Empirical Software Engineering
https://doi.org/10.1007/s10664-019-09735-4

An empirical investigation into merge conflicts and their effect on software quality

Caius Brindescu[1] · **Iftekhar Ahmed**[2] · **Carlos Jensen**[1] · **Anita Sarma**[1]

Published online: 05 September 2019

Abstract
Merge conflicts are known to cause extra effort for developers, but little is known about their effect on software. While some research has been done, many questions remain. To better understand merge conflicts and their impact we performed an empirical study about the types, frequency, and impact of merge conflicts, where impact is measured in terms of bug fixing commits associated with conflicts. We analyzed 143 open source projects and found that almost 1 in 5 merges cause conflicts. In 75.23% of these cases, a developer needed to reflect on the program logic to resolve it. We also found that the code associated with a merge conflict is twice as likely to have a bug. When the code associated with merge conflicts require manual intervention, the code is 26× more likely to have a bug.

Keywords Version control · Software merging · Merge conflicts · Software quality · Empirical study · Mining software repositories

1 Introduction

Modern software development effort is, more often than not, a team effort. The complexity of software projects, such as the Linux Kernel has grown exponentially over the past years, and the Kernel now stands at over 21 million lines of code (Corbet and Kroah-Hartman

Communicated by: Sven Apel

✉ Caius Brindescu
brindesc@oregonstate.edu

Iftekhar Ahmed
iftekha@uci.edu

Carlos Jensen
carlos.jensen@oregonstate.edu

Anita Sarma
anita.sarma@oregonstate.edu

[1] Oregon State University, Corvallis, OR 97331, USA

[2] University of California, Irvine, Irvine, CA 92697, USA

2016). Software of such complexity requires a large development team to develop and maintain; last year alone more than 1,500 people made over 70,000 individual code contributions to the Linux Kernel (Corbet and Kroah-Hartman 2016). While most software systems are simpler than the Linux Kernel, the kind of distributed and collaborative work model which powers this project is broadly representative of how much of today's software development is done, especially in open source projects.

Modern Version Control Systems (VCS) have made parallel development easier by streamlining and coordinating code management and merging. For example, in Git, creating parallel branches or cloning an entire project can be done through a single command. This ability to create private development lines makes it easy for developers to experiment without impacting or being impacted by others' work. However, as is well known, isolation of private development lines can cause problems when changes are synchronized (Brun et al. 2011; Sarma et al. 2003; de Souza et al. 2003).

Merge conflicts can occur when developers make concurrent changes to the same code artifacts. The changes are generally authored by two different developers, but merge conflicts can also happen between the edits of one developer. While automatic merge tools help, manual intervention is required when changes overlap. Resolving merge conflicts, even easy ones, can disrupt the flow of programing, forcing developers to shift their focus on the resolution process. Other times, a conflict resolution requires a deeper understanding of the program's structure and goals. For example, when a function's signature or parameters are changed, a developer first needs to understand the rationale behind the change, and any dependencies, before she can resolve the conflict. Prior work has found that in complex merges, developers may not have the expertise or knowledge to make the right decisions (Costa et al. 2014; Nieminen 2012), which might degrade the quality of the merged code.

Another problem associated with merge conflicts is that they often take the developer outside of their development process. For instance, a developer may follow an established process of peer review of code submissions, but the merged code maybe "cobbled" together. This could lead to an increased likelihood of bugs slipping through, as could an incomplete understanding of the nature of changes and dependencies.

Prior research has shown that merging long lived branches can lead to merge conflicts (when the VCS fails to merge the two branches due to current edits to the same lines, also called *direct* conflicts), or *indirect* conflicts (integration failures where the merged code does not compile, or tests fail) (Bird et al. 2009; Perry et al. 2001; Shihab et al. 2012). While others have shown that merge conflict occurrence is frequent (Perry et al. 2001), what is not known is whether the code emerging from the resolution of merge conflicts is sub-par, whether different types of merge conflicts affect the resultant code quality differently, and whether there are any differences in the merged code created by experienced developers' as compared to novices.

Currently, therefore, there is an important gap in our understanding of the nature of merge conflicts and the results of their resolution. Closing these gaps can help us not just build better tools and processes to help developers, but also develop a more nuanced and in depth understanding of the risks and problems associated with merge conflicts.

The goal of this paper is to *characterize the different types of merge conflicts and identify any effects they have on the quality of the resulting code*. We ask the following research questions:

- RQ1: What are the most common types of merge conflicts?
- RQ2: How likely is code resulting from a merge conflict to contain bugs?

– RQ3: What are the factors that affect the quality of the code resulting from a merge conflict resolution?
– RQ4: What are the resolution strategies used by developers when resolving merge conflicts?

To answer these questions, we analyzed a broad sample of 143 open source projects. From these projects, we collected 556,911 commits, 36,122 of which were merge commits. 19.32% of these merges, in turn, resulted in conflicts.

Our research identified six types of direct merge conflicts. Some types of conflicts require resolution that is trivial (e.g. formatting differences), while others are challenging (e.g. those that make changes to the underlying Abstract Syntax Tree). About 60% of conflict resolutions in our dataset involve changes to the Abstract Syntax Tree (AST) that are entangled (henceforth called SEMANTIC merge conflicts). We also found that code associated with a merge conflict is twice as likely to have a bug, and code associated with SEMANTIC merge conflicts are 26 times more likely to have a bug compared to code associated with other conflicts.

In summary, our contributions are:

– A taxonomy of merge conflicts;
– Empirical results that show that direct merge conflicts are indicative of changes that are bug-prone;
– Identification of factors that are associated with direct merge conflicts that are bug-prone;
– Identification of most commonly used resolution strategies.

2 Related Work

2.1 Merge Conflicts

Researchers have looked at strategies for merging code, problems associated with them, and how to proactively avoid them in order to support collaborative development efforts. Mens (2002) present a survey on the state of the art of software merging. Their survey goes into depth regarding merge strategies and ways to reduce conflicts. However, it does not provide evidence of the kinds of problems merge conflicts present.

Empircal Studies: Merge conflicts are known to be costly (Grinter 1995; Perry et al. 2001; de Souza et al. 2003). They delay the project, requiring an examination of the conflict, and developing a consensus solution. Several empirical studies have detailed how merge conflicts are a problem, and the strategies that developers follow to evade having to resolve conflicts. For example, Perry et al. (2001) found that increased parallel work, in addition to causing conflicts, can also lead to an increase in software defects. Developers are known to follow informal processes (e.g., check in partial code, email the team about impending changes) to avoid having to resolve conflicts when committing changes [25], or rush to commit their work in an effort to avoid being the developer who has to resolve the conflicts (de Souza et al. 2003). A developer may also choose to delay the incorporation of others' work, fearing that a conflict may be hard to resolve (de Souza et al. 2003). Such processes can have a detrimental effect on team productivity and morale. Our work is the first to investigate the root cause of a merge conflict, and to try and quantify how problematic merge conflicts are for developers.

Proactive conflict detection: There have been a number of papers that focus on developing tools to proactively detect or avoid merge conflicts. Sarma et al. (2003) and Ripley et al. (2004) presented Palantír, a tool that helps make developers aware of changes to each other's workspaces. Similarly, Biehl et al. (2007) presented FASTDash, and da Silva et al. (2006) introduced Lighthouse, which also try to foster awareness among developers. Kasi and Sarma (2013) presented Cassandra, a tool that schedules tasks in order to minimize the chance of conflicts occurring. Brun et al. (2011, 2013) developed Crystal, which identifies two different types of merges conflicts: "*textual*," which are detected by the version control system's merge tool, and "*higher level conflicts*," which are not detected until a build or test fails. The tool merges changes in a shadow repository as they are committed in order to catch these types of conflict as early as possible. Similarly, Guimarães and Silva (2012) introduced a technique to continuously merge changes in the IDE in order to detect merge conflicts as soon as possible. Servant et al. (2010) presented a tool and visualization that enable developers to understand the impact of their changes, which can then prevent indirect conflicts. Dewan and Hegde (2007) presented a software development model aimed at reducing conflicts by notifying developers when they work on the same file, and allowing them to collaborate when resolving conflicts. While proactive detection tools help developers prevent merge conflicts, it is not always possible (e.g. important security fixes cannot be delayed to avoid such a conflict). Our work aims to bring understanding of the merge conflicts that do occur.

Merge help: Researchers have investigated techniques to manage merging of changes, in order to more efficiently resolve conflicts, either in an automated way, or by preserving and presenting useful context for the developer trying to resolve the conflict. Apel et al. (2011, 2012) and Cavalcanti et al. (2017) presented a new merging technique, semistructured merge, which considers the structure of the code being merged. Lippe and van Oosterom (1992) presented Operation Based Merging, which, when merging, considers the changes performed, not just the end result. However, McKee et al. (2017) have shown that developers do not trust tools if they don't understand how they work, and they prefer to resolve merge conflicts manually. Our work can help researches focus on the merge conflicts that are more "painful" for developers.

Conflict categorization: Finally, researchers have looked at ways of categorizing conflicts. Sarma et al. (2003) categorized conflicts as *direct conflicts*, when parallel files have been changed and *indirect conflicts*, when parallel changes to dependent files cause a conflict. Similarly, Brun et al. (2011), distinguish between first level (textual) conflicts from second level (build and test failure) conflicts. Buckley et al. (2005) proposed a taxonomy of changes based on properties like time of change, change history, artifact granularity etc. Their taxonomy deals with software changes in general or conflicts at a coarser level. We are intersted in creating a taxonomy that provides finer details about merge conflicts and the impact of their resolution. Finally, Accioly et al. (2018) and Menezes (2016) present a classification that considers the types of changes that generate the conflict. Our approach is different, as we are using a human-centered approach at identifying the root cause of a conflict. We are the first to propose and validate a categorization that looks at the root cause of a merge conflict.

2.2 Measuring Software Quality

One goal of our research is measuring the impact that conflicts have on software quality. Various measures of software quality have been proposed. Boehm et al. (1976), and Gorton and Liu (2002), to mention a few, have explored measures including completeness,

usability, testability, maintainability, reliability, efficiency, etc. Some of these metrics are difficult to measure, especially in the absence of requirement documents or other supporting information. Researchers have also used code smells as a measurement of software quality (Marinescu 2001, 2004), though smells are often focused on future maintainability issues.

2.3 Tracking Code Changes and Conflicts

Our analysis requires mining the history of each line of code to determine if and when it was involved in a merge conflict, a software patch/upgrade, or a bug fix. Researchers have proposed various algorithms for tracking individual lines of code across versions of software. Canfora et al. (2007) proposed an algorithm that uses Levenstein edit distance to compute similarity of lines, matching "chunks" of changed code. Zimmermann et al. (2006) proposed annotation graphs which work at the region level for tracking lines. Godfrey and Zou (2005) described "origin analysis," a technique for tracking entities across multiple revisions of a code base by storing inexpensively computed and easily comparable "fingerprints" of interesting software entities in each revision of a file. These fingerprints can then be used to identify areas of the code that are likely to match before applying more expensive techniques to track code entities. Kim et al. (2006) propose a seminal algorithm, SZZ, for tracking the origin of lines across changes. Finally, Falleri et al. (2014) propose the GumTree algorithm for tracking changes at an AST level. This is the approach that we chose for our experiments.

3 Methodology

Our goal here is to quantify and categorize merge conflicts across a wide set of representative open source projects, and the fault-proneness of the resulting code (whether these changes were associated with bug fixes or other improvements). Additionally, we want to collect relevant metrics about the projects themselves, and the contributors (such as whether these contributors were part of the core development team or intermittent/new developers).

To achieve these goals, we first select a project sampling strategy that allows us to gather data from projects that are representative of the kind of development practice we are interested in. We then track the lines of code through versions and code merges in order to study how the code evolved, and which lines were associated with conflicts, updates, and bug fixes. Next, we determine the nature of code updates (e.g. was this a bug fix, or a new feature, etc.). In order to do this, we manually classify a subset of the commits and trained an automated classifier to classify the rest. Finally, we use the data to build a model to predict the total number of bug fixes that would occur on a conflicting line. The following subsections describe each of these steps in detail, and the overall picture is presented in Fig. 1.

We also released the tooling we used for analyzing the projects. The tool for identifying merge conflicts is available here: https://github.com/caiusb/conflict-detector. The tool for collecting the metrics is available here: https://github.com/caiusb/MergeConflictAnalysis. Finally, our statement tracker is available here: https://github.com/caiusb/statement-history.

3.1 Project Sampling

We collected projects from GitHub (2017) for our empirical evaluation. Our goal was to ensure that the projects chosen offered a reasonably unbiased representation of modern software practices. We also tried to reduce the number of variables that could contribute

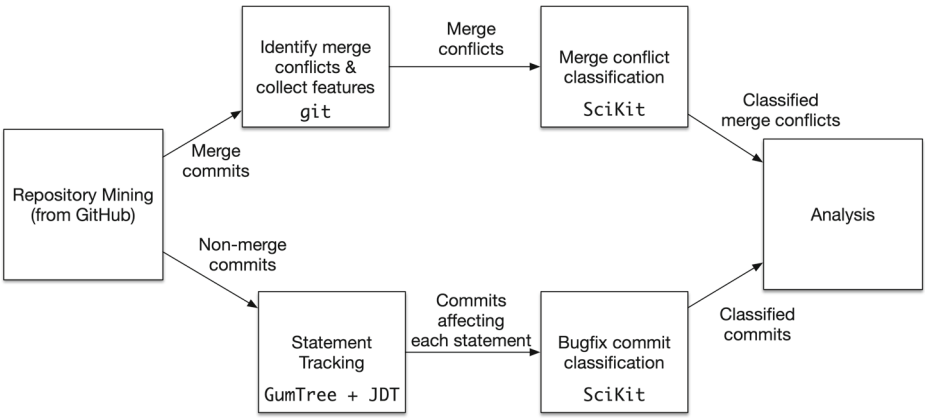


Fig. 1 An overview of our data collection and analysis process. Tools used are listed using a `monospaced typeface` in the lower part of the boxes. JDT stands for Java Development Toolkit (https://www.eclipse.org/jdt/)

to random noise during evaluation. With these goals in mind, we decided to focus on Java projects using the popular Maven build system (Apache Maven 2018). This decision was influenced by the fact that Java is one of the most popular languages (according to the number of projects hosted on GitHub (2017)), and the availability of analysis tools.

We started by randomly selecting 900 projects, the first to show up when using the GitHub search mechanism. From these, we eliminated aggregate projects (which could skew our results), leaving 500 projects. After eliminating projects in which we could not compile more than half of the merge commits (for reasons such as unavailable dependencies, or compilation errors due to syntax or bad configurations), 312 projects remained. Finally, we eliminated projects that our AST difference tool (Falleri et al. 2014) could not handle. This left us with a total of 200 projects.

We followed the guidelines presented by Kalliamvakou et al. (2014) for mining Git repositories. We removed projects that were too small, that is, having fewer than 10 files, or fewer than 500 lines of code, or those projects that were not active in the past 6 months. We also removed projects that had no merge conflicts. This was essential because there is a long tail of small and short-lived projects on GitHub, which include trial projects, projects with single author or no parallel development. Since these projects do not represent the kinds of development efforts we are interested in, we remove them from consideration. The thresholds chosen are based on similar studies (Ahmed et al. 2017).

Our final data set contained 143 projects across different domains. As a check on our sampling, we manually categorized the domains of our projects by looking at their project description, and using the categories used by De Souza and Maia (2013). Table 1 presents the summary of the domains of the projects.

Next, we discuss the individual project characteristics. Table 2 provides a summary of features and other descriptive information of the projects in our study.

The line counts (lines of code in the project) were taken from the version of the code that was in the repository on March 1st, 2016, and the duration is the number of days from the date of the first commits to March 1st, 2016.[1] The number of developers is the number of

[1]Some projects may have migrated to GitHub from other platforms, so this is a lower-bound figure

Table 1 Distribution of projects by domain

Domain	Percentage
Development	61.98%
System Administration	12.66%
Communications	6.42%
Business & Enterprise	8.10%
Home & Education	3.11%
Security & Utilities	2.61%
Games	3.08%
Audio & Video	2.04%

unique individuals that contributed at least once over the life of the project. Individuals were identified by the name in the "Author" field in each Git commit. The standard deviations are high, which suggest that our sample contains a diverse set of projects. This acts in favor of making our findings more generalizable.

From our sample of 143 projects we extracted 556,911 commits. This included 36,122 merge commits. Our data shows a high standard deviation in the total number of commits as well as merges. Therefore, we further investigate the distribution of merge commits (see Fig. 2). From the figure we can see that merge commits are not scarce, as projects have an average of 252.6 merge commits. Out of all the merges, we identified 6,979 (19.32%) conflicts, as described in the next section.

3.2 Conflict Identification

Since Git does not explicitly record information about merge conflicts, we recreate each merge in the corpus to determine if a conflict had occurred. We use Git's default algorithm, the *recursive merge strategy*, as this is likely to be most commonly used by the average Git project.

This also allows us to identify each conflicting commit and the affected file. More specifically, we used the `git merge` command that automatically merges the commits, and flags merges with overlapping changes as merge conflicts. The distribution of merge conflicts is shown in Fig. 3. We see that projects experienced an average of 25 merge conflicts, or 19.32% of all merges. Merge conflicts, therefore, are a common part of the developer experience (in our dataset).

We then collect statistics regarding each file involved in a conflict, including files that have conflict and those that merged cleanly. We track the size of the changes being merged,

Table 2 Project characteristics

Dimension	Max	Min	Average	Std. dev.
LOC	542,571	751	75,795.04	105,280.10
Duration (Days)	6,386	42	1,674.54	1,112.11
# of Developers	105	4	72.76	83.19
Total Commits	30,519	16	3,894.48	5,070.73
Total Merges	4,916	1	252.60	522.73
Total Conflicts	227	1	25.86	39.49

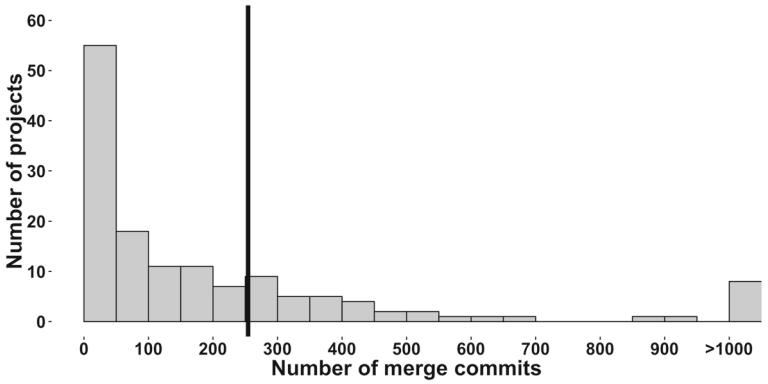


Fig. 2 Distribution of merge commits. The vertical line represents the mean (252.60)

the difference between the two branches (in terms of LOC, AST difference, and the number of methods and classes involved). We track the types of AST nodes involved (e.g., `BinaryExpression`, `MethodInvocation`, `ExpressionStatement` etc.) There were 81 node types in total. We use the Gumtree algorithm (Falleri et al. 2014) to determine the AST differences. We also collect meta information about the merge, for example, if the change was merged into the master branch or not. Finally, we track the number of authors involved in the merge.

In this the paper, we only analyze merge conflicts, that are detected by Git, and we do not consider indirect conflicts, that are not detected by Git, but are noticeable because of build or test failures.

3.3 Conflict Types

To understand the root cause for each conflict we manually investigated and classified 606 randomly sampled commits. We classify each conflict based on the type of changes causing the merge conflict (e.g., whitespace or comment added vs. variable name changed). When classifying a conflict into a category, we chose the most "severe" category. As an example, if a merge contained conflicts in both comments (FORMATTING) and program logic (SEMANTIC), we classify it as SEMANTIC. We do so since we want to identify those

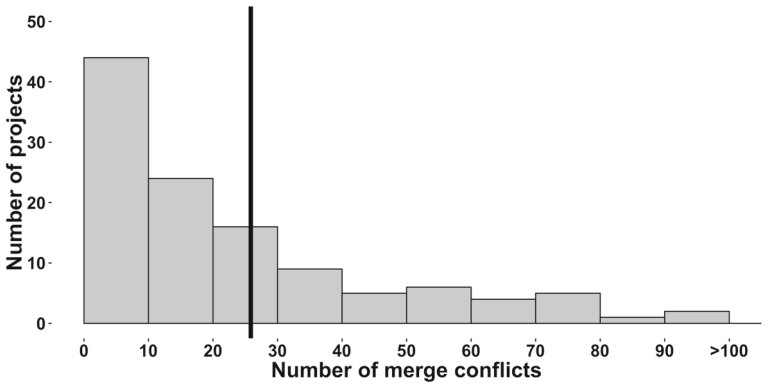


Fig. 3 Distribution of merge conflicts. The vertical line represents a mean of 25.86 conflicts per project

Table 3 Conflict categories. A semantic change is a change that affects the program logic

Category	Definition	C.[a]	Example
SEMANTIC	Two conflicting semantic changes, where two different changes in the program logic overlap	NT[b]	https://github.com/zanata/zanata-server/commit/49fda3
DISJOINT	Semantically unrelated changes that overlap textually	NT[b]	https://github.com/jdktomcat/commit/0cbbd0
DELETE	A conflict in which one of the branches deletes code modified on the other branch	NT[b]	https://github.com/osmandapp/Osmand/commit/defe2e
FORMATTING	Conflicting changes due to formatting (whitespace changes)	T[c]	https://github.com/scudderfish/MSLoggerBase/commit/b495d3
COMMENTS	Conflicting changes are limited to comments only	T[c]	https://github.com/scudderfish/MSLoggerBase/commit/b495d3
OTHER	Not belonging to any of the above	T[c]	

[a]Complexity [b]Non-trivial [c]Trivial

conflicts that require the most developer reasoning (at least for some part of the conflict). The first two authors independently coded 300 of these commits using qualitative thematic coding (Cruzes and Dyba 2011). Using Cohen's Kappa, they achieved an inter-rater agreement of 0.84 on 20% of the data. The first author then classified the remaining 306 commits. The codes and their definitions are given in Table 3. Detailed examples for each category can be found in the companion website (Companion website 2016). Next, we will present an example of each category of NON-TRIVIAL merge conflicts.

Listing 1 presents a SEMANTIC conflict from our manually classified corpus. In this example, we have two refactorings (renaming `resultDate` to `projectLastModificationDate` and `sdf` to `dateFormat`) as well as introducing and using a new

```
90  <<<<<<< HEAD File projectXMLFile = new File(Utils.buildPath
        (Utils.buildProjectPath(projectName), Consts.
        PROJECTCODE_NAME)); SimpleDateFormat dateFormat = new
        SimpleDateFormat("dd.MM.yyyy HH:mm"); Date
        projectLastModificationDate = new Date(projectXMLFile.
        lastModified()); holder.dateChanged.setText(dateFormat.
        format(projectLastModificationDate));
91  =======
92  //set last changed:
93  SimpleDateFormat sdf = new SimpleDateFormat("dd.MM.yy HH:mm
        "); Date resultDate = new Date(projectData.lastChanged)
        ; holder.dateChanged.setText(sdf.format(resultDate));
94  >>>>>>> 20b7c5e
```

Listing 1 An example of a SEMANTIC merge conflict. Taken from the Catdroid project, commit `c03c15` (https://github.com/Catrobat/Catroid/commit/c03c15)

```
74 <<<<<<< HEAD R.string.formula_editor_function_round, R.
       string.formula_editor_function_true, R.string.
       formula_editor_function_false };
75 =======
76 R.string.formula_editor_function_round, R.string.
       formula_editor_function_mod };
77 >>>>>>> f665d49
```

Listing 2 Example of a DISJOINT merge conflict. Taken from the Catdroid project, commit `3ba518` (https://github.com/Catrobat/Catroid/commit/3ba518)

variable (`projectXMLFile`). The developer resolving this merge conflict would have to untangle the changes, in order to keep the correct refactorings and make sure that the new variable is used where required.

Listing 2 presents an example of a DISJOINT merge conflict. In this example, two developers added different values to an existing enum. In this case, the resolution is straight forward, as the correct solution is having all the enum values added on both branches.

Finally, Listing 3 shows the Git's output when a file is deleted on one branch, and modified in the other. In this case, the developers is left with the modified file. It's up to them to find out why was the file deleted. It is also up to them to understand the changes that were made, and decide if to keep the file, or if the changes need to be reimplemented elsewhere, if the code was reorganized as part of a larger refactoring.

3.4 Conflict Type Classification

We use this set of 606 (10%) commits as training data for a machine learning classifier to use on the full set of merge commits. Each class is classified using an Adaptive Boost (AdaBoost) ensemble classifier, where we use 100 Decision Trees as weak classifiers. We choose AdaBoost as it had the highest performance (precision and recall) when compared with Support Vector machine (SVM) for our dataset. This was not surprising as ensemble of classifiers have shown significantly improved performance in other domains such as prediction (Sun et al. 2012). We categorize all of the 6,979 conflicting commits using AdaBoost.

Performance of any prediction is dependent on the features used. Therefore, we wanted to use a comprehensive set of features. We performed a literature search, and we used factors from these 2 papers (Apel et al. 2011; Mens 2002) to decide on the final set. We also included features that are known to influence the comprehension of a program, such as changes size and the spread of the change (number of affected program elements).

```
CONFLICT (modify/delete): src/org/wordpress/android/ui/
    notifications/BigBadgeFragment.java deleted in HEAD and
     modified in fff496c. Version fff496c of src/org/
    wordpress/android/ui/notifications/BigBadge
```

Listing 3 Git's output when encountering a DELETE merge conflict. Taken from the WordPress-Android project, commit `5fe68c` (https://github.com/wordpress-mobile/WordPress-Android/commit/5fe68c)

We gathered these factors from either the Git repository, or we derived them by analyzing the source code, when the factors are related to the process and code metrics (characterized in numerical form)

To train the classifier we use a set of 24 features, including: the total size of the versions involved in a conflict, the size of the conflicting area, the number of statements, methods and classes involved in the conflict. The complete list can be found in Table 4 and in our companion website (Companion website 2016). The features were chosen based on the existing literature. We also considered factors that are known to influence the comprehension of a program, as well as the authors' experience.

Our goal was to achieve high precision and recall. We use 10 fold cross-validation to test the performance of our classifier, measured using the F1-score. The F1-score considers precision and recall by taking their harmonic mean. The average F1-score of the 10 rounds for the conflict type classifier is 0.64, the precision is high at 0.75. The F1-score is defined as the harmonic mean between the precision and recall scores.

We present the confusion matrix in Fig. 4. The diagonal elements represent the number of points for which the predicted label is equal to the true label, while off-diagonal elements are those that are mislabeled by the classifier. While our classifier has decent results, it's worth noting that there are some mislabeled commits. This is happening because for some

Table 4 Features used to train the classifier

Feature	Description
AST Size[a]	The total number of AST Nodes involved in a merge conflict
LOC Size[a]	The sum of the LOC of the files involved in a conflict
AST diff size between the branches	The difference in number of AST nodes involved in the conflict
AST diff branch-solution[a]	The difference in AST nodes between a branch and the merge conflict resolution solution
LOC diff size between the branches	The total number of LOC involved in the conflict
LOC diff branch-solution[a]	The difference in AST nodes between a branch and the merge conflict resolution solution
AST Size of the solution	The total number of nodes for the solved merge conflict (only the files affected by a conflict)
LOC Size of the solution	The LOC size of the solved merge conflict (only the files affected by a conflict)
# authors	The number of authors whose changes are involved in a merge conflict
Merged in master	"*True*" if the branch was merged in the master branch, "*False*" otherwise (was merged in a different branch)
Branch time[a]	The timestamp of the last commit on each branch
Solution time	The timestamp of the merge conflict resolution commit
# methods	The total number of methods involved in the merge conflict
# classes	The total number of classes involved in a merge conflict
# statements	The total number of statements involved in a merge conflict.
AST Nodes in conflict	The total number of AST nodes involved in the merge conflict.
Is AST Conflict	"*True*" if the merge conflict is at the AST level.

[a]Collected for both branches

Predicted classes \ Actual classes	Comments	Delete	Formatting	Other	Semantic	Disjoint
Comments	10	0	1	0	2	0
Delete	4	4	0	1	0	3
Formatting	13	0	3	2	30	2
Other	27	4	4	0	17	18
Semantic	69	0	9	5	219	23
Disjoint	41	2	5	7	68	14

Fig. 4 The confusion matrix for our classifier

categories (e.g. COMMENTS) less training data was available. We discuss this further in the Threats to Validity (Section 5).

3.5 Tracking Statements

We needed to track statements that were involved in merge conflicts in order to track the eventual outcome of merge conflicts. We decided to use GumTree (Falleri et al. 2014) for our analysis, as it allows us to track elements at an AST level. This way we can track only the elements that we are interested in (statements), and ignore other changes that do not effectively change the code. The GumTree algorithm works by determining if any AST node was changed, or had any children added, deleted or modified. The algorithm maps the correspondence between nodes in two different trees, which allows it to accurately track the history of the program elements. This algorithm has unique advantages over other line tracking algorithms, such as SZZ (Kim et al. 2006). These advantages include: ignoring whitespace changes, tracking a node even if its position in the file changes (e.g. because lines have been added or deleted before the node of interest), and tracking nodes across refactorings, as long as the node stays within the same file. Using this technique, we can track a node even when it has been moved, for example, because of an extract method refactoring.

For each statement of interest, we use the version of the source code at the point preceding the merge conflict as the starting point. We use it to identify the AST nodes corresponding to the line of interest. We consider the code as changed if the AST node was changed, or had children that were added, deleted or modified. An example of a change to AST node is: changing `int x = 0;` to `int y = 0;`. This modifies the AST node (`SimpleName`: `x` in the first snippet) by changing its `name` property (`x` to `y`). Similarly, the change `x = x +1;` to `x = 1;` modifies the node because it modifies it's subtree. The algorithm maps the correspondence between nodes in two different trees, which allows us to track the history of any statement.

AST differencing has three advantages over simple line based differencing. The first is that it ignores whitespace changes. Second, we are able to track a node even if its position in the file changes (e.g. lines are added or deleted before the node of interest). Third, we are able to track nodes across refactorings, as long as the node stays within the same file.

For each node involved in a conflict, we identified all future commits that touched the file containing that node. To do so, we track the Java statement that corresponds to the (changed)

AST node. Note that in Java, there might be multiple statements in the same line (e.g., a large `if-else` statement block), therefore, it is important to track only the statements that corresponds to the changed AST node.

We then repeated the same analysis for statements that were not involved in a conflict to determine if there was any difference between lines associated with a merge conflict and those which were not.

3.6 Commit Classification

In order to answer research questions 2 and 3 we needed to group commits into one of two categories: (1) bug-fixes and improvements (modifying existing code), and (2) commits that introduced new functionality (adding new code) or were related to documentation, test code etc.

We investigate bug-fixes as it gives an objective measure for the definition of quality for open source projects, which often lack detailed requirements, roadmaps, or even test harnesses. The more bug fixes and changes a line of code faces over a period of time, the more one can argue that that line of code was incomplete or poorly implemented (Ahmed et al. 2016). As this measure works at the individual line level, just like merge conflicts, we decided to use it as our measure of code quality.

It is not always trivial to determine which category a commit falls under, especially when larger projects see a large amount of activity. Manual classification of commits was therefore not an option, and we decided to use machine learning techniques.

In order to build a classifier, we randomly selected and manually labeled a set of 1,500 commits. Two evaluators worked independently to classify the commits. Their datasets had a 33% overlap, which we used to calculate the inter-rater reliability. This gave us a Cohen's Kappa of 0.90. In our training dataset, the portion of bug-fixes was 46.30%, with 53.70% of the commits assigned to the "Other" category. Some keywords indicating bug-fixes or improvements were "Fix," "Bug," "Resolves," "Cleanup," "Optimize," and, "Simplify," together with their derivatives. Anything that did not fit into this pattern was marked as "Other."

Not all bug-fixing commits include these keywords or a direct reference to an issue-id; commit messages are written by the initial contributor, and there are few guidelines. A similar observation was made by Bird et al. (2009), who performed an empirical study showing that bias could be introduced due to missing linkages between commits and bugs.

We trained a Naive-Bayes (NB) classifier and a Support Vector Machine (SVM) using the SciKit toolset (Pedregosa et al. 2011). The frequencies of the words in the commit message were used as the predictors. We used 10% of the data to train the classifier. We applied the classifiers to the training data with 10-fold cross-validation. As before, we used the F1-score to measure and compare the performance of the models. The NB classifier outperformed the SVM. We used the NB classifier to classify the full set of 16,571 commits.

Table 5 has the quality indicator characteristics of the NB classifier. Tian et al. (2012), suggest that for keyword-based classification the F1-score is usually around 0.55, which happened in our case. While our classifier is far from perfect, it is comparable to "good" classifiers in the literature. Further, we believe it is unlikely for the biases to have a confounding effect on our analysis. Since our analysis only relies on relative counts of bug-fixes for statements, as long as we do not systematically undercount bug-fixes for only some statements, our results should be valid. A manual inspection of the classification results did not show any evidence of systemic over- or under-counting.

Table 5 Details of the bugfix (Naive Bayes) classifier

	Precision	Recall	F1-score
Bug-fix	0.63	0.43	0.51
Other	0.74	0.86	0.80

For each line of code resulting from a merge conflict, we count the number of future commits in which it appears, as long as those commits are identified as bug-fixes. We stop tracking when we encounter a commit that is classified as "Other" (see Fig. 5, where we count commits C_1 and C_2, but not C_n). Our reasoning is that once an element has seen a change that is not a bug-fix, it is no longer fair to assume that subsequent bug fixes are associated with the original merge conflict.

3.7 Core Authors Identification

We categorize developers as core or non-core based on the amount of their contributions. This is because, typically, a small core team is responsible for more than 80% of contributions to open source projects (Mockus et al. 2002). Therefore, those developers who are in the core are likely those with higher experience. We calculate this by first splitting the project history into quarters. We then identify those developers who had the most contributions in a quarter. We do so because in open source there is high developer turnover, or developers become inactive for periods of time, therefore, it is better to gauge experience in shorter time periods. We identify developers as core contributors by evaluating if they are in the top 20% (in terms of number of commits) of the developers in that quarter.

We note that some developers started as non-core and transitioned to core, whereas some went from active to inactive. Therefore, an author can switch between core and non-core across quarters, based on their levels of contribution, and vice-versa.

3.8 Regression Analysis

In order to answer our third research question, we needed to build a regression model to identify the factors that impact the number of bug fixes occurring on lines of code resulting from merge conflicts. We use Generalized Linear Regression (Cohen et al. 2013). In our data, the dependent variable (count of bug fixes occurring on conflict lines) follows a Poisson distribution. Therefore, we use a Poisson regression model with a log linking function.

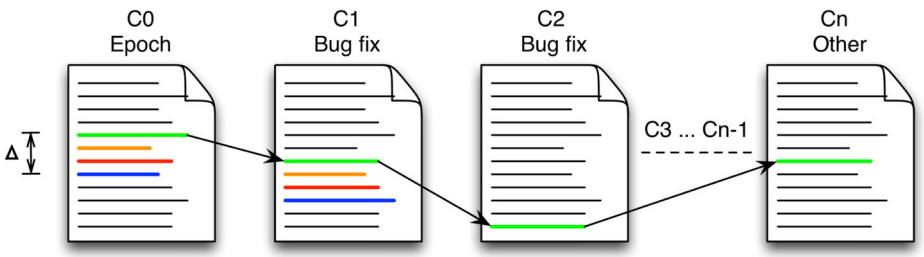


Fig. 5 An overview of the tracking algorithm. The lines marked with Δ represent the conflicting region of a merge. For each line we identify the AST nodes, and track all the modifications (at AST level) forward in time. We stop when we hit a commit that was classified as *Other*

We include the following factors in our regression model: file dependencies, source code and change metrics, and author related metrics. For calculating the dependencies of the files that are involved in a conflict, we use *Understand* (2017) to count the number of references to- and from other files. We collect this information as a proxy for the importance of the file. We assume that the more a file is referenced by other files, the more central that file is, and hence more important. Any change in these central files can increase the chance of a change being required in other files.

For each conflict commit, we record the information about the size of the change – the difference between the two merged branches, in terms of LOC, AST difference, and the number of methods and classes being affected. Our intuition is that larger changes should have a higher chance of causing a conflict. We also calculate the number of authors who made commits to the branches that were merged.

After collecting these metrics, we checked for multi-collinearity using the Variance Inflation Factor (VIF) of each predictor in our model (Cohen et al. 2013). VIF describes the level of multicollinearity (correlation between predictors). A VIF score between 1 and 5 indicates moderate correlation with other factors, so we selected the predictors with VIF score threshold of 5. This step was necessary since the presence of highly correlated factors forces the estimated regression coefficient of one variable to depend on other predictor variables that are included in the model.

3.9 Identifying Resolution Strategies

Not all conflicts are created equal, and their resolution is dependent on the nature of the changes being merged. A developer can choose different strategies for resolving conflicts. The easiest strategy is to simply *select one* of the branches that is being merged. While this might work for trivial conflicts (such as FORMATTING), it might not work for more complex conflicts. In some cases, it might be possible to mix and match the changes by *interleaving* existing code from the two branches. However, in more complicated cases, a developer might need to *adapt* the code from both branches to successfully merge the changes.

In order to determine the strategy that developers had used, we look at the existing solutions to merge conflicts. For each conflict we perform a line difference between the solution and the tips of the two branches being merged. Lines that were different from one branch, but not the other are considered to have been selected by the developer for integration. Lines that are different from both branches implies that they have been changed in order to be successfully integrated. Based on these observations we identified the following three resolution strategies:

1. SELECT ONE: The solution is the same as one of the branches (the difference to one of the branches is 0);
2. INTERLEAVE: The solutions contains lines from both the branches; none of the lines were changed and no new lines were added (we can match each line in the solution to a line in one of the branches);
3. ADAPTED: Existing lines were changed or/and new lines were added.

4 Results

In the following section, we present our results structured around our four research questions.

4.1 Merge Conflict Characteristics (RQ1)

In our dataset, 19.32% of merges resulted in merge conflicts (6,979 merge conflicts in 36,122 merge commits). This means that *almost 1 in 5 merges resulted in conflicts that required human intervention*. Our results are similar to those found by Brun et al. (2013) and Sarma et al. (2003). A merge conflict not only means that developers have to stop their work, reason about the conflicting changes, and figure out the best way to integrate the changes. It also creates a situation where it is possible for bugs to slip through if developers do not refactor and run test cases, or have their proposed (adapted) code peer reviewed.

4.1.1 Types of Merge Conflicts

To further understand how different types of merge conflicts can impact the source code, we examine the type of changes leading to a merge conflict (see Section 3.2 for discussion of categories and methodology). Our results are presented in Table 6, where we present both the results of the automated classifier, as well as the results from our manual classification of the 606 merge conflicts. The distributions of the automatically classified merge conflict types match the distributions of our manual labeling (training data), showing the efficacy of the automated classifier.

The resolution of the first two types of merge conflicts (SEMANTIC and DISJOINT) requires understanding the program logic of the changes in order to successfully resolve the merge conflict. We find that the most common type of merge conflict is SEMANTIC (59.46% of all merge conflicts) where changes are entangled. 14.53% of merge conflicts emerge from concurrent changes to program logic that do not interact with each other and can co-exist (DISJOINT). This means that in the vast majority (at least 75.23% of cases, for the SEMANTIC, DISJOINT and DELETE categories) of merge conflicts, a developer needs to reflect on the program logic when integrating changes, and that for the majority of merge conflicts (at least 59.46% of cases, which represents the SEMANTIC category), new code has to be written.

Merge conflicts due to lines of code being deleted are easier to merge, but still require the developer to reason about the deletion. DELETE conflicts constituted a small percentage (1.24%) of all merge conflicts.

Although resolving merge conflicts due to FORMATTING changes (23.21% of all merge conflicts) or COMMENTS (0.60% of all merge conflicts) require human intervention, they are easier and less risky to resolve since they do not affect the programs' functionality. It is interesting to note that merge conflicts caused by comments are the rarest of all—alluding

Table 6 Merge conflict types and their frequency of occurrence

Category	# of conflicts	% of total (classifier)	% of total (training)	Δ
SEMANTIC	4,150	59.46%	50.86%	+8.6%
DISJOINT	1,014	14.53%	22.74%	−8.21%
DELETE	86	1.24%	2.52%	−1.28%
FORMATTING	1,620	23.21%	11.84%	+11.35%
COMMENTS	42	0.60%	2.21%	−1.61%
OTHER	67	0.96%	3.31%	−2.40%

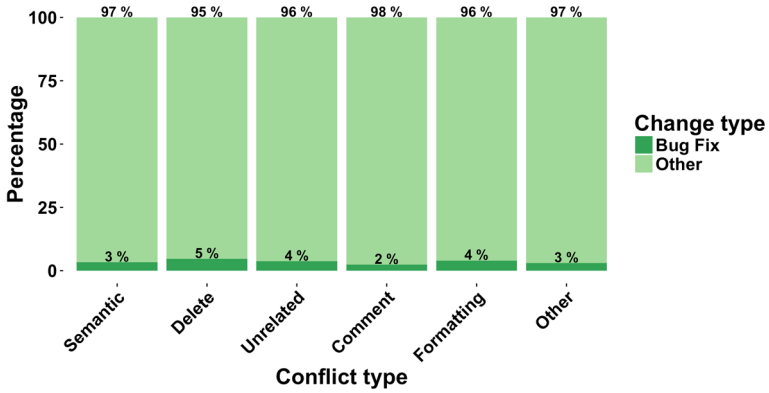


Fig. 6 Frequency of change type by merge conflict type

to the fact that inline comments are rarely changed (if added at all), and most likely only updated when the code is changed.

4.1.2 Change Types

We first examine the kinds of changes that are associated with merge conflicts to understand the types of tasks that are most associated with merge conflicts. We were curious to see if merge conflicts arose from behaviors like multiple developers fixing the same bug. We found this not to be the case. The majority of the changes involved in merge conflicts were non-bug fix changes (see Fig. 6).

4.1.3 Merge Conflict Characteristics

Next we characterize merge conflicts based on the type and size of changes, the number and type of developers making the changes (Table 7). For the former, we determine whether the merger conflicts are non-trivial, that is, if they were associated with changes to the AST. The AST difference captures the impact of changes and is a good estimator of the amount of information a developer must process when resolving merge conflicts. The average size of a Java file is 4,845 AST Nodes, and the maximum is 168,790 AST Nodes. We find that DELETE merge conflicts are associated with the highest AST difference (median of 1,662 nodes, which is 34% of the average file). We believe that a

Table 7 Characteristics of merges conflicts

Conflict Category	Median AST diff (% of median)	Median LOC diff	Median # authors	Overall core developers %
SEMANTIC	387 (8%)	2,351.0	4	89.08%
DISJOINT	88 (2%)	1,037.0	3	89.65%
DELETE	1,662 (34%)	16,102.5	2	86.01%
FORMATTING	0 (0%)	62.0	2	91.67%
COMMENTS	19 (<1%)	427.5	3	80.00%
OTHER	6 (<1%)	238.0	4	86.49%

majority of these changes are associated with refactoring, a result of "chunks of code" that are moved, sometimes across files. Since these (cross-file) moves get counted twice (once for the removal, and once for the addition), it inflates the AST node numbers and LOC differences.

SEMANTIC merge conflicts also led to large AST differences (median 387 nodes). This is intuitive because larger changes have a higher chance of having semantic interactions. In our data set, independent (DISJOINT) changes (affecting a median of 88 AST nodes) are not as complex as those in the prior two categories. As expected FORMATTING merge conflicts have a median AST difference of 0. COMMENTS merge conflicts also have AST node differences (median of 19). This is because Eclipse's JDT has AST nodes for comments. The LOC differences mirror the AST differences.

The median number of authors involved in conflicting commits ranged from 2 to 4. Note that SEMANTIC merge conflicts included changes from a median of four developers, which means that the developer resolving the merge conflict has to reason about changes made by multiple developers—not an easy task. Even DISJOINT merge conflicts included changes from a median of three developers.

Next, we look at the experience level of developers resolving merge conflicts. We classify developers as core or non-core based on their level of contributions as described in Section 3.8. We find that core members are involved in the majority of merge conflict resolutions across all merge conflict types. This is intuitive, as core developers have more knowledge of the system and are therefore the best suited for solving merge conflicts. However, a number of merge conflicts are solved by non-core developers. This can still present problems, as they have less experience and insight into the code, which may make it difficult for them to correctly resolve merge conflicts.

In summary, we identified 6 types of merge conflicts, and the vast majority (75.20%) of merge conflicts impact the program logic, and therefore, require reasoning about the goals of the changes and the best way to integrate them.

4.2 Merge Conflicts and Code Quality (RQ2)

To answer RQ2 we examine whether the changes involved in merge conflicts are more likely to contain bugs than non-conflicting merges. We perform this analysis at the statement level; for every line of code involved in a merge we examine whether it was involved in a future bug fix based on the approach described in Sections 3.6 and 3.8. We use Fisher's Exact Test to compare the number of bug-fix commit between conflicting and non-conflicting merges. The results show that commits that are involved in a *merge conflict are 2.38 times more likely to contain a (future) bug fix* (Fisher's Exact Test, odds ratio $= 2.38$, $p < 0.05$).

Next we determine whether there are differences between different types of merge conflicts, as not all merge conflicts take the same effort to resolve. For our analysis we cluster merge conflicts into two groups:

- NON-TRIVIAL merge conflicts include the SEMANTIC, DISJOINT and DELETE categories. Developers have to determine how to resolve the logical changes in these merge conflicts.
- TRIVIAL merge conflicts include the other categories: FORMATTING, COMMENTS and OTHER. Since these merge conflicts do not involve semantic changes, a trivial merge resolution, such as choosing one version over the other, is feasible.

As before, we use Fisher's Exact Test to examine the difference between TRIVIAL and NON-TRIVIAL merge conflicts and future bug fixes. We find that NON-TRIVIAL *merge*

conflicts are 26.81 times more likely to need a bug fixing commit compared to lines involved in TRIVIAL *merge conflicts* (Fisher's Exact Test, odds ratio $= 26.81$, $p < 0.05$.) This confirms that merge conflicts in the NON-TRIVIAL category are more challenging for developers to resolve, and might either introduce bugs or are associated with changes that themselves are likely to cause bugs.

We also ran Fisher's Exact Test and found a statistical difference between SEMANTIC and DISJOINT merge conflicts (odds ratio $= 0.47$, $p < 0.05$.) This means that merge conflicts arising from disjoint (independent) changes are half as likely to be buggy compared to SEMANTIC merge conflicts. There was no statistical difference between DELETE and SEMANTIC or DELETE and DISJOINT merge conflicts. There was no significant difference between the merge conflicts types in the TRIVIAL group. As we perform repeated tests, we use the Bonferroni adjustment for the α values.

In summary, we find that code that was involved in a merge conflict has a higher likelihood of being involved with a future bug. While some bugs are injected in the merge resolution, others were likely already there (e.g., changes in DISJOINT merge conflicts). In either case, a closer scrutiny of code involved in a merge conflict is warranted.

4.3 Factors Correlated with Bugs (RQ3)

As reported in the defect prediction literature, there are several factors that correlate with the bugginess of code. A critical factor is the size of the module under investigation (El Emam et al. 2001). Therefore, we posit that the size of a change in a merge should be a predictor of bug-proneness. Another factor that has been associated with defects is the number of committers—the "too many cooks" (Weyuker et al. 2008) phenomena. Finally, it has been noted that changes made to central files have a higher likelihood of reducing software quality (Cataldo and Herbsleb 2013). Therefore, we model the total number of bug fixes to conflicting commits by the size of the change, file dependencies, number of authors, and their experience level.

We build a Poisson regression model with a log linking function. After filtering the factors with $VIF \leq 5$, we had a set of eight factors (Table 8) out of 43 total; all eight factors were significant at $p < 0.05$. These factors are: number of references to other files, number of references to the file involved in the merge, number of non-core contributor authors involved in the merge, number of authors involved in the merge, number of AST nodes changed, number of classes involved in the merge, number of methods involved in the merge and the number of LOCs changed.

Table 8 Poisson regression model predicting bug-fix occurrence

Factor	Coefficients
# of References to other files	0.08408
# of References to the file involved in merge	−0.03501
# of Non-core contributor	−1.898
# of authors w/ changes involved in the merge	−0.5634
# of classes involved in the conflict[a]	−0.1636
# of methods involved in the conflict[b]	0.3756
# of AST nodes changed	0.0007278
# of LOC changes	0.00003705

[a]The number of classes containing conflicting changes

[b]The number of methods containing conflicting changes

The McFadden Pseudo R-squared (Hensher and Stopher 1979) of our model is 0.36. We calculated McFadden's Pseudo R-squared as a quality indicator of the model because there is no direct equivalent of R-squared for Poisson regression. The ordinary least square (OLS) regression approach to goodness-of-fit does not apply for Poisson regression. Moreover, pseudo R-squared values like McFadden's cannot be interpreted as one would interpret OLS R-squared values. McFadden's Pseudo R-squared values tend to be considerably lower than those of the R-squared. Values of 0.2 to 0.4 represent an excellent fit (Hensher and Stopher 1979).

The effect of outward dependencies (number of external references from the file involved in a merge conflict) is positive, therefore changes that refer to external file (class) elements are more likely to contain bugs. We found a negative effect of inward dependencies (number of references to the file involved in merge). We also found a negative coefficient for the size of the change (number of classes involved in the merge conflict), the number of authors involved in the change and the number of non-core contributor. The correlation between the number of authors with changes involved in a merge conflict and the number of future bugfixes is also negative.

Finally, in line with previous research (Kim et al. 2008; Mockus and Weiss 2000; Weyuker et al. 2008) we find a positive effect of number of methods involved in the merge conflict, number of LOC changes and number of AST nodes changed on future bug fixing commit counts.

4.4 Resolution Strategies (RQ4)

In this section, we analyze the strategies developers use when resolving merge conflicts. Overall, we find that ADAPTED was the most commonly used resolution strategy (60.82%), compared to INTERLEAVE (26.38%) and SELECT ONE (12.80%). Figure 7 presents the results detailed by merge conflict type. Inr the case of SEMANTIC merge conflicts, we observe that about 80% of the resolutions modify the existing code (ADAPTED) in order to successfully resolve the merge conflict. As expected, merge conflicts that have a semantic nature require a bit of "coercion" for them to work together correctly.

Another category where a high percentage of resolutions use the ADAPTED strategy is COMMENTS. We hypothesize that this is because merge conflicts in comments are actually

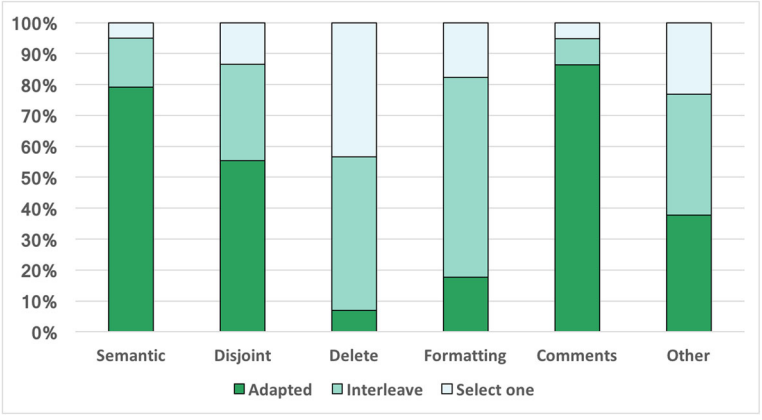


Fig. 7 Resolution strategy based on commit type

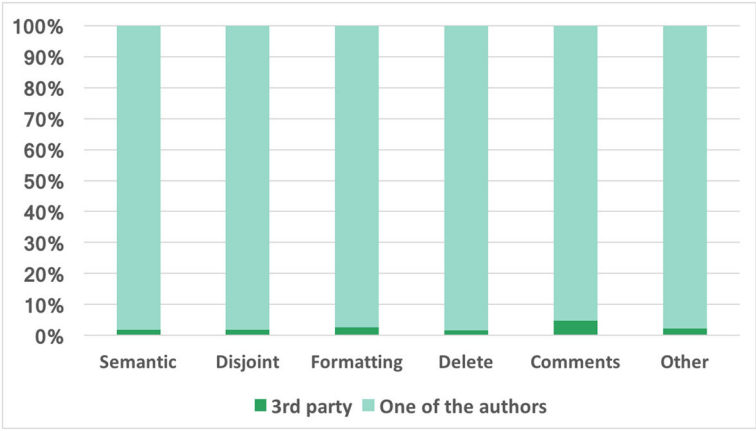


Fig. 8 Developer category for each merge conflict category

conflicting edits in two English texts, which require changing the text to make the sentence structure coherent when integrating the changes.

We see that in FORMATTING merge conflicts, the vast majority (over 80%) involve a resolution by either SELECT ONE or by INTERLEAVE strategies. The small amount of ADAPTED strategy is probably because developers were reformatting the code to integrate the changes.

An interesting result is that more than half of the merge conflicts that are generated by DISJOINT changes still require the developers to use ADAPTED in order to resolve it. Our hypothesis that while the changes themselves might be disjoint, simply interleaving them might not be sufficient.

We now investigate who resolves merge conflicts. We found that the vast majority of merge conflicts are resolved by one of the authors that contributed to one of the branches, as shown in Fig. 8. Core developers solve most of them (89.47%), while non-core developers solve only a small fraction (10.53%). We also saw that about 2% of the merge conflicts were solved by THIRD PARTY developers (who have not contributed to any of the branches). We also analyze whether these third party contributors were core or non-core developers. We present the data in Table 9. For the SEMANTIC category, over 80% of the third party

Table 9 Distribution of developer types resolving merge conflicts

Category	Third party		One of the authors	
	Core	Non-core	Core	Non-core
SEMANTIC	80.97%	19.03%	90.19%	9.80%
DISJOINT	75.00%	25.00%	91.38%	8.62%
DELETE	91.67%	8.33%	91.57	8.33 %
FORMATTING	81.25 %	18.75%	86.61%	13.39%
COMMENTS	85.71%	14.29%	76.92%	23.08%
OTHER	76.06 %	23.94 %	88.47%	11.53%

developers resolving a merge conflict were core contributors. We postulate that these developers are "integrators" for the project. Only a very small fraction of the merge conflicts are resolved by a third party developer who is a non-core developer. The same trend can be seen for the other merge categories as well.

In summary, we find that ADAPTED is the most used strategy for resolving merge conflicts. When dealing with NON-TRIVIAL merge conflicts the percentage increases significantly, possibly indicating the higher difficulty these merge conflicts pose.

5 Discussion

Merge conflicts are far from a solved problem. Despite advances in VCS tools and prescribed development practices, such as frequent commits and review workflows, merge conflicts still occur frequently. We found that 1 out of every 5 merge commits in our dataset resulted in a merge conflict that required human intervention. Additionally, lines of code involved in a merge conflicts were 2× more likely to be buggy. Our taxonomy of merge conflicts shows that in 60% of cases, merge conflicts arise because of interacting, semantic changes. Furthermore, lines of code involved in these types of merge conflicts are 26× more likely to be buggy than in other types of merge conflicts.

Our results have implications for developers, tool builders, and researchers. Developers should focus their testing and code reviews on code resulting from merge conflicts, as they are more likely to be buggy. This is even more critical when the merge involves more changes, or when the conflicting changes span multiple methods.

Tool builders can use our merge conflict taxonomy and conflict (commit) classifier to automatically identify and flag lines of code resulting from merge conflicts, so that these receive higher test coverage or increased code review. Similarly, tools can use the information of the files, methods, lines of code involved in a merge conflict to prioritize testing, especially in the context of swarm testing (Groce et al. 2012; Holzmann et al. 2011).

Our results indicate that 24% of merge conflicts (FORMATTING, COMMENTS) are trivial to resolve. However, they still require human intervention, which interrupts developers' workflow. Tool builders should aim to better support automated code integration for such cases, so that it doesn't require human intervention. For example, Git, has the option of performing a merge that ignores whitespace changes. However, this is not a default option. Research has shown (Johnson et al. 2002; Johnson and Goldstein 2003; Sunstein and Thaler 2003; Samuelson and Zeckhauser 1988) that users, and developers (Parsons and Saunders 2004), are affected by default and anchoring biases. This indicates a preference of using the default options, instead of changing them. Perhaps the description of the option to ignore whitespace while merging should be more prominent, or enabled by default. As we see that a fairly large number of merge conflicts (over 20%) are caused by formatting changes, enabling the option by default could significantly reduce the number of merge conflicts developers face.

Ours is the first empirical study to investigate the effects of merge conflicts on the bugginess of code. Some of our results are counterintuitive and present opportunity for new lines of research. For example, we observe that an increased number of non-core developers is correlated to fewer bug-fixes in the future. We posit that this happens because when non-core developers introduce changes, core developers review them when merging, "eliminating" some of the bugs in the process. We have observed this behavior while analyzing the merge conflict resolution strategies in Section 4.4 where we see that most merges are solved by core developers.

As shown in Section 4.3, we found a negative effect of inward dependencies (number of references to the file involved in merge.) We posit that changes to central files might have to pass more tests or that bugs are identified earlier because more people depend on them.

In the case of the number of authors with changes involved in the merge conflict, we speculate that we are seeing the effect of Linus' law, which states that "*many eyes make all bugs shallow*" (Raymond 1999). We plan to perform further research to investigate how many of the bug fixes that were found after the merge conflict existed before the merge (were dormant), and how many were introduced as part of the merge conflict resolution. The negative correlation between number of classes and bugginess can be attributed to the fact that changes that span multiple classes are more likely to be refactorings, compared to other changes.

When looking at the resolution strategies, we find that developers primarily use ADAPTED as the resolution strategy when dealing with SEMANTIC merge conflicts. Lines involved in SEMANTIC merge conflicts are also most likely to be involved in future bug fixes.

We hypothesize that changes introduced by ADAPTED are not subject to the same level of review as regular commits. Therefore, there is a chance that the ADAPTED merge resolution strategy itself could introduce new bugs. Also, they might contain preexisting bugs. Further studies are needed to answer these questions.

We also saw that non-core developers who were not involved in the development of either of the branches (3rd party) resolve merge conflicts. This is counterintuitive, as non-core developers are less likely to have an in depth understanding of the code base and they are more likely to introduce bugs. On top of that, not being involved in the development of either branch makes their ADAPTED resolution strategy bug prone. It might also be the case that a developer is classified as non-core, but she might have localized experience. Further investigation is needed to explore this.

We posit that because resolving merge conflicts take developers outside of the established workflow, they may not adhere to the strict testing or reviewing process that they would otherwise follow when making their original changes. Developers may pay more attention to changes that include multiple authors, because of which lines of code resulting from such merge resolution are negatively associated with bugginess. Further investigation of the review and testing processes that developers follow for merge resolution is needed to validate this idea.

Our study has included a broad spectrum of projects from different domains. However, they are all open source. Commercial projects have different constraints, development priorities, and practices. Future research comparing the differences between these two types of development with respect to the types of merge conflicts, merge resolution processes, and bugginess of resulting code needs to be performed.

6 Threats to Validity

Our empirical study, like any, has threats to validity. Where possible, we have taken steps to ameliorate their impact.

Identifying merges: Not all projects use `git merge` to integrate their change. Some use `rebase` or `squash` for that purpose. It is currently impossible to tell, from the public history alone, when rebasing or squashing occurs, as all we see is a clean, linear history. This presents the threat that we might under-sample the total number of merge conflicts that

occur in practice. However, we believe this thread has minimal impacts on our results. This just reduces the number of merge conflicts available for analysis, which is compensated by the increased corpus size.

Sampling Bias: All our projects are sampled from a single source—GitHub (GitHub 2017), so our findings may be limited to open source programs. To ensure representativeness of our samples, we used search results from the Github repository of Java projects that use the Maven build system. So, our sample of programs could be biased by skew in the projects returned by Github. Github's selection mechanisms favoring projects based on some unknown criteria may be another source of error. While this makes our results less generalizable, the threat is minimal since we analyze a large number of projects: 500 projects were extracted, from which 143 were selected spanning eight different domains.

Fitness of our approach: We use Gumtree algorithm (Falleri et al. 2014) to track program elements across commits when calculating AST differences. Gumtree, however, does not track program elements across renames or moves to other folders. Despite this drawback, Gumtree is a robust algorithm used to track refactoring and moves within the same file, and is better than line differencing tools.

Machine learning classifiers: We use machine learning to group merger conflicts into the six categories, and to determine whether a commit was a bug-fix. As with any classifier, we may have some mislabeling. This threat is low as our classifiers have good F1-measure and high precision. The confusion matrix (Fig. 3) shows that the highest misclassification is likely to happen when SEMANTIC merge conflicts are identified as DISJOINT merge conflicts. This does not affect our findings for RQ2 (effect of merge conflict on code quality) since both (SEMANTIC and DISJOINT) merge conflict types are in the same NON-TRIVIAL category. Also, our misclassification is asymmetrical because we are classifying more TRIVIAL (COMMENTS) merge conflicts as NON-TRIVIAL (SEMANTIC, DISJOINT) than vice versa. So this makes our results conservative, because any misclassified NON-TRIVIAL commits reduces the future bug-fix count used to answer RQ2. Regarding the bug-fix classifier, our recall and precision measures are on par with past work (Bird et al. 2009). Since our analysis relies on relative count of bug fixes, as long as we do not systematically undercount bug fixes, our results are valid.

Bug-fix commit categorization: It's not always possible to untangle bug fixing changes from other kinds (refactorings, formatting, even adding new features). The same is true of *other* commit types, where bug fixes might have "crept in." As mention before, as long as we don's systematically over- or under-count the number of bug-fixing commits, our results should still be valid.

Finally, we have assumed that all bugs were found and fixed by developers when we use it as a metric of bugginess of merged lines of code. This may not always be true, and hence our results are conservative.

7 Conclusions

Our empirical study spanning 143 open source projects found that merge conflicts occur frequently. Moreover, code involved in a conflict had a 2× higher chance of being buggy. To create a better understanding of the merge conflicts, their frequency, and impact, we create a taxonomy of conflicts, which includes six categories of conflicts depending on the type of resolution effort needed. We found about 74% of merge conflicts include interacting semantic changes (to the underlying AST). Moreover, code resulting from these conflicts

was 26× times more likely to be buggy as compared to that from other conflicts. They are also more likely to be solved using an ADAPTED strategy, compared to other types of conflicts.

Our analysis of the factors associated with merge conflicts show that conflicts that involved files with high (outward) dependencies were correlated with more bugs. However, inward dependencies, number of non-core contributors, and the number of authors were associated with fewer bugs.

We conjecture that the merge resolution process disrupts the normal review and testing workflow, which might be the reason why lines involved in a merge conflict have a higher likelihood of being buggy. We plan to perform more research to investigate the review and testing workflow around a merge resolution. We also plan to study the frequency and distribution of bugs that are actually introduced as part of the resolution process as opposed to bugs that were dormant and remaining in the original code.

Acknowledgements We would like to thank the Software Engineering and HCI research groups at Oregon State University for their input and feedback on this project. This work was in part made possible through support from the National Science Foundation (Grants IIS:1559657, CCF:1253786), and IBM.

References

Accioly P, Borba P, Cavalcanti G (2018) Understanding semi-structured merge conflict characteristics in open-source java projects. Empir Softw Eng 23(4):2051–2085. https://doi.org/10.1007/s10664-017-9586-1

Ahmed I, Gopinath R, Brindescu C, Groce A, Jensen C (2016) Can testedness be effectively measured? In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE 2016. ACM, Seattle, pp 547–558, https://doi.org/10.1145/2950290.2950324

Ahmed I, Brindescu C, Mannan UA, Jensen C, Sarma A (2017) An empirical examination of the relationship between code smells and merge conflicts. In: Proceedings of the 11th ACM/IEEE international symposium on empirical software engineering and measurement, ESEM '17. IEEE Press, Piscataway, pp 58–67. https://doi.org/10.1109/ESEM.2017.12

Apache Maven (2018) Maven. The Apache Software Foundation. https://maven.apache.org/. Accessed 1 May 2018

Apel S, Liebig J, Brandl B, Lengauer C, Kästner C (2011) Semistructured merge: rethinking merge in revision control systems. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering, ESEC/FSE '11. ACM, New York, pp 190–200, https://doi.org/10.1145/2025113.2025141

Apel S, Lessenich O, Lengauer C (2012) Structured Merge with auto-tuning: balancing precision and performance. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering, ASE 2012. ACM, Essen, pp 120–129, https://doi.org/10.1145/2351676.2351694

Biehl JT, Czerwinski M, Smith G, Robertson GG (2007) FASTDash: a visual dashboard for fostering awareness in software teams. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '07. ACM, New York, pp 1313–1322, https://doi.org/10.1145/1240624.1240823

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009) Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, pp 121–130

Boehm BW, Brown JR, Lipow M (1976) Quantitative evaluation of software quality. In: Proceedings of the 2nd international conference on software engineering, ICSE '76. IEEE Computer Society Press, Los Alamitos, pp 592–605

Brun Y, Holmes R, Ernst MD, Notkin D (2011) Proactive detection of collaboration conflicts. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering, ESEC/FSE '11. ACM, Szeged, pp 168–178, https://doi.org/10.1145/2025113.2025139

Brun Y, Holmes R, Ernst MD, Notkin D (2013) Early detection of collaboration conflicts and risks. IEEE Trans Softw Eng 39(10):1358–1375. https://doi.org/10.1109/TSE.2013.28

Buckley J, Mens T, Zenger M, Rashid A, Kniesel G (2005) Towards a taxonomy of software change. J Softw Maint Evol Res Pract 17(5):309–332. https://doi.org/10.1002/smr.319

Canfora G, Cerulo L, Penta MD (2007) Identifying changed source code lines from version repositories. In: Fourth international workshop on mining software repositories (MSR'07:ICSE Workshops 2007), pp 14–14, https://doi.org/10.1109/MSR.2007.14

Cataldo M, Herbsleb JD (2013) Coordination breakdowns and their impact on development productivity and software failures. IEEE Trans Softw Eng 39(3):343–360. https://doi.org/10.1109/TSE.2012.32

Cavalcanti G, Borba P, Accioly P (2017) Evaluating and improving semistructured merge. Proc ACM Program Lang 1(OOPSLA):59:1–59:27. https://doi.org/10.1145/3133883

Cohen J, Cohen P, West S, Aiken L (2013) Applied multiple regression/correlation analysis for the behavioral sciences. Taylor & Francis

Companion website (2016) http://caius.brindescu.com/research/mc-classification

Corbet J, Kroah-Hartman G (2016) 2016 Linux Kernel Development Report. Tech. rep., The Linux Foundation

Costa C, Figueiredo JJC, Ghiotto G, Murta LGP (2014) Characterizing the problem of developers' assignment for merging branches. Int J Softw Eng Knowl Eng 24(10):1489–1508. https://doi.org/10.1142/S0218194014400166

Cruzes DS, Dyba T (2011) Recommended steps for thematic synthesis in software engineering. In: 2011 international symposium on empirical software engineering and measurement, pp 275–284. https://doi.org/10.1109/ESEM.2011.36

da Silva IA, Chen PH, Van der Westhuizen C, Ripley RM, van der Hoek A (2006) Lighthouse: coordination through emerging design. In: Proceedings of the 2006 OOPSLA workshop on Eclipse Technology eXchange, eclipse '06. ACM, New York, pp 11–15, https://doi.org/10.1145/1188835.1188838

de Souza CRB, Redmiles D, Dourish P (2003) "Breaking the code", Moving between private and public work in collaborative software development. In: Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work, GROUP '03. ACM, New York, pp 105–114, https://doi.org/10.1145/958160.958177

De Souza LBL, Maia MDA (2013) Do software categories impact coupling metrics? In: Proceedings of the 10th working conference on mining software repositories, MSR '13. IEEE Press, Piscataway, pp 217–220

Dewan P, Hegde R (2007) Semi-synchronous conflict detection and resolution in asynchronous software development. In: Bannon LJ, Wagner I, Gutwin C, Harper RHR, Schmidt K (eds) ECSCW 2007. Springer, London, Limerick, pp 159–178. https://doi.org/10.1007/978-1-84800-031-5_9

El Emam K, Benlarbi S, Goel N, Rai SN (2001) The confounding effect of class size on the validity of object-oriented metrics. IEEE Trans Softw Eng 27(7):630–650. https://doi.org/10.1109/32.935855

Falleri JR, Morandat F, Blanc X, Martinez M, Montperrus M (2014) Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE international conference on automated software engineering, ASE '14. ACM, New York, pp 313–324, https://doi.org/10.1145/2642937.2642982

GitHub (2017) https://www.github.com/

Godfrey MW, Zou L (2005) Using origin analysis to detect merging and splitting of source code entities. IEEE Trans Softw Eng 31(2):166–181. https://doi.org/10.1109/TSE.2005.28

Gorton I, Liu A (2002) Software component quality assessment in practice: successes and practical impediments. In: Proceedings of the 24th international conference on software engineering, ICSE '02. ACM, New York, pp 555–558, https://doi.org/10.1145/581339.581408

Grinter RE (1995) Using a configuration management tool to coordinate software development. In: Proceedings of conference on organizational computing systems, COCS '95. ACM, New York, pp 168–177, https://doi.org/10.1145/224019.224036

Groce A, Zhang C, Eide E, Chen Y, Regehr J (2012) Swarm Testing. In: Proceedings of the 2012 international symposium on software testing and analysis, ISSTA 2012. ACM, Minneapolis, pp 78–88, https://doi.org/10.1145/2338965.2336763

Guimarães ML, Silva AR (2012) Improving early detection of software merge conflicts. In: Proceedings of the 34th international conference on software engineering, ICSE '12. IEEE Press, Zurich, pp 342–352

Hensher D, Stopher P (1979) Behavioural travel modelling, Croom Helm, London

Holzmann GJ, Joshi R, Groce A (2011) Swarm verification techniques. IEEE Trans Softw Eng 37(6):845–857. https://doi.org/10.1109/TSE.2010.110

Johnson EJ, Goldstein D (2003) Do defaults save lives? Science 302(5649):1338–1339. https://doi.org/10.1126/science.1091721

Johnson EJ, Bellman S, Lohse GL (2002) Defaults, framing and privacy: why opting in-opting out1. Mark Lett 13(1):5–15. https://doi.org/10.1023/A:1015044207315

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining GitHub. In: Proceedings of the 11th working conference on mining software repositories, MSR 2014. ACM, New York, pp 92–101, https://doi.org/10.1145/2597073.2597074

Kasi BK, Sarma A (2013) Cassandra: proactive conflict minimization through optimized task scheduling. In: Proceedings of the 2013 international conference on software engineering, ICSE '13. IEEE Press, Piscataway, pp 732–741

Kim S, Zimmermann T, Pan K, Whitehead EJJ (2006) Automatic identification of bug-introducing changes. In: Proceedings of the 21st IEEE/ACM international conference on automated software engineering, ASE '06. IEEE Computer Society, Washington, DC, pp 81–90, https://doi.org/10.1109/ASE.2006.23

Kim S, Whitehead EJ Jr, Zhang Y (2008) Classifying software changes: clean or buggy? IEEE Trans Softw Eng 34(2):181–196. https://doi.org/10.1109/TSE.2007.70773

Lippe E, van Oosterom N (1992) Operation-based merging. In: Proceedings of the fifth ACM SIGSOFT symposium on software development environments, SDE 5. ACM, New York, pp 78–87, https://doi.org/10.1145/142868.143753

Marinescu R (2001) Detecting design flaws via metrics in object-oriented systems. In: Proceedings of the 39th international conference and exhibition on technology of object-oriented languages and systems (TOOLS39), TOOLS '01. IEEE Computer Society, Washington, DC, p 173

Marinescu R (2004) Detection strategies: metrics-based rules for detecting design flaws. In: 20th IEEE international conference on software maintenance, 2004. Proceedings, pp 350–359

McKee S, Nelson N, Sarma A, Dig D (2017) Software practitioner perspectives on merge conflicts and resolutions. In: 2017 IEEE international conference on software maintenance and evolution, ICSME 2017, Shanghai, China, September 17–22, 2017, pp 467–478. https://doi.org/10.1109/ICSME.2017.53

Menezes GGLD (2016) On the nature of software merge conflicts. Ph.D. thesis, Universidade Federal Fluminense

Mens T (2002) A state-of-the-art survey on software merging. IEEE Trans Softw Eng 28(5):449–462. https://doi.org/10.1109/TSE.2002.1000449

Mockus A, Weiss DM (2000) Predicting risk of software changes. Bell Labs Tech J 5(2):169–180. https://doi.org/10.1002/bltj.2229

Mockus A, Fielding RT, Herbsleb JD (2002) Two case studies of open source software development: Apache and Mozilla. ACM Trans Softw Eng Methodol 11(3):309–346. https://doi.org/10.1145/567793.567795

Nieminen A (2012) Real-time collaborative resolving of merge conflicts. In: Proceedings of the 2012 8th international conference on collaborative computing: networking, applications and worksharing (CollaborateCom 2012), COLLABORATECOM '12. IEEE Computer Society, Washington, DC, pp 540–543. https://doi.org/10.4108/icst.collaboratecom.2012.250435

Parsons J, Saunders C (2004) Cognitive heuristics in software engineering applying and extending anchoring and adjustment to artifact reuse. IEEE Trans Softw Eng 30(12):873–888

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in python. J Mach Learn Res 12:2825–2830

Perry DE, Siy HP, Votta LG (2001) Parallel changes in large-scale software development: an observational case study. ACM Trans Softw Eng Methodol 10(3):308–337. https://doi.org/10.1145/383876.383878

Raymond ES (1999) The Cathedral and the bazaar, 1st edn. O'Reilly & Associates, Inc., Sebastopol

Ripley RM, Yasui RY, Sarma A, van der Hoek A (2004) Workspace awareness in application development. In: Proceedings of the 2004 OOPSLA workshop on Eclipse Technology eXchange, eclipse '04. ACM, New York, pp 17–21, https://doi.org/10.1145/1066129.1066133

Samuelson W, Zeckhauser R (1988) Status quo bias in decision making. J Risk Uncertain 1(1):7–59

Sarma A, Noroozi Z, Hoek AVD (2003) Palantír: raising awareness among configuration management workspaces. In: Proceedings of the 25th international conference on software engineering, May 3-10, 2003, Portland, Oregon, USA, pp 444–454. https://doi.org/10.1109/ICSE.2003.1201222

Servant F, Jones JA, van der Hoek A (2010) CASI: preventing indirect conflicts through a live visualization. In: Proceedings of the 2010 ICSE workshop on cooperative and human aspects of software engineering, CHASE '10. ACM, New York, pp 39–46, https://doi.org/10.1145/1833310.1833317

Shihab E, Bird C, Zimmermann T (2012) The effect of branching strategies on software quality. In: Proceedings of the ACM-IEEE international symposium on empirical software engineering and measurement, ESEM '12. ACM, New York, pp 301–310, https://doi.org/10.1145/2372251.2372305

Sun Z, Song Q, Zhu X (2012) Using coding-based ensemble learning to improve software defect prediction. IEEE Trans Syst Man Cybern Part C Appl Rev 42(6):1806–1817

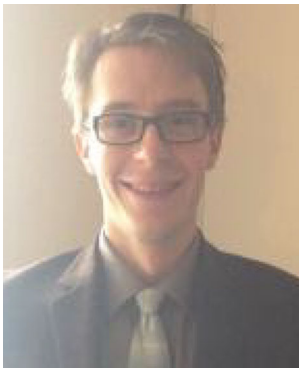
Sunstein CR, Thaler RH (2003) Libertarian paternalism is not an oxymoron. The University of Chicago Law Review, pp 1159–1202

Tian Y, Lawall J, Lo D (2012) Identifying linux bug fixing patches. In: 2012 34th international conference on software engineering (ICSE), pp 386–396

Understand ™ (2017) Static Code Analysis Tool

Weyuker EJ, Ostrand TJ, Bell RM (2008) Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. Empir Softw Eng 13(5):539–559. https://doi.org/10.1007/s10664-008-9082-8

Zimmermann T, Kim S, Zeller A, Whitehead EJ Jr (2006) Mining version archives for co-changed lines. In: Proceedings of the 2006 international workshop on mining software repositories, MSR '06. ACM, New York, pp 72–75, https://doi.org/10.1145/1137983.1138001



Caius Brindescu is a Computer Science PhD Candidate at Oregon State University. He received a BS in Computer Science from the "Politehnica" University of Timişoara, Romania. His research focuses on understanding the problems developers face when working collaboratively, particularly while resolving merge conflicts. Prior work has been published at top peer-review conferences (ICSE, FSE, ESEM) and journals (EMSE). He has also served as a PC member for the MSR'17 Data Mining Challenge, SCAM'15 Tool Demo track, and the VL/HCC '17 Showpiece track.

Iftekhar Ahmed is an Assistant Professor of Informatics in the Donald Bren School of Information and Computer Science at the University of California, Irvine. His research focuses on software engineering in general and combining software testing, analysis and data mining to come up with better tools and techniques in particular. He finished his B.Sc. in Computer Science and Engineering from Shahjalal University of Science and Technology, Bangladesh and after working in the industry for 4 years, did his Ph.D. at Oregon State University. He was advised by Carlos Jensen.

Carlos Jensen is currently an associate dean in the College of Engineering and an associate professor in the School of Electrical Engineering and Computer Science (EECS) at Oregon State University (OSU). He received a B.S. in Computer Science from the State University of New York (SUNY) Brockport, and a Ph.D. in Computer Science from the Georgia Institute of Technology in 2005, where he was a member of the Graphics, Visualization and Usability Center (GVU).

His areas of research are in Usable Privacy and Security (HCISec), with particular focus on making online decisions about privacy and security understandable and meaningful to users. As part of this work he examines policies and business practices currently in use, as well as the technologies available and their implications, and users' understanding and concerns. His main research project in this area; iWatch, tracks website privacy practices, and how your online information spreads online.

An emerging area of research is in the usability aspects of open source systems and open source development. More specifically, he is interested in studying the barriers for entry into open source, and finding ways we can get more people, including students, involved in open source.

Anita Sarma is an Associate Professor at Oregon State University. Before this she was an Assistant Professor at University of Nebraska, Lincoln; a post-doctoral scholar at Carnegie Mellon University, and a doctoral student at University of California, Irvine. Dr. Sarma's research crosscuts Software Engineering and Human Computer Interaction and focuses on how software can be designed to support developers and end user programmers in their development efforts. Her work explores how socio-technical dependencies affect team work, how onboarding barriers in OSS can be alleviated, and how software can be made gender-inclusive. Overall, Dr. Sarma's research has resulted in 46 peer-reviewed publications (36 conferences, 10 journals), four of which have won awards. Her work has been funded through NSF and Airforce (AFOSR). She has been the recipient of the NSF CAREER award. She regularly serves on program committees in software engineering conferences (ICSE, ASE, VL/HCC, ICGSE), has been PC co-chair for ICGSE 2016, VL/HCC 2016, ICSE NIER 2019, ICSE SEIS 2020, ICSE Demo 2014, and is an Associate Editor for TSE.