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

Indicators for Merge Conflicts in the Wild: Survey and Empirical Study

Olaf Leßenich · Janet Siegmund · Sven Apel · Christian Kästner · Claus Hunsen

Received: date / Accepted: date

Abstract While the creation of new branches and forks is easy and fast with modern version-control systems, merging is often time-consuming. Especially when dealing with many branches or forks, a prediction of merge costs based on lightweight indicators would be desirable to help developers recognize problematic merging scenarios before potential conflicts become too severe in the evolution of a complex software project. We analyze the predictive power of several indicators, such as the number, size or scattering degree of commits in each branch, derived either from the version-control system or directly from the source code. Based on a survey of 41 developers, we inferred 7 potential indicators to predict the number of merge conflicts. We tested corresponding hypotheses by studying 163 open-source projects, including 21,488 merge scenarios and comprising 49,449,773 lines of code. A notable (negative) result is that none of the 7 indicators suggested by the participants of the developer survey has a predictive power concerning the frequency of merge conflicts. We discuss this and other findings as well as perspectives thereof.

Olaf Leßenich
University of Passau, Germany E-mail: lessenic@fim.uni-passau.de

Janet Siegmund
University of Passau, Germany E-mail: siegmunj@fim.uni-passau.de

Sven Apel
University of Passau, Germany E-mail: apel@fim.uni-passau.de

Christian Kästner
Carnegie Mellon University, USA E-mail: kaestner@cs.cmu.edu

Claus Hunsen
University of Passau, Germany E-mail: hunsen@fim.uni-passau.de

1 Introduction

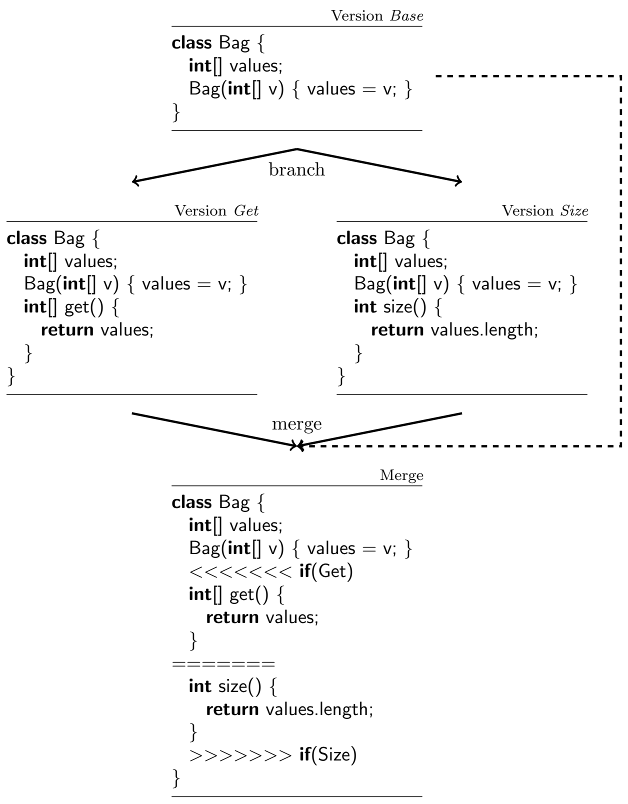
Today, the evolution of a software project is typically managed by means of a version-control system. Distributed version-control systems, such as Git, owe their increasing popularity to the fact that they allow developers to work independently by making branching easy [Muşlu et al, 2014]. Although parallel development increases a team's productivity, merging branches can be a difficult and time-consuming task [Mens, 2002; Bird and Zimmermann, 2012]. Figure 1 shows a typical merge conflict, in which a simple Java class is changed independently in the two versions *Get* and *Size*. As the changes have been introduced in the same location, common text-based merge tools cannot merge the versions automatically—a conflict is reported instead.

Previous studies of popular open-source systems have shown that merge conflicts are frequent and persistent, even when working with modern version-control systems and advanced merge tools [Brun et al, 2011; Apel et al, 2011; Bird and Zimmermann, 2012; Leßenich et al, 2014]. Merge conflicts are especially challenging in large-scale, long-living software projects, with many branches, forks, and clones [Mens, 2002; Bird and Zimmermann, 2012] and where branches are used for customer-specific features [Staples and Hill, 2004; Rubin et al, 2013; Dubinsky et al, 2013; Antkiewicz et al, 2014].

To mitigate problems of (late) merging, several approaches aim at increasing awareness of changes and thereby encourage early merging, for example, *continuous merging* [Guimarães and Silva, 2012] and *speculative analysis* [Brun et al, 2011]. Speculatively executing merges becomes quickly very expensive, especially when many branches are involved (the numbers of comparisons grow quadratically) or when advanced but computationally expensive structure-based or n-way merging techniques are used [Mens, 2002; Leßenich et al, 2014; Rubin and Chechik, 2013]. Especially, in scenarios with many forks and branches and frequent n-way merges—which is not uncommon in practice [Mens, 2002; Bird and Zimmermann, 2012; Antkiewicz et al, 2014; Dubinsky et al, 2013; Rubin et al, 2013; Stanciulescu et al, 2015]—the cost of continuous and speculative merging would be high. Hence, we strive for an alternative approach. A further motivation is that some approaches of continuous and speculative merging rely on dedicated IDE support, which limits the approach to a specific setting, which we would like to avoid.

We aim at predicting the effort of merging branches *without* actually executing the merges and at visualizing the project status in an aggregated and intuitive manner on a project-wide dashboard.[1] To this end, we exploit the fact that a lot of information can be retrieved at very low cost from either the version-control system or from patches. Such information can be computed for each branch separately, so costs grow linearly (not quadratically) with the number of branches. Predictions could even work as an initial filter to reduce the effort of a speculative-merging infrastructure, such that only candidates with likely conflicts are analyzed further.

[1] `http://www.infosun.fim.uni-passau.de/spl/pythia/`



```
class Bag {
  int[] values;
  Bag(int[] v) { values = v; }
}
```

```
class Bag {
  int[] values;
  Bag(int[] v) { values = v; }
  int[] get() {
    return values;
  }
}
```

```
class Bag {
  int[] values;
  Bag(int[] v) { values = v; }
  int size() {
    return values.length;
  }
}
```

```
class Bag {
  int[] values;
  Bag(int[] v) { values = v; }
  <<<<<<< if(Get)
  int[] get() {
    return values;
  }
=======
  int size() {
    return values.length;
  }
  >>>>>>> if(Size)
}
```

Fig. 1 A merge conflict between two variants of class Bag, which introduce different functionalities at the same position.

To learn more about the factors that lead to conflicts in a scenario as outlined above, we conducted a survey, in which 41 developers shared their typical problems during merging and the reasons for conflicts in their projects. Several developers (38 %) stated that they sometimes avoid synchronization, this way, fostering late merging, because they fear to run into conflicts, which would interrupt their work flow. At the same time, the developers largely agreed that late merging is one of the primary causes that lead to merge conflicts and the negative implications thereof.

Several of our questions aimed at learning about the workflow of the developers with regards to the use of the version-control system, and to estimate whether merge conflicts are a persistent obstacle or not. Among others, we asked: "What is the policy on how to deal with conflicts?" One common practice is, as one participant put it: "Whoever commits second has to fix it." This also means that, following this approach, the maintainer of an upstream branch may end up having to solve merge conflicts in the code of a feature that someone else wrote, just because there were later changes in the upstream branch. Some participants mentioned additional constraints when using this method, for example, a "no branch lives longer than a day" policy, which is employed by the development team to restrict possible damage. Such rigorous measures indicate that there have been excessive problems related to merge conflicts in past projects. These responses suggest that merge conflicts are still a common nuisance in software development, which confirms the observations of previous studies [Mens, 2002].

Beside understanding the role of merge conflicts, the majority of our survey questions aimed at what precisely causes many of the conflicts. In essence, the participants of our survey suggested several indicators (consolidated into 7 indicators, in total) that should have an influence on merge conflicts. For example, they suggest that branches with more commits, larger commits, and more scattered changes are more likely to cause merge conflicts, just as cases where more developers work in parallel. These indicators form the basis for analysis.

The overall goal of our study is to explore *whether* a prediction based on these (and other) potential indicators can be used to predict problematic merge situations, possibly as input for open-source dashboard solutions, such as CODEFACE.[2] To test the feasibility and predictive power of the indicators, we analyze 21,488 real merge scenarios in the version history of 163 substantial open-source JAVA projects. Our goal is to *predict* difficult merges by obtaining and analyzing statistical data of the changes that were introduced in concurrent branches. For evaluation, we executed the actual merges and determined the number of conflicts, as an operationalization of merge difficulty. In addition to the information available in standard version-control systems, we analyzed the structure and granularity of changes at the level of the abstract syntax tree to test our hypotheses.

As a key result, we found that *none* of the 7 indicators that have been suggested by the developer survey have a predictive power concerning (i.e., correlating strongly with) the number of conflicts. This is clearly a *negative* result that surprised us, as it contradicts our intuition as well as the prevailing opinion of the developers who took part in our survey. Nevertheless, despite this overall negative result, our study provides important insights into the feasibility of a lightweight approach of predicting merge conflicts. The results of our study can be understood as a warning to practitioners and researchers when making assumptions about merging and merge conflicts. Furthermore, our study, data, analysis, and experiment infrastructure form a solid basis for replication and follow-up studies as well as for research on alternative conflict-prediction approaches (e.g., domain-specific) and conflict-avoidance strategies (e.g., speculative merging).

Overall, we make the following contributions:

- We present the results of a survey among 41 developers, who we asked about the causes of merge conflicts in their projects.

[2] `http://siemens.github.io/codeface/`

– We make our infrastructure and data publicly available for replication and follow-up studies. We collected and published 21,488 merge scenarios from 163 open-source projects.
– We evaluated the predictive power of the indicators suggested by the developers in a study on the selected subject projects.
– We found that none of the 7 indicators correlates strongly with the number of merge conflicts. We did not expect this negative result, and we analyze possible reasons and discuss implications thereof.

2 Survey and Hypotheses

In this section, we present the results of our survey and our research hypotheses. All material (including all survey questions and answers and the collected data) is available at the project's website.[3]

2.1 Survey on Merge Conflicts

Objective We started the investigation with an intuition that several factors should plausibly influence the chance of merge conflicts (making the factors indicators), including the age of a branch, the number of commits and committers, and so forth. To get a broader picture of whether practitioners share our intuition and what other indicators are plausible, we conducted a developer survey. In this survey, we asked developers for factors related with merge conflicts in their projects. We use this survey to ground our hypotheses about merge-conflict indicators, which we evaluate in Section 3.

Material For the purpose of our investigation, we designed an online survey, including open and closed questions. To get an unbiased opinion of participants, we started with a general open question to ask developers what, in their opinion, leads to merge conflicts ("In your opinion, what are typical factors and situations that most likely lead to merge conflicts?"). Then, we proceeded with specific questions in the line of our intuition (e.g., "Do you sometimes avoid a pull/update because of potential conflicts?", "Where do conflicts typically occur?").

Participants To acquire participants, we published the survey on several platforms, including Twitter and Reddit. We received 41 responses from developers aged 21 to 48 years (31.5 years, on average), having 3 to 20 years of programming experience (9.6 years, on average). The participants are involved in projects with mostly 3 to 5 or 6 to 10 developers per team. 98 % stated that they use GIT as version-control system.

[3] `http://www.infosun.fim.uni-passau.de/spl/papers/conflict-prediction/`

Evaluation The majority of our questions aimed at what precisely causes merge conflicts. Typical answers to the open questions refer to formatting changes, large-scale refactoring, and structural changes in long-living forks. Some of the more specific responses mention import statements—which are maintained automatically by an IDE—and changes related to release versioning (e.g., hard-coded strings in the code base).

We derived the hypotheses by conducting a form of card sorting [Hudson, 2013] (i.e., we put answers into buckets and created a hypothesis for each bucket).

Next, we present our hypotheses and describe how they relate to the participants' responses to the corresponding survey questions.

2.2 Hypotheses

The reason for merge conflicts is parallel development with overlapping changes. Therefore, in the context of a distributed version-control system, we suspect that branches with a large number of commits are more likely to induce conflicts during a merge than branches with few commits. Our survey shows that almost all developers we polled (83 %) share this view, which leads us to our first hypothesis:

H₁ *Active, diverted branches (in terms of number of commits) are more likely to result in conflicts than inactive branches that remain close to each other.*

When a deadline is near, there might be a situation where many people commit quite often in short intervals to "get their changes out". At the same time, the awareness of changes made by other team members decreases because of time pressure. Among the participants of our survey, 29 % named release pressure as a cause of conflicts. One participant mentioned "end-of-day commits" as an example. We summarize this view in the following hypothesis:

H₂ *Many commits within a small time span are more likely to produce conflicts than the same number of commits over longer time spans.*

The majority of developers in our survey (73 %) share our following expectation: They mentioned particularly "developers working on the same files" as source for potential merge conflicts. We expect that the more files are changed by both concurrent branches, the higher the probability that developers were working on the same locations—and thereby cause conflicts. We summarize the rationale in the following hypothesis:

H₃ *The more files are changed by both branches, the more likely a conflict occurs.*

The following assumption is a rather intuitive one as well: The more code is changed, the higher is the likelihood of overlap and thereby conflicts. 51 %

of our survey participants stated that “large”/“huge”/“big” commits likely have an influence on the occurrence of merge conflicts. We capture this view in our next hypothesis:

H_4 *Larger changes that modify more lines of code are more likely to cause conflicts than smaller changes.*

Crucial for successful automated merging of changes is the unambiguity of insertion locations. When there are multiple changed, non-cohesive locations (chunks) within a project, that is, tangled changes, it seems natural that the chances for conflicts are higher, because there is an increased chance of an overlapping change at one of these locations within another branch. As an example, consider a file containing 100 lines. For a diff/merge algorithm, it makes a difference whether the first 50 lines are changed, or every second line. So, while the same number of lines have been changed in both scenarios, the merge result is a different: one large conflict versus 50 small conflicts. Two survey participants shared this concern and proposed to “modularise code to keep it in reasonably small files” and “keep files [. . .] well structured as soon as possible” to prevent/avoid potential conflicts. We capture this situation in the following hypothesis:

H_5 *More code fragmentation (tangled changes) of the committed changes results in more conflicts than lesser code fragmentation.*

We expect that scattered changes have a higher chance of interfering with the changes of another developer than cohesive ones. In our survey, such issues were often mentioned in conjunction with a lack of organization and communication. A survey participant stated that “usually tasks that crosscut different components of code lead to conflicts”, another one observed that “[they] used to have more conflicts before [they] started using a more modular approach”. Overall, 56 % of the participants also stated that conflicts typically occur in more than one file. Such scattered changes can be introduced at different granularities, which we incorporate in our next hypothesis:

H_6 *Scattered changes (across classes or methods) are more likely to lead to conflicts than cohesive changes.*

During an earlier study on syntax-based versus line-based merging in Java projects [Apel et al, 2011; Leßenich et al, 2014], we observed that a common cause of conflicts with line-based merges seemed to be caused by changed, or just reordered, lists of import statements. Such operations are often performed automatically by IDEs, without the explicit awareness of the developer. However, the merge itself is typically executed by a text-based, language-unaware merge tool, leaving the resolution of the conflict to the developer. While this can be annoying when using conventional merge tools, especially when a lot of files are affected, advanced, language-aware techniques, such as a structured, syntax-based merge algorithm, can resolve those changes automatically [Leßenich et al, 2014]. This leads us to our last hypothesis:

$\mathbf{H_7}$ *Changes above the level of class declarations (which are often inserted and maintained automatically) are more likely to lead to merge conflicts than changes inside class declarations (introduced by human developers).*

3 Empirical Study

To test our hypotheses, we conducted an empirical study, in which we observe real merge conflicts in open-source software projects. We describe the study setup in this section and the results in Section 5, following standard guidelines of empirical research [Jedlitschka and Pfahl, 2005]. On the project's website, we provide a replication package, including survey responses, the corpus of merge scenarios, and additional information.

3.1 Objective and Variables

The objective of this study is to evaluate the hypotheses that we formulated in Section 2. We operationalize the hypotheses through several metrics (indicators) summarized in Table 1. For each indicator, we determine the respective value for each branch (by comparing the head of the branch with the common ancestor) and then compute the geometric mean ($\sqrt{a \cdot b}$) of those values. We chose geometric mean as our aggregation function, because we want to combine individual values to a meaningful and interpretable indicator. Geometric mean is the most natural choice, because the product of the separate values is interpretable in our case. This is particularly evident by looking at corner cases: If one branch has, for instance zero changes, we would not expect a conflict and therefore want the aggregated value to be zero, as well—as is the geometric mean in this case.

In a nutshell, we correlate the individual metrics (at least, one for each hypothesis), aggregated based on both branches of a merge scenario, with the number of conflicts occurring in that merge scenario. If we observe a strong correlation, the corresponding metric qualifies as a potential indicator. We create a stepwise-regression model to see whether a weighted combination of those metrics is useful for prediction.

Independent Variables As a first step, we collect several metrics on commits in both branches. For hypothesis $\mathbf{H_1}$, we simply count the number of commits in both branches between the base revision and the merge point in each merge scenario; then, we compute the geometric mean of the commits in both branches. For hypothesis $\mathbf{H_2}$, we take the number of commits in the last week before the merge into account, derived from the timestamps in the respective commits—again reported as geometric mean. The same is done analogously for the last two weeks, so we compute two metrics for $\mathbf{H_2}$. For hypothesis

Table 1 Indicators for the number of merge conflicts; for each indicator, we compare the head of each branch with the common ancestor.

Metric	Description	Hypoth.
	Dependent variable	
Number of conflicts	Number of conflicts reported by a line-based merge tool actually performing the merge; conflicts in consecutive lines are reported as a single conflict.	
	Independent variables: Commit metrics	
Number of commits	Number of commits between the common ancestor and the merge point of each branch; reported as geometric mean of both branches.	$\mathbf{H_1}$
Commit density	Number of commits in last week/last two weeks; reported as geometric mean of both branches.	$\mathbf{H_2}$
Number of files changed by both branches	Number of files modified by, at least, one developer in both branches.	$\mathbf{H_3}$
	Independent variables: Change-size metrics	
Number of changed lines of code	Size of the difference between the common ancestor and the end version of each branch, in terms of added and removed lines; reported as geometric mean of both branches.	$\mathbf{H_4}$
Number of AST nodes changed	Size of the difference between the common ancestor and the end version of each branch, in terms of added and removed AST nodes; reported as geometric mean of both branches.	$\mathbf{H_4}$
Number of code chunks changed	Number of locations in the code that have been changed in any commit between the ancestor and the merge point in each branch; measured in terms of consecutive code blocks (AST subtrees or siblings); reported as geometric mean of both branches.	$\mathbf{H_5}$
Number of changes inside class declarations	Number of changed AST nodes that belong to class declarations; reported as geometric mean of both branches.	$\mathbf{H_7}$
Number of changes above class declarations	Number of changed AST nodes above class declarations, including package and import declarations; reported as geometric mean of both branches.	$\mathbf{H_7}$
	Independent variables: Scattering metrics	
Scattering degree (classes) = $\frac{\text{number of changed classes}}{\text{number of all classes}}$	Percentage of classes affected by any changes between the ancestor and the merge point in either branch; reported as geometric mean of both branches.	$\mathbf{H_6}$
Scattering degree (meth.) = $\frac{\text{number of changed methods}}{\text{number of all methods}}$	Percentage of methods affected by any changes between the ancestor and the merge point in either branch; reported as geometric mean of both branches.	$\mathbf{H_6}$

$\mathbf{H_3}$, we count the number of files changed by both branches. Hypothesis $\mathbf{H_4}$ addresses the size of changes. In line with previous work [Hattori and Lanza, 2010; Leßenich et al, 2014], we analyze changes in the abstract syntax trees

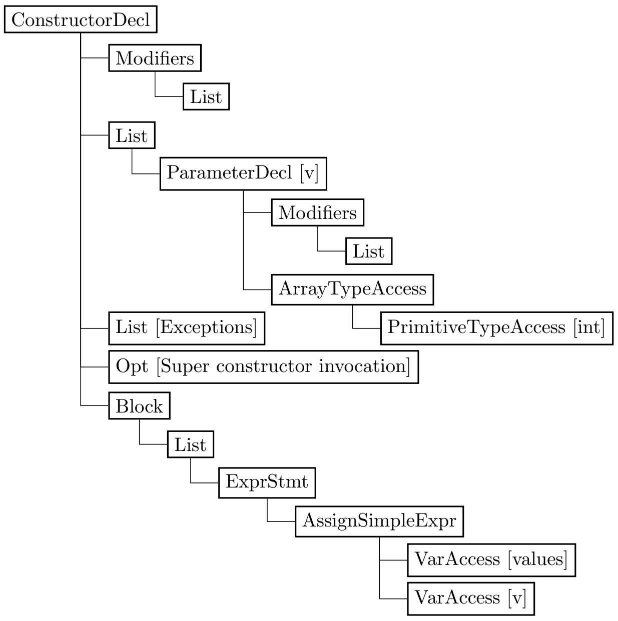


Fig. 2 Abstract syntax tree of constructor of class Bag (base revision) of Figure 1

(ASTs) of the files, not on their textual, line-based representation. This enables a more accurate characterization of the changes independent of layouting issues. Change sizes are counted in terms of added or deleted lines as well as added or deleted nodes in the corresponding ASTs (e.g., adding an entire subtree involves adding multiple nodes). We call a sequence of adjacent nodes with their subtrees a *code chunk*. For illustration, Figure 2 shows an AST for the constructor of the base revision of class Bag of Figure 1. This class has 4 lines of code as formatted in Figure 1, whereas its AST contains 35 nodes. Version *Size* adds 14 nodes (method declaration, modifiers, type access, ...), and version *Get* adds 13 nodes. On average, there are 7 nodes per line of code.

Hypotheses $\mathbf{H_5}$ and $\mathbf{H_6}$ are concerned with the fragmentation and distribution of changes. The scattering degree describes which percentage of classes and methods have been affected by changes in either branch. Changes that are more scattered affect more classes or methods.

For hypothesis $\mathbf{H_7}$, we analyze the location of changes structurally (whether above class declaration, or not), which influences the difficulty of an automatic merge.

Dependent Variable We operationalize the merge effort in terms of number of conflicts per merge scenario. The intuition is that, the more conflicts are reported, the more situations a developer has to manually inspect while resolving the conflicts.

Note that, we also experimented with a number of alternative operationalizations, for example, measuring merge effort in terms of conflicting lines of code, measuring change sizes in lines of code, or measuring the distribution in

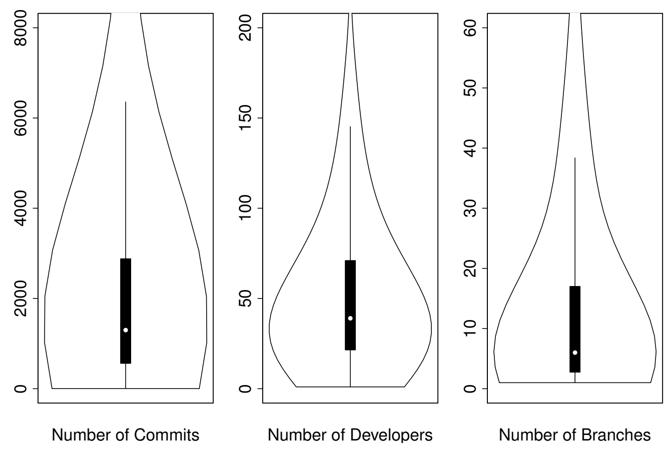


Fig. 3 Descriptive violin plots for sample projects (about 10% of the data points are outliers beyond the scale of the plots, not shown for a better visualization)

terms of files, classes, or methods changed in both branches. As we discuss in Section 6, these alternative metrics yield different numbers, but do not change the overall picture.

Confounding Variables Finally, confounding variables include programming languages, version-control system, and domains of the projects under study [Siegmund and Schumann, 2014]. We controlled their influence by keeping the programming language and version-control system constant (Java and Git) and by selecting a broad range of subject projects from many domains, including IDEs, databases, application and testing frameworks, interpreters, games, and many more.

3.2 Subject Projects

Overall, we selected 163 subject projects from a variety of domains from the hosting platform GitHub. We decided to limit our analysis to git repositories because it makes it easy to identify merge situations in retrospect. We provide a full list (and our raw data) on our supplementary website.

We selected the corpus as follows: We retrieved the 500 most popular Java projects on GitHub, as determined by the number of watchers, queried from GitHubArchive.[4] Among these 500 projects, we filtered out projects with less than 50 merge commits in their version history (see Section 3.3) that have actual changes in Java files, which reduced the number of projects to 163. We analyzed a total of 501,311 commits, including 21,488 merge scenarios.

The selected subject projects range from 700 lines of code to 3.6 million lines of code. The average time span that we analyzed was 3.9 years of development. The median number of forks per project on GitHub is 315, each

[4] `http://www.githubarchive.org/`

project has a median of 6 branches (257 max) and 39 contributors (497 max). Per month, changes introduced by developers are causing a solid median code churn of 0.1. In essence, we were analyzing substantial and active projects that are developed in parallel by several active developers. The distribution of number of commits, developers, and branches is shown in Figure 3. An overview with all of the individual descriptive metrics for each project can be found on our supplementary website.

3.3 Procedure

In Git, merge commits can be easily identified: Their number of parent commits is greater than one.

After cloning the subject project's repository, we first identified merge scenarios by filtering commits with multiple parent commits. Each *merge scenario* consists of three commits (two competing revisions and their common ancestor) and can be identified by such merge commits. We excluded octopus merges,[5] because they are a very special case and we found only very few of them anyway. For the subject projects, we identified 21,488 merge scenarios, in total, each representing a real merge that has been completed by the project's developers. We excluded fast-forward merges, as they cannot lead to conflicts.

To identify *merge conflicts* and to gain information about the induced changes, we automatically re-ran and analyzed each merge scenario. Many of these merge scenarios can be merged automatically, whereas others result in conflicts. In our case, we found that in 2361 of the 21,488 merge scenarios Git reports a merge conflict. Considering only actual Java code (to increase internal validity) of the repositories, Git reports a merge conflict in 1379 of the 21,488 merge scenarios.

For the hypotheses $\mathbf{H_4}$–$\mathbf{H_7}$, we need more information about the nature of differences and conflicts. For this purpose, we merged all 21,488 merge scenarios again using our syntax-based structured-merge tool JDime.[6] JDime parses the code of all three revisions relevant for a merge scenario and characterizes changes and differences structurally in terms of the context-free syntax [Leßenich et al, 2014]. This is important for our work, because it allows us to easily and precisely determine which kind of structures have been changed.

Finally, we compute Spearman's correlation coefficient between the number of merge conflicts and the potential indicators that we derived from our hypotheses (cf. Table 1). We use the Spearman correlation as it is based on rank data and does not assume linear relationships.

[5] n-way merges in Git

[6] `http://fosd.de/JDime/`

3.4 Execution and Deviation

Our analysis framework is open-source and written in JAVA. All of the extracted and computed data are stored in a MYSQL database. The statistical tests are performed using GNU R.[7] All the tools, links to the subject projects, and a dump of our database (50 GB) are available on the supplementary website.

4 Threats to Validity

Construct Validity A threat to construct validity is our operationalization of merge effort (dependent variable). Using the number of conflicts is a natural metric, as it represents the number of situations a developer has to inspect manually while merging. However, other factors have also an influence on the difficulty of resolving a conflict, for example, the size of the conflict. We also experimented with other definitions of merge effort, including the size of the conflicts, but did not get a different picture, as we discuss in Section 6.

Another threat to construct validity is the selection and definition of our metrics (independent variables). While we selected the metrics to plausibly capture causes of merge conflicts, different operationalizations may have resulted in different correlations, leading to accepting or rejecting different hypotheses. However, when exploring different operationalizations (see Section 6), the overall picture did not change, so we have sufficiently controlled this threat. Also, the aggregation function (geometric mean) we use to compare the branches threatens construct validity. Still, we deliberately chose geometric mean, because the product of the separate values is interpretable in our case (e.g., zero changes in one of the branches and there cannot be a conflict) and it better captures our data.

Internal Validity To test our hypotheses, we used real-world merge scenarios that we extracted from open-source projects on GITHUB. This selection procedure threatens internal validity, as only those merge commits made it into the repository that were actually successful in the end. It may be that the real nightmare scenarios, where developers resigned and rather rewrote the code or refrained from merging at all, are not included in the version history of our subject projects. This is a technical limitation by the version-control system, which is only tracking actual changes rather than failed attempts to incorporate changes. Still, we encountered several merge scenarios with many conflicts in our sample, which mitigates this threat. An alternative would have been to merge arbitrary branches at the risk that developers might not have intended to ever merge them in practice, which would threaten internal validity even more. We deliberately selected only real merge scenarios, accepting this threat, in exchange for being able to draw conclusions from real-world data of many projects from different domains.

[7] `http://www.r-project.org/`

Another threat is that we have no knowledge about the development workflows used in the subject projects and whether they had any influence on merging. For example, it might be that a developer had to resolve conflicts while rebasing branches, which we cannot replicate anymore because the version history has been rewritten. In a way, rebasing hides conflicts from later repository analysis, as the new version history is linear. Workflows including rebasing were also mentioned by participants of our survey. However, development models such as git-flow (which encourages a lot of merges) are very popular, which alleviates this threat. In scenarios where consumer-specific features are developed in branches [Staples and Hill, 2004; Rubin et al, 2013; Dubinsky et al, 2013; Antkiewicz et al, 2014], we expect more merging than rebasing, as the developers who want the features, would have to integrate them into their branches.

In our experiments, we observed a considerable number of conflicts in non-Java files, which we excluded from our analysis. While, this way, we reduced threats to validity that arise from differences between programming languages, this choice threatens external validity in the sense that we cannot generalize our findings to arbitrary artifacts. There is clearly a trade-off between internal and external validity [Siegmund et al, 2015], and we decided in favor of internal validity.

Finally, regarding our survey participants, there is the possibility of selection bias in that only motivated participants who are interested in merge conflicts took part. However, this does not affect the indicators that the participants identified, only the number of participants per indicator.

External Validity External validity is threatened by our restriction to Git and GitHub as platform and by focusing on Java projects. Thus, generalizability to other platforms and programming languages is limited. Both were necessary to reduce the influence of confounds, increasing internal validity [Siegmund et al, 2015]. Subsequent studies are needed to generalize to other version-control systems and programming languages. However, we are confident that we selected and analyzed substantial software systems from various domains, developed in parallel by multiple active developers.

Statistical Conclusion Validity Statistical conclusion validity is threatened by the distribution of conflicts in our data set, such that the values for parametric correlations would be biased. To minimize that threat, we used the Spearman correlation, which is non-parametric, because it is based on rank data. To check whether our indicators are capable of predicting the presence rather than the number of conflicts, we also computed the point biserial correlation coefficient. However, it turned out to be almost exactly the same result as computed with Spearman's correlation coefficient. However, our data are from a large number of real project repositories and, therefore, the distribution of conflicts is representative. Also, we found strongly correlating indicators for projects with few ($<$10) conflicts, and no correlating indicators for projects with a large number of conflicts. Therefore, we are confident that the distribution of conflicts has no significant influence on our findings.

As explained in Section 3, we chose geometric mean as summary statistic for our metrics. We experimented with various alternative statistics (e.g., mean and median), especially in the early phase of the project, but also later on as a sanity check. However, regarding correlations, the big picture did not change.

5 Results

In this section, we present the results of our empirical study, structured according to our hypotheses. An interpretation of our findings as well as explanations for the most extreme outliers is provided in Section 6. For each hypothesis, we present some descriptive statistics, Spearman's correlation coefficient (*cor*), and its statistical significance (p value). To accept a hypothesis, we look—beyond statistical significance[8]—at the effect size in terms of the correlation's strength (i.e., the value of *cor*). Since correlations with an absolute value of more than 0.6 are considered as strong, we set this as the threshold for accepting a hypothesis [Anderson and Finn, 1996]. As weak (0.2 to 0.39) and medium (0.4 to 0.59) correlations can potentially be used as predictors in a weighted combination with each other, we build a stepwise-regression model with according indicators in Section 6.

H_1 *Active, diverted branches (in terms of number of commits) are more likely to result in conflicts than inactive branches that remain close to each other.*

The merge scenarios extracted from our subject projects have, on average, 19.48 commits per merge scenario. In Figure 4, we show a scatter plot of the number of commits and the merge effort in terms of number of conflicts. There is almost no correlation (*cor*=0.16, $p<0.05$), so we reject this hypothesis.

H_1: Rejected. Number of commits does not correlate strongly with the number of merge conflicts.

H_2 *Many commits within a small time span are more likely to produce conflicts than the same number of commits over longer time spans.*

On average, the merge scenarios had 9.35 commits in the last week and 11.26 commits within the last two weeks. We did not observe any strong correlation (*cor*=0.13, $p<0.05$ and *cor*=0.14, $p<0.05$) with the number of conflicts, as illustrated in Figure 5. Therefore, we reject this hypothesis.

[8] large data set, easily significant, no value for any claims

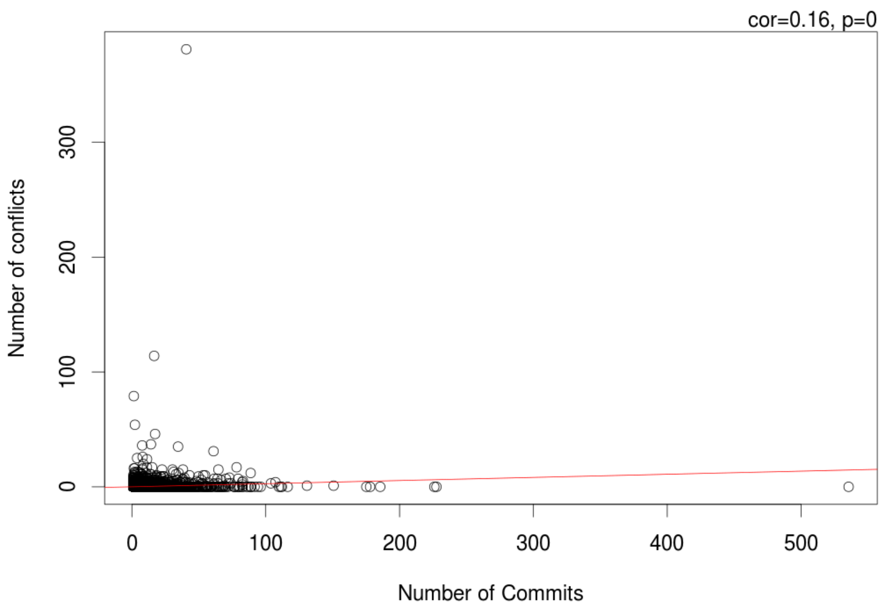


Fig. 4 Number of commits vs. number of conflicts

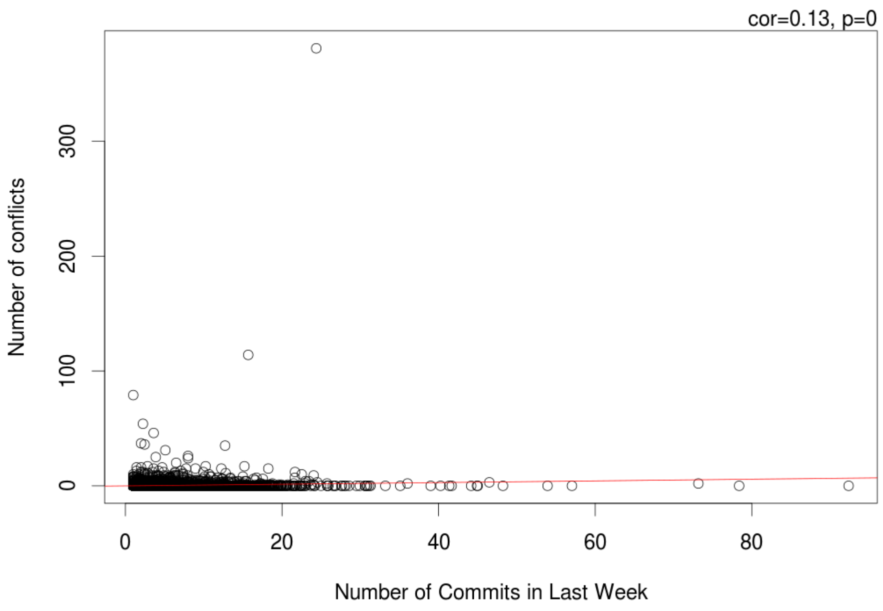


Fig. 5 Commits last week vs. number of conflicts

H_2: Rejected. Commit density does not correlate strongly with the number of conflicts.

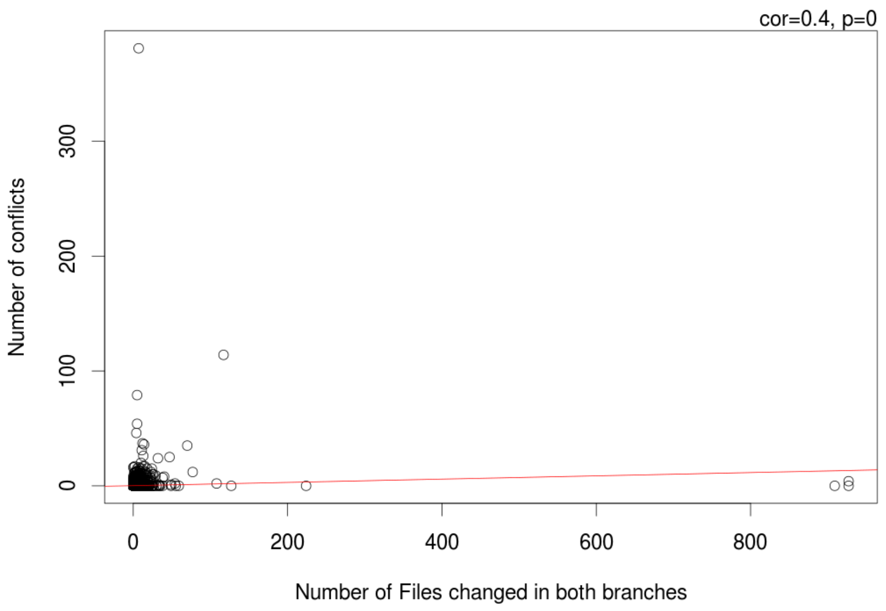


Fig. 6 Files changed by both branches vs. number of conflicts

H₃ *The more files are changed by both branches, the more likely a conflict occurs.*

On average, 0.63 files have been changed in both branches for a merge scenario, as shown in Figure 6. Here, we observe a medium correlation (cor=0.40, $p<0.05$), but still we have to reject this hypothesis.

H_3: Rejected. The number of files changed in both branches does not correlate strongly with the number of conflicts.

H_4 *Larger changes that modify more lines of code are more likely to cause conflicts than smaller changes.*

In an average change scenario, 180.09 lines of code are changed. Our assumption that large changes correlate with the number of conflicts is again only reflected with a medium correlation (cor=0.43, $p<0.05$, see Figure 7). Therefore, we reject the hypothesis.

H_4: Rejected. Large changes do not correlate strongly with the number of conflicts.

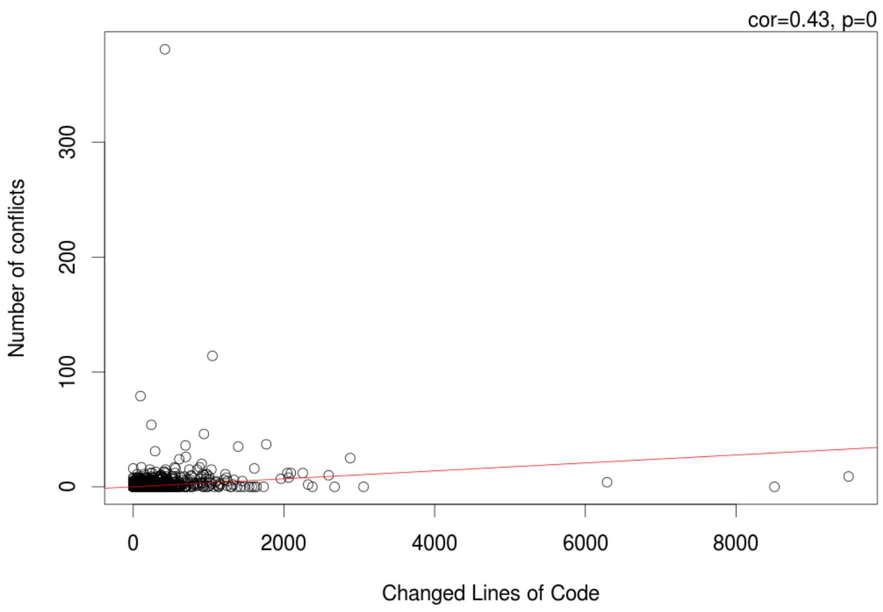


Fig. 7 Change size vs. number of conflicts

H_5 *More code fragmentation (tangled changes) of the committed changes results in more conflicts than lesser code fragmentation.*

On average, 100.12 chunks with an average size of 13.05 AST nodes per scenario are changed. This shows that, on average, many small changes (one to two lines) had to be merged. We observed only a low correlation regarding this indicator (cor=0.24, p<0.05), which is illustrated in Figure 8. Thus, we reject this hypothesis.

H_5: Rejected. Fragmentation of changes does not correlate strongly with the number of conflicts.

H_6 *Scattered changes (across classes or methods) are more likely to lead to conflicts than cohesive changes.*

The average scattering degree per merge scenario is 0.19 for classes and 0.06 for methods. We only found a low correlation to merge conflicts for classes (cor=0.21, p<0.05) and for methods (cor=0.24, p<0.05), which is illustrated in Figures 9 and 10. Therefore, we reject this hypothesis.

H_6: Rejected. Scattered changes across classes or methods do not correlate strongly with the number of conflicts.

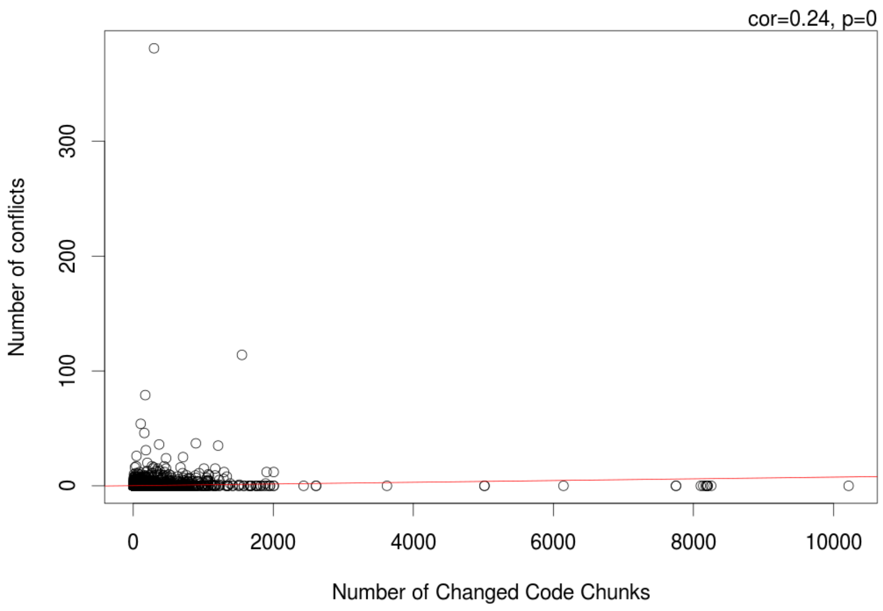


Fig. 8 Number of chunks vs. number of conflicts

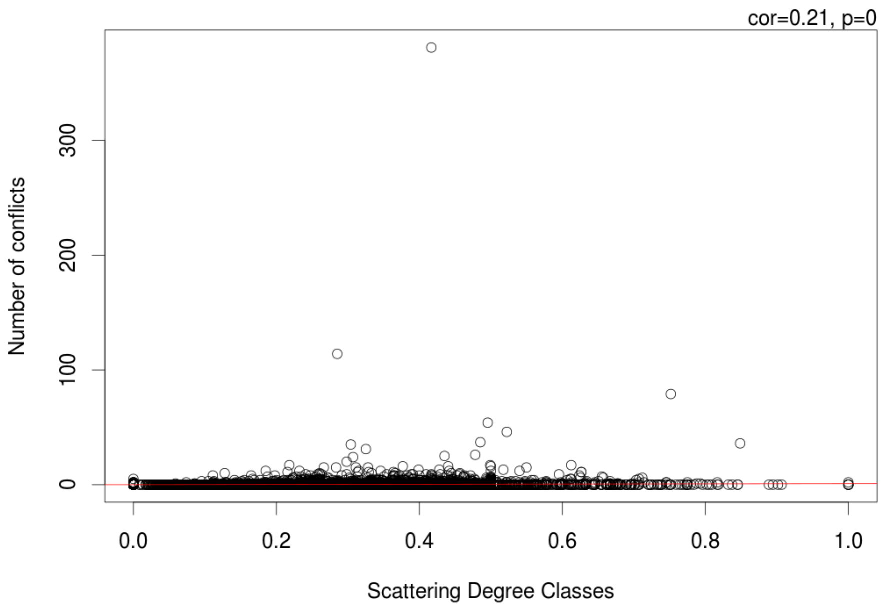


Fig. 9 Scattering over classes vs. number of conflicts

H$_7$ *Changes above the level of class declarations (which are often inserted and maintained automatically) are more likely to lead to merge conflicts than changes inside class declarations (introduced by human developers).*

On average, 10.00% of all changed AST nodes per scenario are above the class level (e.g., package declarations or import statements). Concerning the number

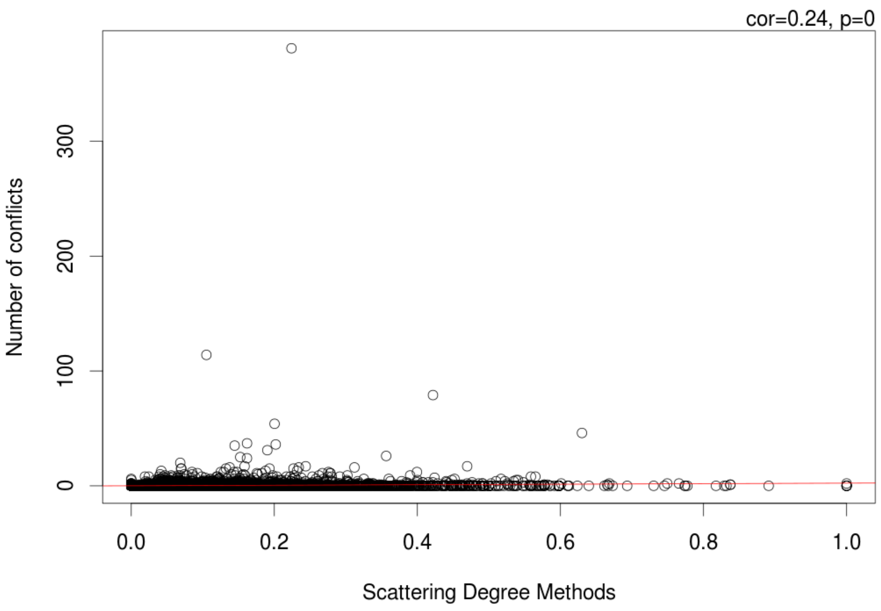


Fig. 10 Scattering over methods vs. number of conflicts

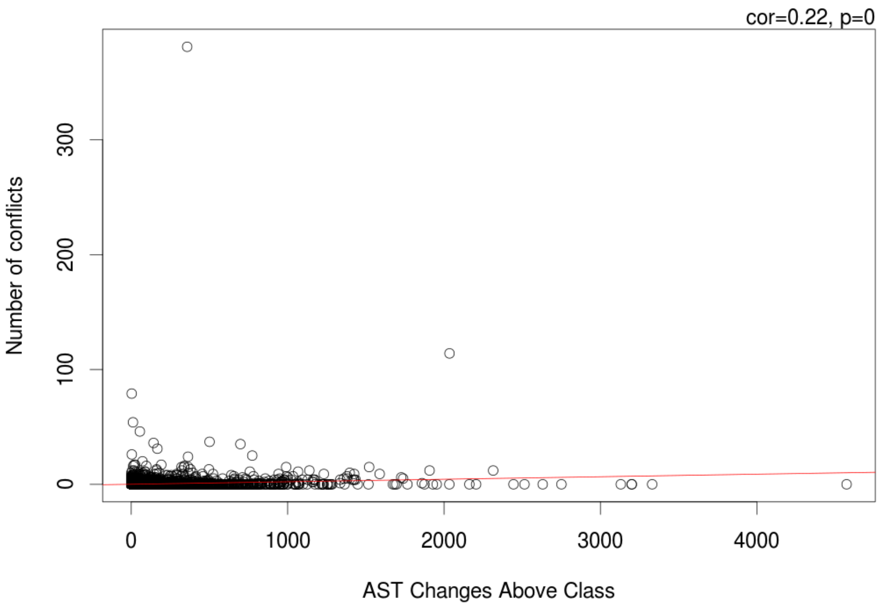


Fig. 11 Rate of changes above class level vs. number of conflicts

of merge conflicts, we again found only low correlations (cor=0.22, p<0.05) for the percentage of changes above the class level, and (cor=0.26, p<0.05) for the percentage of changes within class declarations, as shown in Figure 11 and Figure 12, respectively. Therefore, we reject this hypothesis.

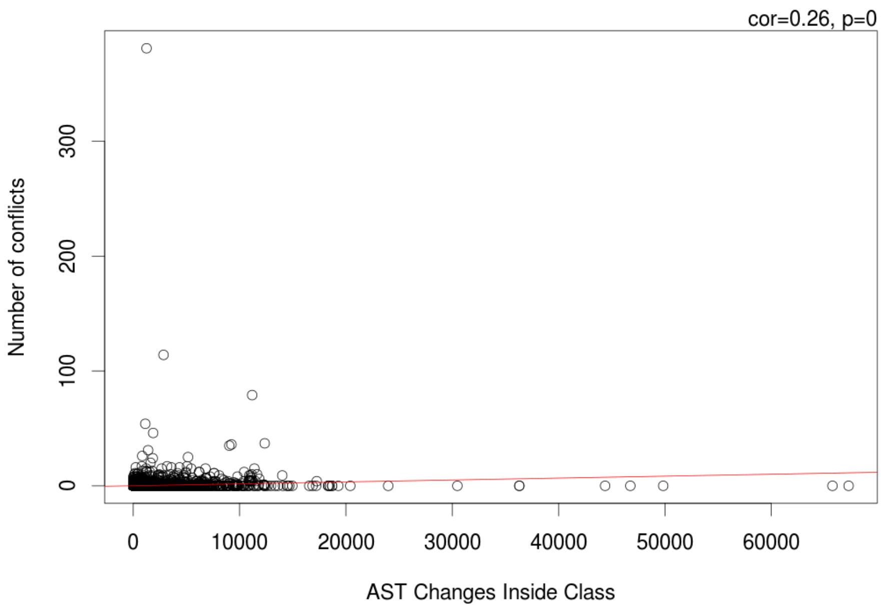


Fig. 12 Rate of changes in classes vs. number of conflicts

H_7: Rejected. Granularity of changes (above or within class declarations) has no strong correlation with the number of merge conflicts.

6 Discussion and Perspectives

6.1 Diving into the Data

We—and the participants of our survey—expected that several indicators, as introduced in Section 2, show a clear correlation with the number of merge conflicts in practice. However, in our study, we observed a different picture: None of the indicators exhibit a strong correlation that we can use for prediction. Actually, we were surprised by the low correlations, for example, with the number of commits or the number of files changed by both branches, making these indicators useless as predictors. We expected, at least, some indicators exhibiting strong correlations, allowing us to combine them to predict the likeliness of conflicts.

To better understand the absence of correlation, we took a closer look at our data set. In particular, we looked for correlations on a per-project basis. We found that some indicators—that correlate only weakly or at most medium with the number of conflicts in general—exhibit a strong correlation for some projects. For example, the number of changed lines shows only a medium correlation overall, but is a significant and strong indicator for 14 projects; for 80 projects, we found, at least, a medium correlation. A similar case is the number of files that were touched by both revisions: Overall, we observe a medium correlation (*cor*=0.40, $p<0.05$); however, for 5 of the subject projects, we found a strong correlation with the number of conflicts, and a medium correlation for 86 projects.

These results suggest that indicators are not (or cannot be) project-independent, leading naturally to the question of how to come up with project-specific or, possibly, domain-specific indicators. This is an interesting avenue of further research. Our data set can provide a good start for this, but is beyond the scope of this study, in which we specifically looked for project-independent indicators with predictive power for the occurrence of merge conflicts. A preliminary attempt to cluster our sample systems (by size and application domain) did not result in additional insights.

Furthermore, we looked into the scenarios that caused extreme outliers. In one merge scenario of the project BroadleafCommerce, 927 files were changed by both of the competing branches. Also, all classes and methods of the project were changed, resulting in an unusually high number of changed lines. Therefore, according to several of our hypotheses, we expected a rather high number of merge conflicts. But when merging this scenario, our tool did not report a single conflict. So, what might be the reason? During our manual investigation, it turned out that this project converted its code base from tabulators to spaces, and apparently did so for each of its branches via independent commits (and, therefore, before the merge commit we encountered). As a result of this conversion, the competing branches actually had a larger common code base as the diffs to their common ancestor in the version history suggest. This is a good example of why prediction of merge conflicts based on cheap-to-compute metrics is not straight forward. Another scenario that we inspected, taken from project groovy-core, had a large number of commits (372 in sum, 295 in one branch and 77 in the other), but again, no conflicts. In this case, we found that one branch committed a number of small and concise bug fixes and documentation changes in the code base, whereas the other branch mainly cleaned up the benchmark suite. So, despite the large number of commits, only 13 files were changed by both branches and did not contain overlapping changes, so they could be merged without conflict. The other extreme was a scenario taken from AndEngine with only 6 commits in total (5 in one branch and a single one in the other) and 5 simultaneously changed files resulting in 54 conflicts. Here, both branches renamed several parameters differently. For example, variable halfDeltaX from the base revision was renamed to rotateX by one revision, and rotationCenterX by the other. Therefore, a conflict is produced at the definition and all uses for each of these variables, resulting in a high number of conflicts. Several merge scenarios included conflicts due to layouting changes (tabs vs. spaces and automatic code formatters). In some scenarios, complex refactorings were performed, which led to conflicts.

6.2 Model of Conflict-Prediction Indicators

To determine whether the indicators in combination can predict the occurrence of merge conflicts, we build a stepwise-regression model[9]. Stepwise regression determines a weighted combination of factors that best predict frequencies of merge conflicts. It is important to note that we use stepwise regression not as an evaluatory strategy, but an exploratory one that may point to future research directions. In a nutshell, the model is significant ($p<0.05$), but the adjusted R^2 is 0.04, meaning that 4% of the variance in the merge conflicts can be explained by the model. In other words, for our projects, a weighted combination is as good/bad as a single indicator, and therefore not necessarily useful for predicting merge conflicts.

6.3 Operationalization

Since our operationalization of merge difficulty and effort covers this dependent variable only partially, we experimented with other operationalizations, including the size of conflicts (hypothesis: conflicts spanning more lines of code contribute more to merge effort than conflicts spanning fewer lines of code). However, the big picture remained the same: For some projects, we found higher correlations, but none that hold for the whole data set. Nevertheless, in future work, further aspects of merge difficulty and effort should be included in the investigation.

Likewise, one could play with the aggregation functions used to compute the indicators from the values computed for the respective branches. But, as our intent was to compare the branches, using the geometric mean is straightforward. In general, one has to be careful that the aggregated value remains interpretable, otherwise this might end up in “fishing for results”.

6.4 Other Indicators

Furthermore, it is certainly advisable to include more potential indicators in the analysis. However, to avoid a trial-and-error process and fishing for results, we need to apply a systematic process of identifying promising indicators, be it by means of more extensive developer surveys or by using sophisticated mining techniques. For example, it could be useful to detect overlapping changes, which might be established by a code-change analysis that compares the regions edited by specific developers. Or, existing information concerning communication and collaboration among developers [Joblin et al, 2015] could be extracted from mailing lists, bug trackers, etc., to detect developers working on the same code blocks, functions, or features, as proposed by recent awareness approaches [Brun et al, 2011]. This information could be augmented by knowledge of the organizational structure of the development teams or other

[9] `https://stat.ethz.ch/R-manual/R-devel/library/MASS/html/stepAIC.html`

social information, such as the reputation of a developer, which have been used in other work to predict software quality (cf. Section 7).

It would also be interesting to present our results to the survey participants and ask for explanations. However, since the responses were submitted anonymously, we cannot ask the same developers again. Nevertheless, a new survey based on these results could also help to generate hypotheses about indicators and their aggregation.

Finally, on a personal note, in previous presentations of the results, we repeatedly observed resistance, suspecting that the negative results are merely an artifact of our data set or our operationalization. We perceived a deeply held believe that correlations should exist, with "the study done right", similar to our own initial expectations and the expectations of our survey participants. We received various suggestions (including from reviewers of prior submissions of this work), including additional metrics, different operationalizations of our measures (e.g., commit density), normalizing by project size, or only studying scenarios with conflicts. In fact, we have rechecked our data repeatedly and tried many alternatives to our analysis, including all the suggested ones without any substantially different results. [10] While we appreciate alternative suggestions—every suggestion that we try and results in the same conclusion strengthens our results—we are confident and tested our results on a substantial data set. We have made all data and infrastructure available and invite others to explore other metrics or replicate the study on a different data set. In fact, the kind of resistance we observed in presentations and reviews emphasizes that the results are indeed surprising and counter common assumptions held by the community, giving a strong incentive to also communicate such negative result.

7 Related Work

There is substantial work that strives for increasing the developers' *awareness* of changes in parallel development. PALANTÍR was one of the first approaches raising the developers' awareness of concurrent changes [Sarma et al, 2003, 2012]. It informs a developer which other developers change which other artifacts, calculates a simple measure of severity of changes, and visualizes this information properly. FASTDASH provides an interaction visualization augmenting existing software tools with information about what other developers in a team are doing [Biehl et al, 2007]. In the same vein, COLLABVS [Dewan and Hegde, 2007] aims at detecting conflicts as early as possible—even before the changes are committed—by propagating possible overlaps when corresponding code sections are edited by developers. SYDE uses information from the AST (as we do with JDIME [Leßenich et al, 2014]) to make the analysis of changes more precise [Hattori and Lanza, 2010]. All these awareness

[10] Only the last suggestion, studying only merge scenarios with conflicts, leads to high correlations that, however, are meaningless because this approach neglects the fact that merge scenarios indeed often have zero conflicts.

approaches require a dedicated infrastructure for monitoring, analyzing, and notifying, that developers are supposed to use in their every-day work; we focus on a lightweight, tool-independent solution.

A further approach to avoid complex merge scenarios is *speculative merging.* CRYSTAL [Brun et al, 2011] speculates what a developer will do in the future and checks whether those actions will result in conflicts. WECODE [Guimarães and Silva, 2012] integrates continuous merging of committed and uncommitted changes into an IDE and reports conflicts via a team view. While promising, these speculative approaches require dedicated tool support, and they extensively perform merges in the background, of which is unclear whether that scales to large-scale software projects with many long-living branches.

Bird and Zimmermann [2012] studied branches intensively at Microsoft. By means of a survey, they found that developers spend significant time dealing with merge conflicts, and they identified the prevalence of several anti-patterns. They identified the metrics *liveness* and *isolation* to classify the importance of branches, and introduced a what-if analysis to approximate the consequences of removing a branch. They use the number of actual merge conflicts as input for their isolation metric, whereas we attempt to predict the number of merge conflicts with lightweight metrics.

There have been some attempts to *predict* certain properties or situations by means of metrics gathered from the development artifacts, their context, and their history. In particular, fault prediction received significant attention [Nagappan and Ball, 2010; Catal and Diri, 2009; Gyimothy et al, 2005; Bettenburg and Hassan, 2010; El Emam et al, 2001], but many other areas have been explored, such as predicting the chance that a pull request gets accepted on GITHUB [Tsay et al, 2014], that a patch is a bug fix [Tian et al, 2012], and predicting self-admitted code hacks [Potdar and Shihab, 2014]. These approaches typically analyze social aspects of software projects regarding their predictive power for software quality, but they are also possibly useful for predicting merge conflicts. A meta-study comparing factors across different kinds of prediction efforts may yield insights for more reliable predictors across different projects.

Furthermore, other kinds of conflicts could be examined, for example, build or test-suite conflicts [Brun et al, 2011]. Our setup could be adopted as a foundation for such an analysis.

Merging is a key operation in *product-line engineering* based on version-control systems. Staples and Hill [2004] explicitly documented the role of branching and merging in product-line engineering. This early work has been extended toward the vision of a *virtual platform* for product-line engineering, which provides flexible views and operations, to which diffing and merging is central [Rubin et al, 2013; Dubinsky et al, 2013; Antkiewicz et al, 2014]. Similarly, many product-line projects start with a code base of several forked or loosely coordinated products that should be integrated into a single implementation [Berger et al, 2013]. In this context, first assessing the severity of the differences among the products and then reverse engineering them are central challenges [Duszynski et al, 2011; Kim et al, 2007; Rubin and Chechik, 2013;

Ryssel et al, 2010; Faust and Verhoef, 2003; Pinzger et al, 2003]. Indicators—if they had predictive power—would be valuable information within the virtual platform and useful to coordinate integration efforts.

Advanced merging techniques, such as syntactic or operation-based merging, can reduce the number of conflicts significantly [Mens, 2002; Dig et al, 2008; Leßenich et al, 2014; Rubin and Chechik, 2013] as compared to traditional text-based merge tools. During experiments, we could verify that our syntactic merge tool, which resolves formatting issues and ordering conflicts, is able to reduce the number of reported conflicts. Finally, such approaches enable the detection of refactorings, which are likely to introduce merge conflicts [Dig et al, 2008; Mahouachi et al, 2013].

8 Conclusion

Software merging is a challenging and tedious task in the practice of software engineering [Mens, 2002; Bird and Zimmermann, 2012]. Recent work on speculative merging [Brun et al, 2011; Guimarães and Silva, 2012] and awareness tools [Biehl et al, 2007; Dewan and Hegde, 2007; Hattori and Lanza, 2010; Sarma et al, 2012; Guimarães and Silva, 2012] for parallel development suggests that complex merge scenarios should be avoided. While this work is promising, we strive for a solution that avoids assumptions that these approaches make, including the usage of a specific IDE and about the complexity of the merge scenarios involved (many small merges vs. few large merges).

In a survey, 41 developers shared their experience on which factors cause merge conflicts. From the survey responses and our own intuition, we extracted a set of 7 indicators for predicting the number of conflicts in merge scenarios. By means of an empirical study on 163 open-source Java projects, involving 21,488 merge scenarios and 49,449,773 lines of code, we computed correlations between the indicators and the number of conflicts in a merge scenario.

As a key result, we found that none of the 7 indicators—as suggested by the developer survey—can predict the number of merge conflicts. Nevertheless, despite this overall negative'result, our study, data, analysis, and experiment infrastructure form a solid basis for replication and follow-up studies (e.g., involving further indicators and more aspects of merge difficulty) as well as for research on alternative conflict-prediction approaches (e.g., domain-specific) and conflict-avoidance strategies (e.g., speculative merging).

Finally, the results of our study shall serve as a warning to practitioners and researchers when making assumptions about merging and merge conflicts.

Acknowledgments

We thank all survey participants for their insightful comments and suggestions. This work has been supported by the German Research Foundation (AP 206/4, AP 206/5, and AP 206/6).

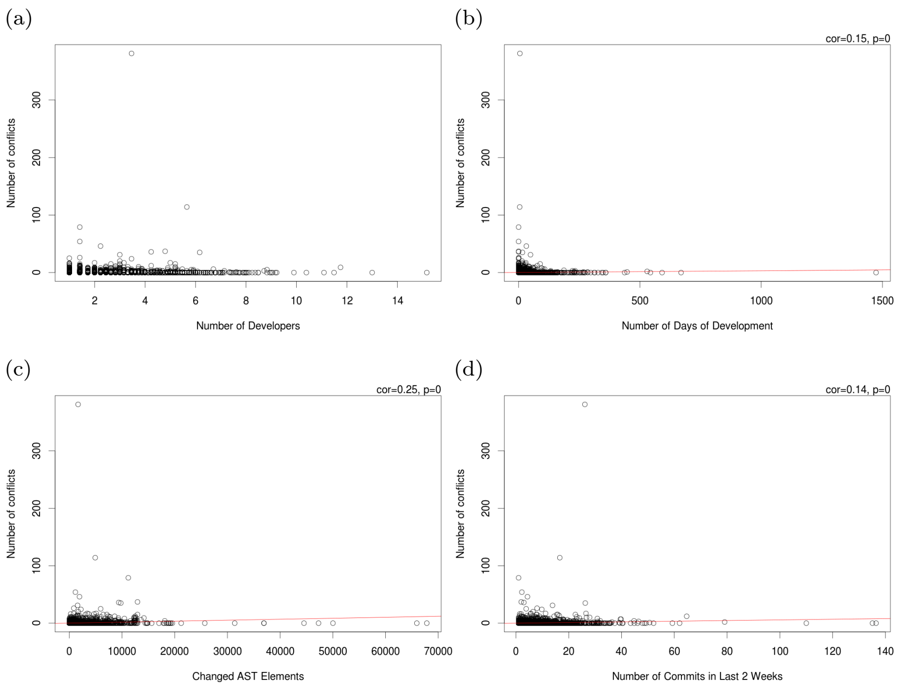


Fig. 13 (a) Number of developers vs. number of conflicts, (b) Number of days of development vs. number of conflicts, (c) Number of changes AST nodes vs. number of conflicts, (d) Commits last two weeks vs. number of conflicts

Appendices

A Further Indicators

In addition to the indicators derived from our hypotheses in Section 2, we experimented with other potential indicators as well. Here, we present the results that we computed using these alternative indicators.

First, we computed the number of developers that were involved in each of the two competing branches of a merge scenario. Our intention was that the more developers contribute to a merge scenario, the more likely it is that there will be conflicts, as developers are unaware of what the others are changing. Following the procedure we applied to our other indicators, we computed the geometric mean of both values and correlated it with the number of merge conflicts. We observed almost no correlation (cor=0.08, p<0.05), so we assume that this indicator is not useful for predicting merge conflicts. The corresponding scatter plot is shown in Figure 13a.

Another assumption also mentioned in the survey is that branches that are developed over a long time without a merge are more likely to lead to merge conflicts. To test this hypothesis, we computed the number of days of development for both branches, and correlated the geometric mean with the number of conflicts. As shown in Figure 13b, we did not even observe a weak correlation here (cor=0.15, $p<0.05$), so we had to reject this idea as well.

To test $\mathbf{H_4}$, we use the number of changed lines of code to capture the size of changes within a merge scenario. An alternative representation of that can be expressed via the number changed nodes in the abstract syntax trees that are merged. A scatter plot of this variation of $\mathbf{H_4}$ can be seen in Figure 13c. As for the results, the correlation we observed is even lower (cor=0.25, $p<0.05$) than the line-based metric presented in Section 5 (cor=0.43, $p<0.05$).

As mentioned in Section 5, we tested a variation of $\mathbf{H_2}$ by looking at the last two weeks before the merge instead of only the last week. The result is displayed in Figure 13d.

B Developer Survey

In what follow, we show the complete questionnaire from which we derived our hypotheses.

Introduction

In this survey, we assess the frequency, cause, and nature of merge conflicts in the context of version-control systems. There are 35 small questions—when asked for numbers, a rough estimate is sufficient. You can leave comments on most questions, but you do not have to. Also, each question is optional. Your data will of course be anonymized. If you are interested in the results, you can leave your e-mail address at the end of this survey. If you have any questions, please contact us.

Olaf Leßenich[1], Janet Siegmund[1], Christian Kästner[2], Sven Apel[1], Claus Hunsen[1]

[1] University of Passau, [2] Carnegie Mellon University

1. In your opinion, what are typical factors and situations that most likely lead to merge conflicts?

Conflicts

We have some questions regarding commit conflicts that you encountered. If you are working on more than one project, please consider the project you mainly work on. All questions are optional.

2. What is done in your project to prevent/reduce conflicts?

3. What is the policy on how to deal with conflicts?

4. Do you sometimes avoid a pull/update because of potential conflicts?

◯ Yes
◯ No

Optional Comments:

5. How often do conflicts occur?

◯ Almost never ◯ Not so often ◯ Sometimes ◯ Very often ◯ Almost at each commit

Optional Comments:

6. How large are the conflicts typically?

◯ Small ◯ Medium ◯ Large

Optional Comments:

7. Where do conflicts typically occur?

◯ Only in one file ◯ In more than one file

Optional Comments:

8. How long does it take to resolve a typical conflict?

◯ < 5 minutes
◯ 6 to 30 minutes
◯ > 30 minutes
◯ > 1 hour

Optional Comments:

9. What induces conflicts in your project?

◯ Almost never ◯ Not so often ◯ Sometimes ◯ Very often ◯ Almost at each commit

Optional Comments:

10. Of what kind are typical conflicts?

☐ Organization (or lack of it, that is)
☐ Wrong use of version-control system
☐ Commit size
☐ Number of commits
☐ Time between synchronization

☐ Heterogeneous environments (different OS, editors/IDEs, ...)
☐ Customer specifications
☐ Release pressure
☐ Conflicts on a certain day of the week (e.g., fridays)
☐ Other: __________

Optional Comments:

11. Do you see any relations between causes and different types of conflicts?

12. Which kind of conflicts do you find most difficult to resolve? Do you see a pattern?

13. Are there hotspots for conflicts in the code-base, i.e., are there regions or files that are very often part of conflicts?

◯ Yes ◯ No

Optional Comments:

14. Are the same or different developers causing most of the conflicts?

◯ Same developers ◯ Different developers

Optional Comments:

15. In your opinion, what could be done to improve the situation?

Project-Specific Questions

Now for some project-specific questions. If you are working on more than one project, please consider the project you mainly work on. All questions are optional.

16. Which version-control system(s) do you use?

☐ Git
☐ Mercurial
☐ Subversion
☐ CVS
☐ Bazaar
☐ DARCS
☐ BitKeeper
☐ Perforce
☐ Other: __________

17. How many developers work on your project?

◯ 1 to 2
◯ 3 to 5
◯ 6 to 10
◯ 11 to 30
◯ 31 to 50
◯ 51 to 100
◯ 101 to 500
◯ > 500

18. With how many developers do you typically collaborate?

◯ None
◯ 1 or 2
◯ 3 to 5
◯ 6 to 10
◯ 11 to 20
◯ > 20

19. How often does a commit occur from anybody on the project?

◯ Daily
◯ Weekly
◯ Monthly
◯ Other: ___________

20. How large is a typical commit? (number of changed lines)

◯ Less than 5 LOC
◯ 5 to 20 LOC
◯ > 20 LOC
◯ > 50 LOC
◯ > 100 LOC

Optional Comments:

21. How often does a release occur?

____

22. What commit policies are typically enforced?

☐ Must build
☐ Must adhere to coding style
☐ Must pass test suite
☐ Must pass code review
☐ Other: ___________

Optional Comments:

23. Are there branches or forks/clones of your project?

Number of branches: ___________

Number of forks/clones: __________

24. Are there forks or clones of your project that are maintained by a different community/company?

◯ Yes ◯ No

Optional Comments:

25. What are the release policies? Which criteria must be fulfilled before the release? (E.g., no critical bugs, code reviews, testing by customer)

26. What describes your development model best?

☐ Agile
☐ Waterfall
☐ "Code and fix"
☐ Other: __________

Optional Comments:

Personal Information

Finally, some personal background information to set your responses into context. If you are working on more than one project, please consider the project you mainly work on. All questions are optional.

27. Gender:

◯ Male
◯ Female

28. Age:

____

29. Educational Background:

◯ No degree ◯ Bachelor's Degree
◯ Master's Degree
◯ Ph.D.
◯ Other: __________

30. Education Background: Area (e.g., computer science):

____

31. Since how many years are you programming professionally?

____

32. With how many programming languages do you have passing familiarity? (Rough estimate)

____

33. What is the size of the project you work on?

◯ Small ($<$ 900 lines of code) ◯ Medium (900 - 45 000 lines of code)
◯ Large ($>$ 45 000 lines of code)

34. Are you working on an open-source project or a proprietary project?

☐ Open-source project
☐ Proprietary project

35. What's the domain of your project (e.g., database, embedded system) If multiple domains apply, please separate each domain by comma.

Further Comments

36. Do you have any further comments on commit conflicts or this survey?

37. You can leave your e-mail address if you are interested in the results (but of course you do not have to):

____

Thank you very much for you time. We highly appreciate your input. If you have any questions, please contact us.

Olaf Leßenich[1], Janet Siegmund[1], Christian Kästner[2], Sven Apel[1], Claus Hunsen[1]

[1] University of Passau, [2] Carnegie Mellon University

References

Anderson T, Finn J (1996) The New Statistical Analysis of Data. Springer

Antkiewicz M, Ji W, Berger T, Czarnecki K, Schmorleiz T, Lämmel R, Stănciulescu t, Wąsowski A, Schaefer I (2014) Flexible Product Line Engineering with a Virtual Platform. In: Companion Volume ICSE, ACM, pp 532–535

Apel S, Liebig J, Brandl B, Lengauer C, Kästner C (2011) Semistructured Merge: Rethinking Merge in Revision Control Systems. In: Proc. ESEC/FSE, ACM, pp 190–200

Berger T, Rublack R, Nair D, Atlee JM, Becker M, Czarnecki K, Wąsowski A (2013) A survey of variability modeling in industrial practice. In: Proc. Int. Workshop on Variability Modelling of Software-intensive Systems (VaMoS), ACM, pp 7:1–7:8

Bettenburg N, Hassan A (2010) Studying the Impact of Social Structures on Software Quality. In: Proc. ICPC, IEEE, pp 124–133

Biehl JT, Czerwinski M, Smith G, Robertson GG (2007) FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. In: Proc. CHI, ACM, pp 1313–1322

Bird C, Zimmermann T (2012) Assessing the Value of Branches with What-if Analysis. In: Proc. ACM SIGSOFT FSE, ACM, pp 45:1–45:11

Brun Y, Holmes R, Ernst MD, Notkin D (2011) Proactive Detection of Collaboration Conflicts. In: Proc. ESEC/FSE, ACM, pp 168–178

Catal C, Diri B (2009) A Systematic Review of Software Fault Prediction Studies. Expert Systems with Applications 36(4):7346–7354

Dewan P, Hegde R (2007) Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. In: Proc. ECSCW, Springer, pp 159–178

Dig D, Manzoor K, Johnson R, Nguyen TN (2008) Effective Software Merging in the Presence of Object-Oriented Refactorings. IEEE TSE 34(3):321–335

Dubinsky Y, Rubin J, Berger T, Duszynski S, Becker M, Czarnecki K (2013) An Exploratory Study of Cloning in Industrial Software Product Lines. In: Proc. CSMR, IEEE, pp 25–34

Duszynski S, Knodel J, Becker M (2011) Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In: Proc. WCRE, IEEE, pp 303–307

El Emam K, Benlarbi S, Goel N, Rai SN (2001) The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. IEEE TSE 27(7):630–650

Faust D, Verhoef C (2003) Software Product Line Migration and Deployment. Software: Practice and Experience 33(10):933–955

Guimarães ML, Silva AR (2012) Improving Early Detection of Software Merge Conflicts. In: Proc. ICSE, IEEE, pp 342–352

Gyimothy T, Ferenc R, Siket I (2005) Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. IEEE TSE 31(10):897–910

Hattori L, Lanza M (2010) Syde: A Tool for Collaborative Software Development. In: Companion Volume ICSE, ACM, pp 235–238

Hudson W (2013) Card Sorting. In: Guide to Advanced Empirical Software Engineering, The Interaction Design Foundation

Jedlitschka A, Pfahl D (2005) Reporting Guidelines for Controlled Experiments in Software Engineering. In: Proc. ESE, IEEE, pp 95–104

Joblin M, Mauerer W, Apel S, Siegmund J, Riehle D (2015) From Developer Networks to Verified Communities: A Fine-Grained Approach. In: Proc. ICSE, IEEE, pp 563–573

Kim M, Notkin D, Grossman D (2007) Automatic Inference of Structural Changes for Matching Across Program Versions. In: Proc. ICSE, IEEE, pp 333–343

Leßenich O, Apel S, Lengauer C (2014) Balancing Precision and Performance in Structured Merge. Automated Software Engineering pp 1–31

Mahouachi R, Kessentini M, Cinnéide MÓ (2013) Search-based refactoring detection. In: Proc. Int. Conference Genetic and Evolutionary Computation Conference (GECCO), pp 205–206

Mens T (2002) A State-of-the-Art Survey on Software Merging. IEEE TSE 28(5):449–462

Muşlu K, Bird C, Nagappan N, Czerwonka J (2014) Transition from Centralized to Decentralized Version Control Systems: A Case Study on Reasons, Barriers, and Outcomes. In: Proc. ICSE, ACM, pp 334–344

Nagappan N, Ball T (2010) Making Software: What Really Works, and Why We Believe It, O'Reilly, chap Evidence-based Failure Prediction, pp 415–434

Pinzger M, Gall H, Girard JF, Knodel J, Riva C, Pasman W, Broerse C, Wijnstra JG (2003) Architecture Recovery for Product Families. In: Proc. Workshop Software Product-Family Engineering, Springer, pp 332–351

Potdar A, Shihab E (2014) An Exploratory Study on Self-Admitted Technical Debt. In: Proc. ICSME, IEEE

Rubin J, Chechik M (2013) N-way Model Merging. In: Proc. ESEC/FSE, ACM, pp 301–311

Rubin J, Czarnecki K, Chechik M (2013) Managing Cloned Variants: A Framework and Experience. In: Proc. SPLC, ACM, pp 101–110

Ryssel U, Ploennigs J, Kabitzsch K (2010) Automatic Variation-point Identification in Function-block-based Models. In: Proc. GPCE, ACM, pp 23–32

Sarma A, Noroozi Z, van der Hoek A (2003) Palantír: Raising Awareness Among Configuration Management Workspaces. In: Proc. ICSE, IEEE, pp 444–454

Sarma A, Redmiles D, van der Hoek A (2012) Palantír: Early Detection of Development Conflicts Arising from Parallel Code Changes. IEEE TSE 38(4):889–908

Siegmund J, Schumann J (2014) Confounding Parameters on Program Comprehension: A Literature Survey. Empirical Software Engineering 20(4):1159–1192

Siegmund J, Siegmund N, Apel S (2015) Views on Internal and External Validity in Empirical Software Engineering. In: Proc. ICSE, IEEE, pp 9–19

Stanciulescu S, Schulze S, Wasowski A (2015) Forked and integrated variants in an open-source firmware project. In: Proc. ICSME, pp 151–160

Staples M, Hill D (2004) Experiences Adopting Software Product Line Development without a Product Line Architecture. In: Proc. APSEC, IEEE, pp 176–183

Tian Y, Lawall J, Lo D (2012) Identifying Linux Bug Fixing Patches. In: Proc. ICSE, ACM, pp 386–396

Tsay J, Dabbish L, Herbsleb J (2014) Influence of Social and technical Factors for Evaluating Contribution in GitHub. In: Proc. ICSE, ACM, pp 356–366