edit types applied to them [61], or the complexity of the changed code [65]. Our intuition is that our new branch edit features could capture these concepts and better inform RPredictor to make its predictions.

For our new Branch Edit features, we identified a collection of 66 fine-grained elements in Java, and 3 types of edits that could have been performed in them, *i.e.,* Insertion, Update, Deletion (Table 1). We studied the 66 elements that we observed being inserted, updated, or deleted in our studied dataset (see §3.1). Our features define Java elements, but they could be extended to other languages.

For a given conflicting chunk, our new branch edit features (*BEF*s) measure, separately, the number of Insertions (I), Updates (U), or Deletions (D) that were performed to each Java element (JE_i), within each of the Local (L) or Remote (R) branch. For example feature BEF_{LIJE_1} measures the number of *Variable Declaration Statements* (JE_1) that were inserted (I) in the local branch (L). In total, our *BEF*s capture 396 aspects of a conflicting chunk (2 branches × 3 edit types × 66 Java elements). We represent them in Table 2.

We also divide our Branch Edit Features into 6 subsets, to separately represent each edit type and branch. For example, BEF_{LI} is the subset of 66 features that capture the number of Insertions (I) that were performed in the Local (L) branch for each one of the 66 Java elements. That is, BEF_{LI} contains the 66 features BEF_{LIJE_1} – $BEF_{LIJE_{66}}$. We represent them in the bottom row of Table 2.

Table 1: The 66 fine-grained Java elements (JE_i) for which we measure our new proposed branch edits (*BEF*s).

Java elements (JE_i)
(JE_1) *Variable Declaration Statement*, (JE_2) *Expression Statement*, (JE_3) *Enhanced For Statement*, (JE_4) *If Statement*, (JE_5) *Method Invocation Receiver*, (JE_6) *Simple Name*, (JE_7) *Method Invocation*, (JE_8) *Break Statement*, (JE_9) *Return Statement*, (JE_{10}) *Class Instance Creation*, (JE_{11}) *Number Literal*, (JE_{12}) *Simple Type*, (JE_{13}) *Method Invocation Argument*, (JE_{14}) *Assignment*, (JE_{15}) *For Statement*, (JE_{16}) *Super Constructor Invocation*, (JE_{17}) *Block*, (JE_{18}) *String Literal*, (JE_{19}) *Type Declaration Statement*, (JE_{20}) *Variable Declaration Fragment*, (JE_{21}) *Boolean Literal*, (JE_{22}) *Infix Expression*, (JE_{23}) *Method Declaration*, (JE_{24}) *Type Declaration*, (JE_{25}) *Type Literal*, (JE_{26}) *Modifier*, (JE_{27}) *Field Declaration*, (JE_{28}) *Single Variable Declaration*, (JE_{29}) *Try Statement*, (JE_{30}) *Qualified Name*, (JE_{31}) *Primitive Type*, (JE_{32}) *Array Access*, (JE_{33}) *Variable Declaration Expression*, (JE_{34}) *Continue Statement*, (JE_{35}) *Synchronized Statement*, (JE_{36}) *Null Literal*, (JE_{37}) *Prefix Expression*, (JE_{38}) *Throw Statement*, (JE_{39}) *Marker Annotation*, (JE_{40}) *Cast Expression*, (JE_{41}) *Parenthesized Expression*, (JE_{42}) *Parameterized Type*, (JE_{43}) *Anonymous Class Declaration*, (JE_{44}) *Infix Expression Operator*, (JE_{45}) *While Statement*, (JE_{46}) *Switch Statement*, (JE_{47}) *Field Access*, (JE_{48}) *Assignment Operator*, (JE_{49}) *Array Type*, (JE_{50}) *Post fix Expression*, (JE_{51}) *Conditional Expression*, (JE_{52}) *Catch Clause*, (JE_{53}) *Initializer*, (JE_{54}) *This Expression*, (JE_{55}) *Dimension*, (JE_{56}) *Array Initializer*, (JE_{57}) *Array Creation*, (JE_{58}) *Normal Annotation*, (JE_{59}) *Member Value Pair*, (JE_{60}) *Empty Statement*, (JE_{61}) *Union Type*, (JE_{62}) *Character Literal*, (JE_{63}) *Constructor Invocation*, (JE_{64}) *Prefix Expression Operator*, (JE_{65}) *Labeled Statement*, (JE_{66}) *Single Member Annotation*.

Table 2: Our new proposed Branch Edit Features (*BEF*s). They measure the number of edits performed for each: branch, edit type, and Java element. We divide them into 6 subsets, each representing all Java elements for a branch and edit type.

Branch		**Local (L)**			**Remote (R)**		
Edit type		**Insertions (I)**	**Updates (U)**	**Deletions (D)**	**Insertions (I)**	**Updates (U)**	**Deletions (D)**
Java element (JE_i)	JE_1	BEF_{LIJE_1}	BEF_{LUJE_1}	BEF_{LDJE_1}	BEF_{RIJE_1}	BEF_{RUJE_1}	BEF_{RDJE_1}
	...	...	...	...	...	...	...
	JE_{66}	$BEF_{LIJE_{66}}$	$BEF_{LUJE_{66}}$	$BEF_{LDJE_{66}}$	$BEF_{RIJE_{66}}$	$BEF_{RUJE_{66}}$	$BEF_{RDJE_{66}}$
***BEF* subset with all Java elements (JE_1–JE_{66}) for a branch and edit type**		BEF_{LI}	BEF_{LU}	BEF_{LD}	BEF_{RI}	BEF_{RU}	BEF_{RD}

Table 3: The Evolution History Features (*EHF*s) used by the original design of RPredictor for its predictions.

Evolution History Features (EHF_i)
(EHF_1) Size of Chunk
(EHF_2) Size of Local Version
(EHF_3) Size of Remote Version
(EHF_5) Number of Conflicting Chunks
(EHF_6) Number of Conflicting Files
(EHF_7) Number of Commits before Local
(EHF_8) Number of Commits before Remote
(EHF_9) Date Difference between Local and Remote
(EHF_{10}) Number of Commits by The Owner of Local
(EHF_{11}) Number of Commits by The Owner of Remote
(EHF_{12}) Number of Commits by The Resolver of Conflict

Table 4: The dataset used in our research

	# of Repositories	**# of Conflicts resolved by KL**	**KR**	**ME**	**Total**
Data used in our characterization study (RQ1)	70	1,102	774	1,511	3,387
Data used in the tool evaluation (RQ2–RQ5)	377	5,139	3,823	6,937	15,899
Total	477	6,241	4,597	8,448	19,286

3 RESEARCH METHOD

We study the impact of our new *BEF*s over the effectiveness of RPredictor, using multiple experiments. First, we measure the correlation of these features with the resolution strategies of developers in a large dataset of merge conflicts. Then, we evaluate RPredictor's predictions when using different combinations of feature sets. We study the following research questions:

RQ1: What *BEF*s differ significantly for different strategies?
RQ2: How effective is using *BEF*s vs. *EHF*s?
RQ3: How effective is combining *EHF*s with *BEF*s?
RQ4: How effective is using subsets of *BEF*s?
RQ5: How effective is combining *EHF*s with subsets of *BEF*s?

3.1 Dataset Construction

Ghiotto *et al.* [25] conducted an empirical study on merge conflicts and created a dataset of conflicts from 2,731 GitHub repositories. For the original evaluation of RPredictor, we created our dataset based on Ghiotto *et al.*'s, because of its comprehensiveness and representativeness [8]. Our focus in this paper is only on Java files. Therefore, we refined the dataset that we described in [8] by taking two steps. First, we eliminated all conflicting chunks that were not of the Java type. Second, we removed projects for which their codebases were no longer accessible on GitHub. As shown in Table 4, after refining the prior dataset to align with our new study scope, we obtained 477 software repositories. Among the 19,286 conflicts contained by these repositories, there are 6,241 conflicts separately resolved via KL, 4,597 by KR, and 8,448 by ME.

To perform our study of which features differ significantly for different strategies (RQ1), we randomly selected 100 projects from our dataset. Then, we applied the two filtering criteria mentioned above, which resulted in 70 repositories. This sample set includes 3,387 conflicts, among which 1,102 conflicts were resolved via KL, 774 via KR, and 1,511 via ME.

We studied RQ1 over this sample of projects to avoid overfitting. As in our previous work [8], we used the features that differ significantly (observed in RQ1) as prediction features to use for RPredictor. We studied RQ1 in these randomly selected 70 projects, and we studied RQ2–RQ5 in the remaining 377 projects (see Table 4). That way, the selection of features in RPredictor's predictions was not influenced by the projects in which we evaluated it.

For each one of the merge conflicts in our studied dataset, we measure the numeric value of our studied features (see §2). We measure the value of *EHF*s as in our original evaluation of RPredictor [8], and we measured *BEF*s using Gumtree v3.0.0 [23].

3.2 Method for RQ1

We first study which ones of our newly proposed *BEF*s differ significantly for different conflict resolution strategies, applying statistical analysis. We measured the value of our *BEF*s for each one of 3,387 conflicting chunks, from 70 randomly sampled repositories (§3.1). We separate these conflicts into three groups, according to the resolution strategy that was applied to them (KL, KR, or ME).

To study which of these features differ significantly for different resolution strategies, we applied the Kruskal-Wallis H test [3, 41, 44] as we did to originally design RPredictor [8]. The Kruskal-Wallis H test assesses if three or more groups of samples come from the same distribution on a variable of interest. Our studied features do not follow a normal distribution, and the Kruskal-Wallis H test is non-parametric (*i.e.,* it does not assume a normal distribution of the data). For each group of samples, the H test sorts data into ascending order, assigns ranks to the sorted data points, and thus converts the given values into their ranks. Namely, in the conversion process, the smallest value gets a rank of 1, the next smallest gets a rank of 2, and so on. Among the given three or more sample groups, the H test validates the following hypotheses:

- H_0: The mean ranks of different groups are the same.
- H_1: The mean ranks of different groups are not the same.

3.3 Method for RQ2–RQ5

To study RQ2–RQ5, we run multiple variants of RPredictor to obtain a prediction for each one of the merge conflicts in our studied dataset. We study these variants in a within-project and cross-project usage context. We evaluate them with various metrics.

Training and Testing Process. We train and test our evaluated variants of RPredictor in two different ways, to study the within-project and the cross-project usage context. For variants that we report as using *BEF*s or one of their 6 subsets (or a combination including them), we in fact only use (or combine) for training their specific Java elements that we observed in RQ1 to differ with statistical significance (reported in Table 5).

Within-Project Prediction. For each software project in our dataset, we leveraged 90% of the oldest resolved conflicts for training, and then used the remaining 10% of resolved conflicts for testing. We intentionally used older data for training and newer data for testing. This is because such a setting can mimic real-world scenarios, where a technique can only refer to a project's history data to suggest resolutions for future conflicts of that project.

Cross-Project Prediction. In this experiment, we evaluated the real-world scenarios where a given project has little version history to leverage. In such scenarios, RPredictor can train a classifier with the conflict data from other repositories and use that classifier to predict resolutions for the given project. We conducted 10-fold cross validation to evaluate the effectiveness of each variant of RPredictor. Namely, we divided the 377 software projects randomly into 10 groups roughly evenly. For each group $G_i (i \in [1, 10])$, we ran an experiment by using the conflict data in the remaining nine groups for training, and adopting the data in G_i for testing. We calculated the effectiveness for each of the 10 runs, and then also aggregated it among all runs.

Ground Truth. As ground truth, we used the resolution strategy that the developer applied to resolve each conflict in our dataset.

Evaluation Metrics. In this paper, we will focus on measuring the effectiveness of a technique by its F-score. However, for completion, we will also measure the metrics used in the original evaluation of RPredictor [8]: Precision, Recall, and C-score. To facilitate discussion, in this section, we index the three conflict resolution strategies and refer to them as $S_i (i \in [1, 3])$. Namely, S_1 refers to KL (keep the local version); S_2 refers to KR (keep the remote version); S_3 refers to ME (resolution with manual edits). We measured all metrics on a scale ranging from [0%, 100%], and higher values indicate better performance.

Precision (P_i) measures, among all the conflicts labeled with S_i by a technique, what ratio of them were actually resolved by S_i.

$$P_i = \frac{\text{\# of conflicts correctly labeled as "}S_i\text{"}}{\text{Total \# of conflicts labeled as "}S_i\text{"}} \quad (1)$$

Recall (R_i) measures, among all conflicts that were resolved by S_i, what ratio of them were labeled by a technique as S_i.

$$R_i = \frac{\text{\# of conflicts correctly labeled as "}S_i\text{"}}{\text{Total \# of conflicts that were resolved via } S_i} \quad (2)$$

F-score (F_i) is the harmonic mean of precision and recall. It allows us to measure technique effectiveness in a single metric.

$$F_i = \frac{2 \times P \times R}{P + R} \quad (3)$$

Aggregated (Overall) metrics (P, R, F). We also measured our metrics by computing the *weighted* average across all strategies. Specifically, if we denote Γ as either precision (P) or recall (R) and use n_i to represent the number of testing samples in S_i, then the overall effectiveness in terms of precision and recall can be computed as follows:

$$\Gamma_{overall} = \frac{\sum_{i=1}^{3} \Gamma_i * n_i}{\sum_{i=1}^{3} n_i} \quad (4)$$

Thus, the overall F is computed with:

$$F_{overall} = \frac{2 \times P_{overall} \times R_{overall}}{P_{overall} + R_{overall}} \quad (5)$$

Conservativeness Score (C) or C-score. Different prediction mistakes have different consequences. If a conflict resolved by KL or KR is incorrectly predicted as ME, the technique makes a *conservative* mistake: it misses the opportunity of saving developers' manual effort, but does not mislead developers to blindly take resolution suggestions. However, if a conflict resolved by ME is incorrectly predicted as KL or KR, the technique makes a more serious mistake: it automatically resolves the conflict using a different strategy than what the developer would have preferred, and thus produces an incorrectly merged version. We created a C metric to measure the ratio of predictions that are *conservative*, *i.e.,* that do not cause any incorrect automatic resolution. Conservative predictions include (1) correct predictions, and (2) any conflict resolved via KL or KR but labeled as ME.

$$C = \frac{\text{\# of conflicts conservatively labeled}}{\text{All predictions}} \quad (6)$$

Table 5: RQ1. The Java Elements in each *BEF* subset that showed statistically significant differences for different merge conflict resolution strategies (Kruskal-Wallis H test). We use these for prediction.

Branch	**Local (*L*)**			**Remote (*R*)**		
Edit type	**Insertions (*I*)**	**Updates (*U*)**	**Deletions (*D*)**	**Insertions (*I*)**	**Updates (*U*)**	**Deletions (*D*)**
***BEF* subset**	BEF_{LI}	BEF_{LU}	BEF_{LD}	BEF_{RI}	BEF_{RU}	BEF_{RD}
Statistically Significant Java Elements (JE_i)	2, 5, 6, 7, 10, 12, 13, 14, 17, 36, 57	1, 6, 12, 17, 20, 48, 58, 59	1, 2, 4, 5, 7, 9, 10, 13, 18, 19, 20, 22, 23, 24, 25, 27, 28, 29, 30, 31, 34, 37, 38, 39, 40, 41, 43, 44, 45, 49, 51, 52, 53, 54, 55, 57, 64	5, 6, 7, 10, 13, 14, 17, 26, 32, 36, 37, 47, 48, 58, 59, 64, 66	1, 6, 12, 17, 20, 26	1, 2, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 18, 19, 20, 21, 22, 23, 24, 25, 27, 28, 29, 30, 33, 34, 36, 37, 38, 39, 40, 42, 43, 44, 48, 49, 52, 53, 55, 58, 59, 62, 64

Table 6: RQ2. Within-project effectiveness of *BEF*s vs. *EHF*s.

	***BEF*s only**				***EHF*s only**			
	P	**R**	**F**	**C**	**P**	**R**	**F**	**C**
KL	37%	27%	31%	-	49%	40%	44%	-
KR	34%	29%	31%	-	37%	34%	35%	-
ME	44%	57%	49%	-	53%	63%	58%	-
Overall	**39%**	**40%**	**39%**	**71%**	**47%**	**48%**	**47%**	**72%**

Table 7: RQ2. Cross-project effectiveness of *BEF*s vs. *EHF*s.

	***BEF*s only**				***EHF*s only**			
Test Fold #	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**
1	40%	43%	41%	90%	40%	43%	41%	80%
2	38%	42%	40%	90%	39%	41%	40%	78%
3	34%	43%	38%	90%	47%	49%	48%	81%
4	53%	58%	55%	92%	63%	65%	64%	84%
5	43%	44%	43%	92%	39%	40%	39%	81%
6	42%	50%	46%	89%	42%	44%	43%	72%
7	35%	39%	36%	81%	34%	38%	36%	72%
8	33%	31%	32%	93%	41%	35%	38%	87%
9	40%	49%	44%	87%	53%	57%	55%	86%
10	42%	46%	44%	89%	46%	48%	47%	77%
All Folds	**39%**	**43%**	**41%**	**90%**	**43%**	**44%**	**44%**	**80%**

4 RESULTS

RQ1: What *BEF*s Differ Significantly for Different Strategies? We applied the Kruskal-Wallis H test to all our 396 newly proposed *BEF*s (see §2) for all our 3,387 studied conflicting chunks (see §3.1). We found that 122 of them showed values with a statistically significant difference (p-value lower than 0.05) for chunks that were resolved with different strategies (KL, KR, or ME). This indicates that these *BEF*s may be useful to predict developers' resolution strategies. We report them in Table 5.

Most of the *BEF*s with a statistically significant difference belong to the BEF_{LD} and BEF_{RD} subsets. This means that there were many Java elements that, when deleted on either branch, could be predictive of the final resolution strategy. Similarly, there were relatively few Java elements that, when updated on either branch, they could be predictive of the final resolution strategy — few Java elements with a statistically significant difference belong to the BEF_{LU} and BEF_{RU} subsets.

For the remaining experiments (RQ2–RQ5), for variants that we report as using *BEF*s or one of their 6 subsets (or a combination including them), we in fact only use (or combine) for training their specific Java elements that we observed in RQ1 to differ with statistical significance (reported in Table 5).

Finding 1: *122 of our 396 proposed BEFs differ with statistical significance for different conflict resolution strategies.*

RQ2: Effectiveness of *BEF*s vs. *EHF*s. First, to understand the effectiveness that RPredictor would provide when using *BEF*s vs. *EHF*s, we created two variants of RPredictor: one that uses only *BEF*s to make its predictions, and one that uses only *EHF*s (this latter one is the original design of RPredictor [8]). We separately trained and tested both variants for the within-project and cross-project usage contexts (see §3.3) over 15,899 conflicts from 377 projects (see §3.1). We report the results obtained by both technique variants for all our studied metrics (see §3.3) in Table 6 for within-project prediction and in Table 7 for cross-project prediction. We should note that *EHF*s obtained slightly different results than in its original evaluation [8], since we now use a slightly different dataset (it only contains Java projects).

Effectiveness of Within-project Prediction. When trained over the same software project (within-project context), RPredictor provides lower effectiveness when using *BEF*s than when using its original *EHF* features, for all our studied metrics (Table 6). This means that our proposed *BEF*s did not help RPredictor improve its effectiveness. Even though the research literature suggests that the strategy used to resolve merge conflicts could be influenced by the specific branch edits (*BEF*) that happened on each branch, *e.g.,* [25, 61, 65], our results show that they did not help the predictions of our machine-learning predictor RPredictor as much as its original features did (*EHF* features).

Finding 2: *In the within-project usage context, RPredictor provided lower effectiveness when using our new BEFs than when using its original EHFs.*

Effectiveness of Cross-project Prediction. When trained over different software projects (cross-project context), RPredictor also provided lower effectiveness when using *BEF*s than when using its original *EHF* features, in terms of precision, recall, and f-score, aggregated for all test folds (Table 7). This trend was also clear in most individual test folds. The counterbalance to these results is that RPredictor using *BEF*s provided higher Conservativeness (C) Score (90% vs. 80%) than when using *EHF*s, for all our evaluated (and aggregated) test folds.

The higher C score provided by RPredictor using *BEF*s can be explained by it being more effective at predicting the ME resolution strategy. We observed this by checking its results in more detail. However, due to the space limit, we do not report the individual effectiveness measurements for each resolution strategy. We observed that RPredictor using *BEF*s predicted ME more often, which meant that it less often predicted KL and KR incorrectly

Table 8: RQ3. Within-project effectiveness of *BEF*s and *EHF*s combined vs. *EHF*s only.

	*BEF*s and *EHF*s				*EHF*s only			
	P	**R**	**F**	**C**	**P**	**R**	**F**	**C**
KL	47%	38%	42%	-	49%	40%	44%	-
KR	38%	35%	36%	-	37%	34%	35%	-
ME	53%	64%	58%	-	53%	63%	58%	-
Overall	**47%**	**48%**	**47%**	**72%**	**47%**	**48%**	**47%**	**72%**

Table 9: RQ3. Cross-project effectiveness of *BEF*s and *EHF*s combined vs. *EHF*s only.

	*BEF*s and *EHF*s				*EHF*s only			
Test Fold #	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**
1	42%	45%	43%	72%	40%	43%	41%	80%
2	36%	41%	39%	81%	39%	41%	40%	78%
3	52%	50%	51%	88%	47%	49%	48%	81%
4	67%	68%	68%	93%	63%	65%	64%	84%
5	42%	43%	43%	88%	39%	40%	39%	81%
6	40%	46%	43%	78%	42%	44%	43%	72%
7	32%	37%	34%	81%	34%	38%	36%	72%
8	41%	35%	38%	90%	41%	35%	38%	87%
9	52%	56%	54%	84%	53%	57%	55%	86%
10	45%	50%	48%	85%	46%	48%	47%	77%
All Folds	**44%**	**45%**	**44%**	**82%**	**43%**	**44%**	**44%**	**80%**

(increasing its C score), but also less often predicting them correctly (decreasing all other metrics). This shows that, overall, the *BEF*s also did not improve the effectiveness of RPredictor for the cross-project usage context.

Finding 3: *In the cross-project usage context, RPredictor also provided lower effectiveness (Precision, Recall, and F-score) when using BEFs for its prediction than when using its original EHFs.*

RQ3: Effectiveness of *BEF*s and *EHF*s vs. *EHF*s only. In RQ2, we observed that RPredictor did not provide higher effectiveness when using *BEF*s than when using its original set of *EHF* features. Next, we study whether *BEF*s and *EHF*s can help each other, *i.e.,* whether RPredictor can provide higher effectiveness by using combining both. Thus, we evaluated a new variant of RPredictor that uses the union of *BEF*s and *EHF*s for its predictions. We report the results that we obtained for the within-project and cross-project usage contexts in Tables 8 and 9, respectively. We also provide the results of the original design of RPredictor (using *EHF*s only) in these tables, for ease of comparison.

Effectiveness of Within-project Prediction. Table 8 shows that combining *BEF*s and *EHF*s in the feature set of RPredictor provided very similar effectiveness than when RPredictor used only its original *EHF*s, for the within-project usage context. The variant of RPredictor that used both *BEF*s and *EHF*s was slightly better at predicting the KR resolution (and higher precision and recall) and slightly worse at predicting KL. However, overall, the precision, recall, and f-score provided by using *BEF*s and *EHF*s was about the same as when using only *EHF*s.

This result is consistent with what we observed in RQ2, since *BEF*s by themselves made RPredictor provide lower effectiveness. However, RQ3 shows us that, when combining *BEF*s and *EHF*s, RPredictor was able to not allow the worse prediction power of *BEF*s hurt the better prediction power of *EHF*s, *i.e.,* combining *BEF*s and *EHF*s did not produce worse results than using *EHF*s only. It is also possible that combining *BEF*s and *EHF*s could not improve the effectiveness of RPredictor because the within-project usage context provides a limited amount of data for training (it only uses the same project's historical data). Therefore, it is also worth investigating the cross-project usage context.

Table 10: RQ4. Within-project effectiveness of *BEF* Subsets.

Feature Set	**P**	**R**	**F**	**C**
*EHF*s only	47%	48%	47%	72%
*BEF*s only	39%	40%	39%	71%
*EHF*s and *BEF*s	47%	48%	47%	72%
BEF_{LI} only	39%	40%	39%	72%
BEF_{LU} only	37%	39%	37%	72%
BEF_{LD} only	37%	39%	37%	72%
BEF_{RI} only	37%	38%	37%	69%
BEF_{RU} only	38%	40%	38%	72%
BEF_{RD} only	38%	40%	38%	72%

Table 11: RQ4. Cross-project effectiveness of *BEF* Subsets.

Feature Set	**P**	**R**	**F**	**C**
*EHF*s only	43%	44%	44%	80%
*BEF*s only	39%	43%	41%	90%
*EHF*s and *BEF*s	44%	45%	44%	82%
BEF_{LI} only	38%	43%	40%	93%
BEF_{LU} only	31%	44%	36%	99%
BEF_{LD} only	38%	43%	40%	98%
BEF_{RI} only	38%	43%	41%	93%
BEF_{RU} only	21%	44%	29%	99%
BEF_{RD} only	39%	44%	41%	98%

Finding 4: *For the within-project usage context, combining BEFs and EHFs provided about the same effectiveness as when RPredictor used EHFs only.*

Effectiveness of Cross-project Prediction. Table 9 shows that when RPredictor combined *BEF*s and *EHF*s in the cross-project usage context, it again provided about as high effectiveness as when it used *EHF*s only. We can only observe a small difference: overall, combining *BEF*s and *EHF*s provided 1% higher precision, 1% higher recall, and 2% higher C-score, but the same F-score.

This time, combining *BEF*s and *EHF*s provided a small effectiveness improvement to RPredictor in the cross-project usage context. In very few cases, the RPredictor variant that predicts based on both *BEF*s and *EHF*s predicted the KL and KR resolutions correctly when the original variant based on *EHF*s only did not. However, the overall f-score of the predictions stayed the same for both technique variants, keeping them equally effective. We conclude that the technique variant that combines *BEF*s and *EHF*s also did not provide an important increase in effectiveness.

Finding 5: *For the cross-project usage context, combining BEFs and EHFs provided 1pp. higher precision, 1pp. higher recall, and 2pp. higher C-score, but the same F-score.*

RQ4: Effectiveness of the 6 Separate *BEF* Subsets. In RQ1, we noticed that the majority of *BEF*s that showed statistically significant differences for different conflict resolution strategies belonged to the BEF_{LD} and BEF_{RD} subsets, *i.e.,* they captured the Deletion (D) Edit Type, in the Local (L) and Remote (R) branches. This observation motivated us to explore the separate influence of each *BEF* subset into the effectiveness of RPredictor.

We created 6 separate variants of RPredictor, each one using only one of the *BEF* subsets for its predictions, namely: BEF_{LI}, BEF_{LU}, BEF_{LD}, BEF_{RI}, BEF_{RU}, and BEF_{RD}. We described these *BEF* subsets in §2. We evaluated each variant in both the within-project and cross-project usage contexts, as we did in previous research questions. We report the results that we obtained for the within-project and cross-project usage contexts in Tables 10 and 11, respectively. For ease of comparison, we also include in these tables the overall results of the other three variants of RPredictor that we studied in previous research questions, *i.e.,* that used: *EHF*s only, *BEF*s only, or *EHF*s and *BEF*s combined.

Effectiveness of Within-project Prediction. We can observe in Table 10 that, in the within-project usage context, the variant of RPredictor that used only BEF_{LI} for prediction was the one that produced the highest effectiveness, for all metrics, among all the variants that we evaluated in this research question. In fact, BEF_{LI} slightly improved over *BEF*s in terms of C-score (72% vs. 71%, respectively), while also maintaining the same effectiveness (in terms of precision, recall, and f-score). Still, when compared with RPredictor's original design of *EHF*s-only, all variants provided lower effectiveness, *i.e.,* an F-score lower than 47%.

Finding 6: *For within-project prediction, all of our studied BEF subsets provided lower effectiveness than the original design of RPredictor that used EHFs only.*

Effectiveness of Cross-project Prediction. In the cross-project usage context, the RPredictor variant that performed best was BEF_{RD}. Both BEF_{RD} and BEF_{RI} provided the highest F-score among variants, but BEF_{RD} provided a higher C-score than BEF_{RI}.

In this case, we found it interesting that the BEF_{RD} variant provided better scores than the variant using all *BEF*s. They both provided the same F-score, but using BEF_{RD} only was highly conservative (its C-score reached 98%). This means that, when BEF_{RD} mispredicted truly-KL or truly-KR conflicts, it was more inclined to mispredict them as ME. This high conservativeness (98% C-score) of BEF_{RD} may be a welcome characteristic for developers. However, it may not be enough for them to prefer BEF_{RD} over the higher-effectiveness *EHF*s variant, since *EHF*s provides higher effectiveness (44% vs. 41% F-score).

In this study we focus on the effectiveness of our studied RPredictor variants, *i.e.,* in their precision, recall, and f-score metrics. Therefore, we conclude that, in cross-project prediction, none of the studied variants was able to provide higher effectiveness than the original design of RPredictor using *EHF*s only (they all provided f-score lower than 44%).

Finding 7: *For cross-project prediction, all of our studied BEF subsets provided lower effectiveness than the original design of RPredictor that used EHFs only.*

RQ5: Effectiveness of *EHF*s and each Separate *BEF* Subset. In RQ4, we found that, in some cases, predicting with only one of the *BEF* subsets improved the results of predicting with all *BEF*s (although only in terms of C-score). For within-project prediction BEF_{LI} provided higher 1% C-score, and for cross-project prediction, BEF_{RD} provided higher 8% C-score. In RQ2, we found that combining *EHF*s with *BEF*s provided higher effectiveness than using *BEF*s only (even though not enough to improve over *EHF*s only). These two findings motivated us to study the effectiveness of combining *EHF*s with separate *BEF* subsets.

Table 12: RQ5. Within-project effectiveness of *EHF*s and *BEF* Subsets combined.

Feature Set	P	R	F	C
*EHF*s only	47%	48%	47%	72%
*BEF*s only	39%	40%	39%	71%
*EHF*s and *BEF*s	47%	48%	47%	72%
*EHF*s and BEF_{LI}	50%	51%	50%	73%
*EHF*s and BEF_{LU}	48%	49%	48%	72%
*EHF*s and BEF_{LD}	46%	47%	46%	71%
*EHF*s and BEF_{RI}	49%	50%	50%	73%
*EHF*s and BEF_{RU}	48%	48%	48%	72%
*EHF*s and BEF_{RD}	49%	50%	49%	73%

Table 13: RQ5. Cross-project effectiveness of *EHF*s and *BEF* Subsets combined.

Feature Set	P	R	F	C
*EHF*s only	43%	44%	44%	80%
*BEF*s only	39%	43%	41%	90%
*EHF*s and *BEF*s	44%	45%	44%	82%
*EHF*s and BEF_{LI}	44%	45%	45%	82%
*EHF*s and BEF_{LU}	42%	44%	43%	81%
*EHF*s and BEF_{LD}	44%	46%	45%	83%
*EHF*s and BEF_{RI}	43%	44%	44%	81%
*EHF*s and BEF_{RU}	43%	45%	44%	80%
*EHF*s and BEF_{RD}	43%	45%	44%	82%

We now evaluated 6 new variants of RPredictor, each one using for prediction: the *EHF*s and one of the 6 *BEF* subsets. We used the same settings as in previous experiments, for within-project and cross-project prediction. We report the results of this evaluation in Tables 12 and 13 for within-project and cross-project, respectively. As in previous RQs, we also include the results of *EHF*s only, *BEF*s only, and *EHF*s and *BEF*s combined for comparison.

Effectiveness of Within-project Prediction. In this research question, the variant that provided the highest effectiveness was the combination of *EHF*s and BEF_{LI} (Table 12). In fact, *EHF*s and BEF_{LI} provided higher effectiveness than RPredictor's original design that uses *EHF*s only: 50% vs. 47% F-score, respectively. It also provided higher C-score: 73% vs. 72%.

We had observed in RQ2 and RQ3 that our proposed *BEF*s did not improve the effectiveness of RPredictor in its original design (using *EHF*s only) neither by themselves nor in combination with *EHF*s. We wondered if, by using so many features (122 *BEF*s), those that were less-useful for prediction were not letting RPredictor take the full advantage of the most-useful ones. After observing the results of RQ5, that seems possible. Combining *EHF*s only with BEF_{LI} was more effective than combining *EHF*s with all *BEF*s.

Finally, it is worth noting that, while combining *EHF*s and BEF_{LI} improved the effectiveness of *EHF*s, such improvement was relatively minor, and thus the practical applicability of this variant may be up for discussion.

Finding 8: *For within-project prediction, the combination of EHFs and BEF_{LI} provided a minor improvement over RPredictor's original design using EHFs only. EHFs and BEF_{LI} provided higher: precision by 3pp., recall by 3pp., f-score by 3pp., and C-score by 1pp.*

Effectiveness of Cross-project Prediction. We can observe in Table 13 that, in the cross-project setting, the variant that produced highest effectiveness was *EHF*s and BEF_{LD}. This variant provided higher results for all metrics when compared with RPredictor's original *EHF*s only: 44% vs. 43% precision, 46% vs. 44% recall, 45% vs. 44% f-score, and 83% vs. 80% C-score.

These results again show us that combining *EHF*s with individual *BEF* subsets can be more beneficial than combining them with all *BEF*s. *EHF*s and BEF_{LD} provided better results than *EHF*s and all *BEF*s. However, we also again observe that the best-performing variant of this experiment only provided minor improvements over the original *EHF*s. *EHF*s and BEF_{LD} improved *EHF*s only by: 1pp. in precision, 2pp. in precision, 1pp. in f-score, and 3pp. in C-score. Furthermore, more generally, all the technique variants that we studied in the cross-project usage context provided lower effectiveness than the original *EHF*s did in the within-project setting, *i.e.,* they all provided f-scores that were lower than 47%.

Finding 9: *For cross-project prediction, the combination of EHFs and BEF_{LD} provided a minor improvement over RPredictor's original design using EHFs only. EHFs and BEF_{LD} provided higher: precision by 1pp., recall by 2pp., f-score by 1pp., and C-score by 3pp.*

5 DISCUSSION

The best feature set provided only an incremental improvement. After multiple experiments, we found that the set of features that provided the highest effectiveness for RPredictor was the combination of *EHF*s and BEF_{LI} in the within-project prediction context. This shows that RPredictor can be more effective when, in addition to its original *EHF*s, it also considers in its predictions the number of Java elements that were inserted in the local branch, for a some Java elements.

However, unfortunately, the improvement in effectiveness that the new RPredictor *EHF*s and BEF_{LI} variant provided over RPredictor's original design (using *EHF*s only) was relatively minor: only 3 pp. in precision, recall, and f-score. Such an incremental improvement may make it hard for developers to justify the usage of this new variant. Capturing the additional fine-grained features of BEF_{LI} imposes the usage of additional parsing software (Gumtree [23]). Writing such code for using Gumtree, and maintaining it over time for compatibility with future releases comes with its own cost. Thus, we expect that developers may be wary of increasing the complexity (and maintenance cost) of their software developement pipeline in exchange for the small effectiveness increment that the *EHF*s and BEF_{LI} variant provides.

Past studies suggested that *BEF*s could be useful predictors. We originally set out to study whether *BEF*s could improve the effectiveness of RPredictor motivated by previous studies that observed that the specific changes performed the branches of a merge conflict could influence developer decisions to resolve it. For example, Shen *et al.* [61] found that when both branches of a merge conflict contained many *inserted* elements, developers tended to resolve it by keeping the content of both branches. However, they also observed that when one branch deleted code and the other one updated it, developers tended to exclusively keep edits from one branch. Similarly, Ghiotto *et al.* [25] found that developers tended to resolve conflicts via either combining edits from both branches or introducing new code when conflicts consisted of comments, if statements, or method invocations. They also observed that developers tended to combine edits from both branches, or choose KL, or KR when conflicts consisted solely of variables or imports. Another study by Vale *et al.* [65] found that conflict resolution time has a correlation with the complexity of the code in conflict.

Lesson learned: *EHF*s and *BEF*s are both useful predictors, but they only lightly complement each other. We observed in RQ2 that our motivation for this work was not entirely misguided. In fact, RPredictor provided very similar levels of effectiveness when using *BEF*s than it did originally by using *EHF*s. The results obtained by both variants differed in less than 10 pp. for all metrics.

Unfortunately, the predictive power of *BEF*s do not seem to complement well the one provided by *EHF*s. We observed in RQ3 that combining both *EHF*s and *BEF*s still provided about the same effectiveness as using *EHF*s only. This may mean that *BEF*s were helpful in predicting the same kinds of conflicts for which *EHF*s were already good predictors, but they were not as helpful in predicting the ones that *EHF*s mispredicted. Therefore, future efforts to improve the effectiveness of RPredictor would have to use a different strategy.

Lesson learned: Future work may study developer preferences of the trade-off between C-score and F-score. Finally, we also discovered in this study an unexpected potentially useful future direction: increasing C-score at the expense of F-score. We discovered that some of the variants that we studied provided much higher C-score as a trade-off for their lower f-score. The most clear example of this were the results of the BEF_{RD} variant for cross-project prediction. It provided 41% f-score, which is 3pp. lower effectiveness than *EHF*s, but it also provided 98% C-score, which is 18 pp. higher than *EHF*s. This means that this technique variant was less effective than *EHF*s at recommending resolution decisions for developers. However, it also means that it made *safer* mistakes, *i.e.,* it was much more inclined to wrongly recommend ME (as opposed to wrongly recommending KL or KR).

In this study, our focus was on the prediction effectiveness provided by RPredictor when using different feature sets. However, the discovery of the BEF_{RD} variant makes us wonder if some developers would prefer sacrificing (some amount of) f-score for a higher (some amount of) C-score. We believe that it would be valuable to run a future human study to understand developers opinions about different trade-offs of effectiveness (f-score) and recommendation safety (C-score) they prefer, and under what circumstances they do so. The results of such study could inspire future technique designs.

6 RELATED WORK

Empirical Studies on Merge Conflicts. Several studies have been conducted to investigate the relationship between merge conflicts and various aspects of software maintenance [5, 22, 40, 43, 50]. For instance, *et al.* [22] conducted a survey involving 105 student developers, discovering that the lack of awareness, specifically the knowledge of "who's changing what," occurs more frequently than merge conflicts. Leßenich *et al.* [40] conducted a survey involving

41 developers and pinpointed seven potential indicators, including metrics like the number of changed files in both branches, which could be used to anticipate the occurrence of merge conflicts. Following a further investigation of the indicators, the researchers discovered that none of them had the capability to predict the frequency of merge conflicts. Likewise, Owhadi-Kareshk *et al.* defined nine features, including metrics such as the number of added and deleted lines in a branch, to describe merging scenarios. They developed a machine-learning model that could predict conflicts with an accuracy ranging from 57% to 68% [50].

Like these previous studies, our study also characterizes merge conflicts. Nonetheless, it distinguishes itself in two key ways. First, our study explores the various features that characterize how developers approach conflict resolution. Second, Our study is driven by the goal of automating the prediction of resolution strategies, whereas previous studies have primarily focused on automating the prediction of conflict occurrences.

Other studies also characterize the underlying reasons and/or solutions for *textual* conflicts [12, 25, 48, 51, 66]. In particular, Yuzuki *et al.* inspected hundreds of textual conflicts [66]. Their observations revealed that 44% of conflicts resulted from conflicting updates to the same line of code, and developers resolved 99% of these conflicts by choosing either the left- or right version of the code. Brindescu *et al.* [12] conducted a manual inspection of 606 textual conflicts. They categorized merge conflicts based on the differences in the Abstract Syntax Tree (AST), the size of lines of code (LOC), and the number of authors involved. They identified three distinct resolution strategies: "SELECT ONE" (retaining edits from one branch), "INTERLEAVE" (retaining edits from both branches), and "ADAPTED" (modifying existing edits and/or introducing new edits). Pan et al. [51] delved into the analysis of merge conflicts within the context of Microsoft Edge. They categorized these conflicts according to file types, conflict locations, conflict sizes, and patterns of conflict resolution. Building on the insights from their empirical research, the researchers explored the application of program synthesis techniques for conflict resolution. The initial prototype of their resolution tool only attempts to combine edits from both branch versions and can't suggest "KL" (left-hand version) or "KR" (right-hand version) resolutions.

Although these studies inspired us to define and investigate potential features that can be valuable in predicting developers' strategies for conflict resolution, none of these prior studies conducted statistical analyses to investigate the relationships between these identified features and developers' resolution strategies.

Awareness-Raising Tools. Tools [11, 13, 14, 26, 35, 38, 42, 53, 60] were developed to keep track of and assess programmers' progress in their work, aiming to enhance team collaboration. For example, CASI [60] and Palantír [53] notify a developer about the modifications made by their colleagues, determine the significance of these modifications, and present this information visually. Cassandra [35] serves as a method to reduce conflicts in software development. It examines the relationships of super-sub and caller-callee dependencies among program components. By considering these dependencies as limitations on tasks involving editing files, Cassandra detects tasks that could clash when executed concurrently. Subsequently, it schedules tasks in a way that suggests development paths that avoid conflicts. Crystal [13, 14] and WeCode [26] take a proactive approach to identify collaboration conflicts using speculative analysis. They proactively merge program modifications made in various software branches before these changes are fully integrated into the main repository within the distributed version control system (DVCS). These tools employ a sequence of textual merging, automatic building, and automatic testing to uncover potential conflicts between branches.

The previously mentioned tools can proactively detect and notify developers about merge conflicts. However, they do not characterize developers' resolution preferences, nor do they automatically recommend any resolution strategy.

Automated Software Merge. Various tools have been introduced to either detect or resolve merge conflicts [2, 9, 10, 15, 19, 39, 45, 49, 62–64, 67, 68]. Mens *et al.* [45] conducted a survey on techniques for merging software. FSTMerge [2, 10, 15] analyzes code to create Abstract Syntax Trees (ASTs) and matches nodes between versions L and R using only class or method signatures as criteria. It subsequently incorporates the modifications within each pair of matched method nodes through a textual merging process. IntelliMerge [62] enhances the effectiveness of FSTMerge by identifying and resolving conflicts related to code refactoring. Much like FSTMerge , JDime [9, 39] also employs the matching of Java methods and classes based on syntax trees. However, JDime differs in combining changes within matched methods; it accomplishes this by matching and manipulating ASTs. AutoMerge [68] builds upon JDime's approach. When branch edits cannot be merged, AutoMerge endeavors to resolve conflicts by suggesting different strategies for merging versions L and R. SafeMerge [63] verifies whether a merging scenario has introduced new semantics to the codebase. RPredictor is complementary to these techniques by modeling and predicting developers' preferences for resolutions.

MergeHelper [49] records the chronological sequence of edit operations programmers perform using the Eclipse Java editor. When faced with conflicting branch versions, L and R, MergeHelper examines the recorded edit sequences preceding both versions. Its goal is to identify the most recent snapshot in the version history that aligns with both L and R. In simpler terms, MergeHelper rolls back the edits made by both branches until it reaches an intermediate version just before the initial conflict arises. While it offers detailed edit information to help developers understand conflicts, it does not provide resolution strategies as RPredictor does.

DeepMerge [19], MergeBERT [64], and GMerge [67] employ deep-learning techniques to resolve conflicts automatically, but DeepMerge is designed explicitly for conflicts involving fewer than 30 lines of code [64]. It may not be suitable for more complex conflicts. When dealing with textual conflicts, both DeepMerge and MergeBERT are designed to integrate partial edits from versions L and R for resolution rather than proposing entirely new solutions represented as KL or KR. GMerge addresses a distinct type of merge conflict, which involves edits that can be applied concurrently to the merged version but result in semantic errors when combined. In contrast, RPredictor complements the learning-based methods discussed earlier and offers additional capabilities or solutions beyond those methods. That is, RPredictor is capable of predicting

conflicts that can be resolved using either KL or KR, which, according to existing literature, are the most common resolution strategies. Additionally, when RPredictor anticipates a Merge Error (ME), it can be complemented with an alternative approach such as DeepMerge or MergeBERT to facilitate an automated resolution by merging lines from conflicting versions.

MESTRE [21] has been introduced recently as a recommender that predicts the merge resolution strategy. MESTRE predicts the resolution strategy considering options such as using version 1, and version 2, combining version 1 and 2 in different ways, merging lines from both versions, or creating new code manually. While both MESTRE and RPredictor aim to assist with merge conflict resolution, they differ in their approach and the granularity of their predictions. MESTRE focuses on high-level merge strategies, whereas RPredictor provides fine-grained predictions for conflicting chunks, by considering a larger number of code elements and the nature of edits, considering developer behavior and history.

7 THREATS TO VALIDITY

Construct Validity. Threats to construct validity refer to the confidence in our measurements and conceptual framework.

In our study, the accuracy and completeness of our measurements of changes to fine-grained code elements heavily depends on the performance of the Gumtree tool [23]. Any limitations or inaccuracies in Gumtree's parsing algorithms could introduce errors into our measurements. To mitigate this threat, we conducted validation checks through the manual analysis of a subset of our studied code elements. First, we conducted a meticulous manual analysis for each sampled change hunk, and manually determined the fine-grained change operations and changed elements that happened between the two code versions. Then, we compared our assessment with the results produced by Gumtree. We found no contradictions between our manually extracted differences and the results generated by Gumtree. So, while inaccuracies in Gumtree's analysis algorithm are possible, they did not seem to be common, as per our validation process.

Another threat to construct validity is the fact that developers may not accept the additional computational cost of computing ASTs that *BEF*s require, particularly since they provide little improvement in RPredictor's predictions. To mitigate this threat, RPredictor could perform its AST analysis in the background and cache its results to reuse them at prediction time. Still, it would be useful to better understand developer preferences and limitations in the automatic resolution of merge conflicts through an interview-based developer study. We plan to perform such a study in the future (the cost to humans of getting automated recommendations can be high *e.g.,* [29]).

External Validity. Threats to external validity refer to the generalizability of the observations of our study.

While we didn't observe a strong improvement in the prediction results of RPredictor with our studied *BEF*s, it is still possible that *BEF*s defined for other code elements, or for other programming languages may provide different results. It is also possible that our studied *BEFs* (or other *BEF*s) show different impact when studied over other different software projects. This might impact the generalizability of our findings. However, we believe that our study provides reasonable generalizability, since it included a large number of studied samples and features. We evaluated *EHF*s and *BEF*s over 15,899 code conflicts within 377 software projects. Also, our *BEF*s captured 396 combinations of 3 types of changes applicable to 66 fine-grained Java code elements over 2 code branches. Still, in the future, we plan to conduct larger-scale experiments with more projects and features for more types of code elements and changes.

Another threat to the generalizability of our approach is that *BEF*s only cover the Java programming language. Therefore, our observations may or may not be applicable to other programming languages. However, most of our studied *BEF*s capture Java elements that also exist in many popular programming languages (see Table 1). So, while our study only covered Java, we believe that it could be easily replicated to cover other languages.

8 CONCLUSION

Resolving merge conflicts is a tedious and error-prone process in software development. Although many tools were proposed to detect and even resolve merge conflicts, little tool support is available to automatically resolve conflicts by observing and mimicking developers' resolution strategies. In past work, we proposed RPredictor, which recommends developers strategies to resolve merge conflicts, based on a set of Evolution History Features (*EHF*s) [8].

In this paper, we proposed a new set of Branch Edit Features (*BEF*s) to try to improve the effectiveness of RPredictor, inspired by observations in the literature of the specific fine-grained edits of the conflict influencing developer decisions to resolve it.

We performed an extensive evaluation of Branch Edit Features (*BEF*s), studying various scenarios in which they could improve effectiveness: by themselves, combining them with *EHF*s, using only subsets of them, and combining *EHF*s with subsets of *BEF*s. Overall, we found that only one of our studied set of features improved the effectiveness of RPredictor, but did so very slightly: the *EHF*s and BEF_{LI} set improved over *EHF*s by 3pp. f-score. We believe that such an incremental improvement may be hard to justify for developers to spend the effort to implement and maintain the usage of the new *BEF*s. In the future, we will explore more features and more ML algorithms, to further try to improve the effectiveness of RPredictor. For example, we will explore additional prediction features related to, *e.g.,* code-change history [54–56, 58, 59], testing activity, *e.g.,* [24, 36, 37], decision-making metadata, *e.g.,* [6, 7, 46], developer expertise, *e.g.,* [16, 57], build failure prediction *e.g.,* [28, 30–34], security issue prediction *e.g.,* [17, 27] or cross-language issues, *e.g.,* [18, 20].

ACKNOWLEDGEMENTS

We thank all reviewers for their valuable feedback. This work was partially funded by awards NSF CCF-1845446, NSF CCF-2046403, Virginia Tech's hiring package, by International Distinguished Researcher award C01INVESDIST by Universidad Rey Juan Carlos, by Saudi Arabian Cultural Mission (SACM), and by grant PID2022-142964OA-I00 funded by MCIN/AEI/10.13039/501100011033/FEDER, UE.

REFERENCES

[1] 2021. git merge - Integrating changes from another branch. https://www.git-tower.com/learn/git/commands/git-merge.

[2] 2021. jFSTMerge. https://github.com/guilhermejccavalcanti/jFSTMerge.

[3] 2023. Kruskal-Wallis Test. https://www.statisticssolutions.com/kruskal-wallis-test/.

[4] 2024. Research Artifact for Paper: Understanding the Impact of Branch Edit Features for the Automatic Prediction of Merge Conflict Resolutions. https://zenodo.org/doi/10.5281/zenodo.10553235.

[5] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. 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)*. 58–67. https://doi.org/10.1109/ESEM.2017.12

[6] Khadijah Al Safwan, Mohammed Elarnaoty, and Francisco Servant. 2022. Developers' Need for the Rationale of Code Commits: An In-breadth and In-depth Study. *Journal of Systems and Software* (2022).

[7] Khadijah Al Safwan and Francisco Servant. 2019. Decomposing the Rationale of Code Commits: The Software Developer's Perspective. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[8] Waad Aldndni, Na Meng, and Francisco Servant. 2023. Automatic prediction of developers' resolutions for software merge conflicts. *Journal of Systems and Software* 206 (2023), 111836. https://doi.org/10.1016/j.jss.2023.111836

[9] 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* (Essen, Germany) *(ASE 2012)*. ACM, New York, NY, USA, 120–129. https://doi.org/10.1145/2351676.2351694

[10] Sven Apel, Jorg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kastner. 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* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/2025113.2025141

[11] Jacob T. Biehl, Mary Czerwinski, Mary Czerwinski, Greg Smith, and George G. Robertson. 2007. FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '07)*. ACM, New York, NY, USA, 1313–1322. https://doi.org/10.1145/1240624.1240823

[12] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. 2020. An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering* 25, 1 (2020), 562–590. https://doi.org/10.1007/s10664-019-09735-4

[13] 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* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 168–178. https://doi.org/10.1145/2025113.2025139

[14] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. 2013. Early Detection of Collaboration Conflicts and Risks. *IEEE Transactions on Software Engineering* 39, 10 (Oct 2013), 1358–1375. https://doi.org/10.1109/TSE.2013.28

[15] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and Improving Semistructured Merge. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 59 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133883

[16] Lykes Claytor and Francisco Servant. 2018. Understanding and Leveraging Developer Inexpertise. In *International Conference on Software Engineering: Companion Proceeedings*.

[17] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: an Empirical Study at the Ecosystem Scale. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[18] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-Use and Portability of Regular Expressions. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[19] Elizabeth Dinella, Todd Mytkowicz, Alexey Svyatkovskiy, Christian Bird, Mayur Naik, and Shuvendu Lahiri. 2023. DeepMerge: Learning to Merge Programs. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1599–1614. https://doi.org/10.1109/TSE.2022.3183955

[20] Mohammed El Arnaoty and Francisco Servant. 2024. OneSpace: Detecting cross-language clones by learning a common embedding space. *Journal of Systems and Software* 208 (2024), 111911. https://doi.org/10.1016/j.jss.2023.111911

[21] Paulo Elias, Heleno de S. Campos, Eduardo Ogasawara, and Leonardo Gresta Paulino Murta. 2023. Towards accurate recommendations of merge conflicts resolution strategies. *Information and Software Technology* 164 (2023), 107332. https://doi.org/10.1016/j.infsof.2023.107332

[22] H Christian Estler, Martin Nordio, Carlo A Furia, and Bertrand Meyer. 2014. Awareness and merge conflicts in distributed software development. In *2014 IEEE 9th International Conference on Global Software Engineering*. IEEE, 26–35.

[23] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[24] Aakash Gautam, Saket Vishwasrao, and Francisco Servant. 2017. An Empirical Study of Activity, Popularity, Size, Testing, and Stability in Continuous Integration. In *International Conference on Mining Software Repositories*.

[25] Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and André 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), 1–1. https://doi.org/10.1109/TSE.2018.2871083

[26] M. L. Guimarães and A. R. Silva. 2012. Improving early detection of software merge conflicts. In *2012 34th International Conference on Software Engineering (ICSE)*. 342–352. https://doi.org/10.1109/ICSE.2012.6227180

[27] Sk Adnan Hassan, Zainab Aamir, Dongyoon Lee, James C. Davis, and Francisco Servant. 2023. Improving Developers' Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1238–1255. https://doi.org/10.1109/SP46215.2023.10179442

[28] Xianhao Jin. 2021. Reducing Cost in Continuous Integration with a Collection of Build Selection Approaches. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[29] Xianhao Jin and Francisco Servant. 2018. The hidden cost of code completion: understanding the impact of the recommendation-list length on its efficiency. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 70–73. https://doi.org/10.1145/3196398.3196474

[30] Xianhao Jin and Francisco Servant. 2020. A Cost-efficient Approach to Building in Continuous Integration. In *International Conference on Software Engineering*.

[31] Xianhao Jin and Francisco Servant. 2021. CIBench: A Dataset and Collection of Techniques for Build and Test Selection and Prioritization in Continuous Integration. In *International Conference on Software Engineering: Companion Proceedings*.

[32] Xianhao Jin and Francisco Servant. 2021. What Helped, and What Did Not? An Evaluation of the Strategies to Improve Continuous Integration. In *International Conference on Software Engineering*.

[33] Xianhao Jin and Francisco Servant. 2022. Which Builds are Really Safe to Skip? Maximizing Failure Observation for Build Selection in Continuous Integration. *Journal of Systems and Software* (2022).

[34] Xianhao Jin and Francisco Servant. 2023. HybridCISave: A Combined Build and Test Selection Approach in Continuous Integration. *ACM Transactions on Software Engineering and Methodology* 32, 4, Article 93 (may 2023), 39 pages. https://doi.org/10.1145/3576038

[35] B. K. Kasi and A. Sarma. 2013. Cassandra: Proactive conflict minimization through optimized task scheduling. In *2013 35th International Conference on Software Engineering (ICSE)*. 732–741. https://doi.org/10.1109/ICSE.2013.6606619

[36] Ayaan M Kazerouni, James C Davis, Arinjoy Basak, Clifford A Shaffer, Francisco Servant, and Stephen H Edwards. 2021. Fast and Accurate Incremental Feedback for Students' Software Tests using Selective Mutation Analysis. *Journal of Systems and Software* (2021).

[37] Ayaan M. Kazerouni, Clifford A. Shaffer, Stephen H. Edwards, and Francisco Servant. 2019. Assessing Incremental Testing Practices and Their Impact on Project Outcomes.

[38] Michele Lanza, Marco D'Ambros, Alberto Bacchelli, Lile Hattori, and Francesco Rigotti. 2013. Manhattan: Supporting real-time visual team activity awareness. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 207–210.

[39] Olaf Leßenich, Sven Apel, and Christian Lengauer. 2015. Balancing Precision and Performance in Structured Merge. *Automated Software Engg.* 22, 3 (Sept. 2015), 367–397. https://doi.org/10.1007/s10515-014-0151-5

[40] Olaf Leßenich, Janet Siegmund, Sven Apel, Christian Kästner, and Claus Hunsen. 2018. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering* 25, 2 (2018), 279–313.

[41] Thomas W. MacFarland and Jan M. Yates. 2016. *Kruskal–Wallis H-Test for Oneway Analysis of Variance (ANOVA) by Ranks*. Springer International Publishing, Cham, 177–211. https://doi.org/10.1007/978-3-319-30634-6_6

[42] Chandra Maddila, Nachiappan Nagappan, Christian Bird, Georgios Gousios, and Arie van Deursen. 2021. ConE: A Concurrent Edit Detection Tool for Large Scale Software Development. *arXiv preprint arXiv:2101.06542* (2021).

[43] M. Mahmoudi, S. Nadi, and N. 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)*. 151–162. https://doi.org/10.1109/SANER.2019.8668012

[44] J.H. McDonald. 2014. *Handbook of Biological Statistics (3rd ed.)*. Sparky House Publishing, Baltimore, Maryland, 157–164.

[45] T. Mens. 2002. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering* 28, 5 (2002), 449–462. https://doi.org/10.1109/TSE.2002.1000449

[46] Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. In *International Conference on Automated Software Engineering*.

[47] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. 2019. The life-cycle of merge conflicts: processes, barriers, and strategies. *Empirical Software Engineering* (02 2019). https://doi.org/10.1007/s10664-018-9674-x

[48] 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, 3 (01 Dec 2018), 741–765. https://doi.org/10.1007/s10606-018-9323-3

[49] Yuichi Nishimura and Katsuhisa Maruyama. 2016. Supporting Merge Conflict Resolution by Using Fine-Grained Code Change History. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* 1 (2016), 661–664.

[50] Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. 2019. Predicting Merge Conflicts in Collaborative Software Development. https://arxiv.org/pdf/1907.06274.pdf.

[51] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu Lahiri, and Mike Kaufman. 2021. Can Program Synthesis Be Used to Learn Merge Conflict Resolutions? An Empirical Analysis. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 785–796. https://doi.org/10.1109/ICSE43902.2021.00077

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[53] Anita Sarma, David F Redmiles, and Andre Van Der Hoek. 2011. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering* 38, 4 (2011), 889–908.

[54] Francisco Servant. 2013. Supporting Bug Investigation using History Analysis. In *International Conference on Automated Software Engineering*.

[55] Francisco Servant and James A Jones. 2011. History Slicing. In *International Conference on Automated Software Engineering*. IEEE.

[56] Francisco Servant and James A Jones. 2012. History Slicing: Assisting Code-evolution Tasks. In *International Symposium on the Foundations of Software Engineering*.

[57] Francisco Servant and James A Jones. 2012. WhoseFault: Automatic Developer-to-Fault Assignment through Fault Localization. In *International Conference on Software Engineering*.

[58] Francisco Servant and James A Jones. 2013. Chronos: Visualizing Slices of Source-code History. In *Working Conference on Software Visualization*.

[59] Francisco Servant and James A Jones. 2017. Fuzzy Fine-grained Code-history Analysis. In *International Conference on Software Engineering*.

[60] Francisco Servant, James A Jones, and André Van Der Hoek. 2010. CASI: preventing indirect conflicts through a live visualization. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*. 39–46.

[61] Bowen Shen, Muhammad Ali Gulzar, Fei He, and Na Meng. 2022. A Characterization Study of Merge Conflicts in Java Projects. *ACM Trans. Softw. Eng. Methodol.* (jun 2022). https://doi.org/10.1145/3546944 Just Accepted.

[62] 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 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360596

[63] Marcelo Sousa, Isil Dillig, and Shuvendu Lahiri. 2018. Verified Three-Way Program Merge. In *Object-Oriented Programming, Systems, Languages & Applications Conference (OOPSLA 2018)*. ACM. https://www.microsoft.com/en-us/research/publication/verified-three-way-program-merge/

[64] Alexey Svyatkovskiy, Sarah Fakhoury, Negar Ghorbani, Todd Mytkowicz, Elizabeth Dinella, Christian Bird, Jinu Jang, Neel Sundaresan, and Shuvendu K. Lahiri. 2022. Program Merge Conflict Resolution via Neural Transformers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 822–833. https://doi.org/10.1145/3540250.3549163

[65] Gustavo Vale, Claus Hunsen, Eduardo Figueiredo, and Sven Apel. 2021. Challenges of Resolving Merge Conflicts: A Mining and Survey Study. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3130098

[66] R. Yuzuki, H. Hata, and K. Matsumoto. 2015. How we resolve conflict: an empirical study of method-level conflict resolution. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*. 21–24. https://doi.org/10.1109/SWAN.2015.7070484

[67] Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K. Lahiri. 2022. Using Pre-Trained Language Models to Resolve Textual and Semantic Merge Conflicts (Experience Paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/3533767.3534396

[68] Fengmin Zhu and Fei He. 2018. Conflict Resolution for Structured Merge via Version Space Algebra. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 166 (Oct. 2018), 25 pages. https://doi.org/10.1145/3276536