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

Waad Alldndni
Virginia Tech
Blacksburg, VA, U.S.A.
wada@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 (EHFs) 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 (BEFs), that capture the fine-grained edits that were performed on each branch of the conflict. We learned multiple lessons. First, BEFs provided lower effectiveness (F-score) than the original EHF. Second, combining BEFs with EHF still did not improve the effectiveness of EHF. Third, the feature set that provided highest effectiveness in our experiments was the combination of EHF with a subset of BEF that captures the number of insertions performed in the local branch, but this combination only improved EHF 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.

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 96% of developers postponed resolving merge conflicts for various reasons, most of them related to the complexity of the conflicts.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICPC ’24, April 15–16, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
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 (EHFs) 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 (BEFs), 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 BEFs 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 BEFs 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 BEFs, in different ways. First, we compared the effectiveness provided by BEFs as features in RPRedictoR with that provided by its original design using EHF. Then, we measured the effectiveness of combining both BEFs and EHF. Then, we also studied in finer granularity the effectiveness provided by 6 subsets of BEFs, (based on each branch and edit type) as well as their combination with EHF. 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 BEFs for conflict resolution prediction did not improve the effectiveness of RPRedictoR— BEFs provided slightly lower effectiveness than EHF. Second, combining EHF and BEF for prediction also did not improve the effectiveness of using only EHF — both approaches provided the same f-score. Third, among all our studied feature sets, the best-performing one was EHF and BEF, but it only improved the effectiveness of EHF 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 features, and maintaining it over time. Finally, some of the feature sets that could not improve the effectiveness of EHF 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](image)

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 scikit-learn [52].

Original: Evolution History Features (EHFs). 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 (BEFs). In this paper, we propose a new set of Branch Edit Features (BEFs) and study whether they could improve RPRedictoR’s effectiveness. BEFs 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 (BEFs) measure, separately, the number of Insertions (I), Updates (U), or Deletions (D) that were performed to each Java element ($E_j$), within each of the Local (L) or Remote (R) branch. For example feature $BEF_{LJE}$ measures the number of Variable Declaration Statements ($JE_l$) that were inserted ($I$) in the local branch ($L$). In total, our BEFs 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, $BEF_{LJ}$ 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, $BEF_{LJ}$ contains the 66 features $BEF_{LJE} - BEF_{RJE}$. We represent them in the bottom row of Table 2.
We study the impact of our new Branch Edit Features (BEFs) 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 features in its original design for its predictions. For each one of the merge conflicts in our studied dataset, 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, 1,574 via KR, and 1,561 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 EHFs as in our original evaluation of RPRedictoR [8], and we measured BEFs using Guntree v3.0.0 [23].

### Table 1: The 66 fine-grained Java elements (JE) for which we measure our new proposed branch edits (BEFs).

<table>
<thead>
<tr>
<th>Java elements (JE)</th>
<th>BEF subset with all Java elements (JE1–JE6a) for a branch and edit type</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE1</td>
<td>BF1JJE1, BF1JJEu, BF1JJEa, BF1JJEb, BF1JJEk, BF1JJEh, BF1JJEa, BF1JJEb, BF1JJEk, BF1JJEh</td>
</tr>
<tr>
<td>JE2</td>
<td>BF2JJE1, BF2JJEu, BF2JJEa, BF2JJEb, BF2JJEk, BF2JJEh, BF2JJEa, BF2JJEb, BF2JJEk, BF2JJEh</td>
</tr>
<tr>
<td>JE3</td>
<td>BF3JJE1, BF3JJEu, BF3JJEa, BF3JJEb, BF3JJEk, BF3JJEh, BF3JJEa, BF3JJEb, BF3JJEk, BF3JJEh</td>
</tr>
<tr>
<td>JE4</td>
<td>BF4JJE1, BF4JJEu, BF4JJEa, BF4JJEb, BF4JJEk, BF4JJEh, BF4JJEa, BF4JJEb, BF4JJEk, BF4JJEh</td>
</tr>
<tr>
<td>JE5</td>
<td>BF5JJE1, BF5JJEu, BF5JJEa, BF5JJEb, BF5JJEk, BF5JJEh, BF5JJEa, BF5JJEb, BF5JJEk, BF5JJEh</td>
</tr>
<tr>
<td>JE6</td>
<td>BF6JJE1, BF6JJEu, BF6JJEa, BF6JJEb, BF6JJEk, BF6JJEh, BF6JJEa, BF6JJEb, BF6JJEk, BF6JJEh</td>
</tr>
</tbody>
</table>

### Table 2: Our new proposed Branch Edit Features (BEFs). 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.

<table>
<thead>
<tr>
<th>Branch</th>
<th>Local (L)</th>
<th>Remote (R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertions (I)</td>
<td>BF1LIJE1, BF1LJIE1, BF1LIJEa, BF1LJIEa, BF1LIJEb, BF1LJIEb, BF1LIJEk, BF1LJIEk, BF1LIJEh, BF1LJIEh</td>
<td></td>
</tr>
<tr>
<td>Updates (U)</td>
<td>BF1LJUE1, BF1LJUEu, BF1LJUEa, BF1LJUEa, BF1LJUEb, BF1LJUEb, BF1LJUEk, BF1LJUEk, BF1LJUEh, BF1LJUEh</td>
<td></td>
</tr>
<tr>
<td>Deletions (D)</td>
<td>BF1LJDE1, BF1LJDEa, BF1LJDEb, BF1LJDEk, BF1LJDEh, BF1LJDEa, BF1LJDEb, BF1LJDEk, BF1LJDEh</td>
<td></td>
</tr>
</tbody>
</table>

### Table 3: The Evolution History Features (EHFs) used by the original design of RPRedictoR for its predictions.

<table>
<thead>
<tr>
<th>Evolution History Features (EHFs)</th>
<th># of Repositories</th>
<th># of Conflicts resolved by</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KL</td>
<td>KR</td>
</tr>
<tr>
<td>Data used in our characterization study (RQ1)</td>
<td>70</td>
<td>1,102</td>
</tr>
<tr>
<td>Data used in the tool evaluation (RQ2–RQ5)</td>
<td>377</td>
<td>5,139</td>
</tr>
<tr>
<td>Total</td>
<td>477</td>
<td>6,241</td>
</tr>
</tbody>
</table>

### Table 4: The dataset used in our research

<table>
<thead>
<tr>
<th>Dataset</th>
<th># of Repositories</th>
<th># of Conflicts resolved by</th>
</tr>
</thead>
<tbody>
<tr>
<td>KL</td>
<td>774</td>
<td>1,511</td>
</tr>
<tr>
<td>KR</td>
<td>1,102</td>
<td>774</td>
</tr>
<tr>
<td>ME</td>
<td>8,448</td>
<td>6,937</td>
</tr>
<tr>
<td>Total</td>
<td>19,286</td>
<td>4,597</td>
</tr>
</tbody>
</table>
3.2 Method for RQ1

We first study which ones of our newly proposed BEFs differ significantly for different conflict resolution strategies, applying statistical analysis. We measured the value of our BEFs for each one of the 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 BEFs 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 \( S_i \).

\[
P_i = \frac{\# \text{ of conflicts correctly labeled as } \text{"} S_i \text{"}}{\# \text{ of conflicts labeled as } \text{"} S_i \text{"}} \tag{1}
\]

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

\[
R_i = \frac{\# \text{ of conflicts correctly labeled as } \text{"} S_i \text{"}}{\# \text{ of conflicts that were resolved via } S_i} \tag{2}
\]

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

\[
F_i = \frac{2 \times P_i \times R_i}{P_i + R_i} \tag{3}
\]

Aggregated (Overall) metrics (\( P, R, F \)). We also measured our metrics by computing the weighted average across all strategies. Specifically, if we denote \( \Gamma \) 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:

\[
\Gamma_{\text{overall}} = \frac{\sum_{i=1}^{3} \Gamma_i \times n_i}{\sum_{i=1}^{3} n_i} \tag{4}
\]

Thus, the overall \( F \) is computed with:

\[
F_{\text{overall}} = \frac{2 \times P_{\text{overall}} \times R_{\text{overall}}}{P_{\text{overall}} + R_{\text{overall}}} \tag{5}
\]

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.

\[
C = \frac{\# \text{ of conflicts conservatively labeled}}{\# \text{ of all predictions}} \tag{6}
\]

This allows us to measure technique effectiveness in a single metric.
**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.

<table>
<thead>
<tr>
<th>Branch</th>
<th>Edit type</th>
<th>Local (L)</th>
<th>Remote (R)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insertions (I)</td>
<td>Updates (U)</td>
<td>Deletions (D)</td>
</tr>
<tr>
<td>BEF subset</td>
<td>BEFLD</td>
<td>BEFLU</td>
<td>BEFLD</td>
</tr>
<tr>
<td>Statistically Significant Java Elements (JEi)</td>
<td>2, 5, 6, 7, 10, 12, 13, 14, 17, 36, 57</td>
<td>1, 6, 12, 17, 20, 48, 58, 59</td>
<td>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, 44, 45, 49, 51, 52, 53, 54, 55, 57, 64</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Branch</th>
<th>BEFs only</th>
<th>EHF s only</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>R</td>
</tr>
<tr>
<td>KL</td>
<td>37%</td>
<td>27%</td>
</tr>
<tr>
<td>KR</td>
<td>34%</td>
<td>29%</td>
</tr>
<tr>
<td>ME</td>
<td>44%</td>
<td>57%</td>
</tr>
<tr>
<td>Overall</td>
<td>39%</td>
<td>40%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Test Fold #</th>
<th>BEFs only</th>
<th>EHF s only</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>40%</td>
<td>43%</td>
</tr>
<tr>
<td>2</td>
<td>38%</td>
<td>42%</td>
</tr>
<tr>
<td>3</td>
<td>34%</td>
<td>43%</td>
</tr>
<tr>
<td>4</td>
<td>53%</td>
<td>58%</td>
</tr>
<tr>
<td>5</td>
<td>43%</td>
<td>44%</td>
</tr>
<tr>
<td>6</td>
<td>42%</td>
<td>50%</td>
</tr>
<tr>
<td>7</td>
<td>35%</td>
<td>39%</td>
</tr>
<tr>
<td>8</td>
<td>35%</td>
<td>31%</td>
</tr>
<tr>
<td>9</td>
<td>40%</td>
<td>49%</td>
</tr>
<tr>
<td>10</td>
<td>42%</td>
<td>46%</td>
</tr>
<tr>
<td>All Folds</td>
<td>39%</td>
<td>43%</td>
</tr>
</tbody>
</table>

**4 RESULTS**

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

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

**Finding 3**: Effectiveness of Cross-project Prediction. When trained over different software projects (cross-project context), RPredictor provided lower effectiveness when using BEFs 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 BEFs provided higher Conservativeness (C) Score (90% vs. 80%) than when using EHF s, for all our evaluated (and aggregated) test folds.

For the remaining experiments (RQ2–RQ5), for variants that we report as using BEFs 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).
Table 8: RQ3. Within-project effectiveness of BEFs and EHF\textsubscript{s} combined vs. EHF\textsubscript{s} only.

<table>
<thead>
<tr>
<th>Feature Set</th>
<th>P</th>
<th>R</th>
<th>F</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFs and EHF\textsubscript{s}</td>
<td>47%</td>
<td>38%</td>
<td>42%</td>
<td>72%</td>
</tr>
<tr>
<td>EHF\textsubscript{s} only</td>
<td>40%</td>
<td>43%</td>
<td>41%</td>
<td>80%</td>
</tr>
</tbody>
</table>

Table 9: RQ3. Cross-project effectiveness of BEFs and EHF\textsubscript{s} combined vs. EHF\textsubscript{s} only.

<table>
<thead>
<tr>
<th>Test Fold</th>
<th>P</th>
<th>R</th>
<th>F</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFs and EHF\textsubscript{s}</td>
<td>42%</td>
<td>45%</td>
<td>43%</td>
<td>82%</td>
</tr>
<tr>
<td>EHF\textsubscript{s} only</td>
<td>43%</td>
<td>45%</td>
<td>44%</td>
<td>80%</td>
</tr>
</tbody>
</table>

Finding 3: In the cross-project usage context, RP\textsuperscript{redictor} also provided lower effectiveness (Precision, Recall, and F-score) when using BEFs for its prediction than when using its original EHF\textsubscript{s}.

RQ3: Effectiveness of BEFs and EHF\textsubscript{s} vs. EHF\textsubscript{s} only. In RQ2, we observed that RP\textsuperscript{redictor} did not provide higher effectiveness when using BEFs than when using its original set of EHF features. Next, we study whether BEFs and EHF\textsubscript{s} can help each other, i.e., whether RP\textsuperscript{redictor} can provide higher effectiveness by using combining both. Thus, we evaluated a new variant of RP\textsuperscript{redictor} that uses the union of BEFs and EHF\textsubscript{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 RP\textsuperscript{redictor} (using EHF\textsubscript{s} only) in these tables, for ease of comparison.

Effectiveness of Within-project Prediction. Table 8 shows that combining BEFs and EHF\textsubscript{s} in the feature set of RP\textsuperscript{redictor} provided very similar effectiveness than when RP\textsuperscript{redictor} used only its original EHF\textsubscript{s}, for the within-project usage context. The variant of RP\textsuperscript{redictor} that used both BEFs and EHF\textsubscript{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 BEFs and EHF\textsubscript{s} was about the same as when using only EHF\textsubscript{s}.

This result is consistent with what we observed in RQ2, since BEFs by themselves made RP\textsuperscript{redictor} provide lower effectiveness. However, RQ3 shows us that, when combining BEFs and EHF\textsubscript{s}, RP\textsuperscript{redictor} was able to not allow the worse prediction power of BEFs hurt the better prediction power of EHF\textsubscript{s}, i.e., combining BEFs and EHF\textsubscript{s} did not produce worse results than using EHF\textsubscript{s} only. It is also possible that combining BEFs and EHF\textsubscript{s} could not improve the effectiveness of RP\textsuperscript{redictor} 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.

Finding 4: For the within-project usage context, combining BEFs and EHF\textsubscript{s} provided about the same effectiveness as when RP\textsuperscript{redictor} used EHF\textsubscript{s} only.

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

This time, combining BEFs and EHF\textsubscript{s} provided a small effectiveness improvement to RP\textsuperscript{redictor} in the cross-project usage context. In very few cases, the RP\textsuperscript{redictor} variant that predicts based on both BEFs and EHF\textsubscript{s} predicted the KL and KR resolutions correctly when the original variant based on EHF\textsubscript{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 BEFs and EHF\textsubscript{s} also did not provide an important increase in effectiveness.

Finding 5: For the cross-project usage context, combining BEFs and EHF\textsubscript{s} 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 BEFs that showed statistically significant differences for different conflict resolution strategies belonged to the BEF\textsubscript{LD} and BEF\textsubscript{RG} 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 RP\textsuperscript{redictor}.

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

<table>
<thead>
<tr>
<th>Feature Set</th>
<th>P</th>
<th>R</th>
<th>F</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFs only</td>
<td>47%</td>
<td>48%</td>
<td>47%</td>
<td>72%</td>
</tr>
<tr>
<td>EHF\textsubscript{s} only</td>
<td>39%</td>
<td>40%</td>
<td>39%</td>
<td>71%</td>
</tr>
<tr>
<td>BEF\textsubscript{LD} only</td>
<td>47%</td>
<td>48%</td>
<td>47%</td>
<td>72%</td>
</tr>
<tr>
<td>BEF\textsubscript{RG} only</td>
<td>37%</td>
<td>38%</td>
<td>37%</td>
<td>69%</td>
</tr>
<tr>
<td>BEF\textsubscript{LD} only</td>
<td>38%</td>
<td>40%</td>
<td>38%</td>
<td>72%</td>
</tr>
<tr>
<td>BEF\textsubscript{RG} only</td>
<td>38%</td>
<td>40%</td>
<td>38%</td>
<td>72%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Feature Set</th>
<th>P</th>
<th>R</th>
<th>F</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFs only</td>
<td>43%</td>
<td>44%</td>
<td>44%</td>
<td>80%</td>
</tr>
<tr>
<td>EHF\textsubscript{s} only</td>
<td>39%</td>
<td>43%</td>
<td>41%</td>
<td>90%</td>
</tr>
<tr>
<td>BEF\textsubscript{LD} only</td>
<td>44%</td>
<td>45%</td>
<td>44%</td>
<td>82%</td>
</tr>
<tr>
<td>BEF\textsubscript{RG} only</td>
<td>58%</td>
<td>43%</td>
<td>40%</td>
<td>93%</td>
</tr>
<tr>
<td>BEF\textsubscript{RU} only</td>
<td>38%</td>
<td>43%</td>
<td>41%</td>
<td>93%</td>
</tr>
<tr>
<td>BEF\textsubscript{RD} only</td>
<td>39%</td>
<td>44%</td>
<td>41%</td>
<td>98%</td>
</tr>
</tbody>
</table>
We created 6 separate variants of RPredictor, each one using only one of the BEF subsets for its predictions, namely: BEF\_LI, BEF\_LD, BEF\_RD, BEF\_RI, BEF\_RL, 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 EHF\_s 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 EHF\_s 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.

We now evaluated 6 new variants of RPredictor, each one using for prediction: the EHF\_s and one of the 6 BEF\_s subsets. We used the same settings as in previous experiments, for within-project and cross-project prediction. We report the results of these 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 EHF\_s and BEF\_LI provided a minor improvement over RPredictor’s original design using EHF\_s only. EHF\_s 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 EHFs and BEFLD. This variant provided higher results for all metrics when compared with RPRedictor’s original EHFs 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 EHFs with individual BEF subsets can be more beneficial than combining them with all BEFs. EHFs and BEFLD provided better results than EHFs and all BEFs. However, we also again observe that the best-performing variant of this experiment only provided minor improvements over the original EHFs. EHFs and BEFLD improved EHFs 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 EHFs 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 BEFLD provided a minor improvement over RPRedictor’s original design using EHFs only. EHFs and BEFLD 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 EHFs and BEFL in the within-project prediction context. This shows that RPRedictor can be more effective when, in addition to its original EHFs, 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 EHFs and BEFL variant provided over RPRedictor’s original design (using EHFs 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 BEFL 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 development pipeline in exchange for the small effectiveness increment that the EHFs and BEFL variant provides.

Past studies suggested that BEFs could be useful predictors. We originally set out to study whether BEFs 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: EHFs and BEFs 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 BEFs than it did originally by using EHFs. The results obtained by both variants differed in less than 10 pp. for all metrics. Unfortunately, the predictive power of BEFs do not seem to complement well the one provided by EHFs. We observed in RQ3 that combining both EHFs and BEFs still provided the same effectiveness as using EHFs only. This may mean that BEFs were helpful in predicting the same kinds of conflicts for which EHFs were already good predictors, but they were not as helpful in predicting the ones that EHFs 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 was the results of the BEFLD variant for cross-project prediction. It provided 41% f-score, which is 3pp. lower effectiveness than EHFs, but it also provided 98% C-score, which is 18 pp. higher than EHFs. This means that this technique variant was less effective than EHFs 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 BEFLD 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. Lešenić 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, Owkhadi-Karesht 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 Palantir [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. Intellimize [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 BEFs 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 BEFs, it is still possible that BEFs defined for other code elements, or for other programming languages may provide different results. It is also possible that our studied BEFs (or other BEFs) 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\textsubscript{s} and BEF\textsubscript{s} over 15,899 code conflicts within 377 software projects. Also, our BEF\textsubscript{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 BEFs 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\textsubscript{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 (EHFs) [8].

In this paper, we proposed a new set of Branch Edit Features (BEFs) 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 (BEFs), studying various scenarios in which they could improve effectiveness: by themselves, combining them with EHF\textsubscript{s}, using only subsets of them, and combining EHF\textsubscript{s} with subsets of BEFs. Overall, we found that only one of our studied set of features improved the effectiveness of RPREDICTOR, but did so very slightly: the EHF\textsubscript{s} and BEF\textsubscript{II} set improved over EHF\textsubscript{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 BEFs. 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 meta-data, 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

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

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


https://doi.org/10.1145/3533767.3534396