How (Not) to Find Bugs: The Interplay Between Merge Conflicts, Co-Changes, and Bugs

Luís Amaral*, Marcos C. Oliveira†, Welder Luz*, José Fortes*,
Rodrigo Bonifácio*, Daniel Alencar‡, Eduardo Monteiro*, Gustavo Pinto§, and David Lo¶
*University of Brasília. Brasília, Brazil
†Brazilian Ministry of Economy. Brasília, Brazil
‡University of Otago. Dunedin, New Zealand
§Federal University of Pará. Belém, Brazil
¶Singapore Management University. Singapore

Abstract—Context: **In a seminal work, Ball et al. [1] investigate if the information available in version control systems could be used to predict defect density, arguing that practitioners and researchers could better understand errors "*if [our] version control system could talk*". In the meanwhile, several research works have reported that conflict merge resolution is a time consuming and error-prone task, while other contributions diverge about the correlation between co-change dependencies and defect density.** ***Problem:*** **The correlation between conflicting merge scenarios and bugs has not been addressed before, whilst the correlation between co-change dependencies and bug density has been only investigated using a small number of case studies—which can compromise the generalization of the results.** ***Goal:*** **To address this gap in the literature, this paper presents the results of a comprehensive study whose goal is to understand whether or not (a) conflicting merge scenarios and (b) co-change dependencies are good predictors for bug density.** ***Method:*** **We first build a curated dataset comprising the source code history of 29 popular Java Apache projects and leverage the SZZ algorithm to collect the sets of bug-fixing and bug-introducing commits. We then combine the SZZ results with the set of past conflicting merge scenarios and co-change dependencies of the projects. Finally, we use exploratory data analysis and machine learning models to understand the strength of the correlation between conflict resolution and co-change dependencies with defect density.** ***Findings:*** **(a) conflicting merge scenarios are not more prone to introduce bugs than regular commits, (b) there is a negligible to a small correlation between co-change dependencies and defect density—contradicting previous studies in the literature.**

***Index Terms*—software defects, software integration, merge conflicts, co-change dependencies**

I. Introduction

Software teams spend a significant amount of time trying to locate defects and fixing bugs [2]. Actually, fixing a bug involves isolating the part of the code that causes an unexpected behavior of the program and changing it to correct the error [3, 4]. This is a challenging task, and developers often spend more time fixing bugs and making the code more maintainable than developing new features [5, 6, 7].

To mitigate the time spent fixing bugs, it is crucial to better understand the development practices and the properties of the systems that are more likely to introduce bugs. Existing research works have investigated the correlation between structural properties of the systems (such as object-oriented metrics) and defect density [8, 9, 10]. Researchers have also investigated whether the complexity of code changes could be used to estimate the incidence of bugs in software assets [11, 12]; while others have leveraged information available in version control systems (VCSs) either to (a) characterize the properties of changes that may introduce bugs [13] or to (b) investigate if co-change metrics are good predictors for defect density [1, 14, 15, 16].

Although some studies investigated the characteristics of bug-introducing changes (e.g., [13, 17]), there are many other categories of these changes that have not been explored before. Exploring specific categories of bug-introducing changes is essential to aid developers in avoiding them. In our paper, we investigate two potential categories of bug-introducing changes: merge conflict resolution and co-changes dependencies. To the best of our knowledge, we are the first to address the relationship between merge conflicts and bug-introducing changes. Furthermore, while previous research works have explored the relation between co-change dependencies metrics and defect density, the conclusions have been drawn from a small number of samples and are inconclusive—some works claim that co-change dependencies might be used to predict defects [14], while others claim the contrary [1]. The lack of a general understanding of these two aspects brings the general research questions we address in this paper:

- RQ1 *To what extent conflicting merge scenarios induce bugs?* Answering this research question is important because it could reveal the impact of merging operations on the correctness of the systems—particularly because previous research works suggest that resolving merge conflicts is a complicated and error-prone task [18, 19, 20].
- RQ2 *To what extent co-change dependencies metrics correlate to defect density?* Answering this research question is important because it could reveal a negative side of a system decomposition that leads to co-change dependencies, either confirming or refuting results of previous studies [14, 21, 22].

To achieve our goal, we first mine the source code history of a curated dataset comprising 29 popular Java Apache projects hosted on GitHub. We then leverage the SZZ algorithm to identify the bug-fixing changes (BFCs) and bug-introducing changes (BICs). We relate the outcomes of the SZZ algorithm with the information about conflicting merge scenarios and co-change dependencies. Finally, we use statistical methods to answer the research questions.

Our study brings several findings, for instance: we evidence that commits that solve conflicting merge scenarios are not more likely to introduce bugs than regular commits. Apart from that, resolving 9.43% of the conflicting merge scenarios lead to bug-introducing changes, which represents 0.5% of all BICs. Curiously, our results also suggest that the continuous integration of code does not necessarily reduce the amount of bugs caused by conflict resolution. Finally, contrasting to previous research [14], we did not find evidence that co-change dependency metrics are good predictors for defect density.

II. Related Work

In this paper, we leverage the SZZ algorithm to investigate to what extent (a) conflicting merge scenarios introduce bugs and (b) co-change dependency metrics relate to defect density. Because our work investigates whether merge conflicts and co-changes relate to bugs, we surveyed the related research regarding merge conflicts and co-changes.

A. Research on Merge and Conflict Resolution

Recent workflows for collaborative development support the use of different branches to implement each feature or bug fix, for instance. The contributions from different branches must be integrated at some point, through operations such as *merge* or *rebase*—both available in popular distributed version control systems (VCSs), such as Git and Mercurial. Nonetheless, existing research works report that solving conflicting merge scenarios are both tedious and error prone [18, 19, 20]. To mitigate this problem, researchers have proposed techniques to predict, prevent, and help developers during activities of conflict resolution.

Regarding conflict prevention and resolution, Guimarães and Silva [23] introduce the idea of continuous merge inside the IDE. The goal is to continuously merge *uncommitted* and *committed* changes in order to avoid conflicts—using a fine-grained solution. Previously, awareness tools have also been designed to deal with merge conflicts [24, 25] at a coarser grained level file (e.g., file). More recently, Apel et al. [26] propose an approach for structured merging, trying to find a balance between precision and performance. The rationale for their approach is that unstructured merge scenarios (using a line-based strategy, for instance) have great performance but poor precision. Similarly, other research contributions aim to improve the precision of merge tools [27, 28, 29] to reduce the number of conflicts.

Regarding predictive models for integration conflicts, Leßenich et al. [30] explore whether or not a set of merge scenario features (obtained from a survey with practitioners) could be used to estimate the *size of merge conflicts*. That is, the goal is to verify if merge conflict prediction based on those features could help developers during the activity of code integration. As a result of the empirical analysis, the authors report that none of the features are good predictors concerning conflict size (e.g., number of files in conflict, lines of code in conflict, and so on). Despite this overall negative result, their study formed a solid basis for replication and follow-up studies such on conflict-avoidance strategies (e.g., speculative merging).

Following a similar approach, Owhadi-Kareshk et al. [31] built some classifiers to decrease the cost of speculative merging running in background, by avoiding to perform speculative merging [23, 32, 33] in the safe scenarios. Proactive conflict detection is based on speculative merging, which retrieves all available branches and merges them in background. While it is cheap to perform a single textual merge operation, the cost can increase exponentially according to the number of active branches. Differently from the work by Leßenich et al. [30], Owhadi-Kareshk et al. [31] argue that a lack of correlation does not necessarily mean that it is not possible to classify safe versus conflicting merge scenarios. Their results confirm the lack of a significant correlation between the characteristics of a merging scenario and the number of conflicts. Nonetheless, their prediction results show that their classifiers did not perform poorly. The results are useful to check safe merge scenarios and reduce the costs of proactive speculative merging, reducing the computational costs.

Accioly et al. [34] also analyzed the prediction power of two features for early conflict detection—editions to the same method (EditSameMC) and editions to directly dependent methods (EditDepMC). As a result of the conflict awareness tool, considering EditSameMC and EditDepMC, the precision indicates that the tool triggered the alarm 57.99% of the merge scenarios. Moreover, the recall indicates they have captured 82.67% of the merge scenarios with conflicts (merge, build, or test) when considering both predictors. Furthermore, when analyzing the predictors individually, EditSameMC has a precision of 56.71% and EditDepMC, only 8.85%, and recall of 80.85% and 13.15% respectively. As a conclusion, their study is useful to guide conflict awareness strategies and provide a better notion of the real frequency of merge conflicts.

Ahmed et al. [35] investigated strength of the correlation between entities that contain code smells, the code smells they contain, and the merge conflicts surrounded by smelly entities. The goal is to obtain metrics about code changes and conflicts. First, they divided conflicting merge scenarios into two categories—semantic conflicts (requires to understand the logic of the program to resolve, such as variable name changed), and non-semantic (more natural and less risky to resolve, such as comments and white space). As a result, the authors report that, on average, elements involved in merge conflicts present three times more code smells than elements not involved in merge conflicts. Since code smells are more expected to be related to bugs in the future [36], they conclude that entities involving code smells and merge conflicts are more likely to be buggy, and practitioners should pay more attention to code smells to reduce the number of merge conflicts.

B. Research on Co-change Dependencies

Ball et al. [1] present one of the first research works that explore the use of co-change dependencies (a.k.a, change coupling or logic coupling) to analyze the structure of systems. In fact, the research on co-change dependencies have focused on getting new insights about the structure of systems [16, 37, 38, 39] and finding opportunities to rethink architectural decisions [40, 41]. For instance, Beyer and Noack [37] use information from version control systems to build a graph from assets that frequently change together. The goal is to find clusters in this graph that correspond to *subsystem candidates*. Other research works focus on the interplay between structural dependencies and co-change dependencies [16, 39], highlighting that there is no linear correlation between these types of dependencies—classes that are statically dependent do not necessarily change together. Other research works find opportunities to change the decomposition of the systems using co-change dependencies [40, 41].

Besides reasoning about the structure of the systems, other research works investigate the relationship between co-change dependencies and defect density. However, we could not find a consensus about this topic. Some findings reported in the literature [21, 42] claim that there is no correlation between highly co-change coupled assets (such as files or classes) and the bug incidence in these assets (i.e., frequency that these assets change due to bug fixes). For instance, Knab et al. [21] use *decision trees* to find rules that can be used to predict defect density. Using data extracted from the Mozilla Web Browser source code history, the authors conclude that "*change couplings are of little value for the prediction of defect density*" [21].

Contrasting, other research works [14, 43, 22] suggest that there is a correlation between co-change dependency metrics (such as the number of co-dependent classes of a given class) and bug density. For instance, D'Ambros et al. [14] present the results of an empirical study using three open source systems (ArgoUML, JDT Core, and Mylyn). The authors investigate the correlation between five *change coupling metrics* and the number of bugs of the components (Java classes)—reporting a moderate to a high correlation between change coupling metrics and defects. Other research works explored the same question, though using a small number of systems [22, 43], and concluded that co-change dependencies could be used to predict defect density.

Our second research question also investigates whether or not co-change dependency metrics correlate to defect density. Nonetheless, differently from previous studies [14, 43, 22] that draw conclusions from one or two systems, here we consider a curated dataset with the source code history from a set of 29 Apache open source systems, increasing the generalization of the results.

C. The SZZ algorithm and its limitations

The SZZ algorithm was introduced by Śliwerski et al. [13], to identify the potential changesets (commits) responsible for introducing defects. It is a well-known algorithm, being widely used in the Just-in-Time Defect Prediction research agenda to label historical-changes as *bug-introducing* or *clean*. Rodríguez-Pérez et al. [44] present the results of a literature review, assessing 187 papers that made use of the SZZ algorithm to evaluate the reproducibility and credibility of these publications in Empirical Software Engineering.

Several limitations of the SZZ algorithm have been reported, including technical (e.g., mislabeled changes) and methodological ones (e.g., difficulty to reproduce the studies). For instance, the first SZZ [13] variant has several problems. In particular, it considers cosmetic changes (as indentation, blank lines, and comments) as possible bug-introducing commits. Nonetheless, cosmetic changes do not modify the software behavior.

To deal with the technical limitations of the original SZZ design, researchers developed new variants of the SZZ algorithm [17, 45, 46], in order to reduce noise. When considering the first phase of the algorithm (finding bug-fixing commits), the limitation relies on how bug reports are linked to commits, i.e., if the bug fix is not identified, the bug commit cannot be determined, causing a false negative. False-positive happens when a bug report does not describe a real bug, but a fixing commit is linked to it. As reported by early studies 33.8% [47] to 40% [48] of the bugs in issue tracking system are miss-classified.

The second part of the algorithm, which is concerned with identifying the bug-introducing commits, can also produce false positives and negatives. Addressing these limitations requires a manual and tedious validation process [44], and da Costa et al. [17] proposed a framework to evaluate and compare different implementations of SZZ.

Neto et al. [46] showed that discarding cosmetic changes and refactoring contributions improve the precision of the second phase of the original SZZ, from 37% using their `RA-SZZ` implementation to 97% using the `RA-SZZ`* variant. Moreover, `RA-SZZ`* outperforms another recent SZZ implementation (MA-SZZ [17]). After experimenting with other implementations, and reading these results in the literature, we decided to use `RA-SZZ`* in our research.

Although the SZZ algorithm has been used to relate work practices and bug introducing change, we are the first to investigate this aspect considering conflict merge resolution.

III. Study Settings

In this section, we present the settings of our study, whose main goal is to investigate whether syntactic merge conflicts and commits that lead to co-change dependencies relate to bugs. As such, we answer the research questions we introduce in Section I.

A. Project Selection

Our procedures for project selection consider the existence of tools for mining bug introducing commits and tools that we could use to reproduce merge scenarios, identify non-cosmetic changes (e.g., changes that go beyond adding a comment of a piece of code), and compute co-change dependencies. To mine bug-introducing commits, we leverage in our research the RA-SZZ* [46] tool—a refactoring aware implementation of the SZZ algorithm. RA-SZZ* collects project information from a git source code repository and from a JIRA database with the project issues. RA-SZZ* then populates a relational database with all necessary information to find bug-fixing commits, and link bug-fixing commits to bug-introducing commits, taking into account refactoring and cosmetic changes. The decision of using RA-SZZ* led us to consider the Apache community as an initial project population, since a set of Apache projects use JIRA as an issue management system, and developers of Apache projects often link code contributions to the JIRA issues—a requirement for improving the performance of RA-SZZ*. By mining from Apache we are controlling for the quality of our dataset as we are much less likely to perform our study on unrepresentative projects.

We then focused on Apache Java projects, due to the availability of tools to compute structured merge conflicts [49, 26] and co-change dependencies [41]. Besides this, we found some datasets about syntactic merge conflicts in Java projects, which we could use to validate some of our procedures and scripts we use to mine the change history of the projects. Furthermore, following existing recommendations for mining GITHUB repositories [50], we include the number of stars as a measurement of popularity. As a result, we selected Apache Java projects hosted on GITHUB having more than 200 stars. Applying this filter on the APACHE GITHUB organization revealed 101 repositories, which we considered as our initial dataset. This initial dataset includes projects with different characteristics, from medium size libraries and web frameworks (e.g., Struts and Wicket) to full-fledged textual search engines and database systems (e.g., Lucene and Cassandra).

B. Finding Bug-introducing commits

We mined software repositories to detect bug-introducing commits (BICs) from the source code history of the selected projects. To this end, we leveraged the RA-SZZ* [46] tool to identify BICs. Several reasons support our choice of using RA-SZZ*. First, RA-SZZ* removes both refactoring and cosmetic changes from bug-introducing candidates, reducing the number of false positives. Second, previous results in the literature show that RA-SZZ* outperforms other implementations [17, 46].

In this section, we use the Apache Nifi project as a running example to describe our methodology. Apache Nifi is hosted on GITHUB and uses JIRA as the issue tracking system (as all instances of our initial project population). We follow the steps below to mine the bug introducing commits:

(S1) **Fetch bug issues:** The first step is to collect bug issues from JIRA, using its REST API, and filtering the issues using the issue type = **bug**, the status = (**resolved** or **closed**), and the resolution = **fixed**. As an output, we collected 1988 issues from Apache Nifi.

(S2) **Clone the project:** The second step is to clone the project repository locally to get its source-code history.

(S3) **Find Bug Fixes:** The third step is to use the resulting files from previous steps to link bug-fixing commits (BFCs) to *issues*. In this case, it is necessary to specify how a bug fix should mention the issue in a commit message, and then RA-SZZ* finds some patterns to decide whether or not a commit is a bug-fix. As a result, we obtain a file containing all BFCs necessary as input to the second phase of the SZZ algorithm (that finds the bug-introducing commits). For the running example, we found 1847 bug-fixing commits, mapping 92% of the issues from JIRA to BFCs on git-log.

(S4) **Find Bug Introducing Commits:** Finally, after computing the bug-fixing commits, we run the second phase of RA-SZZ* to identify the commits responsible for introducing bugs. In the case of Apache Nifi, we obtained 2406 pairs of bug-fixing commits and their respective bug-introducing commits. Notice that a bug-introducing commit might be responsible for inducing more than one issue, and one bug-fixing commit might have more than one BIC.

In summary, considering our running example, SZZ identified 2406 pairs of BFC $\times$ BIC—having 1025 unique BFCs and 920 unique BICs. We created additional scripts to replicate the pilot study, through running RA-SZZ* for the remaining 100 project repositories. After assessing the results in these repositories, we filter out several outlier projects from our analysis, as we discuss in Section IV-A.

C. Identifying and Re-playing Merge Scenarios

To answer RQ1, we identify all merge scenarios (conflicting and non-conflicting ones) of a project. Using the git log capabilities, we collected information about the commits that correspond to merge scenarios, including information about the commit hash of two parents of a merging scenario (left and right). We also collect additional information regarding: (a) if the merge scenario leads to conflict, (b) the time span between the commit date of the left/right shared ancestor and the commit date of the merge, and (c) the number of contributors in the *left* and *right* development branches. It is necessary to replay every merge scenarios to collect all this information because git does not keep a record of past merge conflicts. We use these merge attributes to investigate the performance of models to predict when a conflicting merge scenario is more likely to introduce a bug.

Our procedure to answer RQ1 consists of first collecting all hash commits that correspond to conflicting merge scenarios (creating a dataset with all conflicting merge scenarios from the projects in our projects' population). Based on the outputs of RA-SZZ*, we create a second dataset with all bug introducing commits. After that, we merge these two datasets and use exploratory data-analysis [51] to estimate the frequency in which conflicting merge scenarios introduce bugs.

We further explore RQ1 by building and comparing the performance of three machine learning algorithms to predict the likelihood with which a conflicting merge scenario introduces a bug. The sequence of steps necessary to collect information about merge conflicts is as follows. It is important to note that all steps have been performed using variations of the `git log`, `git reset`, and `git merge` commands.

(S1) **Get all Commits:** In the first step we collect the **hash** information, the **date**, and the **author's name** of all commits. This information is relevant while performing the exploratory data analysis.

(S2) **Get the Merge Commits:** In the second step, we collect the hash of all merge commits and their respective parents' hashes, including the *left parent hash* (left hash) and *right parent hash* (right hash). We represent these commits with the labels `LP` and `RP` in Figure 1.

(S3) **Find the Base Commit:** After finding the hash of the parents' commits (`LP` and `RP`), we are able to find the common ancestor (the `CA` commit in Figure 1). In this research we only considered 3-way merge scenarios.

(S4) **Re-play Merge Commits:** In this step we verify if there were files in conflict in a merge scenario; if so, we also collect more detailed information, by hard resetting git to the base common ancestor (`CA`) and then merging that base with the parent right (`RP`), and finally merging the results with the parent left (`LP`).

(S5) **Record the outcome:** Finally, when a conflict occurs, we collect and treat the outcome of Step 4 to collect several features from a merge scenarios (e.g., number of modified files, number of lines added, number of lines removed, number of files in conflict, and number of contributors), which we use to build prediction models. We decided only to use features that could be collected using `git` commands, similarly to previous studies [30, 31].

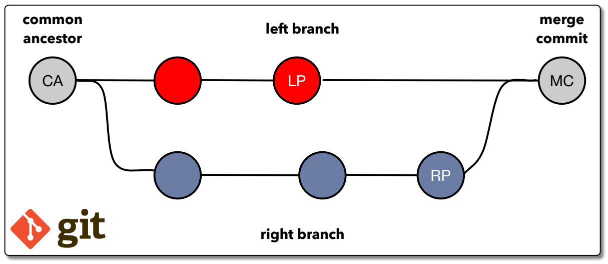

Fig. 1. Components of a merge scenario

We recreated 363 merges scenarios for the Apache Nifi project using the above procedures, from which 30 of them led to conflicts (8.26%). By merging both datasets, we found that SZZ blamed seven conflict merge scenarios (that is, they are potential bug-introducing commits).

D. Computing Co-change Dependencies

To answer our second research question, we first have to compute the co-change dependencies of the systems. A co-change dependency arises when two source-code entities, such as classes, interfaces, methods, or fields, frequently change together. We compute co-change dependencies using the source code history of the systems. Popular VCSs such as `git` and Subversion maintain the evolution of source-code artifacts (typically files), and the history of changes can be described as a sequence of commits $H = (c_1, c_2, \ldots, c_n)$, where each commit contains a subset of artifacts in the form $c_i \subseteq A$. From this sequence of commits, we can build a (di)graph whose vertexes correspond to the source code entities of a system and whose edges correspond to the co-change dependencies. Although it is possible to compute co-change dependencies for finer-grained entities [41], in this study we focus on coarse-grained entities (i.e., the vertexes of our graph correspond to Java classes).

We then use two metrics to determine if two entities e_a and e_b change frequently together: *support count* and *confidence*. The first counts the number of commits in which both e_a and e_b appear together; while the second corresponds to the ratio of the *support count* between e_a and e_b and the number of commits containing e_a. Note that, while the support count is commutative, i.e., the support count between e_a and e_b is the same of the support count between e_b and e_a, the confidence is not, i.e., the confidence between e_a and e_b might differ from the confidence between e_b and e_a. We consider that e_a and e_b change frequently together if their support count and confidence are above the threshold for support count S_{min} and confidence C_{min} at least in one direction. Several studies on co-change dependencies use the values $S_{min} = 2$ and $0.4 \leq C_{min} \leq 0.5$ (e.g., [40, 38]).

We also compute two additional metrics [14] from the co-change dependencies: Number of Coupled Classes (NOCC) and Sum of Class Coupling (SOCC). The first computes the *number of classes n-coupled with a given class*—where n specifies a dependency threshold corresponding to the minimum number of changes between two components. The second is the *sum of the shared transactions* (commits) *between a given class c and all the classes n-coupled with c*. Accordingly, SOCC considers the strength of the coupling between the two components. Finally, we use statistical methods (hypothesis testing and regression analysis) to estimate the strength of the relationship between these metrics and metrics that estimate how a given component is prone to bugs.

IV. Results

In this section we present the results of our empirical study. We first report the outcomes of an exploratory data analysis, and then we answer our research questions using statistical methods (either hypothesis testing or regressions models).

A. Data Description

We conduct an exploratory data analysis to get a general understanding about the frequency of merge scenarios and conflicting merge scenarios, as well as to refine and build curated datasets we use to answer our research questions. To curate our dataset, we removed projects that neither have merge scenarios nor conflicting merge scenarios. Interesting, in nine projects, we did not find any merge commit (e.g., COMMONS-IO). Although we do not investigate this issue in details, we conjecture that some projects employ alternative procedures to integrate software changes (e.g., rebase).

Furthermore, we eliminated projects that do not have at least 26 (first quartile) merge scenarios and filtered out projects in which it was not possible to collect at least 200 (first quartile) closed bug-issues, to guarantee that we would have linked a representative number of issues to bug-introducing commits. Finally, we classified the merge scenarios either as *extra, extra large* merge scenarios (`XXL`) or *non-extra, extra large* merge scenarios (`non-XXL`). To this end, we roughly estimate the complexity of a merge scenario (Cm) as the geometric mean between the number of changed files from its parents (left and right). In our dataset, `XXL` scenarios are those scenarios with $Cm > 15.780$ (third quartile). Having this separation is important because we found many merge scenarios changing a huge number of files. For instance, the merge scenario with commit ID `3b21d1db4109939450dc400faebe568222ab4758` from NETBEANS changed more than 70,000 files. Nonetheless, it is important to note that even the `non-XXL` group contains merge scenarios involving more than 34 files on the average, with contributions made by more than four authors (also on average). Table I summarizes some features of the `non-XXL` merge scenarios.

TABLE I
SUMMARY OF THE CHARACTERISTICS OF OUR DATASET WITH `NON-XXL` MERGE SCENARIOS

Statistic	Mean	St. Dev.	Min	Max
Number of files changed	34.68	316.61	0	25,142
Number of contributors	4.45	5.527	2	175
Number of commits	14.52	50.368	2	2497

Altogether, our curated dataset, which is the intersection of the outcomes generated by our three procedures (see Sections III-B, III-C, and III-D), contains information about 21,189 merge scenarios of 29 Java Apache projects, from which we collected 29,245 bug-introducing commits. The average number of issues and bug-fixing commits per project is 2445 and 1911, respectively. For instance, we have mined 15,465 closed bug issues and linked 14,333 bug-fixing commits in APACHE AMBARI; while we got only 158 bug-fixing commits for 203 closed bug issues collected from JIRA in APACHE FINERACT. Figure 2 shows a histogram that considers the rate of bug-fixing commits over the number of issues per project. Overall, the first phase of `RA-SZZ`* linked 78.16% of the issues to bug-fixing commits. In seven projects, `RA-SZZ`* linked more 90% of the issues to BFCs (e.g., ZEPPELIN and LUCENE-SORL). Nonetheless, in the APACHE CORDOVA-ANDROID project, `RA-SZZ`* linked only 508 bug-fixing commits to a total of 4709 issues (which represents 10.79%). This situation occurs because APACHE CORDOVA-ANDROID is a submodule of APACHE CORDOVA, which shares the same JIRA repository with other modules. Nonetheless, in our analysis we only considered APACHE CORDOVA-ANDROID.

The outcomes of the second phase of `RA-SZZ`* revealed

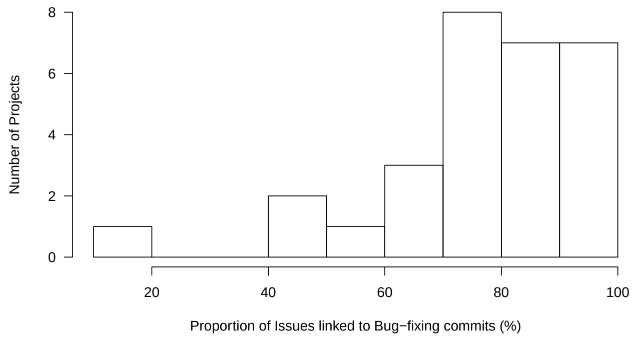

Fig. 2. Proportion of bug issues linked to bug-fixing commits over the projects

59,910 pairs of BFC-BIC over the projects, comprising a set of 22,532 bug-fixing and 29,245 bug-introducing distinct commits. It is important to remember that a bug-introducing commit might introduce bugs in more than one place, and a bug-fixing commit might fix bugs introduced by multiple BICs. For instance, APACHE CAMEL is the project with more BICs—`RA-SZZ`* blamed 3336 commits for 2204 BFCs. By comparing with its first phase, where `RA-SZZ`* linked 3354 bug-fixing commits for APACHE CAMEL, it means that SZZ could not find BICs for 1150 BFCs. Considering the NETBEANS project, SZZ revealed 66 BICs for 145 bug-fixing commits while 128 BICs were responsible for introducing errors in 123 BFCs on project APACHE PARQUET-MR. Overall, according to the `RA-SZZ`* outcomes, 51% of BFCs fixed errors caused by bug-introducing commits, with a rate higher than 70% on FINERACT and CXF. The lowest rate value happened in APACHE HIVE, in which `RA-SZZ`* linked only 667 BFCs to bug-introducing commits (9.88% of the total number of BFCs). We found 1590 conflicting merge scenarios (7.5% of the total number of merge scenarios). Considering the merge scenarios, APACHE AVRO has 47 (the lowest) and BEAM has 7365 (the highest). Finally, APACHE JAMES has only one conflicting merge scenario, while STORM presents 239 conflicting merge scenarios. More than 40% of the merge scenarios of HIVE led to a conflict. Considering the rate of conflicted merge scenarios over the number of merges, Figure 3 shows that, in most of the projects (86.2%), conflicts occur in less than 20% of merge scenarios.

B. *To what extent conflicting merge scenarios induce bugs?*

From the conflicting merge scenarios (1590 observations), `RA-SZZ`* blamed 150 commits (9.43%) as bug-introducing which introduced bugs in 21 projects. *Apache Lucene* is the project with the highest number of bug-introducing commits linked to conflicting merge scenarios (30), followed by *Apache Ambari* with 26 conflicting merge commits that introduced errors. `RA-SZZ`* did not blame any conflicting merge scenario in eight projects, and four projects have only one blamed conflicting merge commit. Figure 4 shows the percentage of BICs linked either to *conflicting merge scenarios* and to *regu-*

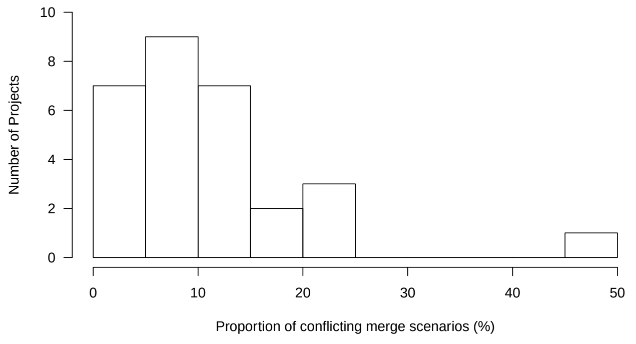

Fig. 3. Histogram with the proportion of conflicting merge scenarios per project

lar commits over the projects. Most of the projects (72.24%) had less than 10% of conflicting merge scenarios linked to bug-introducing commits. On the other hand, eight projects have more than 10% of the conflicting merge scenarios linked to bug-introducing commits (see Figure 4). Project *Apache Phoenix* presents the highest ratio—`RA-SZZ*` blamed all the six conflicting merge commits—followed by *Apache Ambari* with a ratio of 26 over 66 (39.39%)—in *Lucene*, the one with the highest number of bug-introducing commits linked to conflicting merge scenarios, the rate is 22.39%.

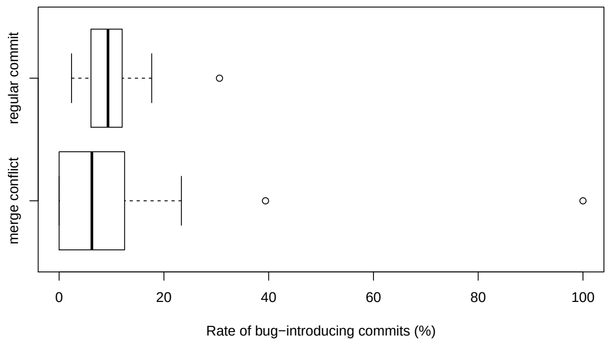

Fig. 4. Percentage of bug-introducing commits (either due to conflicting merge scenarios or regular commits)

When considering *r*egular commits (305,262 observations), `RA-SZZ*` blamed 29,095 commits that possibly introduced bugs, which represents 9.53%. Figure 5 shows that the percentage of bug-introducing commits related to conflict resolution, in the worst case, correspond to less than 4% of all bug-introducing commits.

Some general information is necessary to understand our findings. First, conflicting merge scenarios represent 0.52% of the total number of all commits. Second, conflicting merge scenarios blamed by `RA-SZZ*` represent 0.51% of all bug-introducing commits (see Figure 5). These percentages indicate that the occurrence of a bug introduced by a conflicting merge scenario is equal to the occurrence of a conflicted merge commit. We compare the distribution of the percentages of bug-introducing commits emerging either (a) from the set of conflicting merge scenarios or (b) from the set of all the remaining commits (that is, the set of commits that do not solve a merge conflict), using the `Wilcoxon signed-rank test`. According to the analysis, we cannot reject the null hypothesis that the two populations have a similar distribution of bug-introducing commits (p-value = 0.2652 with confidence interval of 95%). Moreover, we applied the `Cliff's Delta effect size` test to verify the significance of the differences between both distributions, revealing a small magnitude of the differences between the bug introducing contribution of regular commits vs. conflicting merge commits (Cliff's delta = 0.2628).

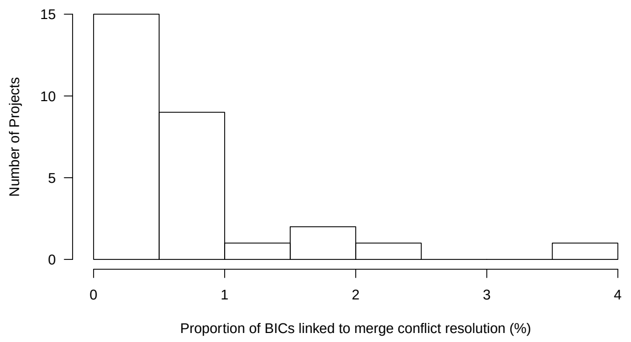

Fig. 5. Proportion of bug-introducing commits also linked to merge conflicts resolution

Therefore, conflicting merge scenarios are not more prone to introduce bugs than usual commits. That is, 9.43% of the conflicting merge scenarios introduce bugs, while `RA-SZZ*` linked 9.53% of all commits as bug-introducing changes. Besides, conflict resolution represents 0.51% of all bug-introducing commits. Nonetheless, this finding suggests that merge conflict resolution is an important source of bugs, and 150 bugs have been introduced due to conflicting merge resolution.

We also replicated our analyzes considering only `non-XXL` and `XXL` scenarios, separately. Interestingly, when considering only `non-XXL` scenarios—which corresponds to 75% of our curated dataset of merge scenarios, the number of conflicting merge scenarios drops from 1590 to 369 (23.20%). This might suggest that `XXL` merge scenarios are more likely to introduce conflicts—different from what has been reported in previous research [30]. Moreover, the number of bug-introducing commits linked to conflicting merge scenarios drops from 150 to 20 (13.3% of all BICs linked to conflicting merge scenarios). This suggests that complex merge scenarios (`XXL`) cause more than 85% of all conflicting merge scenarios linked to bug-introducing commits.

When considering the `non-XXL` group, `RA-SZZ`* blamed 5.42% of conflicting merge scenarios. On the other hand, 10.64% of the conflicting merge scenarios were linked to bug-introducing commits in the `XXL` group. Since the number of BICs linked to conflicting merge scenarios appears more frequently in `XXL` scenarios, we ran a new hypothesis testing on this group. The results of the paired `Wilcoxon signed-rank test` over this sample compared to the sample of all commits, show that we still cannot reject the null hypothesis that both distributions are identical (p-value is 0.6089 with 95% confidence interval). The estimated Cliff's Delta between the distribution of conflicting merge scenarios (the more complex ones) and regular commits is 0.277, meaning that the difference is small.

`XXL` conflicting merge scenarios are responsible for more than 85% of the the conflicting merge scenarios linked to bug-introducing commits. Nonetheless, we did not find a significant difference in the likelihood of a bug-introducing come from a regular commit or from a `XXL` conflicting merge scenario.

Next, we investigate if the characteristics of a conflicting merge scenario could help to predict when a conflict merge scenario is more likely to introduce bugs (according to the `RA-SZZ`* algorithm). To this end, we first filter out the non-conflicting merge scenarios of our curated dataset and then selected a couple of features from the literature [30, 31] to use as predictors. Our feature set comprehends:

- The total of files changed in both branches
- The total number of contributors in both branches
- The total number of commits in both branches
- The total number of conflicting files
- The total number of days of development

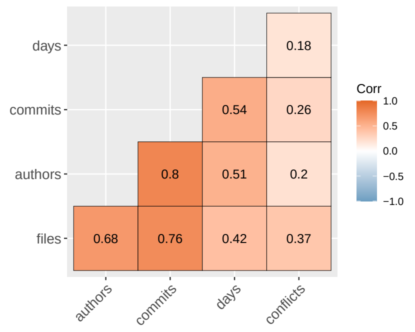

Fig. 6. Spearman correlation of the features collected from conflicted merge scenarios

As mentioned before, we compute these features by replaying all merge scenarios. We then investigate the Spearman correlation among these features (see the results in Figure 6). Similar to previous studies [30, 31], the number of changed files presents a moderate correlation coefficient (p-value = 0.37) while the number of active authors, the number of commits, and the number of days have weak correlations with

TABLE II
PERFORMANCE OF THE TRAINING CLASSIFIERS, CONSIDERING ALL CONFLICTING MERGE SCENARIOS (GROUP A) AND THE COMPLEX CONFLICTING MERGE SCENARIOS (GROUP B)

Group	Model	Precision	Recall	`f1-score`
A	Logistic Regression	0.26667	0.14286	0.18605
	Decision Trees	0.15789	0.3000	0.20690
	Random Forest	0.16667	0.46667	0.24561
B	Logistic Regression	0.4000	0.3077	0.3478
	Decision Trees	0.1159	0.8000	0.2025
	Random Forest	0.1500	0.6000	0.2400

the number of conflicting files (coefficients < 0.3). On the other hand, we found a strong correlation (coefficients ≥ 0.5) among the other features (number of changed files, contributors, and commits) and a moderate correlation (coefficient $=$ 0.42) between the number of days and the number of files changed.

In the process of data preparation and feature engineering, we explored our dataset to treat skewness on the predictors. The distribution of the number of files in conflict is equal three in the third quartile and we decided to run the classification models for all conflicting merge scenarios and for the complex merge scenarios—more than 2 files in conflict—separately. According to the curated dataset, `RA-SZZ`* linked 9.43% of all conflicting merge scenarios to BICs and linked 14.57% of bug-introducing commits to complex conflicting merge scenarios. Finally, we experimented with three classifiers—Logistic Regression, Decision Trees, and Random Forest—considering first all merge scenarios and then the complex merge scenarios only.

Table II shows the performance results of the classifiers we trained in this investigation. The results show that, overall, based on the outcomes of the three classifiers, it is hard to predict if a conflicting merge scenario will be responsible for introducing bugs. That is, when considering all conflicting merge scenarios, the Random Forest classifier presented the best performance with a `f1-score` of 0.256, followed by Decision Trees (`f1-score` = 0.207). Logistic Regression led to a best performance when considering the complex merge scenarios (`f1-score` = 0.348 and precision = 0.4), while Decision Trees presented higher recall (80%) with f1-score (0.20).

Based on the results of different classification models, we consider that it is hard to predict when a conflicting merge scenario is more likely to introduce a bug. In the best scenario, the classification models achieved an `f1-score` of 0.348.

C. To what extent co-change dependencies metrics correlate to defect density?

The goal of this research question is to investigate the relationship between bug incidence and co-change dependencies. This question has been investigated before by D'Ambros et al.

[14], though using only three systems, while we collected data from 29 projects. According to their findings, bug predictions models can be improved when considering co-change dependencies (change-coupling in the previous work).

To answer this research question, we first use the change history of the systems to compute the co-change dependencies between software components (see Section III-D)—at the coarse-grained level only (i.e., files and classes). From the co-change dependencies, we compute two additional metrics, similarly to the work by D'Ambros et al. [14]: Number of Coupled Classes (NOCC) and Sum of Class Coupling (SOCC), using $n = 2$ as threshold—which showed the best performance in the previous work [14]. We use three datasets in this analysis. The first dataset contains the co-change data, consisting of observations with the name and the metrics NOCC and SOCC of the components. The second dataset contains the change history of all components—each row indicating that a commit changed a given component. The third dataset contains all bug-fixing commits of the systems, which we compute using the first phase of `RA-SZZ*`. We then merge these datasets and compute the number of non bug-fixing (NBC) and bug-fixing commits (BC) of every component. After that, we estimate the buggy ratio (Br) of a component c using Eq. (1).

$$Br(c) = \frac{BC(c)}{NBC(c) + BC(c)} \tag{1}$$

We use the Spearman correlation and simple linear regression analysis to estimate the strength of the relationships between NOCC and SOCC with the buggy ratio and the total number of bug-fixing commits of a component. Simple linear regression allows us to (a) investigate if there is a relationship between NOCC and SOCC with the defect density of the components (buggy ratio and number of bug-fixing commits) and also (b) explain how strong the relationship between these features and defect density are [52].

Table III shows some descriptive statistics from the co-change metrics observations. Interestingly, considering our final dataset, most of the observations rely on the interval from four to 25 co-change dependencies (first and third quartiles, respectively)—although we found a specific component with 1022 co-change dependencies. Since these unusual cases increase the mean value of NOCC and SOCC, we decided to remove the components having either NOCC > 25 or SOCC > 75 from our dataset.

TABLE III
DESCRIPTIVE STATISTICS FOR THE NOCC AND SOCC

Metric	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
NOCC	1	4	11	19.40	25	1022
SOCC	2	10	30	68.04	75	5984

Figure 7 shows a matrix of correlation for the metrics NOCC, SOCC, BFCs (number of bug-fixing commits), and Ratio (Buggy Ratio). In our research, different from the work by D'Ambros et al. [14], we found a small correlation between the number of bug-fixing commits and the metrics NOCC and SOCC. This might contradict their findings and suggest that co-change dependencies are not effective predictors for defect density. In addition, a correlation between NOCC and SOCC with the total number of bug-fixing commits might actually suggest that NOCC and SOCC correlate to the total number of commits of a component—something that is expected. That is, considering only the total of bug-fixing commits might mislead the conclusions, since there is a difference in the error proneness of a given component A with three bug-fixing commits and 10 non-bug-fixing commits (a buggy ratio of 23%) and another component B with the same number of bug-fixing commits and 20 non-bug-fixing commits (a buggy ratio of 13%). Previous work only consider the absolute value of number of bug-fixing commits. Accordingly, in Figure 7, we find a negligible correlation between the metrics NOCC and SOCC with the Buggy Ratio of a component, which might better characterize defect density.

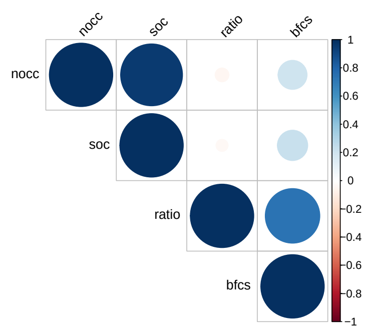

Fig. 7. Correlation matrix between the metrics NOCC, SOCC, BFCs (number of bug-fixing commits), and Ratio (Buggy Ratio)

We use linear regression models (m1.: $Ratio \approx \beta_1 \times NOCC + \beta_0$ and m2.: $Ratio \approx \beta_1 \times SOCC + \beta_0$) to investigate if we could predict *buggy ratio* using the metrics NOCC and SOCC. Although the p-values for both models suggest that exist associations between these predictors and the *buggy ratio*, the *adjusted* R^2 is also close to zero (for both models), supporting our findings that one cannot truly explain the buggy ratio variation in terms of NOCC and SOCC.

Altogether, we argue that components with high co-change dependencies are not more subject to defect density than other components. This result contradicts previous studies [14, 22, 43] that claim that co-change dependencies metrics are good predictors to bug introducing changes.

We also replicate the correlation analysis to all individual projects, considering the total number of bug-fixing commits of a component and the metrics NOCC and SOCC. In more than 80% of the projects, we found either a small (< 0.5) or a negligible correlation (< 0.3) for both metrics.

V. Discussion and Implications

Our research reveals that merge conflict resolution is responsible for a significant number of bug-introducing changes. From a total of 22,532 issues (characterized as bug-fixing commits), 336 (1.49%) are due to conflict resolution. Although one might consider this a small percentage, we argue that the source of these bugs arises exclusively due to conflict merge resolution. Researchers and tools developers could investigate the use of better approaches to help developers on the task of conflict resolution.

Besides that, we found that 9.43% of conflicting merge scenarios lead bug-introducing commits, revealing that commits related to conflict resolution are not more likely to introduce bugs (than regular commits that implement new features or fix other bugs, for instance). This might indicate that conflicting resolution is less error-prone than the literature suggests. Practitioners can benefit from this result, being more confident about the risks of introducing a bug when resolving merge conflicts and before deciding to postpone merge-operations.

We also show evidence that co-change dependency metrics do not correlate with defect density, contrasting with the findings of previous studies [14, 43, 22]. A possible reason for this discrepancy is that previous research works ground their conclusions using a smaller set of systems. Due to these conflicting results, we argue that further research should be conducted, either to confirm or refute our findings that co-change dependencies might not be efficient for predicting bugs.

VI. Threats to Validity

We leverage the SZZ framework to identify bug-fixing commits as well as to trace the source of those bugs—and then find the so called bug-introducing commits. Although, SZZ has known limitations, as we present in Section II, some of our study procedures mitigate part of its usage threats.

We highlight that (1) considering our curated dataset, 75% of the issues reported on JIRA were linked to bug-fixing commits in the first phase of SZZ; and (2) the SZZ implementation we used in our research (`RA-SZZ`*) was able to link 51% of the bug-fixing commits to bug-introducing commits. These numbers actually support the use of SZZ in our research, mitigating part of the critics about the use of SZZ. However, the set of bugs introduced and fixed in a source code repository over time might go beyond those handled by SZZ. For instance, a bug can be introduced and fixed before it is even reported in a bug tracker tool. In addition, some contributions might have not been correctly linked by developers with the bug issues associated (using the commit message). In spite of that, we also performed manual validations to mitigate reliability issues in our results.

Trying to improve the quality of our datasets, we removed several projects that we initially collected data from the repositories. For instance, we established a minimum number of 200 issues (1st quartile) on the JIRA issue database for each project. We also excluded from our analysis two projects in which the `RA-SZZ`* execution did not complete the process of linking the bug-fixing commits to bug-introducing commits within a time limit of 24 hours. We also removed from our analysis projects that we considered outliers, for instance due to its huge number of merge conflicts. As a final criteria, we excluded projects in which `RA-SZZ`* linked a small number of: (a) issues to bug-fixing commits, or (b) bug-fixing commits to bug-introducing commits. Therefore, we reduced the number of systems in our corpus, which might compromise the external validity of our study. However, we believe that this decision would not change our findings, because we still collected a reasonable number of issues, and more than 75% of them were linked to bug-fixing commits.

We probably did not collect all merge scenarios of the systems, since it is also possible to integrate contributions in a `git` repository using the `rebase` command—which removes merge operations from the log history. There is some controversial recommendations in the grey literature about considering rebase as either a harmful or a good practice. Nonetheless, our curated dataset presents a significant number of merge scenario, which might suggest that rebase is not a widespread practice in these projects.

To investigate the second research question, we had to estimate the number of bugs related to each component (i.e., Java classes). As such, we identified the commits that (a) affect a given class and (b) that also relate to bug fixes in the commit message. To this end, we leverage the first phase of the SZZ algorithm only. Although previous research works did not use the SZZ algorithm [14, 22, 43], we believe that our methodologies are quite similar (since previous works also associate commits to the issues databases). Therefore, the divergence of our findings is not full explained by our decision to use the first phase of SZZ. Instead, this divergence is more likely to occur due to our larger dataset of projects.

Also, to increase our confidence in our toolset for reproducing merge scenarios, we cross-validate the information of our dataset comprising merges and conflicts scenario with other datasets available in the literature [26, 31]. Nonetheless, despite using a curated dataset, we still believe that we cannot generalize our results to scenarios that do not explore the development practices of open source projects and use languages different than Java.

VII. Final Remarks

This paper has investigated the correlation between (a) conflict merge scenarios with bug-introducing commits and (b) co-change dependencies with defect density. We extracted 22,532 bug-fixing commits and 21,189 merge scenarios from 29 Java Apache projects and leverage `RA-SZZ`* to detect 29,245 bug-introducing commits. We gave evidence that commits that solve conflicting merge scenarios are not more likely to introduce bugs than regular commits—though we found that 9.43% of the conflicting merge scenarios lead to bug-introducing changes. In addition, contrasting to previous research [14], we found a small to negligible correlation between co-change dependency metrics and defect density.

REFERENCES

[1] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy, "If your version control system could talk," in *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, vol. 11, 1997.

[2] A. Zeller, *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[3] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 572–583.

[4] M. Beller, G. Gousios, and A. Zaidman, "How (much) do developers test?" in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 559–562.

[5] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[6] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design space of bug fixes and how developers navigate it," *IEEE Transactions on Software Engineering*, vol. 41, no. 1, pp. 65–81, 2014.

[7] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 492–501.

[8] K. E. Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *J. Syst. Softw.*, vol. 56, no. 1, p. 63–75, Feb. 2001.

[9] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010*, J. Wang, W. K. Chan, and F. Kuo, Eds. IEEE Computer Society, 2010, pp. 23–31. [Online]. Available: https://doi.org/10.1109/QSIC.2010.58

[10] A. Bacchelli, M. D'Ambros, and M. Lanza, "Are popular classes more defect prone?" in *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010. Paphos, Cyprus*, ser. Lecture Notes in Computer Science, D. S. Rosenblum and G. Taentzer, Eds., vol. 6013. Springer, 2010, pp. 59–73. [Online]. Available: https://doi.org/10.1007/978-3-642-12029-9_5

[11] A. E. Hassan, "Predicting faults using the complexity of code changes," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 78–88.

[12] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 181–190.

[13] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30. ACM, 2005, pp. 1–5.

[14] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *2009 16th Working Conference on Reverse Engineering*, Oct 2009, pp. 135–144.

[15] E. Kouroshfar, "Studying the effect of co-change dispersion on software quality," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 1450–1452.

[16] G. A. Oliva and M. A. Gerosa, "Experience report: How do structural dependencies influence change propagation? an empirical study," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 250–260.

[17] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.

[18] S. Horwitz, J. Prins, and T. Reps, "Integrating noninterfering versions of programs," *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 3, p. 345–387, Jul. 1989. [Online]. Available: https://doi.org/10.1145/65979.65980

[19] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE'12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2393596.2393648

[20] H. C. Estler, M. Nordio, C. A. Furia, and B. Meyer, "Awareness and merge conflicts in distributed software development," in *2014 IEEE 9th International Conference on Global Software Engineering*, 2014, pp. 26–35.

[21] P. Knab, M. Pinzger, and A. Bernstein, "Predicting defect densities in source code files with decision tree learners," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR'06. New York, NY, USA: Association for Computing Machinery, 2006, p. 119–125. [Online]. Available: https://doi.org/10.1145/1137983.1138012

[22] S. Kirbas, A. Sen, B. Caglayan, A. Bener, and R. Mahmutogullari, "The effect of evolutionary coupling on software defects: An industrial case study on a legacy system," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM'14. New York, NY, USA: Association for Computing Machinery, 2014.

[23] M. L. Guimarães and A. R. Silva, "Improving early detection of software merge conflicts," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 342–352.

[24] A. Sarma, G. Bortis, and A. Van Der Hoek, "Towards supporting awareness of indirect conflicts across software configuration management workspaces," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 94–103.

[25] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson, "Fastdash: a visual dashboard for fostering awareness in software teams," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007, pp. 1313–1322.

[26] S. Apel, O. Leßenich, and C. Lengauer, "Structured merge with auto-tuning: balancing precision and performance," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 120–129.

[27] O. Leßenich, S. Apel, and C. Lengauer, "Balancing precision and performance in structured merge," *Automated Software Engineering*, vol. 22, no. 3, pp. 367–397, 2015.

[28] G. Cavalcanti, P. Borba, and P. Accioly, "Should we replace our merge tools?" in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 325–327.

[29] P. Accioly, P. Borba, and G. Cavalcanti, "Understanding semi-structured merge conflict characteristics in open-source java projects," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2051–2085, 2018.

[30] 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, no. 2, pp. 279–313, 2018.

[31] M. Owhadi-Kareshk, S. Nadi, and J. Rubin, "Predicting merge conflicts in collaborative software development," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–11.

[32] J. Baumgartner, R. Kanzelman, H. Mony, and V. Paruthi,

"Incremental speculative merging," Apr. 26 2011, uS Patent 7,934,180.

[33] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 732–741.

[34] P. Accioly, P. Borba, L. Silva, and G. Cavalcanti, "Analyzing conflict predictors in open-source java projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 576–586.

[35] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma, "An empirical examination of the relationship between code smells and merge conflicts," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 58–67.

[36] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[37] D. Beyer and A. Noack, "Clustering software artifacts based on frequent common changes," in *13th International Workshop on Program Comprehension (IWPC'05)*, 2005, pp. 259–268.

[38] M. C. de Oliveira, R. Bonifácio, G. N. Ramos, and M. Ribeiro, "Unveiling and reasoning about co-change dependencies," in *Proceedings of the 15th International Conference on Modularity, MODULARITY 2016, Málaga, Spain*. ACM, 2016, pp. 25–36.

[39] N. Ajienka and A. Capiluppi, "Understanding the interplay between the logical and structural coupling of software classes," *Journal of Systems and Software*, vol. 134, pp. 120 – 137, 2017.

[40] L. L. Silva, M. T. Valente, and M. de Almeida Maia, "Co-change clusters: Extraction and application on assessing software modularity," *LNCS Trans. Aspect Oriented Softw. Dev.*, vol. 12, pp. 96–131, 2015.

[41] M. C. de Oliveira, D. Freitas, R. Bonifácio, G. Pinto, and D. Lo, "Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings," *J. Syst. Softw.*, vol. 158, 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2019.110420

[42] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[43] M. Steff and B. Russo, "Co-evolution of logical couplings and commits for defect estimation," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, 2012, pp. 213–216.

[44] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm," *Information and Software Technology*, vol. 99, pp. 164–176, 2018.

[45] C. Williams and J. Spacco, "Szz revisited: verifying when changes induce fixes," in *Proceedings of the 2008 workshop on Defects in large software systems*, 2008, pp. 32–36.

[46] E. C. Neto, D. A. da Costa, and U. Kulesza, "Revisiting and improving szz implementations," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–12.

[47] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proceedings of the 2013 international conference on software engineering*. IEEE Press, 2013, pp. 392–401.

[48] G. Rodríguez-Pérez, J. M. Gonzalez-Barahona, G. Robles, D. Dalipaj, and N. Sekitoleko, "Bugtracking: A tool to assist in the identification of bug reports," in *IFIP International Conference on Open Source Systems*. Springer, 2016, pp. 192–198.

[49] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: rethinking merge in revision control systems," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 190–200.

[50] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 92–101. [Online]. Available: https://doi.org/10.1145/2597073.2597074

[51] J. Maindonald and W. J. Braun, *Data Analysis and Graphics Using R: An Example-Based Approach*, 3rd ed., ser. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2010.

[52] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.