Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Predicting Merge Conflicts Considering Social and Technical Assets

Received: date / Accepted: date

Abstract Concurrent contributions to a code base may introduce merge conflicts. Whereas merge conflicts are easy and common to introduce, resolving them is a difficult, time-consuming, and often error-prone task. Previous research concentrated on the emergence of merge conflicts considering technical assets in their analyses and often ignored the social perspective (e.g., developer roles). Our goal is to understand and predict merge conflicts considering social and technical assets. We devise three models for predicting merge conflicts based on common measures used by developers. The first model focuses on the social assets, the second on technical assets, and the third on technical and social assets. To evaluate our predictors, we report on a large-scale empirical study analyzing the histories of 66 real-world software systems. Specifically, we categorize developers into *top* or *occasional* contributors at project and merge-scenario level. We found that top contributors at project level and occasional contributors at merge-scenario level cause more merge conflicts than the other roles. Hence, the coordination of top contributors at project level and occasional contributors at merge-scenario level is a good starting point to minimize the occurrence of merge conflicts (especially because when these two developers work on the source branch, the chances of merge conflicts are 32.31%). Overall, we show that predicting merge conflicts incorporating developer roles is possible in practice with high accuracy (0.92) and recall (1.00) when combining technical and social assets, which is vital information to guide improvements on speculative merging techniques.

Keywords Collaborative Software Development, Version Control Systems, Developers Role, Three-way Merge, Merge Conflicts

Address(es) of author(s) should be given

1 Introduction

Successful collaborative software development depends on the ability to coordinate technical and social assets [34]. Version control systems help developers to manage concurrent contributions across a project's evolution [68]. Although typically a large number of commits cleanly merge, concurrent changes can overlap, leading to *merge conflicts*. While merge conflicts are easy to introduce, resolving them is difficult, time-consuming, and often error-prone [40].

Given the costs involved in the merge conflict life-cycle [45], researchers have proposed merge strategies (e.g., structured [3], semi-structured [4]), avoidance strategies (e.g., continuous integration [29], speculative merging [11]), awareness tools (e.g., CollabVS [17], Palantír [52], Cassandra [36], FASTDash [5]), investigated the nature of merge conflicts (e.g., identifying the types of code changes that lead to merge conflicts) [1,26,40,61], asked how developers have resolved merge conflicts [41, 45, 62], and tried to predict them [2, 40, 47].

When merge strategies are inefficient in reducing the number of merge conflicts, developers should continuously integrate their changes and keep aware of what others are doing. To support awareness, researchers have developed tools to alert developers about potential merge conflicts before they become too complex [5, 17, 36, 52]. Awareness tools speculatively pull and merge all combinations of available branches. The downside is that, constantly pulling and merging a large number of branch combinations, can quickly get prohibitively expensive [11]. One opportunity for decreasing this cost is to reduce the number of speculative merging operations in merge scenarios concentrating only the ones that are prone to conflict. To achieve this, researchers use machine learning techniques for predicting merge conflicts [47].

There are six studies predicting merge conflicts. Leßenich et al. [40] look for correlations between various technical measures and merge conflicts. None of their measures have a strong correlation with merge conflicts (e.g., varying from 0.13 to 0.43). Accioly et al. [2] investigate the relationship between two types of code changes (i.e., changes to the same method and changes to directly dependent methods) and merge conflicts. They found a precision of 57.99% and a recall of 83.62%. Rocha et al. [51] look at whether it is possible to use acceptance tests to predict files changed by programming tasks assuming that choosing the right tasks to work on in parallel will decrease the number of merge conflicts. As a results, they found a relation between acceptance tests and files changed. Dias et al. [18] investigate the relation of modularity (in term of model-view-controller (MVC) layers), size, and timing of code changes and merge conflicts. As a result, they found that cross MVC layer, large, and long-living changes are conflict-prone. Owhadi-Kareshk et al. [47] build a machine learning classifier (using decision trees and random forests) based on 9 Git feature sets. They obtained precision, recall, and f1-score of 1.00, 0.96, and 0.97 for safe merge scenarios and 0.63, 0.96, and 0.68 for conflicting merge scenarios. Similarly, Trif et al. [59] use 4 machine learning classifiers (SVM, Naive Bayes, random forests, and neural networks) to predict merge conflicts.

Their best performance results using neural networks were 0.77 and 0.93 for precision and recall, respectively.

It is important to note that previous work concentrated on the prediction of merge conflicts considering technical assets and often ignored the social perspective (i.e., developers and their relationship). Thus, as current merge conflict predictions, in terms of recall, are low, we hypothesise that information on social aspects might increase recall when predicting merge conflicts. Since coding is a social task, it might be simple for developers to know their role and relationship with other developers in a merge scenario. Hence, in addition to reducing the costs of speculative merging techniques, an understanding of the influence of the social dimension (e.g., developer role) on the emergence of merge conflicts might be useful to guide the coordination of developers aiming at reducing the number of merge conflicts. To illustrate how useful knowing the developer's role who caused the merge conflict can be for project coordination, we selected a merge scenario of project *create-react-app*[1]. In this merge scenario, 62 developers changed 651 chunks distributed into 121 files. Despite the high number of developers involved, the top contributors of the two merged branches introduced all conflicting code. Therefore, by making these developers aware of the other code changes, they could have communicated to understand the changes avoiding the merge conflicts or, at least, simplify the conflict resolution since they could explain their changes to each other and decide together what should remain in the target branch.

Our overall goal is to predict merge conflicts taking the social dimension into account. To achieve our goal, we have conducted a large empirical study analyzing the history of 66 repositories of popular software projects with a total of 78 740 merge scenarios. We classified developers as top and occasional based on their code contributions with distinct granularity (project and merge-scenario level). Aiming at increasing our knowledge on developer roles, we first look at the relation of each role separately, then we combine project- and merge-scenario-level information. Later, after getting this initial understanding of the relation between developer roles and merge conflicts, we devised three models to predict merge conflicts. We used three classifiers (decision tree, random forest, and KNN), and seven balancing techniques (e.g., SMOTE, Adasyn, and over-sampling). The first model is composed of only social measures, the second is composed of only technical measures, and the third model is composed of all (social and technical) measures. Creating these three models enables us to pin down how different measures influence the predictions and if social measures are useful in practice.

We found that top contributors slightly contribute to more merge conflicts at project level, and occasional contributors contribute to more merge conflicts than top contributors at merge-scenario level. When combining the granularity, we found that top contributors at project level that are occasional contributors at merge-scenario level are more related to merge conflicts than all other combinations of developer roles. When these developers touch (i.e., add,

[1] `https://github.com/facebookincubator/create-react-app/commit/1e83e8`

delete, change) the source branch, the chances of merge conflicts are 32.31%. Regarding predictions, random forest performed better in most cases and our models can correctly predict all conflicting scenarios (i.e., it achieved 100% of recall). Looking at other performance measures (e.g., precision, f1-score, accuracy, and AUC), the models with all and only technical measures performed better than the model composed by only social measures.

Albeit technical assets have proven essential to predict merge conflicts, our findings shall call the attention of researchers and practitioners to focus on social assets and the branches developers are touching in their analyses.

Overall, we make the following contributions:

- We provide evidence that it is possible to predict merge conflicts by looking only at social measures (e.g., developer roles, the number of developers involved, and the branch the developers touch);
- We analyze the relation between developer roles and merge conflicts from three different perspectives: (i) with developer roles investigated individually, (ii) with developer roles at project and branch level combined, and (iii) using machine learning classifiers with three models (i.e., social measures vs. technical measures vs. all measures).
- We show that code changes in the *source* branch are more conflict-prone than code changes in the *target* branch. For instance, when top and occasional contributors at merge-scenario level touch the source branch, 4.36% and 24.60% of the merge scenarios lead to conflicts, respectively. On the other hand, only 4.88% and 8.32% of the merge scenarios lead to conflicts when top and occasional contributors touch the target branch;
- We make our infrastructure and data publicly available for replication and follow-up studies on a supplementary Web site [63].

2 Background and Related Work

In this section, we present an overview of studies that investigate merge conflicts and classify developers into their roles.

2.1 The Three-way Merge

The three-way merge pattern also known as *pull-based model* and *merge scenario* (from hereon, merge scenario) is a distinct and widely collaborative development pattern [27]. In this model, developers first fork the main repository by creating a branch. Then, developers commit their changes independently to add new features or fix bugs. Finally, they create a merge commit integrating their changes back to the main repository.

There are other ways than the three-way pattern to integrate code to the repository, such as fast-forward, rebase, or squash integrations [37]. However, these integrations damage the project's history, hindering the understanding of how the changes were made in practice. Hence, to understand the evolution

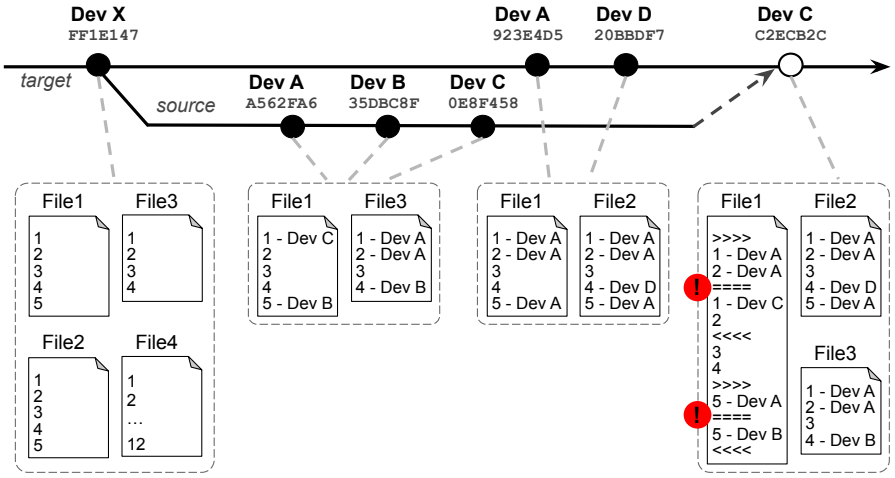

Fig. 1 Illustrative merge scenario

of the project, we use the three-way merge pattern. Even though a branch lives longer, we are considering just the changes from the fork up to the integration (i.e., a merge scenario). If the branch is forked and integrated again, it sets up another merge scenario.

In Figure 1, we illustrate a merge scenario where Dev X created a repository with four files (File 1, 2, 3, an 4). Later, Dev A forked the *target* branch, creating the *source* branch. Together with Dev B and Dev C, Dev A touched File 1 and File 3 in the *source* branch. Concurrently Dev A and Dev D changed File 1 and File 2 in the *target* branch. Finally, Dev C tried to merge the *source* branch into the *target* branch.

Software integrations typically cleanly merge, however, concurrent changes can overlap, leading to merge conflicts. In the example of Figure 1, Dev C faced merge conflicts in File 1. These conflicts appeared given the concurrent changes of Dev A in the *target* branch with the changes of Dev C and Dev B in the *source* branch. In the next section, we discuss how researchers and practitioners have investigated and dealt with merge conflicts.

2.2 Merge Conflicts

Merge conflicts are easy to introduce, but resolving them is a difficult, time-consuming, and error-prone task [40]. There are dozens of studies investigating the whole merge conflict life-cycle. In this section, we give an overview of studies that try to: (i) avoid or minimize the emergence of merge conflicts, (ii) investigate how merge conflicts and code changes that cause merge conflicts look like, (iii) estimate their resolution or difficulty, and (iv) predict them.

Avoiding merge conflicts. Researchers have investigated *merge strategies* (e.g., [3, 4, 12, 25, 32, 64]) and *awareness approaches and tools* (e.g., [5,

11, 17, 29, 31, 36, 52]). Regarding merge strategies, researchers proposed structured strategies that leverage information about the underlying code structure by analyzing the corresponding Abstract Syntax Tree (AST) [3]. Some merge conflicts such as due to formatting changes or renaming often can be avoided using AST information. Since differentiating a complete AST is expensive, semi-structured merge strategies improve performance by producing a partial AST that expands only until the method level, with complete method bodies in the leaves [3]. In this line Dinella et al. [19] propose DeepMerge, a tool that uses deep learning algorithm to merge code that an unstructured merge technique (DIFF3) failed to merge. Regarding awareness approaches and tools, Guimarães et al. [29] proposed to continuously merge, compile, and test committed and uncommitted changes to detect merge conflicts as early as possible. As examples of tools, CollabVS [17] is a semi-synchronous distributed computer supported model that allows programmers creating code asynchronously to synchronously collaborate with each other to detect and resolve potentially conflicting tasks before they have completed the tasks. Crystal [11] is a visual tool that uses speculative analysis to help developers detect, manage, and prevent merge conflicts. FASTDash [5] is an interactive visualization tool that seeks to improve team activity awareness using a spatial representation of the shared code base that highlights team members' current activities (e.g., what methods and classes are currently changed). Similarly, Syde [31] is a tool for increasing awareness by sharing the code changes from other developers' workspaces. Similar to FASTDash and Syde, Palantír [52] visually illustrates code changes and helps developers avoid conflicts by making them aware of changes in private workspaces. Finally, Cassandra [36] is a tool to minimize conflicts by optimizing task scheduling to minimize simultaneous edits to the same files.

Investigating merge conflicts. A few studies have investigated how merge conflicts look exactly and which type of code changes lead to merge conflicts [1, 46, 61, 65]. Accioly et al. [1] investigated the structure of code changes that lead to merge conflicts with semi-structured tools. Their results show that in most of the conflicting merge scenarios, more than two developers are involved and code cloning can be a root cause of merge conflicts. Nishimura et al. [46] proposed MergeHelper, a tool that helps developers to find the root cause of merge conflicts by providing them with the historic edit operations that affected a given class member. Vale et al. [61] investigated the relation between GitHub communication activity and merge conflicts. As a result, they found no correlation between communication measures and the occurrence of merge conflicts. However, when investigating only the 10% largest merge scenarios, they found that merge scenarios' size (i.e., changed lines of code) and the number of developers involved influence the strength of the relation between GitHub communication activity and the occurrence of conflicts. Wuensche et al. [65] developed an approach to find potential higher-order merge conflicts (e.g., test and build conflicts) using a statistically constructed call graph which reuses data from previous runs to scale well with very large source code repositories. As a result, they did not find any test conflicts in

their 22 month analysis and they found that the top three causes of build conflicts are: 1) changes to the signature, 2) missing include statements, and 3) duplicated definitions.

Estimating the merge conflict resolution. A few studies have tried to measure the time/difficulty of resolution of merge conflicts [8, 62]. Brindescu et al. [8] conducted an *in-situ* observation of 7 developers resolving 10 merge conflicts. Their results show that developers search for information on seven sources (e.g., diff between merged versions and commit history), the conflicts resolution took from 40 to 2 190 seconds (36.5 minutes) and developers normally follow 6 steps to conflict resolution: (1) look at external data sources, (2) open a particular file to work on, (3) read or scroll through the source code, (4) edit source code, (5) read a chunk on either side, and (6) run the build or perform test. Vale et al. [62] conducted a mining and survey study to identify the challenges of resolving merge conflicts. As a result, they found that measures indirectly related to merge conflicts (i.e., measures related to the merge scenario changes) are stronger correlated with merge conflict resolution time than measures directly related to merge conflicts (i.e., merge conflict characteristics). Cross-validating their results, survey participants mentioned 25 measures used to quantify how hard/time-consuming is the resolution of merge conflicts mentioning measures indirectly related to merge conflicts. The challenges on merge conflict resolution includes: lack of coordination, lack of tool support, flaws in the system architecture, and lack of testing suite or pipeline for continuous integration.

Predicting merge conflicts. Looking at the main *venues* of software engineering (e.g., Transactions on Software Engineering, Empirical Software Engineering, and International Conference on Software Engineering) and searching for papers in the references of the selected papers (i.e., snowballing technique), we found six studies predicting merge conflicts. Leßenich et al. [40] investigated the correlation between seven source code measures and the likelihood of merge conflicts. Note that practitioners indicated these measures to be related to the emergence of merge conflicts (e.g., scattering degree among classes, commit density, and number of files). As a result, none of the investigated factors had a strong correlation with the occurrence of merge conflicts. Accioly et al. [2] computed recall and precision identifying merge conflicts related to two types of code changes (i.e., editions to the same method and editions to directly dependent methods). In addition, they manually investigated false positives and false negatives. Their results show recall and precision of 83.62% and 57.99% in the best case. Related to the manual analysis, they did not find a *silver bullet* to improve their predictions. Still, they realized that removing different spacing instances, decreases the number of false positives from 226 to 203. Owhadi-Kareshk et al. [47] tried to predict merge conflicts by building a classifier with nine measure sets (e.g., number of developers in a branch, number of simultaneously changed files in two branches, and number of added and deleted lines in a branch) for projects developed in seven programming languages (e.g., C, C#, Java, PHP, and Python). Their results agree with Leßenich et al., in most of the cases showing weak or no correlation

between subject metrics and the occurrence of merge conflicts. Only the number of simultaneously changed files in two branches had strong correlation for projects written in Java and PHP. Furthermore, the analysis using a *random forest* classifier successfully predicted merge conflicts (precision, recall, and f1-score of 1.00, 0.96, and 0.97 for safe merge scenarios and 0.63, 1.00, 0.68 for conflicting merge scenarios). Note that the *f1-score* of non-conflict scenarios is much higher than conflicting scenarios which suggests that it is easier to predict conflict-free merge scenarios than merge scenarios with conflicts. Rocha et al. [51] investigated whether it is feasible to use acceptance tests to predict files changed by programming tasks in Behaviour-Driven Development (BDD) projects. The idea behind is that choosing which tasks to work on in parallel, a development team could likely reduce conflict occurrence. As a result, they found that tests associated to a task might help to predict application files changed by developers responsible for the task. Furthermore, they found that the better the test coverage of a task, the better the predictive power. Dias et al. [18] conducted a study to understand merge conflicts three aspects of developers contributions: modularity, size, and timing. As a result, they found that: i) conflicts occur even when merging modular contributions, but the occurrence of merge conflicts increases when contributions are not modular (i.e., across model, view, and controller (MVC) layers), ii) large contributions involving more developers, commits, changed files are more likely associated with merge conflicts than small contributions, and iii) contributions over longer periods of time are more likely associated with conflicts than short ones. Trif et al. [59] predicted conflicts using machine learning (SVM, Naive Bayes, random forest) and deep learning (neural networks). As a result, their random forest analysis show 5 top factors that cause conflicts: 1) the number of parallel lines changed, 2) whether a pull request was opened before the merge, 3) the number of commits on a branch, 4) the active time of development, and 5) the minimum length of commit messages in a branch. Their best performance results were for random forest and neural networks with 0.75 and 0.77 of precision and 0.69 and 0.93 of recall, respectively.

Despite the number of studies investigating merge conflicts, we did not find studies investigating which developer roles are prone to introduce merge conflicts. Most studies that use some social measure (e.g., [1, 40, 47, 61]) look at the number of developers in the merged branches or in the merge scenario. Only Vale et al. [61] investigate the relationship among developers (i.e., their communication). Still, they do not classify developers into roles nor try to predict merge conflicts. Therefore, looking at the software engineering literature, we are the first study trying to understand and predict merge conflicts using social assets (especially the developer roles) in collaborative software development. Furthermore, only two studies use sophisticated machine learning based techniques to predict merge conflicts. Hence, we are the first study predicting merge conflicts by creating multiple classifiers with multiple machine learning based techniques and taking the social perspective into account.

2.3 Human Factor Investigations

There are several studies showing that human factors play an important role in software quality. These studies include investigations on developers productivity when they learn from experience of other developers individually, from groups, and from organisational-unit level [9] and the influence of the number of developers [42, 66], organisational structure [44], and code ownership [7, 13, 23, 28, 48, 49, 58] on the number of failures.

To mention a few of them, Bird et al. [7] investigated whether ownership influences the number of pre-release faults and post-release failures in the context of two commercial systems: Windows Vista and Windows 7. As a results, they found that: i) developers who owns less than 5% of lines of code of components (named minor contributors) is more likely to introduce pre- and post release failures, ii) higher levels of ownerships are related to fewer failures, iii) the number of minor contributors negatively affects software quality, and iv) without minor contributors, the ability to predict failure-prone components is greatly diminished, supporting the hypothesis that minor contributors are related to software quality. Similarly, Businge et al. [13] investigated the influence of ownership on the number of failures in the context of small-sized Android applications. As a result, concurring with Bird et al. [7], they found that minor contributors are related to more failures and applications with few major contributors are more reliable than applications with larger number of minor contributors. At the end, studies investigating the relation between code ownership and the number of failures found similar results and recommend that i) changes made by minor contributors should be reviewed with more scrutiny, ii) potential minor contributors should communicate desired changes to developers experienced with the respective file/binary, and iii) components with low ownership should be given priority by quality assurance resources.

Similar to these studies, we agree that human factors play an important role on software quality. Different from them, we investigate the influence of human factors on merge conflict prediction and not on the failure prediction.

3 Developer Roles

Previous work [6,15,20,34,35,43,50,57] had classified developers into core and peripheral roles aiming at understanding the organizational structure of open source projects. Mockus et al. [43] found empirical evidence for the *Mozilla browser* and the *Apache Web server* that a small number of developers are responsible for approximately 80% of the code modifications. Their approach consists of counting the number of commits made by each developer and then computing a threshold at the 80% percentile. Developers with a commit count above the threshold are considered core and, developers below the threshold are considered peripheral. They rationalized this threshold by observing that the number of commits made by developers typically follows a *Zipf* distribution (which implies that the top 20% of contributors are responsible for 80% of the

contributions) [15]. The *Zipf* distribution was also observed in other studies [20, 50, 57]. Other researchers used network metrics and analyzed core and peripheral developers over the project evolution [6,34,35]. For instance, Joblin et al. [34] empirically classified developers into core and peripheral to model the organizational structure using network metrics (e.g., degree- and eigenvector-centrality) and analyzed how the set of core developers changed over time.

Despite several studies classifying developers into roles, none of them analyze the influence of the developer roles on the emergence of merge conflicts. We use *top contributors* and *occasional contributors* classification instead of core and peripheral developers because, as suggested by a previous study [34], we consider that these terms better represent high- or low-frequency contributors, respectively. Similar to previous work [15, 43, 50, 57], we use the 80% percentile to classify top contributors (core). Furthermore, as suggested by previous work [34, 35], we recompute the developer roles for each merge scenario. Differently from them, we classify developers with distinct granularity: *project* and *merge-scenario level*.

Top and occasional contributors at project level classification. Top and occasional developers at project level are classified based on their code contributions on the whole project at the end of each merge scenario (i.e., at the merge commit). Practically, we follow 5 steps. First, for each merge commit we checked it out using the GIT CHECKOUT SHA command, where SHA is the identifier for the merge commit. Hence, for each merge commit, we run the GIT BLAME command to compute the authorship of each line of code in the whole project. Second, we sum up the lines of code each developer contributed creating a map where each developer has an unique identifier (key) and an object with the developer information as a value. This object includes an attribute informing the number of lines of code this developer changed in the whole project at the moment of the merge commit. Third, get the total lines of code in the project by summing all developer contributions. Fourth, we create a list of developers in descending order based on their code contributions (i.e., developers that contribute most are at the top of the list). Fifth, we get the top developers from the list until the sum of their contributions makes up 80% of the total contributions at merge commit time. These developers are classified as top contributors. All other developers are considered occasional contributors.

Top and occasional contributors at merge-scenario level classification. Top and occasional developers at merge-scenario level are classified based on their code contributions in a merge scenario. The classification at merge-scenario is similar to the project level, the only difference is in the first step. Instead of measuring the authorship of each developer in the whole project, we measure the code contribution of each developer in the merge scenario. In other words, for each merge commit, we measure only the lines of code changed between the base and merge commit. Hence, top contributors at merge scenario level are the developers that contribute to 80% of the changed lines of code in the merge scenario and all other developers are occasional contributors.

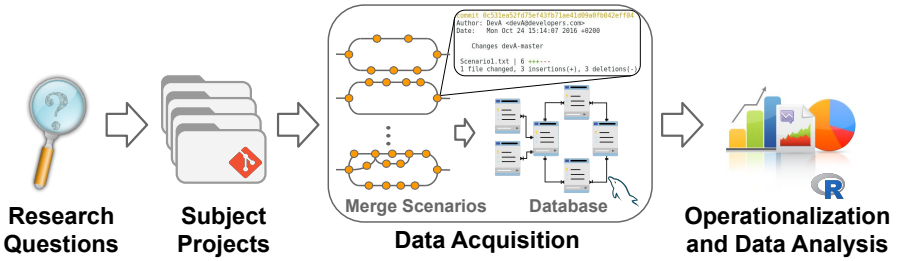

Fig. 2 Methodology Overview.

The distinction of project and merge-scenario level is essential because, while the developer roles at project level give a more global view of the code contributions, developer roles at merge-scenario level give a more focused view on merge scenario code changes and on merge conflicts. In Section 4.3, we describe the investigated measures as well as exemplify how the developer roles are computed in practice.

4 Study Setting

In Figure 2, we illustrate our four steps, which consist of (i) defining our goals and research questions, (ii) selecting subject projects, (iii) acquiring data, (iv) operationalizing and analyzing data. We describe these processes in the following four sections.

4.1 Goals and Research Questions

Our overall goals are threefold:

- **To understand which developer roles cause proportionally more merge conflicts**. Knowing which developer roles are more often involved in merge conflicts can: i) avoid or minimise conflicting merge scenarios since project coordinators and developer themselves can increase the coordination and communication where conflict-prone developer roles are working on. Hence, they can be aware sooner of other changes and fix conflicts in its earlier stages or even avoid them. For instance, as seen in the example merge scenario presented in Section 1, properly coordinating specific developer roles (i.e., making them aware or other changes and communicate with each other) can be enough for avoiding merge conflicts and ii) support on the conflict resolution since project coordinators and developer themselves can increase the communication of developer roles often involved in conflict to support the merge conflicts resolution.
- **To find whether it is feasible to predict merge conflicts using only social measures**. Showing that it is possible to predict merge conflicts using social measures can minimise the number of speculative merging as

motivated in Section 1, but also highlight the importance of social measures in software analysis. Hence, we show evidence of why researchers should consider social measures more often in their analyses.
– **To find whether combining social and technical assets improve the state-of-the-art of predicting merge conflicts**. Previous work has predicted merge conflicts using technical measures, adding the social perspective might improve previous results improving the state-of-the-art of merge conflict predictions.

We investigate the relationship between the developer role and the emergence or avoidance of merge conflicts in four ways, represented by the following research questions:

RQ$_1$: *Which developer role is more often related to merge conflicts considering project and merge-scenario level* ***separately****?*

RQ$_{1.1}$: *Are top contributors at project level proportionally related to more merge conflicts than occasional contributors?*

RQ$_{1.2}$: *Are top contributors at merge-scenario level proportionally related to more merge conflicts than occasional contributors?*

RQ$_2$: *Which combination of developer roles is related to merge conflicts* ***combining*** *project and merge-scenario level classification?*

RQ$_3$: *Are merge conflicts predictable using only social measures?*

RQ$_4$: *Is a model combining social and technical measures better than a model composed of only social measures to predict merge conflicts?*

Note that the first research question is simple such that developers can identify developer roles without tool support. In the second research question, we increase the complexity, but developers with a comprehensive understanding of the project can still identify developer roles without tool support. In the third and fourth research questions, we use more information and a more sophisticated approach. It makes manual identification difficult. Answering these four research questions, we expect an actionable insights overview of the influence of the subject developer roles on the occurrence of merge conflicts, especially when triangulating social and technical measures.

4.2 Subject Projects

We selected the corpus of subject projects by retrieving the 100 most popular projects on GitHub, as determined by the number of stars [10] and, then, we applied the following four filters created based on Kalliamvakou et al. work [38]: (i) projects that do not have a classified programming language as the main file extension since we are interested in programming language projects; (ii) projects with less than two commits per month in the last six months, since we are interested in active community projects on GitHub; (iii) projects in which it was not possible to reconstruct at least 50% of the merge scenarios,

since we are interested in projects that use the three-way merge pattern in the majority of integrations (see Section 2.1). The inclusion of projects that follow other development patterns could bias our analysis. In Section 4.3, we detail how we rebuilt merge scenarios; and, (iv) balancing the programming language of projects consists of excluding less popular JavaScript projects until they are not the majority of subject projects. Including most projects of a programming language could bias our analysis, as we explain in Section 7.

We restricted our selection to GitHub because it is one of the most popular platforms to host repositories, and it has been investigated and used in prior work [16,27,55,56,60–62]. We limited our analysis to Git repositories because it simplifies the identification of merge scenarios in retrospect and is a popular practice as well.

After applying all filters, we obtained 66 projects, developed in 12 programming languages (e.g., JavaScript, Python, Java, Go, and C++), containing 78 740 merge scenarios that involve more than 1.5 million files changed, 10.4 million chunks, and 3 950 conflicting merge scenarios. bootstrap[2], react[3], TypeScript[4], redis[5], and lantern[6] are examples of selected projects. In Figure 3, we show the distribution of merge scenarios (ms), conflicting merge scenarios (cms), number of files, number of chunks, number of commits, and number of developers by each subject project. In other words, each project represents a dot in the graphs and the number of files, for instance, is the sum of all files of a project. The complete list of projects with URL, programming language, and descriptive statics is available at our supplementary Web site [63].

4.3 Data Acquisition

In this section we show: i) how we acquire data for each merge scenario, ii) details about the developer roles classification, iii) the investigated measures, iv) how we computed the investigated measures, v) an example of how the investigated measures are computed, and vi) where our data and framework is available.

Acquiring data. We followed a similar approach from previous work [61, 62] to acquire data from merge scenarios which consists of the following 5 steps. First, we cloned a subject project's repository. Second, we got all merge commits by filtering commits with multiple parent commits. Third, we retrieved the base commit (i.e., the common ancestor for both parent commits), for each merge commit (see Section 2.1). Fourth, we rebuilt merge scenarios by (re)merging parent commits and retrieving information from each

[2] `https://github.com/twbs/bootstrap`
[3] `https://github.com/facebook/react`
[4] `https://github.com/microsoft/TypeScript`
[5] `https://github.com/antirez/redis`
[6] `https://github.com/getlantern/lantern`

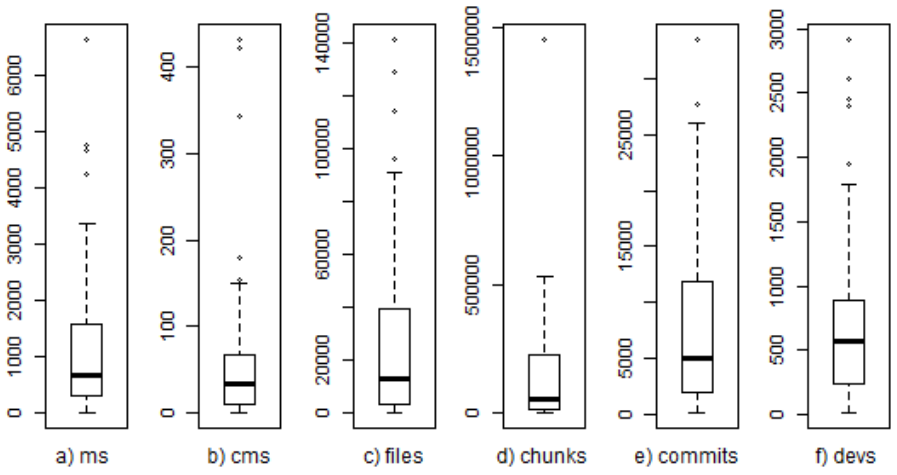

Fig. 3 Descriptive statistics by subject project

commit between the base commit and the merge commit for each merged branch.

Commit information includes author, date, lines of code, and files changed. Therewith, we know which developer (by the commit's author) changed each line of code at each branch. Finally, we stored all data and repeated steps 3 and 4 for each merge scenario found in step 2. Note that we excluded merge scenarios that do not have a base commit (e.g., fast-forward, rebase, or squash integrations [37]), and we ignored binary files because we cannot track changes from them. It is important to highlight that all investigated merge scenarios integrate only two branches (i.e., no octopus merges).

At the end, we obtained a *set* of developers for each merge scenario. For each developer, we retrieved: 1) a unique identifier, 2) the merge scenario identifier, 3) a boolean flag demonstrating if it is or not a conflicting merge scenario, 4) the list of files touched in the target branch, 5) the list of files touched in the source branch, 6) the number of chunks changed in the target branch, 7) the number of chunks changed in the source branch, 8) the number of lines of code changed in the target branch, 9) the number of lines of code changed in the source branch, 10) the number of commits in the target branch, and 11) the number of commits in the source branch.

Classifying developers. To classify developers into top and occasional, we followed the approach described in Section 3. Note that at project level we consider developers contribution in the whole project for each merge commit. Hence, top and occasional contributors at project level might not be active developers in a given merge scenario. By active developers, we mean developers that touched (i.e., created, edited, or deleted) one of the integrated branches of a merge scenario. At merge-scenario level, we consider only code contributions in a merge scenario. Hence, all top and occasional contributors at merge-scenario level are active developers.

Investigated measures. In Table 1, we present the investigated measures. The reasoning behind our choice is related to three factors: i) fine-grained measurement, ii) already used in the literature, and iii) inexpensive computation.

Fine-grained measurement. Considering that the code contributions are normally different on the merged branches which may influence either the occurrence of merge conflicts as well as the developers' role that contribute to the branch [14, 26], we differentiate contributions from *target* and *source* branches for all investigated measures.

Already used in the literature. We selected the measures by surveying the literature on merge conflicts and related areas, such as code evolution or software maintenance (see the Reference column of Table 1). Furthermore, developers reported that most of the selected measures are useful to identify merge conflicts [40]. Note that measures found in the literature are often coarse-grained (i.e., ignore the branch contributions that happened).

Inexpensive computation. We selected measures which extraction is computationally inexpensive aiming at making the prediction used in practice.

Computing measures. To get the measures from a merge scenario, we basically aggregate measures from active developers (i.e., the set mentioned before). For instance, to come up with the value of loc_t, we aggregate the number of source lines of code (i.e., excluding blanks and comments) in the target branch of all developers for a given merge scenario. The counting of *loc* is part of our framework and follows a similar implementation of CLOC TOOL[7]. As another example, to come up with the value of *devs* from a merge scenario, we got a set of all active developers (represented by the unique identifier) of a merge scenario (represented by the merge scenario identifier).

Exemplifying computation of measures. From the example presented in Section 2.1, we see that three files changed in this merge scenario (*files* - File 1, File 2, and File 3) where two changed in the target branch ($files_t$ - File 1 and File 2) and two files changed in the source branch ($files_s$ - File 1 and File 3). The number of chunks is six (*chunks*) where four chunks are in the target branch ($chunks_t$ - two chunks in File 1 from DEV A and two in File 2 from DEV A and DEV D) and four chunks are in the source branch ($chunks_s$ - two chunks in File 1 from DEV C and DEV B and two in File 3 from DEV A and DEV B). The number of lines of code is twelve (*loc*) where seven are in the target branch (loc_t) and five are in the source branch (loc_s). The number of commits is five (*commits*) where two are in the target branch ($commits_t$ - hashes: 923E4D5 and 20BBDF7) and three are in the source branch ($commits_s$ - hashes: A562FA6, 35DBC8F, and 0E8F458).

In Table 2, we illustrate the number of lines of code each developer contributed at the moment of the merge commit (hash: C2ECB2C) at project and merge-scenario level. Despite of in the beginning of the merge scenario, Dev X committed 26 lines of code (hash FF1E147 - 5 *loc* in File 1, 5 *loc* in File 2, 4 *loc* in File 3, and 12 *loc* in File 4), until the merge commit DEV A, DEV B, DEV C, and DEV D changed 8, 2, 1, and 1 lines of code, respectively. Hence,

[7] `https://cloc.sourceforge.net/`

Table 1 Variables of our study

Variable	Description	References
	Dependent variable	
has_conflict	Boolean informing if the merge scenario has conflicts	[40, 61]
	Independent (Technical) variables	
files	Number of files touched in the merge scenario	[61, 62]
$files_t$	Number of files touched in the target branch	[22, 39, 40, 47, 52]
$files_s$	Number of files touched in the source branch	[22, 39, 40, 47, 52]
$files_{t\&s}$	Number of files touched in the target and source branches	[40]
chunks	Number of chunks touched in the merge scenario	[40, 61, 62]
$chunks_t$	Number of chunks touched in the target branch	
$chunks_s$	Number of chunks touched in the source branch	
loc	Number of source lines of code touched in the merge scenario (i.e., code churn)	[61, 62]
loc_t	Number of source lines of code touched in the target branch	[22, 40, 47]
loc_s	Number of source lines of code touched in the source branch	[22, 40, 47]
commits	Number of commits created in the merge scenario	[22, 39, 40, 47, 61]
$commits_t$	Number of commits created in the target branch	
$commits_s$	Number of commits created in the source branch	
	Independent (Social) variables	
top_p	Number of top contributors at project level	[15, 43, 50, 57]
$top_{p\&t}$	Number of top contributors at project level contributing to the target branch	
$top_{p\&s}$	Number of top contributors at project level contributing to the source branch	
occ_p	Number of occasional contributors at project level	[15, 43, 50, 57]
$occ_{p\&t}$	Number of occasional contributors at project level contributing to the target branch	
$occ_{p\&s}$	Number of occasional contributors at project level contributing to the source branch	
top_{ms}	Number of top contributors at merge-scenario level	
$top_{ms\&t}$	Number of top contributors at merge-scenario level contributing to the target branch	
$top_{ms\&s}$	Number of top contributors at merge-scenario level contributing to the source branch	
occ_{ms}	Number of occasional contributors at merge-scenario level	
$occ_{ms\&t}$	Number of occasional contributors at merge-scenario level contributing to the target branch	
$occ_{ms\&s}$	Number of occasional contributors at merge-scenario level contributing to the source branch	
devs	Number of active developers in a merge scenario	[61, 62]
$devs_t$	Number of active developers in the target branch	[21, 22, 39, 47]
$devs_s$	Number of active developers in the source branch	[21, 22, 39, 47]
$devs_{t\&s}$	Number of active developers in target and source branches	

Table 2 Developer code contributions at project and merge-scenario level

Developer	Project	Merge-Scenario
Dev X	17	-
Dev A	8	8
Dev B	2	2
Dev C	1	1
Dev D	1	1
Total loc	**29**	**12**

at the merge commit, Dev X authored 17 lines of code (3 *loc* in File 1, 1 *loc* in File 2, 1 *loc* in File 3, and 12 *loc* in File 4). Note that changes from DevA, Dev B, and Dev C are concurrently causing merge conflicts. Note that Dev X is not an active developer in the exemplified merge scenario since she does not commit between the base and merge commit.

At the merge commit the project had 29 lines of code. Hence, the first developers with the large number of lines of code that touched 23 lines of code are classified as top contributors. Hence, Dev X and Dev A are top contributors and Dev B, Dev C, and Dev D are occasional contributors at project level. We see that 12 lines of code changed in the exemplified merge scenario. Hence, developers that touched 10 lines of code are classified as top contributors. Hence, Dev A and Dev B are top contributors and Dev C and Dev D are occasional contributors at merge scenario level.

Looking at the social measures we see that 4 developers are active in this merge scenario (*devs* - Dev A, Dev B, Dev C, and Dev D) where two touched the target branch ($devs_t$ - Dev A and Dev D) and three touched the source branch ($devs_s$ - Dev A, Dev B, and Dev C). The only developer that touched both target and source branches is Dev A ($devs_{t\&s}$). The number of top contributors at project level is one (top_p - Dev A) since despite of Dev X was classified as top contributor, she is not active in the illustrated merge scenario. As Dev A contributed to the target and source branch, $top_{p\&t}$ and $top_{p\&s}$ is one. The number of occ_p is three (Dev B, Dev C, and Dev D) where Dev D contributed to the target branch (i.e., $occ_{p\&t}$ is one) and Dev B and Dev C contributed to the source branch (i.e., $occ_{p\&t}$ is two). Looking at measures at merge-scenario level, the number of top developers is two (top_{ms} - Dev A and Dev B) where Dev A contributed to the target branch ($top_{ms\&t}$) and Dev A and Dev B contributed to the source branch ($top_{ms\&s}$). The number of occasional contributors is two (occ_{ms} - Dev C and Dev D) where Dev D contributed to the target branch ($occ_{ms\&t}$) and Dev C contributed to the source branch ($occ_{ms\&s}$).

Framework and Data Availability. Our data mining (Java) and analysis frameworks (Python) are open-source. All data necessary for replicating this study are stored in a MySQL database and replicated on spreadsheets (.csv files). All tools, links to subject projects, subject projects filtering process, and data are available at our supplementary Web site [63].

4.4 Operationalization

The operationalization of RQ_1 and RQ_2 consists of getting the set of merge scenarios that a given developer role participated in (*#MS*), a subset of these merge scenarios which have merge conflicts (#Conf. MS), and share of conflicting merge scenarios (i.e., #MS × #Conf. MS) investigated in each research question (see Section 4.1). For instance, in $RQ_{1.1}$ we want to find a subset of merge scenarios from all 78 740 investigated merge scenarios that have top contributors contributing to both *target* and *source* branches. From this subset, we get the number of merge scenarios that have merge conflicts and compute the share of conflicting merge scenarios. We performed a chi-square test to verify whether the developer role (top and occasional) differs significantly. The chi-square test is adequate because we have large and unpaired data (i.e., the number of merge scenarios varies depending on the developer role), variables under analysis are categorical (e.g., top- or occasional-contributors), and the outcome is binomial (i.e., conflicting or safe merge scenarios). The null and alternative hypotheses for RQ_1 and RQ_2 are:

H$_0$*: Developers' role and emergence of merge conflicts are independent.*

H$_a$*: Developers' role and emergence of merge conflicts are not independent.*

where the p-value is below 0.01 (i.e., 99% significance level), we reject the null hypothesis (H_0) and accept the alternative hypothesis (H_a). Accepting the alternative hypothesis suggests that the variables are related, but the relationship is not necessarily causal. As we measured several attributes for each merge scenario, we grouped them using set operations (e.g., union) and treated each influencing factor separately. Aiming at getting a baseline for comparison of our results, we compared the results of each developer role with the overall average of conflicting merge scenarios for all merge scenarios in analysis. Thus, we increased the knowledge over our data and internal validity.

The operationalization of RQ_3 and RQ_4 consists of using data acquired as described in Section 4.3 and follows three steps: (i) to balance our data since merge conflicts happen in only the minority of merge scenarios, (ii) to select the target measures (i.e., features), and (iii) to predict conflicting scenarios using three classifiers. We used multiple balancing techniques, sets of measures, and classifiers to show practitioners which configurations perform better on our data. For data balancing, we chose seven techniques (under, over, both, SMOTE, BorderlineSmote, SVMSmote, Adasyn). For feature selection, in RQ_3, we created a model using only the social measures presented in Table 1 as we want to investigate the prediction of merge conflicts using only social assets. In RQ_4, we created two models, one using only the technical measures and the other one all measures presented in Table 1. We build these two models to be able to compare our results with the model created for RQ_3 and with a previous study [47]. To predict conflicts, we chose three classifiers (decision tree, random forest, and KNN), because they are simple yet achieve good results for binomial classification. Due to the importance of hyper-parameters, we used

grid-search with 10-fold cross-validation to find the right hyper-parameters to use. For each classifier, we tuned it using all possible hyper-parameters. For instance, for *decision tree*, we set the hyper-parameters: *max_depth* (10, 50, 80, 100, 150, 200), *max_feature* (auto, sqrt, log2), *min_samples_split* (2, 3, 5, 10), *min_samples_leaf* (1, 2, 3, 5, 10), *criterion* (gini, entropy) and, *splitter* (best, random). The complete list of hyper-parameters and tuning values, as well as, a description of each balancing technique and classifier are available at our supplementary Web site [63].

Performance Measures. We showed *precision*, *recall*, and *f1-score* for conflicting and safe merge scenarios. Furthermore, even though the previous work [47] mentioned that *accuracy* is not a good performance measure when dealing with a discrepant difference between the majority and minority classes, we also showed *accuracy* and area under the curve (AUC) for our general predictions. Presenting results for conflicting and safe scenarios provides a complete view of how a detector would perform in practice than only presenting general measures. In our case, it is desirable to have higher recall than higher precision for conflicting scenarios since it is better to predict all conflicting scenarios and some false-positives (i.e., reported as conflicting scenarios, but they are safe in practice) than miss some conflicting scenarios (i.e., true-negatives). In other words, it is better to suggest speculative merges for some safe-scenarios than ignore some real conflicting scenarios. We considered f1-score the second most relevant performance measure since its computation combines precision and recall. In cases where we found the same value for the performance measures, we present the results for the model with lower values for the hyper-parameters.

5 Results

In this section, we present the results structured according to our research questions. Overall, we investigated 78 740 merge scenarios of which 3 950 of them have merge conflicts. It corresponds to an average of 5.02% conflicting merge scenarios. We use this percentage in RQ_1 and RQ_2 to compare if a developer role is above or below the general average.

5.1 RQ_1. Which developer role is more often related to merge conflicts considering project and merge-scenario level separately?

We answer this question by looking at data from project and merge-scenario level separately. In Table 3, we present the general result for $RQ_{1.1}$ and the results for each branch. Top contributors at project level contributed to the target and source branches in 45 297 merge scenarios and 3 290 of them have merge conflicts. It represents a share of 7.26% of conflicting merge scenarios. Occasional contributors at project level contributed to 60 609 merge scenarios, 3 409 of them have merge conflicts. It represents a share of 5.62% of conflicting

Table 3 Top and occasional contributors at project level contributions overview

Branch	Dev. Role	#MS	#Conf. MS	%Conf.MS
Target & Source	Top	45 297	3 290	**7.26%**
	Occ.	60 609	3 409	5.62%
Target	Top	33 872	2 301	**6.79%**
	Occ.	46 826	2 745	5.86%
Source	Top	20 925	2 590	**12.38%**
	Occ.	35 086	2 970	8.46%

***Dev.**: developer, **#MS**: number of merge scenarios and **#Conf.MS**: number of conflicting merge scenarios*

Table 4 Top and occasional contributors at merge-scenario level contributions overview

Branch	Dev. Role	#MS	#Conf. MS	%Conf.MS
Target & Source	Top	75 142	3 623	4.82%
	Occ.	21 751	2 880	**13.24%**
Target	Top	62 214	3 039	4.88%
	Occ.	17 100	1 424	**8.32%**
Source	Top	53 812	2 344	4.36%
	Occ.	8 023	1 974	**24.60%**

***Dev.**: developer, **#MS**: number of merge scenarios and **#Conf.MS**: number of conflicting merge scenarios*

merge scenarios. Note that it does not need to be the same developer. It just needs to have at least one given developer role contributing to the target branch and at least one developer contributing to the source branch. With the chi-square test ($X\text{-}squared=103.01$, $df=1$, $p\text{-}value < 2.2^{e-16}$), we reject the null hypothesis and accept the alternative hypothesis. Thus, we conclude that there is a relationship between the developer role and the emergence of merge conflicts. We found a similar result for the target and source branches.

In Table 4, we present the general result for $RQ_{1.2}$ and the results for each branch. Top contributors at merge-scenario level contributed to 75 142 analyzed merge scenarios and 3 623 conflicting merge scenarios. It represents a share of 4.82% of conflicting merge scenarios. Occasional contributors at merge-scenario level contributed to 21 751 merge scenarios and to 2 880 conflicting merge scenarios. It represents a share of 13.24% of conflicting merge scenarios. With the chi-square test ($X\text{-}squared=1600.4$, $df=1$, $p\text{-}value < 2.2^{e-16}$), we reject the null hypothesis and accept the alternative hypothesis. Thus, we conclude that there is a relationship between the developer role and the emergence of merge conflicts. We also found a similar result for the target and source branches.

Comparing the results with the general average, we observed that contributors at project level have a greater percentage for all the cases. For instance, top and occasional contributors have a share of 7.26% and 5.62% conflict-

ing scenarios, respectively. For developer roles at merge-scenario level, we see that occasional contributors have a share of conflicting merge scenarios above the general average (between 8.32%–24.60%) while top contributors do not (between 4.36%–4.88%).

We expected that top contributors are related to more merge conflicts than occasional contributors since the more code a developer changes, the greater the chance of happening conflicting merge scenarios. However, our results of $RQ_{1.2}$ shows that, at merge-scenario level, occasional contributors are more often involved in conflicting merge scenarios than top contributors. To illustrate, let us consider a merge scenario with 4 developers changing 100 lines of code. Dev A changed 50 lines of code (50% chance to be related to conflicts), Dev B changed 40 lines of code (40%), Dev C and Dev D changed 5 lines of code each (5% each). In this merge scenario, Dev A and Dev B are top contributors and Dev C and Dev D are occasional contributors. Note that the chance of Dev C and Dev D being in merge conflict is only 10%. Nevertheless, even despite this small chance, these occasional contributors were responsible for all conflicting changes.

RQ_1 Summary: At project level, top contributors are related proportionally more to conflicting merge scenarios than occasional contributors, and occasional contributors collaborate to more merge scenarios than top contributors. At merge-scenario level, the share of conflicting merge scenarios is greater for occasional contributors than for top contributors. Around one quarter of the contributions of occasional contributors at merge-scenario level in the source branch are related to merge conflicts.

5.2 RQ_2. Which combination of developer roles is related to merge conflicts **combining** project and merge-scenario level classification?

In Table 5, we present the general result for RQ_2 and the results for each branch. As expected, merge scenarios with top contributors at project touching the target and the source branches that are also top contributors at merge-scenario level occurred more often than merge scenarios with top contributors at project level touching the target and source branches and occasional contributors at merge-scenario level (44 497 against 15 834). However, when looking at the proportion of conflicting merge scenarios, top contributors at project level that are occasional contributors at merge scenario level have the higher percentage (15.76%) than all other developer roles. With the chi-square test ($X\text{-}squared = 1229.6$, $df=3$, $p\text{-}value < 2.2^{e-16}$), we reject the null hypothesis and accept the alternative hypothesis. Thus, we conclude that there is a relationship between the developer role and the emergence of merge conflicts. We found a similar result for the target and source branches.

Table 5 Top and occasional contributors combining project and merge-scenario level contributions overview

Branch	Dev. Role	#MS	#Conf. MS	%Conf.MS
Target & Source	$Top_p \circ top_{ms}$	44 497	3 070	6.90 %
	$Top_p \circ occ_{ms}$	15 834	2 496	**15.76%**
	$Occ_p \circ top_{ms}$	57 053	3 084	5.41 %
	$Occ_p \circ occ_{ms}$	21 195	2 800	13.21 %
Target	$Top_p \circ top_{ms}$	23 728	1 579	6.65 %
	$Top_p \circ occ_{ms}$	11 268	1 164	**10.33%**
	$Occ_p \circ top_{ms}$	31 009	1 880	6.06 %
	$Occ_p \circ occ_{ms}$	16 188	1 329	8.21 %
Source	$Top_p \circ top_{ms}$	11 184	1 678	15.00 %
	$Top_p \circ occ_{ms}$	4 943	1 597	**32.31%**
	$Occ_p \circ top_{ms}$	18 793	1 933	10.29 %
	$Occ_p \circ occ_{ms}$	1 914	1 974	24.93 %

***Dev.**: developer, **#MS**: number of merge scenarios and **#Conf.MS**: number of conflicting merge scenarios*

RQ_2 Summary: Looking at developer roles at project and merge-scenario levels together, we found that merge scenarios with top contributors at project level and occasional contributors at merge-scenario level touching the target and source branches have the greatest share of conflicting merge scenarios in general and for the analysis of both branches. Surprisingly, around one-third of the contributions of these developer roles in the source branch are related to merge conflicts. It is surprising because it represents six times more than the general average (i.e., without considering developer roles) and three times more the contributions of the same developer roles in the target branch.

5.3 RQ_3. Are merge conflicts predictable using only social measures?

When answering RQ_3 and RQ_4, we present only the best performance results according to the criteria described in Section 4.4. Presenting only a few results is necessary since we have results for a combination of three models (social vs. technical vs. social and technical measures), seven balancing techniques (i.e., under-, over-, both-, SMOTE-, BorderlineSmote-, SVMSmote-, Adasyn-sampling), and three classifiers (i.e., decision tree, random forest, and KNN). The complete results can be seen in our supplementary Web site [63].

In Table 6, we present the results of our predictions using social measures for each classifier highlighting the best balancing techniques. By best balancing techniques, we mean the balancing techniques that balanced our data in a way that made our classifiers perform better. Reinforcing, we use the recall and f1-score of conflicting scenarios as our main performance measures (see Section 4.4). Hence, once we got the best setup (i.e., the combination of

Table 6 Performance overview for **social** measures

Classifier	Bal. Tech.	Scenario	R	F1	P	Acc.	AUC
Decision Tree	Over	Safe	0.60	0.75	**1.00**	0.62	0.80
		Conflicting	0.99	0.21	0.12		
Random Forest	Under SMOTE Adasyn	Safe	0.58	0.74	**1.00**	0.60	0.79
		Conflicting	**1.00**	0.20	0.11		
KNN	Adasyn	Safe	0.72	**0.83**	**1.00**	**0.73**	**0.83**
		Conflicting	0.94	0.26	0.15		

***Bal. Tech.**: balancing technique, **R**: recall, **F1**: f1-score, **P**: precision, and **Acc.**: accuracy*

classifier and balancing technique) for conflicting scenarios, we highlight their results.

For the predictions of conflicting scenarios using only social measures, the setup with random forest performed better when using balanced data from under, SMOTE, or Adasyn-sampling technique. With this setup, we achieve a recall, f1-score, and precision of 1.00, 0.26, and 0.15, respectively. Regarding safe scenarios, we found a recall, f1-score, and precision of 0.72, 0.83, and 1.00, respectively. In terms of accuracy and AUC this setup achieved 0.60 and 0.79.

Note that the setup using KNN classifier with data from Adasyn-sampling technique achieved better accuracy and AUC are 0.73 and 0.83 than the setup with better recall. Furthermore, note that none of the setups achieved high f1-score and precision for conflicting scenarios (all values below 0.3).

RQ_3 Summary: Using only social measures we created a model in which the random forest classifier achieved 1.00 of recall. Therefore, we conclude that it is possible to predict conflicting merge scenarios using only social measures. Classifying all conflicting scenarios correctly means that we can reduce speculative merging considerably without missing any real conflicting scenarios. In any event, we highlight the low precision of our model which leads to an open challenge of increasing the precision of models using only social measures.

5.4 RQ_4. Is a model combining social and technical measures better than a model composed of only social measures to predict merge conflicts?

Before looking at the results combining social and technical measures, we present the results of a model using only technical measures. As mentioned, we created this model aiming at increasing our understanding on the models as well as fomenting discussions. In Table 7, we present the results of our predictions, similar we did when answering RQ_3. For the predictions of conflicting scenarios, the setup using random forest classifier and balanced data from SMOTE- or Adasyn-sampling techniques performed better. With this

Table 7 Performance overview for **technical** measures

Classifier	Bal. Tech.	Scenario	R	F1	P	Acc.	AUC
Decision Tree	Over Both SMOTE BorderlineSmote Adasyn	Safe Conflicting	0.88 **1.00**	0.93 0.46	**1.00** 0.30	0.88	0.94
Random Forest	SMOTE Adasyn	Safe Conflicting	0.92 **1.00**	**0.96** 0.56	**1.00** 0.39	**0.92**	**0.95**
KNN	Adasyn	Safe Conflicting	0.75 0.93	0.85 0.28	**1.00** 0.16	0.76	0.84

***Bal. Tech.**: balancing technique, **R**: recall, **F1**: f1-score, **P**: precision, and **Acc.**: accuracy*

setup, we achieved a recall, f1-score, and precision of 1.00, 0.56, and 0.39, respectively. Regarding safe scenarios, we found a recall, f1-score, and precision of 0.92, 0.96, and 1.00, respectively. In terms of accuracy and AUC, we found 0.92 and 0.95.

Note that we found the maximum value for the model using only social measures in terms of recall for conflicting scenarios. However, the value for other performance measures increases in the model using technical measures. For instance, f1-score and precision for conflicting scenarios increase from 0.26 to 0.92 and from 0.15 to 0.39, respectively. We also see an increase for safe scenarios and general measures. For instance, the accuracy for the social and technical models are 0.73 and 0.92, respectively.

In Table 8, we present the predictions of our model using social and technical measures similar to Table 6 and Table 7. For the predictions of conflicting scenarios, the setup using random forest classifier and balanced data from under- or over-sampling techniques performed better. With this setup, we found a recall, f1-score, and precision of 1.00, 0.56, and 0.39, respectively. Regarding safe scenarios, we found a recall, f1-score, and precision of 0.92, 0.96, and 1.00 for the same setup, respectively. In terms of accuracy and AUC, we found 0.92 and 0.96, respectively.

As seen, the results of a model using only technical measures and the other using all (social and technical) measures are basically the same. Only the AUC increased from 0.95 to 0.96. For the technical and all measures models, the random forest classifier performed slightly better than the other classifiers and the data from under- and over-sampling presented better results than the data from other balancing techniques.

Observing that no real improvements were obtained adding the technical and social measures, in Figure 4 we show the correlation-matrix to identify whether the investigated measures correlate with each other. Be aware that correlating pairs of investigated variables provide a limited and simpler viewpoint compared to the machine learning classifiers predictions. As we can see in Figure 4, some social measures are correlated with each other and also with some technical measures. For instance, occ_p has a high positive correla-

Table 8 Performance overview for **all (technical and social)** measures

Classifier	Bal. Tech.	Scenario	R	F1	P	Acc.	AUC
Decision Tree	Over	Safe	0.87	0.93	**1.00**	0.87	0.93
		Conflicting	**1.00**	0.44	0.29		
Random Forest	Under Over	Safe	0.92	**0.96**	**1.00**	**0.92**	**0.96**
		Conflicting	**1.00**	0.56	0.39		
KNN	Adasyn	Safe	0.73	0.84	0.99	0.74	0.82
		Conflicting	0.92	0.27	0.16		

***Bal. Tech.**: balancing technique, **R**: recall, **F1**: f1-score, **P**: precision, and **Acc.**: accuracy*

tion with *devs* (0.78) and $occ_{p\&t}$ (0.73) and a moderate positive correlation with $occ_{p\&s}$ (0.65), occ_{ms} (0.64), *commits* (0.60), $occ_{ms\&t}$ (0.55). All correlations were computed using Spearman-rank based correlation with 95% of confidence level. Spearman-rank based correlation is invariant for linear transformations of covariates and is simple and useful to understand the relation among our covariables [33]. Having social measures correlated with each other might have provided similar information to the social model not improving its performance. Having social measures related to technical measures made the addition of social measures to the technical model, introducing only information that technical measures had already provided. We come back with a discussion on this topic in Section 6.2.

RQ_4 Summary: In terms of recall, the three models we built (only social vs. only technical vs. all measures) found 1.00 of recall for conflicting scenarios (i.e., they were able to retrieve all real conflicting scenarios). Considering accuracy, AUC, f1-score, and precision for conflicting and safe scenarios, the models using only technical and all social and technical measures performed better than those using only social measures.

6 Discussion

We divide this section into three parts. First, we compare our results with previous work predicting merge conflicts. Second, we present a reflection upon our results. Finally, we present implications of our results and findings to practitioners, researchers, and tool builders.

6.1 Comparing Results

As mentioned in Section 2.2, there are six studies predicting merge conflicts. As the approach and results from Accioly et al. [2], Leßenich et al. [40], Rocha et al. [51], and Dias et al. [18] differ significantly from ours, it is not fair comparing our results. For instance, while we compare developer roles and the used machine learning classifiers to predict conflicts, Accioly et al. [2] computed

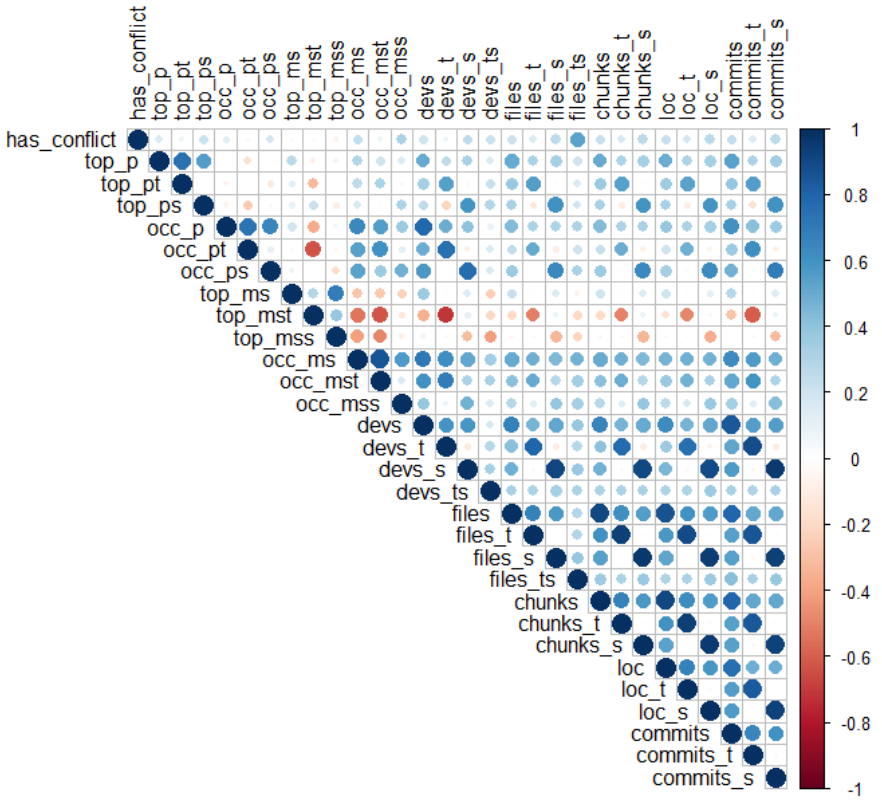

Fig. 4 Correlation matrix of investigated variables.

recall and precision to identify merge conflicts related to two types of code changes. Hence, even though they also compute recall and precision, our results are not comparable. Despite of Trif et al. [59] used machine learning like us, they present just a general recall and precision, i.e., they do not differ safe and conflicting scenarios. Furthermore, they do not show f1-score and AUC. Hence, we opted to not compare their results with ours. Owhadi-Kareshk et al. [47], on the other hand, used machine learning classifiers like us and present recall, precision, and f1-score for safe and conflicting scenarios making our results comparable. Even though we use a different set of measures/variables, subject projects, and they present the results by programming language, we consider our results comparable.

In Table 9, we present the results for the performance measures presented in their study (i.e., recall, f1-score, and precision) which is a subset of our performance measures. Aiming at providing a fair comparison, we show the interval of their results by programming languages for the random forest classifier. Similar to our study, the random forest classifier was the classifier that performed better. Looking at safe scenarios, they presented better recall, similar f1-score, and lower precision. Looking at conflicting scenarios, we presented

Table 9 Comparison of our results with the results of Owhadi-Kareshk et al. [47]

Study	Scenario	Recall	F1-score	Precision
Owhadi-Kareshk et al. [47]	Safe	**[0.93,0.96]**	**[0.95,0.97]**	[0.97,0.98]
Ours		0.92	**0.96**	**1.00**
Owhadi-Kareshk et al. [47]	Conflicting	[0.68,0.83]	**[0.57,0.68]**	**[0.48,0.63]**
Ours		**1.00**	0.56	0.39

higher recall and lower f1-score and precision. It is important to mention that we focus on increasing recall of conflicting merge scenarios since missing real conflicting scenarios might damage speculative merge tools and hurt users' confidence on the predictions making them stop using tool support [30]. Hence, we consider essential to retrieve all real conflicting scenarios. This choice made us decrease precision. In other words, we ensure that all real conflicting scenarios were correctly classified, but we classified some safe scenarios as conflicting scenarios (see Section 4.4).

6.2 Reflecting on Results

Occasional contributors are more related to conflicting scenarios than top contributors. As mentioned, we expected that top contributors are related to more merge conflicts than occasional contributors since the more code a developer changes, the greater the chance of happening conflicting merge scenarios. However, our results when answering RQ_1 (see Table 4) show the opposite. In other words, at merge-scenario level, occasional contributors are conflict-prone when compared to top contributors. For instance, when looking at the source branch, the percentage of conflicting merge scenarios for occasional contributors is 24.60%, while for top contributors the percentage of conflicting merge scenarios is only 4.36%. We speculate that there may be two reasons for this phenomenon: i) occasional contributors normally change more code than necessary to address a task, such as fix a bug and ii) occasional contributors take more time than necessary to complete a task. We plan an *in-situ* investigation in future work to draw a conclusion about it.

One-third of scenarios have merge conflicts when top contributors at project level and occasional contributors at merge-scenario level touch the source branch. Once we know the conflict-prone developer roles, project coordinators or developers themselves should increase awareness when these developer roles are touching the source code. Considering that it is easy for practitioners collecting the required information (i.e., developer roles at project and merge-scenario level and the touched branch), they can use this information in practice without tool support. Looking at our data, we saw that merge conflicts are rare when there is none or one occasional contributor at both project and merge-scenario level touching the source branch. However, when there are two or more occasional contributors, the chances of conflicting

merge scenarios increase considerably. Looking at Table 5, we saw that one-third of the scenarios led to conflicts when there is at least a top contributor at project level and an occasional contributor at merge-scenario level touching the source branch.

Random forest performed better than decision tree and KNN classifiers. Looking at the answers of RQ_3 and RQ_4, we can see that the random forest classifier performed better than the other classifiers. For instance, in the technical- and all measure models, random forest performed better or equally for all performance measures presented in Table 7 and Table 8. Owhadi-Kareshk et al. [47] found similar results, as we discussed in Section 6.1. With all, we suggest this classifier for further analysis and research on conflict predictions. Be aware that all classifiers used the same data to predict the conflicts. So, the performance is indeed related to the competence of a classifier retrieves better recall, f1-score, precision, and AUC.

Adasyn-sampling is a reasonable balancing technique to use for merge scenario data. Adasyn-sampling is one of the newest balancing techniques and performed better in six out of the nine cases we explored. Over- and SMOTE-sampling also performed well, appearing in four and three cases we investigated, respectively. We suggest Adasyn-sampling balancing technique for further analysis and research on merge conflicts.

The touched branch might be insightful for different kinds of analyses. Following our study, we see that the answers of research questions are complementary. Answering RQ_1, we took a simple viewpoint. Answering RQ_2, we combined developer roles at both project and merge-scenario level. This information was fundamental to achieve reasonable performance measures because we increased our knowledge over our data, especially for the touched branch confounding factor. In fact, we are not the first ones exploring the touched branch factor. However, while previous work [14, 26] reports different contribution patterns on the target and source branches, we are the first ones to show that the touched branch influences the emergence of merge conflicts. In some cases, only the fact of considering the touched branch triples the share of conflicting merge scenarios. For instance, while occasional contributors at merge-scenario level touching the target branch have a share of 8.32%, these contributors touching the source branch have a share of 24.60% (see Table 4). Therefore, as the touched branch played an important role in our analyses and on previous work [14,26], We speculate that the touched branch might be useful for studies mining repositories, investigating project quality criteria, predicting bugs, and other anomalies. For instance, it might provide a new perspective and increase performance when predicting bugs on software systems.

Computing social versus technical measures. As mentioned, developers deeper into the project have a great knowledge of what is going on. Hence, they are able to classify developers at project and merge scenario level without formal measurement. Considering the results of RQ_1 and RQ_2, they are able to identify conflict-prone scenarios with informal measurement and without tool support. In the case of top contributors at project level and occasional contributors at merge-scenario level, around one third of the merge

scenarios have merge conflicts (see Table 5). On the other hand, when a formal measurement is preferred, computing technical measures is simpler because social measures are computed based on the lines of code (a technical measure). Therefore, to compute the developer role related measures, we first need to compute technical measures and then, compute social measures.

Why did the model with social and technical measures not perform better than the model with only technical measures? We see two factors influencing the performance of the model with all measures: i) inserting confounds and ii) increasing the complexity of the investigated phenomenon.

Inserting confounds. Confounds are variables related to each other, but which are not positively impacting the predictions. We already showed a discussion on this topic when answering RQ_4. Hence, in Section 5.4, we saw that some social measures are correlated with each other (e.g., Spearman-rank of 0.78 between occasional contributors at project-level (occ_p) and the number of developers (*devs*)) and also with some technical measures (Spearman-rank of 0.60 between occ_p and the number of commits (*commits*)). Having variables correlated with each other in our model is not necessarily bad, however, it does not help improving the performance of our model.

Increasing the complexity of the investigated phenomenon. The model using only technical measures is composed of 13 independent variables. The model with only social measures is composed of 16 independent variables. Hence, the model with all measures is composed of 29 independent variables which increases the complexity of the investigated phenomenon. Machine learning classifiers are able to identify which variables are more relevant to predict the dependent variable (i.e., minimising over-fitting). However, the more complex the phenomenon, the more difficult it will be to find a function that describes that behaviour. Considering that some variables do not add useful information to the model and the great complexity of the investigated phenomenon with all variables, social measures were not able to improve the performance on the predictions of the technical model. At least adding the social measures did not confuse the technical measures decreasing the performance of the all measures model compared with the technical model performance.

6.3 Implications for Practitioners, Researchers, and Tool Builders

Researchers should focus more on the social perspective and on the branch developers touch. The social perspective in general and the touched branch factor are often ignored when dealing with merge conflicts. Even though using only social measures does not perform optimally, our study reinforces that social information and the touched branch influence on the emergence of merge conflicts. We suggest researchers using developer roles and the touched branch information when investigating merge conflicts. The information of the developer roles and the touched branch might be useful also to other kinds of analysis mining software repositories.

Tool builders should use developer roles for building tools that reduce speculative merging. We show evidence that some developer roles are more often related to conflicting scenarios than others. So, we suggest tool builders using this information to reduce speculative merging. Hence, before performing speculative merging, their tools filter merge scenarios that have a chance of having merge conflicts. Developer roles can also be useful to construct awareness tools. For instance, merge scenarios that have top contributors at project level and occasional contributors at merge-scenario level touching the source branch, might be closely coordinated/monitored since one third of them end with merge conflicts.

Social measures are a good alternative to retrieve conflicting scenarios. As seen in Section 2.3, human factors play an important role in software development. In our study, we were able to retrieve all real conflicting merge scenarios using developer roles. As discussed in Section 6.2, developers with a deep understanding of the project collaboration are able to manually classify developers into top and occasional contributors without formal measurement. Hence, they can avoid merge conflicts by coordinating conflict-prone developer roles more closely. Once automated classification is desired, they can use speculative or awareness tools as previously discussed.

Practitioners should care more about the order of development tasks. Once it is clear that a conflict will arise when developers touch the same piece of code in different branches, practitioners might find ways to define an order to perform their tasks in a way that they are not going to touch the same parts of code in different branches (i.e., excluding the chances of merge conflicts arise). Researchers have been investigating and creating tools to support this [22, 26, 36, 52, 62]. So, practitioners can already use the proposed tools.

7 Threats to Validity

In this section, we discuss potential threats to the validity of our study to help further research and replications of this study. In the following, we detail the main internal and external threats to validity.

Internal validity. We discuss three main internal threats to validity. First, we used simple and common metrics to classify developers. This poses the threat that the metrics do not accurately capture reality. This threat is minor, as existing evidence indicates that those metrics accurately reflect the developers' perception [15, 20, 50, 57]. Second, we used a single alias instead of looking at developers' contributions across multiple information sources (i.e., mailing list, social networks, and version-control system). Although contributors in general are interested in the relevance/recognition of their contributions, maintaining multiple aliases would not be productive. For this reason, we think this threat has limited influence on developer classifications. Third, we selected subject projects from different programming languages; hence, one language could have dominated our dataset. To minimize this threat, we checked and

excluded less popular JavaScript projects until they do not dominate our dataset, as presented in filter *iv* of Section 4.2.

External validity. Three factors can contribute to external threats to validity. First, we used Git and GitHub as platforms, the three-way merge pattern, and the set of metrics. Generalizability to other platforms, projects, development patterns, and set of metrics is limited. This sample limitation was necessary to reduce the influence of confounds, increasing internal validity, through [54]. While more research is needed to generalize to other version control systems and development patterns, we are confident that we select and analyze a practically relevant platform and a substantial number of software projects from various domains, programming languages, longevity, size, and coordination practices. In addition, our filters applied during subject project selection guarantee, for instance, that we sample real and active projects (see Section 4.2). Second, we could not retrieve information from binary files; hence, we may miss information from some merge scenarios. Unfortunately, we could not do anything about that, however, the number of binary files is normally small in software projects. Third, performing only automated analyses. Interviewing or surveying developers could make our analyses more trivial; however, considering that developers think they are doing the right thing, their answers could not point to their faults.

8 Conclusions and Future Work

In this study, we investigated the relation of top and occasional contributors on the emergence of merge conflicts and merge conflict predictions using social and technical assets. To achieve our goal, we mined 66 repositories of popular software projects with a total of 78 740 merge scenarios.

As a result of our initial analysis to understand the influence of developer roles on merge conflicts, we saw that those roles are practical and statistically related to the emergence of merge conflicts. When looking at project level, top contributors are more related to merge conflicts than occasional contributors. On the other hand, when looking at merge-scenario level, occasional contributors are more related to merge conflicts than top contributors. Joining the analysis of project and merge-scenario level, we saw those scenarios, where top contributors at project level and occasional contributors at merge-scenario level contribute, are more related to merge conflicts than the other combination of developer roles. We also found that contributions on the source branch are more conflict-prone than contributions on the target branch. For instance, 24.60% of the contributions of occasional contributors in the source branch resulted in merge conflicts, while only 8.32% of these contributors on the target branch resulted in merge conflicts.

Our predictions achieved 100% of recall for the three models we built (social measures vs. technical measures vs. all measures). Predicting merge conflicts using social and technical assets is useful in practice and these models retrieved all real conflicting scenarios. At the end, we reinforce the importance of using

the information of the touched branch and the social perspective in analyses of software repositories. These pieces of information are important since coding is a social task and they played an important role in our analyses.

In the future, we plan to: (i) deeply investigate the influence of the change location on the emergence of merge conflicts, (ii) survey developers with a large share of conflicting contributions to get their perception of practices that cause merge conflicts, (iii) mine repositories from other source and version control systems to compare our results, (iv) retrieve the performance of other classifiers and measures to predict merge conflicts, and (v) perform a deep analysis on a few developers that are involved in the majority of merge conflicts in a few projects to understand from different perspectives which factors (e.g., type of changes and changed files) are more related to merge conflicts.

9 Acknowledgement

This work was supported by the German Research Foundation (AP 206/14-1) and by the Brazilian National Council for Scientific and Technological Development - CNPq (grant 290136/2015-6).

References

1. P. Accioly, P. Borba, and G. Cavalcanti. “Understanding Semi-structured Merge Conflict Characteristics in Open-source Java projects”. In Empirical Software Engineering. Springer, pp. 1–35, 2017.
2. P. Accioly, P. Borba, L. Silva, and G. Cavalcanti, “Analyzing Conflict Predictors in Open-source Java Projects”. In Proceedings of the International Conference on Mining Software Repositories (MSR), ACM, pp. 576–586, 2018.
3. S. Apel, O. Lessenich, and C. Lengauer. “Structured Merge with Autotuning: Balancing Precision and Performance”. In Proceedings of the International Conference on Automated Software Engineering (ASE). ACM, pp. 120–129, 2012.
4. S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. “Semistructured Merge: Rethinking Merge in Revision Control Systems”. In Proceedings of the Symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE). ACM, pp. 190–200, 2011.
5. J. Biehl, M. Czerwinski, G. Smith and G. Robertson “FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams”. In Proceedings of the Conference on Human Factors in Computing Systems (CHI), ACM, pp. 1313–1322, 2007.
6. C. Bird, D. Pattison, R. D’Souza, V. Filkov, and P. Devanbu. “Latent Social Structure in Open Source Projects”. In Proceedings of the International Symposium on Foundations of Software Engineering (FSE). ACM, pp. 24–35, 2008.
7. C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. “Don’t touch my code!: Examining the Effects of Ownership on Software Quality”. In Proceedings of the European Software Engineering Conference and the Symposium on Foundations of Software Engineering (ESEC/FSE), ACM, pp. 4-14. 2011.
8. C. Brindescu, I. Ahmed, R. Leano, and A. Sarma. “Planning for untangling: Predicting the difficulty of merge conflicts”. In Proceedings of the International Conference on Software Engineering (ICSE), ACM, pp. 801-811, 2020.
9. W. Boh, S. Slaughter, and J. Espinosa. “Learning from experience in software development: A multilevel analysis”. Management Science, vol. 53(8), pp. 1315–1331, 2007.

10. H. Borges and M.T. Valente. “What’s in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform”. In Journal of Systems and Software (JSS), Elsevier, vol. 146 (1), pp. 112–129, 2018.
11. Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. “Proactive Detection of Collaboration Conflicts”. In Proceedings of the European Software Engineering Conference and the Symposium on Foundations of Software Engineering (ESEC/FSE). ACM, pp. 168–178, 2011.
12. J. Buffenbarger, “Syntactic Software Merging”. In Software Configuration Management. Springer, pp. 153–172, 1995.
13. J. Businge, S. Kawuma, E. Bainomugisha, F. Khomh, and E. Nabaasa. “Code Authorship and Fault-proneness of Open-Source Android Applications: An Empirical Study”. In Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE). ACM, pp. 33-42, 2017.
14. C. Costa, J. Figueiredo, G. Ghiotto, and L. Murta. “Characterizing the Problem of Developers’ Assignment for Merging Branches”. International Journal of Software Engineering and Knowledge Engineering, 24, 10 (2014), pp. 1489–1508, 2014.
15. K. Crowston, K. Wei, Q. Li, and J. Howison. “Core and Periphery in Free/Libre and Open Source Software Team Communications”. In Proceedings of the Hawaii International Conference on System Sciences (HICSS), IEEE, pp. 45–56, 2006.
16. L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. “Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository”. In Proceedings of the Conference on Computer Supported Cooperative Work (CSCW). ACM, pp. 1277–1286, 2012.
17. P. Dewan and R. Hegde. “Semi-synchronous Conflict Detection and Resolution in Asynchronous Software Development”. In Proceedings of the Conference on European Computer Supported Cooperative Work (ECSCW). ACM, pp. 159-178, 2007.
18. K. Dias, P. Borba, M. Barreto. “Understanding predictive factors for merge conflicts”. In Information and Software Technology, Elsevier, vol. 121, p. 106256, 2020.
19. E. Dinella, T. Mytkowicz, A. Svyatkovskiy, C. Bird, M. Naik, and S. Lahiri. “DEEPMERGE: Learning to Merge programs”. In Transactions on Software Engineering (TSE), IEEE, vol. 49, pp. 1599-1614, 2022.
20. T. Dinh-Trong and J. Bieman. “The FreeBSD Project: A Replication Case Study of Open Source Development”. Transactions on Software Engineering, , IEEE, vol. 31(6), pp. 481–494, 2005.
21. H. Estler, M. Nordio, C. Furia, and B. Meyer, “Awareness and merge conflicts in distributed software development”. In Proceedings of the International Conference on Global Software Engineering (ICGSE), IEEE, pp. 26-35, 2014.
22. Y. Fan, X. Xia, D. Lo, and S. Li, “Early prediction of merged code changes to prioritize reviewing tasks”. Empirical Software Engineering, Springer, pp. 1–48, 2018.
23. M. Foucault, C. Teyton, D. Lo, X. Blanc, and J. Falleri. “On the Usefulness of Ownership Metrics in Open-source Software Projects”. Information and Software Technology. Elsevier, vol. 64, pp. 102–112, 2015.
24. T. Fritz, G. Murphy, and E. Hill. “Does a programmer’s activity indicate knowledge of code?”. In Proceedings of the European Software Engineering Conference and the Symposium on Foundations of Software Engineering (ESEC/FSE). ACM. pp. 341-350, 2007.
25. “Fstmerge tool”. `https://github.com/joliebig/featurehouse/tree/master/fstmerge`.
26. G. Ghiotto, L. Murta, M. Barros, and A. Hoek. “On the Nature of Merge Conflicts a Study of 2,731 Open Source Java Projects Hosted by GitHub”. Transactions on Software Engineering, IEEE, pp. 1-25, 2018.
27. G. Gousios, M.A. Storey, and A. Bacchelli. “Work Practices and Challenges in Pull-based Development: The Contributor’s Perspective”. In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 285-296, 2016.
28. M. Greiler, K. Herzig, and J. Czerwonka. “Code Ownership and Software Quality: A Replication Study”. In Proceedings of the Working Conference on Mining Software Repositories (MSR), IEEE, pp. 2-12, 2015.
29. M. L. Guimarães and A. R. Silva. “Improving Early Detection of Software Merge Conflicts”. In Proceedings of the International Conference on Software Engineering (ICSE). IEEE, pp. 342-352, 2012.

30. A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry. "Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack". In Empirical Software Engineering. Springer, vol. 24, pp. 674-717, 2019.
31. L. Hattori and M. Lanza. "Syde: A Tool for Collaborative Software Development". In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 235-238, 2010.
32. "Jdime tool". `http://fosd.net/JDime`
33. H. Z. Jerrold. "Significance Testing of the Spearman Rank Correlation Coefficient". Journal of the American Statistical Association, vol. 67 (339), Taylor & Francis, Ltd, pp. 578–580, 1972.
34. M. Joblin, S. Apel, C. Hunsen and W. Mauerer. "Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics". In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 164-174, 2017.
35. M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle. "From Developer Networks to Verified Communities: A Fine-grained Approach". In Proceedings of the International Conference on Software Engineering (ICSE). IEEE, pp. 563-573, 2015.
36. B.K. Kasi and Anita Sarma. "Cassandra: Proactive Conflict Minimization through Optimized Task Scheduling". In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 732-741, 2013.
37. S. Just, K. Herzig, J. Czerwonka, and B. Murphy. "Switching to Git: The Good, the Bad, and the Ugly". In Proceeding of the International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 400-411, 2016.
38. E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. German, D. Damian. "The Promises and Perils of Mining GitHub". In Proceedings of the Working Conference on Mining Software Repositories (MSR), ACM, 92-101, 2014.
39. O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen, and B. Water, "Studying pull request merges: a case study of shopify's active merchant". In Proceedings of the International Conference on Software Engineering (ICSE): Software Engineering in Practice. ACM pp. 124-133, 2018.
40. O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen. "Indicators for Merge Conflicts in the Wild: Survey and Empirical Study". Automated Software Engineering, vol. 25, pp. 279-313, 2018.
41. S. McKee, N. Nelson, A. Sarma, and D. Dig. "Software Practitioner Perspectives on Merge Conflicts and Resolutions." In Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 467-478, 2017.
42. A. Meneely and L. Williams. "Secure open source collaboration: an empirical study of linus' law". In Proceedings of the Conference on Computer and Communications Security (CCS), ACM, pp. 453-462, 2009.
43. A. Mockus, R. T. Fielding, and J. D. Herbsleb. "Two case studies of open source software development: Apache and Mozilla". Transactions Software Engineering Methodology, ACM, pp. 309-346, 2002.
44. N. Nagappan, B. Murphy, and V. Basili. "The influence of organizational structure on software quality: an empirical case study". In Proceedings of the International Conference on Software Engineering (ICSE), ACM, pp. 521-530, 2008.
45. N. Nelson, C. Bridescu, S. McKee, A. Sarma, D. Dig. "The Life-cycle of Merge Conflicts: Processes, Barriers, and Strategies". Empirical Software Engineering, Springer, pp. 1-44, 2019.
46. Y. Nishimura and K. Maruyama. "Supporting Merge Conflict Resolution by Using Fine-grained Code Change History". In Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, pp. 661-664, 2016.
47. M. Owhadi-Kareshk, S. Nadi, and J. Rubin. "Predicting Merge Conflicts in Collaborative Software Development". In Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, pp. 1-11, 2019.
48. M. Pinzger, N. Nagappan, and B. Murphy. "Can developer-module networks predict failures?". In Proceedings of the European Software Engineering Conference and the Symposium on Foundations of Software Engineering (ESEC/FSE). ACM, pp. 2-12, 2008.

49. F. Rahman and P. Devanbu. “Ownership, Experience and Defects: A Fine-grained Study of Authorship”. In Proceedings of the International Conference on Software Engineering (ICSE), ACM, pp. 491-500, 2011.
50. G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz. “Evolution of the Core Team of Developers in Libre Software Projects”. In Proceedings of the Mining Software Repositories (MSR), IEEE, pp. 167-170, 2009.
51. T. Rocha, P. Borba and J. Santos. “Using acceptance tests to predict files changed by programming tasks”. In Journal of Systems and Software (JSS), Elsevier, vol. 154, pp. 176–195, 2019.
52. A. Sarma, D.F. Redmiles, A. van der Hoek. “Palantír: Early Detection of Development Conflicts Arising from Parallel Code Changes”. Transactions on Software Engineering. IEEE, vol. 38(4), pp. 889-908, 2012.
53. S. Shafiq, A. Mashkoor, C. Mayr-Dorn and A. Egyed, “TaskAllocator: A Recommendation Approach for Role-based Tasks Allocation in Agile Software Development”. In Proceedings of the International Conference on Global Software Engineering (ICGSE), IEEE, pp. 39-49, 2021.
54. J. Siegmund and J. Schumann. “Confounding Parameters on Program Comprehension: A Literature Survey”. Empirical Software Engineering 20, 4 (2015), Springer, pp. 1159-1192, 2015.
55. L. Singer, F. Figueira Filho, B. Cleary, C. Treude, M.A. Storey, and K. Schneider. “Mutual Assessment in the Social Programmer Ecosystem: An Empirical Investigation of Developer Profile Aggregators”. In Proceedings of the Conference on Computer Supported Cooperative Work (CSCW). ACM, pp. 103-116, 2013.
56. M.A. Storey, A. Zagalsky, F. Figueira Filho, L. Singer, D. M. German. “How Social and Communication Channels Shape and Challenge a Participatory Culture in Software Development”. In IEEE Transactions on Software Engineering. vol. 43 (2), pp. 185-204, 2016.
57. A. Terceiro, L. R. Rios, and C. Chavez. “An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects”. In Proceedings of the Brazilian Symposium on Software Engineering, IEEE, pp. 21-29, 2010.
58. P. Thongtanunam, S. McIntosh, A. Hassan, and H. Iida. “Revisiting code ownership and its relationship with software quality in the scope of modern code review”. In Proceedings of the International Conference on Software Engineering (ICSE), ACM, pp. 1039-1050, 2016.
59. M. Trif and R. Slavescu. “Towards Predicting Merge Conflicts in Software Development Environments”. In Proceedings of the International Conference on Intelligent Computer Communication and Processing (ICCP), IEEE, pp. 251-256, 2021.
60. J. Tsay, L. Dabbish, and J. Herbsleb. “Influence of Social and Technical Factors for Evaluating Contribution in GitHub”. In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 356-366, 2014.
61. G. Vale, A. Schmid, A. Santos, E. Almeida, S. Apel. “On the Relation Between Github Communication Activity and Merge Conflicts”. Empirical Software Engineering, vol. 25, Springer, pp. 402-433, 2020.
62. G. Vale, C. Hunsen, E. Figueiredo and S. Apel. “Challenges of Resolving Merge Conflicts: A Mining and Survey Study”. Transactions on Software Engineering (TSE), IEEE, pp. 1-22, 2021.
63. G. Vale, H. Costa, and S. Apel, “Predicting Merge Conflicts Considering Social and Technical Assets” Available: `https://gustavovale.github.io/predicting-merge-conflicts-considering-social-and-technical-assets/`. [Accessed 27/07/2023].
64. B. Westfechtel, “Structure-oriented Merging of Revisions of Software Documents”. In Proceedings of the International Workshop on Software Configuration Management. ACM, pp. 68-79, 1991.
65. T. Wuensche, A. Andrzejak, and S. Schwedes. “Detecting Higher-Order Merge Conflicts in Large Software Projects”. In Proceedings of the International Conference on Software Testing, Validation and Verification (ICST). IEEE, pp. 353,363, 2020.

66. E. Weyuker, T. Ostrand, and R. Bell. “Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models”. Empirical Software Engineering, Springer, vol. 13(5), pp. 539-559, 2008.
67. R.F. Woolson, “Wilcoxon signed-rank test”. Wiley Encyclopedia of Clinical Trials pp.1-3, 2008.
68. T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. “Mining Version Histories to Guide Software Changes”. In Proceedings of the International Conference on Software Engineering (ICSE). IEEE, pp. 563-572, 2004.