Build Conflicts in the Wild

Léuson Da Silva[a,*], Paulo Borba[a], Arthur Pires[a]

[a]*Informatics Center, Federal University of Pernambuco, Recife, Brazil*

Abstract

When collaborating, developers often create and change software artifacts without being fully aware of other team members' work. While such independence is essential for increasing development productivity, it might also result in conflicts when integrating developers code contributions. To better understand some of these conflicts— the ones revealed by failures when building integrated code— we investigate their frequency, structure, and adopted resolution patterns in 451 open-source Java projects. To detect such build conflicts, we select merge scenarios from git repositories, parse the Travis logs generated when building the commits, and check whether the logged build error messages are related to the merged changes. We find and classify 239 build conflicts and their resolution patterns. Most build conflicts are caused by missing declarations removed or renamed by one developer but referenced by another developer. Conflicts caused by renaming are often resolved by updating the missing reference, whereas removed declarations are often reintroduced. Most fix commits are authored by one of the contributors involved in the merge scenario. We also detect and analyze build failures caused by immediate post integration changes, which are often performed with the aim of fixing merge conflicts but end up leading to build issues. Based on our catalogue of build conflict causes, awareness tools could alert developers about the risk of conflict situations. Program repair tools could benefit from our catalogue of build conflict resolution patterns to auto-

*Corresponding author

Email addresses: lmps2@cin.ufpe.br (Léuson Da Silva), phmb@cin.ufpe.br (Paulo Borba), ahlp2@cin.ufpe.br (Arthur Pires)

matically fix conflicts; we illustrate that with a proof of concept implementation of a tool that recommends fixes for conflicts.
Keywords: Code integration, Conflicting contribution, Build conflicts, Broken builds

1. Introduction

When collaborating, developers create and change software artifacts often without full awareness of changes being made by other team members. While such independence is essential for non-small teams, and might increase development productivity, it might also result in conflicts when integrating developers changes. In fact, high degrees of parallel changes and integration conflicts have been observed in a number of industrial and open-source projects that use different kinds of version control systems [1, 2, 3, 4]. This has been observed even when using advanced merge tools [5, 6, 7, 8, 9] that avoid common spurious merge conflicts reported by state of the practice tools.

Resolving such integration conflicts might be time consuming and is an error-prone activity [10, 11, 12], negatively impacting development productivity. So, to avoid dealing with them, developers have been adopting risky practices, such as rushing to finish changes first [13, 10] and even making partial check-ins [14]. Similarly, partially motivated by the need to reduce conflicts, or at least avoid large conflicts, development teams have been adopting techniques such as trunk-based development [15, 16, 17] and feature toggles [18, 15, 19, 20], which are important to support actual continuous integration (CI) [21], but might lead to extra code complexity [22].

Although evidence in the literature is mostly limited to merge conflicts [2, 5, 23, 8] and other kinds of build errors [24, 25, 26], a couple of studies investigate the frequency of *build conflicts* [3, 4], that is, conflicts revealed after a successful merge, by failures when building the integrated (merged) code. As these two studies observe build conflicts in a small number of projects (three and four, respectively), it is important to observe build conflict *frequency* in a larger con-

text, as we do here analyzing more than 450 open-source projects. Furthermore, previous studies does not investigate further aspects related to build conflicts. In this study, we go further analyzing and classifying unexplored aspects such as conflict *structure* and adopted *resolution patterns*. Better understanding these aspects might help us to derive guidelines for avoiding build conflicts, improve awareness tools to better assess conflict risk, and develop new tools for automatically fixing conflicts.

So, to investigate the frequency, structure, and resolution patterns of build conflicts, we analyze merge scenarios (merge commits and the associated parent commits containing the integrated changes) from 451 GitHub open-source Java projects that use Travis CI— a popular service offering build process information through an open API [27, 28]— for continuous integration, or simply automating build creation and analysis. To collect build conflicts, we select merge scenarios from the git repositories and check through Travis services whether the merge commit build is broken (*errored* status).[1] To make sure the breakage is caused by the integrated changes, we parse the Travis logs generated when building the commits in a scenario, and automatically check whether the logged error messages are related to the merged changes; when our scripts are not able to check that, they confirm conflict occurrence by observing whether the parent commits present superior status (*failed* or *passed*) to the merge commit, and whether this commit contains only the integrated changes. For exceptional cases not supported by our scripts, we report conflicts detected by a manual analysis. Although we investigate conflicts using Travis CI, we focus only on conflicts caused by contribution changes on source code. In this study, we do not target eventual conflicts caused by the environment and configuration of CI services.

We find and classify 239 build conflicts, deriving a catalogue of build conflict causes expressed in terms of the structure of the changes that lead to the

[1] In Travis terminology, *failed* indicates the build and compilation phases of the build process were successfully performed, but at least one test failed. So *passed* is superior to *failed*, which is superior to *errored*.

conflicts. For conflicts with associated fixes, we analyze how developers fixed them, deriving a catalogue of common conflict resolution patterns. We enrich these catalogues by also finding and analyzing build failures caused by immediate post integration changes, which are often performed with the aim of fixing merge conflicts, but ended up creating build issues. Although we conservatively do not consider these as conflicts because the failure is not caused by the integrated changes, they are closely related. The frequencies in both catalogues show that most build breakages are caused by missing declarations removed or renamed by one developer, but referenced by the changes of another developer. Moreover, the conflicts caused by renaming are often resolved by updating the dangling reference, whereas the conflicts caused by deletion are often resolved by reintroducing declarations. To resolve build conflicts developers often fix the integrated code, instead of completely discarding changes and conservatively restoring project state to a previous commit. We have also observe that most fix commits are authored by one of the contributors involved in the merge scenario.

Based on the catalogue of build conflict causes derived from our study, awareness tools could alert developers about the risk of a number of conflict situations they currently do not support. Program repair tools could benefit from the derived catalogue of build conflict resolution patterns, and the study infrastructure, to automatically fix conflicts. We illustrate that with a proof of concept implementation that recommends fixes for build conflicts caused by a developer deleting the declaration of a variable that is referenced by the changes of another developer. The current implementation could have automatically fixed 21% of the build conflicts in our sample, if used by the respective development teams.

The rest of the paper is organized as follows: Section 2 motivates our study and introduces our research questions. Section 3 explains our study setup and the approach we use for detecting build conflicts. In Section 4, we present the results, which are further discussed in Section 5. Threats to the validity of our study are discussed in Section 6. Section 7 discusses related work. Finally, in Section 8, we present our conclusions.

2. Build Conflicts in Practice

2.1. Motivating Example

To illustrate a build conflict occurrence, consider an adapted example from the Quickml project.[2] Hypothetical developers Lucas and Rachel are assigned different, but related, tasks. They start working on their private repositories, which are updated with respect to the main project repository (illustrated in Figure 1). Assuming the latest commit in the repository is *C0*, Rachel finishes her work creating commit *C1*. At this point, she successfully builds and tests her version of the project (build process), and immediately sends her contribution to the main repository.

Later, Lucas finishes his work creating commit *C2*. He also makes sure that his changes do not break the build, successfully building and testing the project at state *C2*. But before sending his contribution to the main repository, Lucas notices Rachel's updates in *C1*. By quickly inspecting that, he is relieved because Rachel changed a disjoint set of files and, consequently, he will not need to fix merge conflicts. He then rushes to send his contribution to the main repository, creating a merge commit *C3* upstream. Before starting his new task, Lucas updates his private repository, checks the new commit *C3*, and decides to run the system to have a look at the new functionalities added by Rachel. He is then worried to see error messages after trying to build the project, and realizes that the project main repository is in an inconsistent state.

By talking to Rachel and confirming that the build process failure (*errored status*) observed for *C3* was not directly inherited from a defect in *C1* or *C2*—builds were fine for both commits—, the developers suspect the changes from one of them unintendedly interfere [7, 29] with the changes of the other. Trying to confirm the interference, Lucas checks the broken build log and observes that a method call for `ignoreAttributeAtNodeProbability`, in the `StaticBuilders` class, is the source of a compilation error because its declaration is missing

[2]Build ID sanity/quickml/53571613, Merge Commit 62a2190.

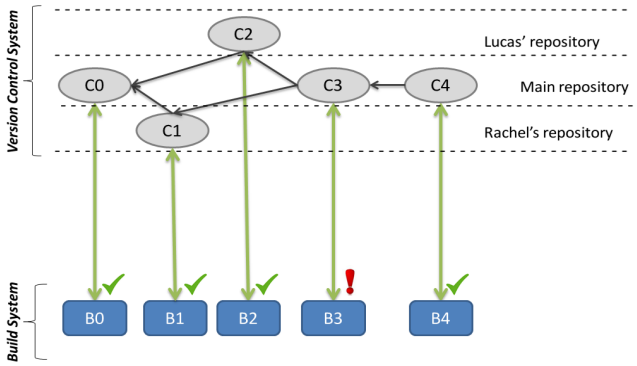

Figure 1: Build conflicts in practice. Black arrows link commits (gray ellipses) to their ancestors. Commits in between dashed lines are in the same repository. Green arrows associate commits with their builds (blue boxes). Above each build, we present its status (✓ and **!**, respectively for successful and errored builds).

.

in the `TreeBuilder` class. Investigating the `TreeBuilder` class, he confirms the method is not declared. However, he is sure this method declaration was available when he was working on his task and added the now problematic method call. Consulting the project history, Lucas notices that Rachel, unaware of Lucas' task, renamed the `ignoreAttributeAtNodeProbability` method in *C1*. He then realizes that the changes interfere, causing a *build conflict*, that is, a build breakage in a merge commit, caused by interaction among integrated changes.

To fix this build conflict, Lucas changes the method call using the new method name `attributeIgnoringStrategy`, and now successfully creates an executable build restoring the main repository consistency by sending a conflict fix (commit *C4*). A more experienced developer would maybe not have rushed as Lucas did, first pulling Rachel changes to her private repository, merging them and making sure she can successfully build and test the integrated code. This would have avoided leaving the main repository in an inconsistent state, but would not avoid the conflict nor the effort to resolve it. Similarly, a more prudent team would maybe have a continuous integration service running on

the main repository, and adopt a pull-request based contribution model. This would avoid the inconsistent state, and help to earlier detect the conflict, but would not necessarily avoid it.

Our motivating example presents a specific context in which build conflicts may occur. However, there are other some possible context and associated consequences, in practice. For example, conflicts may affect productivity and limit the access to resources shared by the development team. Consider a build process that takes a long time to complete, and the build breaks caused because of a build conflict. Despite the time spent by the developer for finding and fixing the conflict, the failed build process held resources that might be relocated for other build processes. In our motivating example, we present a build conflict caused by a merge scenario remotely performed, when the developers have the required permissions to directly update the main repository. However, a build conflict may also happen when the development team does not own these permissions. For example, during the acceptance of a *pull request* (PR) on GitHub, the current repository state may be conflicting with the changes present in the PR. In case the repository adopts CI services like Travis CI, the broken build would be reported, and eventually, the conflict might be detected. Otherwise, the PR is accepted polluting the main repository. As a result, the mentioned breakage state would be observed when a developer locally updates her private repository and builds the project. In this way, the developer would locally apply changes to fix the conflict and push them to the remote repository. If these activities of detecting and fixing conflicts frequently happen, resolving them end up being a tedious and error-prone activity [30].

Although our motivating example has been simplified for space reasons, it illustrates the kind of conflict we consider in our study, and how they might impair team productivity. However, our conflict classification is programming language dependent. Whereas our discussion makes sense for a Java project, in a Ruby project, for example, the illustrated conflict would not be revealed during build time. Due to Ruby dynamic features, there is no pre-execution check about the existence of method declarations. The conflict would be revealed

only during testing or system execution in production.

2.2. Research Questions

Considering the just discussed negative consequences of build conflicts, it is important to study how often they occur in practice. This partially motivated a couple of studies that investigate the frequency of build conflicts [3, 4]. [4] report that build conflict rates substantially vary across projects, ranging from 2% to 15%. [3] find a similar range: 1% to 10%. This is not, however, the main focus of the two mentioned papers; the authors observe build conflicts in a small number of projects— three in one study and four in the other. It is then important to observe conflict frequency in a larger context. In this study we consider a substantially larger sample, with 451 projects and 20 times more merge scenarios (57065) than the aggregated sample of the two studies, to answer the following research question.

RQ1 – *Frequency:* How frequently do build conflicts occur?

To identify build conflict occurrences, we look for merge scenarios with broken merge commit builds (*errored* build status) and an error message not related to Travis environment issues (like timeout of the build process). To make sure the *errored* status was caused by the integrated changes, and is actually a conflict, we further parse the Travis logs generated when building the commits in a scenario, and automatically check whether the logged error messages are related to the changes. If our scripts are not able to check that, due to the limited set of patterns they recognise, they confirm conflict occurrence by observing whether the merge commit parents both present superior status (*failed* or *passed*), and the merge commit contains only the integrated changes. We conservatively do not classify the breakage as a conflict if it is caused by post integration changes, which are often applied to fix merge conflicts but could also cause build breakage.

Although useful as initial evidence of build conflict occurrence, previous work [3, 4] does not bring information about conflict causes. They do not

investigate the *structure* of changes that cause build conflicts. We go further investigating and classifying these causes. Seo et al. [31] present technical causes for broken builds in general, but do not study build conflicts in particular. They mostly explore individual developers changes that break builds. They do not study individual changes that do not break builds, but that interact in unexpected ways and lead to broken builds when integrated. Understanding the structure of parallel changes that lead to build conflicts might help us to derive guidelines for avoiding such conflicts, and to improve awareness tools to better assess conflict risk. Hence, our second research question, which has not been explored before, is the following.

RQ2 – *Causes:* What are the structures of the changes that cause build conflicts?

Based on the results of RQ1, we further analyze build logs and the source code of merge scenarios, classifying and quantifying the kinds of changes we observe. As a result, we derive a catalogue of build conflicts causes, and identify the most common causes.

Identifying conflict causes might help developers to reduce conflict numbers, but most likely will not eliminate them [32]. Fixing merge conflicts might be a demanding and tedious task [11, 2]. Fixing build conflicts might be even harder and risky, since semantic aspects must be taken into account. Build conflicts may have different causes requiring different treatment to detect and also deal with them. As a result, developers should spend more time fixing these conflicts, negatively impacting team productivity and software quality as fixing conflicts is an error-prone activity. So, to reduce build conflict resolution effort, it is important to better understand which resolution patterns are adopted, and maybe automate some of them. It is also important to understand whether conflicts are fixed by both contributors and integrators, especially because integrators might have a harder time fixing them. Thus, our third research question, and a complementary related question, are the following:

RQ3 – *Resolution Patterns:* Which resolution patterns are adopted to fix build conflicts?

RQ3.1 - *Fixer:* Who does fix build conflicts?

To answer these questions, which have also not been explored before by previous work [3, 4], our scripts perform an automated analysis between the broken merge scenario and the closest commit that fixes the conflict. As a result, we derive a catalogue of build conflict resolution patterns, and identify the most common patterns.

3. Study Setup

To answer the just discussed research questions, we structured our experiment in three main steps, as represented in Figure 2. For automating this process, we implemented scripts that perform the analyses required for the experiment. Although most of this process is automatically done using these scripts, we perform manual analysis to detect conflicts for exceptional cases not supported by our scripts due to limitations. We explain later in this section when this manual analysis is performed. In the first step, our scripts mined project repositories in GitHub and Travis to get the information required for the experiment, including the source code and build status of commits that are part of merge scenarios. As explained in Section 2.2, we collected only merge scenarios with broken merge commit builds (*errored* build status). In the second step, our scripts parsed the Travis build logs of the collected scenarios to filter out scenarios with build breakages caused by non integration related causes such as execution timeout of the Travis services. Finally, in the last step, our scripts confirmed conflict occurrence, and classified conflicts and the adopted resolution patterns. The scripts we used to automate these steps are available online [33] to support replications.

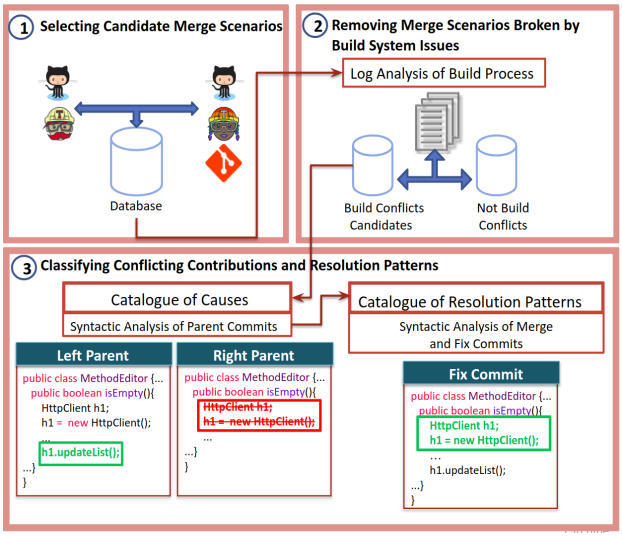

Figure 2: Study setup composed of three steps. The first step (Section 3.1) mines project repositories. Step two (Section 3.2) filters merge scenarios and yield build conflict candidates. Finally, step three (Section 3.3) classifies build conflict causes and resolution patterns.

3.1. Selecting Candidate Merge Scenarios

To explain in more detail how we select candidate merge scenarios for our experiment, we first discuss our project sample. Then we explain the criteria we use to select scenarios from our project sample.

Project Sample

As our experiment relies both on the analysis of source code and build status information, for service popularity reasons we opt for GitHub projects that use Travis CI for continuous integration. As a significant part of our automated analysis is programming language dependent, we consider only Java projects. Similarly, as Travis build log parsing depends on the underlying build automation infrastructure, we analyze only Maven [34] projects; its log reports are very informative compared to the reports of other dependency managers. Considering more languages, build systems, and CI services would demand more implementation effort.

We start with the projects in [35] and [36] datasets, which include a large number of carefully selected open-source projects that adopt continuous integration. We then select Java projects that satisfy the following criteria: (i) presence of the *.travis.yml* file in the root directory of the latest revision[3] of the project (this indicates that the project is configured to use the Travis service); (ii) presence of at least a build process in the Travis service, and confirmation of its active status, which indicates that the project has actually used the service; (iii) presence of the *pom.xml* file in the root directory of the latest revision of the project (this indicates use of Maven), and absence of Gradle [37] configuration files, which could represent early use of Maven, but later migration to Gradle, demanding a different build log parser; and (iv) the project should have at least one merge scenario considering the history interval we were analyzing. This way we ensure only Java projects that use Maven and Travis were selected [27, 28]. This results in a sample of 451 projects.

Although we have not systematically targeted representativeness or even diversity [38], we believe that our sample has a considerable degree of diversity concerning different dimensions like domain, size, and number of collaborators. Our sample has projects from different domains, such as APIs, platforms, and Network protocols varying in size and number of developers. The Truth project has approximately 31.2 KLOC, while Jackson Databind has more than 113 KLOC. Moreover, the Web Magic project has 45 collaborators, while OkHttp has 195. The complete list of the analyzed projects can be found online [33].

Candidate Merge Scenarios

For each selected project, our scripts locally clone the project and select[4] merge commits created after the project adopted Travis CI, that is, the first build appeared in Travis. Since Travis CI is a relatively recent technology, most projects adopted it later in their history. Our filter then makes sure we select

[3] As in August 2018.

[4] We use the `git log --merges` command with an extra parameter specifying the initial date to drive the search.

Table 1: Summary of Merge Scenarios with Build Conflicts

Number of projects	451
Number of Merge Scenarios (MS)	57065
Errored MS	4252
MS with Build Conflicts	65
Number of Build Conflicts	239

merge scenarios that are Travis ready. We also avoid selecting *fast-forward* situations [39, p. 69], as they do not correspond to a code integration scenario. Applying this filter to our project sample, we obtained 57065 merge commits. By collecting each merge commit with its parents (the associated changes it integrates), we obtain a sample with the same number of merge scenarios (second row in Table 1).

For each merge scenario in our sample, we try to identify the Travis builds associated with the merge commit and its two parents. As not all commits have an associated Travis build, we force the creation of builds as needed: our scripts use GitHub's API[5] to fork the corresponding project, and then reset the fork head to the unbuilt commits, triggering Travis service to start the build process. With builds for the commits in a merge scenario, we filter our sample by selecting only the scenarios with broken merge commit builds, that is, scenarios having merge commits with *errored* build status (4252, third row in Table 1). To avoid bias, we also discard duplicate merge scenarios, which involve different merge commit hashes but have the same parents.

3.2. Removing Merge Scenarios Broken by Build System Issues

To ensure the merge scenarios selected so far are associated with conflicts, and to better understand what caused the *errored* status, we further investigate the build status and associated logs of the merge commits. This is needed because the broken build status might have been caused by a number of reasons

[5] `https://developer.github.com/v3/`

not related to the merged contributions. In particular, the breakage might have been caused by build system issues such as execution timeout of the Travis services, or unresolvable dependencies no longer supported by Travis. Another example of broken builds are caused by changes performed in the environment or build script files, not on source code changed by both contributions in a merge scenario. As previously motivated, in our study, we only consider as build conflicts those cases caused by contributions performed on source code files. Eventual conflicts involving no source code files are not targeted in our study. As previously discussed (see Section 2.2), previous studies investigate build conflicts, but they do not explore aspects like the causes and resolution patterns adopted, as we do here. So we further analyzed, for each merge commit build, its Travis log report.

Our scripts parse each log and search for the specific error messages listed in the third column of Table 2. This list was incrementally derived by observing the error messages of broken merge builds in our sample. Initially, whenever our script could not parse a log, we would extend it to consider the new kind of message that appeared in that log, and run it all over again. As a result, we have a list of the most common error messages in our sample, being able to classify most (99%) broken builds we found. Our scripts do this analysis based on matchings between the expected and the observed string in the build file logs; if our scripts fail to perform these checks due to limitations, we miss these conflicts so that we may have false negatives. However, our scripts present a high precision when selecting or removing merge scenarios during this analysis. Thus we believe our scripts would erroneously remove only a small number of eventual conflicts in this analysis. The third column of the Table 2 highlights how we group such error messages into conflict causes expressed in terms of the structure of the changes that lead to the conflicts. These causes are then grouped in the categories that appear in column two.

To filter out scenarios with build breakages caused by build system related causes, we discarded scenarios with broken merge commits whose logs have one of the error messages listed in rows *Environment Resource* and *Dependency* of

Table 2: Build error messages related to broken builds during the build process on Travis.

Error Category	**Error**	**Build Error Message**
Static Semantic	Unimplemented Method	does not override method
	Duplicated Declaration	is already defined in
	Unavailable Symbol	cannot find class
		cannot find method
		cannot find variable
	Incompatible Method Signature	no suitable method found for
		cannot be aplied to
	Incompatible Types	error: incompatible types
		cannot be converted
Other Static Analysis	Project Rules	some files do not have the expected license header
Environment Resource	Remote problems	The job exceeded...
		No output received...
		Your test run exceeded
Dependency	Environment configuration	could not (find OR transfer) artifact

Table 2. For these cases, we have no evidence that the corresponding build breakages are caused by the integrated changes, or are related to build conflicts.

3.3. Classifying Conflicting Contributions and Resolution Patterns

As presented in Table 2, *errored* builds might have many causes, and these are associated to specific error messages. However, even after discarding breakages caused by build system issues, occurrences of these messages do not imply conflict occurrences since the error might not have been caused by the integrated changes. For example, the breakage might also have been inherited from one of the parents, instead of having been caused by conflicting contributions. So, we now discuss the extra checks needed to confirm conflict occurrence and classify conflicts according to the mentioned causes. Additional information about this process can be found online [33].

Checking Build Conflicts

To make sure an *errored* merge commit build in one of the selected scenarios corresponds to a build conflict, we have to check whether the build breakage was caused by the integrated changes. So, we parse the build log and check whether the logged error messages are related to the changes integrated in the merge commit. To capture the integrated changes, we use GumTree [40] to compute syntactic *diffs* among the *merge commit*, its *parents*, and the associated *base*[6] commit. When our scripts, due to the limited set of patterns they recognize, are not able to check that error messages are related to the changes, they confirm conflict occurrence by observing whether the merge commit parents present superior status (*failed* or *passed*), and the merge commit contains only the integrated changes. For that, our scripts locally replicate the merge scenario, merging the parent contributions into a new commit. They then check whether this new replicated commit has any difference when compared with the original merge commit. If no difference is observed, we assume the broken build is caused by the parent contributions revealing a build conflict as the parents present no *errored* build status.

For confirming the *Unavailable Symbol* conflict, for example, we initially look for declarations removed or renamed by the changes of one developer, but referenced by the changes of the other developer.[7] So, after confirming the presence of a "cannot find..." error message (see Table 2, third column, 4-6th rows) in the log, our scripts further parse the log to extract the following information: *the missing symbol* and the *involved classes* (one references the symbol, the other should have declared the symbol). With this information, to confirm the conflict, our scripts check whether one of the parents remove the declaration of the *missing symbol* from one of the *involved classes*, while the other parent

[6]The latest common ancestor to the parent commits.

[7]Conflicts might even happen when integrating changes a single developer made to different branches or repositories. For simplicity, in the explanation we consider just the most common case of conflicts caused by integrating different developers changes.

adds a reference to the *missing symbol* in the other *involved class*. These checks confirm the parent contributions are responsible for the error reported by the broken build. The scripts also consider the case when the removed declaration and added reference are in the same class. The scripts used to perform the analysis for all build conflict causes are based on matchings expected and observed strings in the GumTree logs associated with the conflicting files. These logs are generated based on the parent commits (holding the changes performed by each parent during their contributions) and the base commit (the common ancestor). In case our scripts check the expected matching, a build conflict is detected.

Under the presence of the "cannot find..." error message (see Table 2, third column, 4-6th rows), or in pathological cases of build logs that do not conform to the common expected format parsed by our scripts, we manually analyze the merge scenario to confirm conflict occurrences. We check whether the integrated changes either removed and added references as just explained, or removed a build dependency that declares the missing symbol, causing a build dependency issue. In both cases, we rely on the file name information in the build log. We opt for not automating these cases because they rarely occur in our sample. We actually adopt an on demand approach for evolving the scripts, with a few cycles of first extending script functionality as needed, followed by rerunning the experiment, and then identifying and analyzing common cases not captured by the scripts. Build conflicts associated with *Incompatible Types* and *Project Rules* are manually checked. One single author performs this manual analysis. Although the scripts could not detect the conflict in these cases, they inform the classes the author should look at to mitigate errors or mistakes during this analysis. We discuss the threats related to this manual analysis in Section 6.

In case the logged error messages are not related to the changes integrated in the merge commit, we further investigate the merge scenario. We confirm that such build failures are caused by immediate post integration changes, often performed with the aim of fixing merge conflicts. In this case, the integrator

changes, not the contributors changes, cause the problem.[8] So we do not consider that a build conflict, but a broken build caused by changes applied to fix a merge conflict, or amend a merge commit for other reasons. Nevertheless, as these cases might also benefit from a number of applications of our results for conflicts, our scripts also analyze them. For confirming *Unavailable Symbol* breakages caused by post integration changes, our scripts check whether both parents changes keep the *missing symbol* declaration. This implies that the integrator removed or renamed the symbol declaration. In case we cannot check this, our scripts apply a similar approach used to identify build conflicts. So, our scripts check again whether the merge commit parents present a superior status (*failed* or *passed*), and if the merge commit does not contain only the integrated changes. We check this replicating the merge scenario again, and comparing the new merge commit with the original merge commit.

We follow a similar approach for confirming the other conflict causes. For brevity, they appear only online [33].

Build Conflict Fixes

For each build conflict identified in the previous steps, we analyze the adopted conflict resolution pattern and who (integrator or contributor) was responsible for applying the fix. So we first look for fix commits. For a merge commit with a build conflict, and consequently, a broken build, the fix commit is the first commit that follows the merge commit and has a superior build status. For build conflicts, we consider *failed* and *passed* as superior status for a fix; both status indicates that the source code could at least be compiled.

Our scripts automatically analyze the fix commits extracting common resolution patterns that appear in the third column of Table 4. Similarly to the scripts that identify conflict causes (see Section 3.3), we adopted an on demand and relevance based approach for evolving the scripts that identify conflict reso-

[8]The same developer might be playing both roles in the same merge scenario, but that's not necessarily the case.

lution patterns. The scripts used to detect the resolution patterns also follow the same approach adopted for the detection of build conflict causes. The scripts work based on matching expected and observed strings in the GumTree logs associated with the broken merge and fix commits' files. In case our scripts perform this matching of strings, a resolution pattern is detected. Uncommon and harder to automate cases of build conflict resolution patterns are manually handled. Based on the conflict type and files involved, we analyze the fix commits looking for changes in those files. Similarly to the automated analysis, we use a syntactic diff between the merge and fix commit getting all syntactic differences. The list of resolution patterns in Table 4 covers all fixes we were able to identify in our sample.

4. Results

Following the empirical study design presented in the previous section, we analyze merge scenarios in 451 GitHub Java projects to investigate the frequency, causes, structure, and resolution patterns adopted for build conflicts. This section details our results answering our research questions. The first question, as explained in the previous section, has been explored before, but in a significantly more restricted scope. The other questions are originally explored here.

4.1. RQ1: How frequently do build conflicts occur?

To answer RQ1, we follow the steps in Section 3, select merge scenarios in our sample, discard non conflicting scenarios, and count conflict numbers in the remaining scenarios.[9] We then find 239 build conflicts in 65 scenarios (see Table 1, fifth row). These conflicts were automatically (174 cases) and manually (65 cases) identified.

[9] Each scenario might have a number of conflicts caused by different changes and associated to different error messages in the build log of the scenario merge commit.

To better contextualize that, we observe in Table 1 that roughly 8% of the merge commit builds are broken in our sample. This maybe surprisingly high rate of broken merge builds is due to a number of causes: build environment (Travis timeouts, unavailability of external services such as package manager servers, bugs in bug scripts, and configuration problems) issues; integration conflicts; defects in post integration changes; or defects, and consequently breakages, inherited from the parents. Most breakages are due to the first cause.

Part of the build conflicts, 51%, have parents with superior build status; the build breakage just arises after the merge scenario integration; no error is inherited from the parent commits. The remaining cases, besides build conflicts, may also present additional errors not caused by conflicting contributions but inherited from their parent commits. Other non inherited breakages occur when a merge commit and at least one of its parents have builds with environment issues, but these are less interesting for our discussion.

We believe the low numbers and frequencies of build conflicts in our sample are highly influenced by the use of Maven and continuous integration practices and services in the projects we analyze. With automated build and testing processes in these projects, developers can easily build their contributions before sending them to the main repository [21]. They are often, by project guidelines, required to locally integrate their contributions into the main repository contributions before submission for approval or final integration. This way, we assume most build conflicts are actually detected and resolved locally, in contributors private repositories. For example, Accioly et al. [8] observe that when applying improved merge tools, merge conflicts are reported involving duplicated declaration methods. So traditional tools would merge the code without reporting a merge conflict, but only during the build process, the build conflict would arise. These findings bring evidence that developers locally face build conflicts, but solve them before sending their contributions to the remote repository.

As our study analyses only public repositories, we do not have access to problematic code integration scenarios that were locally amended before reaching the main repository. Consequently, ours numbers reflect the number of

build conflicts that reached public repositories, not the actual number of conflicts that happened and had to be resolved. This justifies the high frequency of merge scenarios with successful build process, and also of non integration related breakages. As these two aspects widely vary across the analyzed projects, we miss existing integration conflicts.

In the end, 37 project of our sample present build conflicts. As presented in Section 3, our analysis considers only the main development line. So we are not able to identify conflicts that occur on local developer workspaces. Despite the low frequency, when compared to related work [4, 3], this frequency is 9 and 12 times, respectively, bigger. Furthermore, these projects cover different domains showing that build conflicts are not restricted to a specific domain (see Section 3.1).

Our results also show that different conflict causes may coexist in a merge scenario. While previous studies consider the build process breakage as one single conflict occurrence, we go further and detail the different causes as the conflict causes occur independently (see Section 3). Analyzing the distribution of conflicts based on merge scenarios, we identify 24 out of 65 merge scenarios present more than one single conflict. Besides the effort and time spent to fix these conflicts, different conflict causes also require different approaches to handle them.

> Our results indicate that build conflicts occur during software development, but many conflicts do not reach the remote repositories as developers fix them before sending their contributions. We also observe that more than one build conflict may occur in a merge scenario as the causes for these conflicts are independent.

4.2. RQ2: What are the structures of the changes that cause build conflicts?

Based on the conflict occurrence results, we proceed with further analysis to answer RQ2 by investigating conflict causes. As a result, we observe six build conflict causes used to define our catalog of build conflicts. Table 3 shows the

Table 3: Catalogue of build conflicts

Build Conflict Category	**Cause**	#
Static Semantic	Unimplemented Method (method from super type or interface)	12
	Duplicated Declaration (elements with the same identifier)	5
	Unavailable Symbol (reference for a missing symbol)	157
	Incompatible Method Signature (unmatched method reference)	26
	Incompatible Types (type mismatch between expected and received parameters)	17
Other Static Analysis	Project Rules (unfollowed project guidelines)	22
Total		**239**

causes, their descriptions and frequencies are shown grouped by cause categories (second and third columns).

Most build conflicts are caused by Unavailable Symbol

We find that 65% of all build conflicts are caused by a reference for a missing declaration (third column, fourth row in Table 3). The most recurrent missing symbols are class names (112 cases), corresponding to 73% of all *Unavailable Symbol* cases. *Missing method* names (22) and variable or parameter names (20) come next, with the missing declaration and the dangling reference possibly associated to the same class. The other six causes occur less frequently (each in less than 35% of the cases).

Concerning the distribution of *Unavailable Symbol* cause, we observe 39 out of 65 merge scenarios present this conflict cause. Looking at its distribution on our project sample, we observe these 39 scenarios belong to 27 out of 37 projects with conflicts. These numbers show how recurrent conflicts of this type occur and reinforces the need for an approach to deal with them. Especially because the effort to fix each conflict directly depends on the type of the missing element, and each type requires specific attention; we explain it in detail in Section 5.3.

Unplanned dependencies also cause build conflicts.

We have observed a number of *Unimplemented Method* conflicts, which often occur when one developer adds a method to an interface while another developer adds, to an existing class, an implements clause referencing the same interface. The build then breaks because the existing class does not declare the method introduced to the interface; the class developer is not aware of such method, even though there is a direct dependence between the class and the interface. So the changes introduce an *unplanned direct dependency* causing the conflict. But we have also observed similar problems when developers change files or program elements that are not directly related.

For example, in the Ontop project,[10] one developer adds the new interface `Var2VarSubstitution`, which extends another interface (`Substitution`), and its implementing class `Var2VarSubstitutionImpl`. The other parent, unaware of the previous changes, adds the new method `composeFunctions` to the `Substitution` interface. Once all contributions are integrated, the build process breaks as the `Var2VarSubstitutionImpl` class does not implement the method added to the interface (`composeFunctions`).

Another example occurs in the project PAC4J.[11] While one developer adds the class `DigestAuthExtractor` implementing interface `Extractor`, the other developer changes the location of the interface class. Besides the expected build conflict occurrence, caused by the dangling reference to interface `Extractor` (*Unavailable Symbol*), the `override` anottations in `DigestAuthExtractor` class also causes build conflicts. In this context, there is no super class or interface associated with `DigestAuthExtractor`. So it is not possible to override a method. Different from the previous example, when the missing method implementation causes the conflict, here the missing method declaration in the interface or super class is the reason for the conflict occurrence.

In the same way, a related case happens in the Ontop project.[12] The build conflict occurs due to the class `MonetDBSQLDialectAdapter`, which presents a method implementation for `strconcat` that was not defined in its super class. So the anotation `Override` above the method declaration could not be resolved. This case occurs because Left updates the method name from `strconcat` to `strConcat`, while Right adds the class `MonetDBSQLDialectAdapter`.

Build conflicts are caused by copy/paste actions involving different branches

The reported build conflicts in this category are caused by the addition of methods with the same signature in the same class, or the same variable added in the same method (third column, third row in Table 3). For duplicated meth-

[10]Build ID: ontop/ontop/59371438 – Merge Commit: c626206
[11]Build ID: pac4j/pac4j/291027337 – Merge Commit: e18dd85
[12]Build ID: ontop/ontop/83689234 – Merge Commit: 0f62121

ods, each parent commit adds its method declaration in a class resulting in one single declaration, but after the merge scenario, the class presents two methods with the same signature. Analyzing these reported cases, we observe, in all occurrences, the duplicated methods have the same implementation (method body). For example, in the Blueprints project, the same method is added in the class `GraphTestSuite`, `testRemoveNonExistentVertexCausesException`.[13] It may happen when, for example, a developer is working in two different branches and decides to use changes previously done in one branch into the other. Instead of integrating her work with the commit holding the desired method, the developer decided to copy/paste it. Accioly et al. [8] also observe this behavior in their study. They bring evidence these operations are done in practice and the local occurrence of build conflicts experienced by developers.

Build conflicts also involve unmatching operations

During her contribution, when a developer adds a new reference for a specific method, it is expected this reference may be resolvable by matching the call reference for the method declaration in the class supposed to hold it. However, when these matchings are not resolvable, build conflicts may happen caused by *Incompatible Method Signature* (third column, fifth row in Table 3). For example, in the Spark project, one parent adds a new method call in the class `MatcherFilter` to the method `modify` of class `GeneralError`.[14] The other parent, unaware of the changes previously done, changes the signature of method `modify` adding a new parameter. It also updates previous method calls using the old signature to the new signature. However, when the parent commits are integrated, the first parent's new method call is not resolvable as the method signature has changed. Similar to *Unavailable Symbol* conflicts, this cause involves a method reference that could not be resolved, but the method is not removed or renamed. This cause requires more attention by the developer as

[13] Build ID: tinkerpop/blueprints/267833702 – Merge Commit: 5a25e3a
[14] Build ID: perwendel/spark/229805514 – Merge Commit: 3fd18a9

there is a slightly different signature that may confuse the developer during analysis. In case this initial analysis is done locally, the developer may be supported by IDEs, which properly indicates the mentioned problem using type checking. If this analysis is remotelly done, the developer may spend more time understanding the problem.

In the same way, consider the occurrence of build conflicts caused by *Incompatible Types* as unbound matching between an expected and an observed type (third column, sixth row in Table 3). For example, in project Elasticsearch-SQL, one parent adds a new local variable `genderKey`, which is initialized by the method call `getKey` of an external class `Bucket` returning a `String`.[15] The other parent updates the version of the jar used in the project, that holds the class `Bucket`. Now the method `getKey` returns no longer a `String` but an `Object`. When the contributions are integrated, the variable `genderKey` can not be initialized with an `Object` type as its type is `String`. This case requires more attention as it is expected the developer to initially explore the classes involved and reported in the broken build log. When no valuable information is identified, the developer might explore the history changes performed by the parent commits and then verifies the change of an old dependency for its new version.

Build conflicts are also caused by non compilation problems

Although most build conflicts are related to programming language static semantic problems (third column, 2-6th rows in Table 3), we find conflicts due to failures during the execution of other static analysis tools (third column, seventh row in Table 3). These failures appear during the so called ASAT phase of the Travis build process.[16] The integrated source code is compilable, but the static analysis presents errors, like name and style conventions adopted by the project, breaking the build process. For instance, in the project HDIV,[17]

[15]Build ID: NLPchina/elasticsearch-sql/99402562 – Merge Commit: eb5fe5f
[16]Travis Documentation: The Build Lifecycle
[17]Build ID: hdiv/hdiv/126140680 – Merge Commit: c620070

one developer updates the license header file, while the other developer adds two new classes (`HTTPSessionCache` and `SimpleCacheKey`). As the new added classes have the old header style, the verifications identify inconsistencies caused by build conflicts.

> Our catalog of build conflicts groups six causes. The most recurrent cause is *Unavailable Symbol*, which can be split into sub-categories, like *Unavailable Symbol Class, Method, and Variable*. These build conflicts are caused not only by static semantic problems but also static analysis performed after the compilation phase during the build process.

4.3. RQ3: Which resolution patterns are adopted to fix build conflicts?

To answer this question, we first tried to identify commit fixes for all observed conflicts. For a number of conflicts, we were not successful for the following reasons. First, the conflict was not fixed because was potentially hard to resolve; as it occurred in an auxiliary branch, developers ignored the changes in the branch and moved on in the master branch. Second, the fix appears only after our limiting date for analyzing project history, either because the conflict occurred not long before this date, or because the project receives only sporadic contributions (or is no longer active). Third, as our scripts try to build fix commits not built yet, it is possible that environment problems occur during this attempt, breaking the build process. So in these cases, we are not able to assess the fix commit because the build process would continue as *errored* status. Finally, as explained before (see Section 3.1), our scripts discard duplicate merge scenarios; the fix could be associated with the scenario we discarded. So we analyzed fixes for just part of the conflicts: 148 fixes out of all build conflicts. Only part of these conflict fixes are automatically analyzed by our scripts (44), while the remaining cases (104) were manually analyzed. The identified resolution patterns, together with their frequencies, appear in the fourth column of Table 4.

Table 4: Catalogue of resolution patterns adopted to fix build conflicts.

Conflict Category	**Cause**	**Resolution Patterns**	#
Static Semantic	Unimplemented Method	Super type class addition	5
		Method implementation	5
	Duplicated Declaration	Duplicated element removal	2
	Unavailable Symbol Class	Missing class import update	47
		Missing class reference removal	27
		Missing class reference update	4
	Unavailable Symbol Method	Missing method reference update	7
		Missing method reference removal	6
		Associated class import update	3
	Unavailable Symbol Variable	Associated class import update	12
		Missing variable reference update	4
		Missing variable reference removal	2
	Incompatible Method Signature	Required method reference update	3
		Required method reference removal	2
		JDK setup	7
	Incompatible Types	Type update	9
Other Static Analysis	Project Rules	License header update/removal	3
Total			**148**

Developers often fix the integrated code preserving parent contributions

We observe that build conflicts are fixed using 17 resolution patterns. Most of these fixes are done aiming to integrate the code involved in the conflict, instead of completely discarding changes and conservatively restoring project state to a previous commit. This practice is adopted in 73% of all analyzed conflict fixes. Build conflicts caused by *Unavailable Symbols* are fixed by removing or updating the dangling reference; not only the direct reference for the missing *unavailable symbol* itself but also the reference for the class, where the *missing element* should be available. However, roughly 62% of the fixes are done by updating the dangling reference. The most recurrent resolution pattern adopted is related to the *Update of the class import* holding the element that caused the conflict. The build conflict may be caused by a missing reference for a field, a method, or even a direct reference for the class itself. For example, in project Jackson-Core, a build conflict happens due to the location change of class `JsonFactory`. While one parent adds a new reference for this class and its import in class `AsyncTokenFilterTest`, the other parent changes the location of this class. After the integration, the import could not be resolved, and the build conflict is detected. Updating the import declaration of the missing class is enough to fix the conflict. Although this resolution pattern is straightforward, redoing the same update operations repeatedly is a tedious activity that could be automatically done.

In other cases, when the conflict is not caused by a change on classes' location, the adopted resolution pattern follows the same change structure. For example, *Unavailable Symbol* conflicts are often resolved by updating the dangling reference. In the Java Driver project, for example, the reference for the missing symbol `builderWithHighestTrackableLatencyMillis` in the request of the `Activator` class was updated to the new method signature `builder`.[18] For the cases that discard the changes instead of adjusting them, we may comment one scenario of the OkHttp project. So the reference to the *missing symbol*

[18] Build ID datastax/java-driver/144856728, Fix Commit 7cec37a.

`deadline` (local variable), in the `GzipSource` class, was just removed.[19]

The other conflicts are also fixed in the same way aiming to preserve all parent contributions, except for build conflicts caused by *Duplicated Declaration*, whose cases are fixed by removing the duplicated method. In the Blueprints project, as previously discussed, in the `GraphTestSuite` class, both contributors added the same method `testRemoveNonExistentVertexCausesException`.[20] The integrator fixes the conflict by removing the *duplicated* test case. In these cases, the duplicated methods share the same body, so no behavior change on the program would be observed after removing one of the duplicated methods.

Conflicts caused by *Incompatible Types* and *Incompatible Method Signature* are fixed by adopting straightforward actions. For example, in project Elasticsearch-SQL, as previously discussed (see Section 4.2), the build conflict is fixed by adjusting the return of method `getKey` for `String.` It is done by adding a call to the method `toString`, which now returns a `String` as expected by the parent contribution changes.[21] For the build conflict observed in project Spark, as previously discussed, the *Incompatible Method Signature* is fixed by adding the new required parameter in the call for the method `modify`.[22] In both cases, the project already has source code showing how the developer could fix the conflict. This source code is available in both scenarios as the parents responsible for adding the unexpected changes also updated the old source code impacted by his changes. They perform these changes before integrating their contributions in the merge scenarios.

In other cases, adopted resolution patterns are not made in source code but configuration files. For example, in project Vavr, the conflict is fixed by forcing the installation of a specific JDK version when building the project (changes on `travis.yml` file).[23] As we previously discuss (see Section 4.2), in this work,

[19]Build ID square/okhttp/19404300, Fix Commit 59b8adc.
[20]Build ID tinkerpop/blueprints/10120795, Fix Commit ae8a2cd.
[21]Build ID NLPchina/elasticsearch-sql/99402657, Fix Commit 0163335.
[22]Build ID perwendel/spark/403188038, Fix Commit 5f1632d.
[23]Build ID vavr-io/vavr/58958884, Fix Commit 63f5fc3.

we do not focus on conflicts that are caused by configuration files. However, this is a particular case caused by changes in source code that requires specific environment configurations to be settled before building the project.

Most fixes are done by parent commit authors

Once the resolution patterns are identified, we investigate who was responsible for the fixes. Most fix commits are authored by one of the contributors involved in the merge scenario (137 out of 148 fixes follow such pattern).

> Our catalogue of resolution patterns is composed of 17 fixes adopted by developers to fix build conflicts. These patterns show that fixes do not only preserve all contributions from the merge scenario but also discard some of them. In most cases, these fixes are applied by one of the developers involved in the merge scenario.

Availability of data and material

Our catalogues of build conflicts and scripts we used to perform this study are available online [33]. These catalogues can be consulted and exported for further analysis. We encourage study replications once our scripts are available for the community. However, exact replications directly depend on data stored in external services (GitHub and Travis CI). Additional instructions and further information can also be found online [33].

5. Discussion

In this section, we discuss our findings and their implications, and how they can be applied to assistive software development tools.

5.1. Findings

Comparing our RQ1 results with previous studies, we conclude that, in our sample, build conflicts occur less frequently. Kasi and Sarma [4] show build conflict occurrence ranges from 2% to 15% of the analyzed merge scenarios across projects. Brun et al. [3] find a similar range: 1% to 10%. We report significantly

inferior numbers. However, our number should not be interpreted as conflict occurrence rates in general as studied in previous work, but as conflict occurrence that reach public repositories and, therefore, are more problematic. So, given the differences in sample size and characteristics, our conflict frequencies results actually complement previous results. This number discrepancy is further justified by limitations and bias in the mentioned studies, as later explored in Section 7. For example, to confirm conflict occurrence, previous studies consider only the build process status of merge commits, while we additionally confirm that the changes are related to the build error message, making sure build breakages actually correspond to conflicts.

In our study, we investigate build conflicts aspects not targeted by related work [4, 3]. While previous studies focus on the frequency of build conflicts, we go further analyzing and classifying the *structure* of changes that cause build conflicts and their *resolution patterns*. In the next Section, we discuss in detail the implication of these new contributions. In our study, *Unavailable Symbol* is the most frequent build conflict cause. Although Seo et al. [31] discuss build errors in general, not relating them to integration conflicts, they report a related finding: *Unavailable Symbol* is the most frequent cause of build error in their sample. This reveals the common mistake of removing or renaming declarations, but possibly not updating all references to the new identifiers. Assistive tools like Palantír [10] could be applied to anticipate the emerging conflicts, or even treat them directly. In this way, developers could be aware during the tasks development that a new reference for a symbol will fail during integration (in case of build conflicts due to contributor changes).

Build conflicts caused by *Duplicated Declaration* have also been observed as semistructured merge conflicts in previous studies [7, 8]. This supports our assumption that build conflicts frequently occur in private repositories, but not so much in public repositories of projects with automated build process and continuous integration.

We believe some build conflicts might be avoided if their associated conflicting contributions are not performed in parallel. For example, a better as-

signment of activities based on the chance of interference among contributions. Rocha et al. [41] propose a tool for predicting the files that could be changed during a task. If two tasks are predicted to change the same set of files, these tasks should be assigned for developers at a different time, not in parallel. As a result, potential merge conflicts (textual) would be avoided as developers would change different files. However, there are no guarantee build conflicts would not arise; for example, our results show that build conflicts occur in the same and dependent files. So we still believe there are situations that conflicting contributions will be required to be executed simultaneously, and consequently, build conflicts will arise impacting the team productivity and the software quality.

Although most source code related build breakages we observed in merge scenarios are caused by build conflicts (239), we also find evidence of build breakages caused by immediate post integration changes (485), often performed with the aim of fixing merge conflicts. Such post integration changes occur when integrators, developers doing merge scenario integrations, change the merge integration result before committing it. For example, in project Traccar, both parents change overlapping line of the class `WebServer`.[24] While left parent adds the new method `initRestApi`, right also adds the method `initConsole`. During the attemp of integration, the merge tool reports a merge conflict that was treated by the integrator. As a result, the method declaration `initRestApi` was removed, and consequenlty a reference for this method could not be satisfied breaking the build process.

This is consistent with our experience that fixing merge conflicts is occasionally challenging and error-prone, possibly introducing other kinds of conflicts that are harder to detect and resolve. Teams adopting improved tools like semi-structured merge [7] would reduce the number of spurious merge conflicts, and possibly reduce the risks of build breakages caused by immediate post integration changes.

In other cases, even the parent contributions not conflicting with each other,

[24] Build ID traccar/traccar/232042524, Merge Commit ca06e8d.

the changes introduced by the integrator during the integration are responsible for the broken build. For example, in project DSpace, the parent contributions do not conflict in Java files.[25] Even though, the broken build is caused by an *Unavailable Symbol File* (`DSpaceSetSpecFilter` class). The class `DSpaceItemRepository` presents a dangling reference that may not be resolved for class `DSpaceSetSpecFilter`. The right parent is the only one changing class `DSpaceItemRepository`. The class file location was changed and new code was added. Despite the changes in class `DSpaceItemRepository` introduced by the right parent, the integrator is the one responsible for the removal of class `DSpaceSetSpecFilter` breaking the build.

As our build classification criteria is programming language dependent, studying, for instance, Ruby projects could lead to rather different build conflicts frequencies. For example, a missing reference for a method leads to a build conflict in Java, since the code cannot be compiled and built. However, in Ruby this would likely be classified as a test conflict,[26] and only if there is a test case that exercises such method reference. Due to Ruby dynamic features, there is no pre-execution check about the existence of method declarations.

5.2. Implications

Based on the catalogue of build conflict causes derived from our study, awareness tools such as Palantír [10] could alert developers about the risk of some conflict situations they currently do not support. For example, suppose a developer renames a method while another developer adds a new reference for it. In this case, an awareness tool would alert the second developer about the renaming performed by the first.

Our results are also supporting evidence for some of the warnings currently supported by Palantír. In another scenario, suppose two developers simultaneously add to the same class two methods with the same signature. Palantír could

[25] Build ID DSpace/DSpace/263379307, Merge Commit 049eb50.

[26] Failures revealed after testing the integrated code.

alert the second developer that such method signature has been introduced by another developer. Our catalogue of build conflict causes could support assistive tools that consider developers workspace as source information for predicting conflicts. Such tool should be aware of changes in dependent classes independent of when changes are performed.

Program repair tools [42] could benefit from the derived catalogue of build conflict resolution patterns, and the study infrastructure, to automatically fix conflicts. For example, part of the *Unavailable Symbol* conflicts could be automatically fixed as follows. Suppose the left parent renames a method, while the right parent adds a call using the old name. The attempt to build the resulting integrated code will break because of the call for an *Unavailable Symbol* (missing method, *build conflict*). A program repair tool could rename the newly added call, fixing the build conflict. Our scripts could support such a tool as they automatically identify the new method name by comparing the base and left commits. The same approach could be adopted if the symbol was deleted by one parent. In this case, a repair tool would reintroduce the deleted declaration that caused the build conflict. *Unavailable Symbol* related to missing class declarations should be more carefully handled, as they could have been moved. Instead of always reintroducing the class, which would introduce code duplication and possibly inconsistencies, after further analysis and confirmation, a program repair tool could simply update an import declaration.

In the same way, a program repair tool could also fix build conflicts caused by *Incompatible Method Signature*. This error may happen in two contexts. First when deleting parameters from a method signature. The tool could then update the broken call adapting it to the new method signature by removing arguments. Second when deleting the declaration of an overloaded method that shares the name, but not the signature, with others in the same class. To fix this error, the tool could reintroduce the missing method declaration.

Most fix commits are authored by one of the contributors involved in the merge scenario. Our assumption is that build conflicts are harder to fix than merge conflicts, requiring one of the contributors to be involved in the fix. This

might also reflect often adopted practices of requiring contributors to successfully integrate code before sending contributions to the main repository.

5.3. Build Conflict Repair Prototype

For better assessing this proposal of using our catalogue and study infrastructure to implement a program repair tool that fixes build conflicts, we implemented a prototype that identifies and recommends fixes for some build conflicts identified in this study. Although the current implementation does not cover all conflict causes reported in this study, we belive the tool could have automatically fixed 21% of the build conflicts in our sample, if used by the respective development teams. Next, we present how the tool handle build conflicts caused by *Unavailable Symbol* of variables.

We structure our prototype in terms of three main execution steps: fault localization, which identifies the *Unavailable Symbol* build conflict and its cause; patch creation, which proposes changes that solve the conflict; and patch validation, which confirms with developers that the proposed solution is valid. To detail these steps, we use as example a broken build in project Swagger Core [27], reflecting an *Unavailable Symbol* build conflict caused by a missing reference for the `op` local variable. One of the developers renames the variable to `apiOperation`, while the other developer adds new code referencing the old variable name.

The main goal of this proposed tool is to handle build conflicts. As verified in this study and previous ones, this kind of problem occur only in merge scenarios. So that is the reason it is required merge commits as input for the tool. In the next sections, we explain how the particularities of build conflicts are reflected in the way the tool works.

5.3.1. Fault Localization

For a given merge scenario, the tool first uses our script to check whether the provided scenario has build conflicts. If there is at least one build conflict,

[27] Build ID swagger-api/swagger-core/65086450, Merge Commit 657b64b.

the tool uses another script for classifying the conflicts according to the conflict categories in our catalogue. In case of a build conflict supported by the tool, we move on to the next step carrying on the information yielded by the classification script. In the illustrated case, a build conflict caused by *Unavailable Symbol* due to a missing variable. So the tool parses the build logs and finds out that a missing reference for the variable `op` causes the conflict.

To ensure the missing variable is a true build conflict, the tool adopts our scripts analyzing syntactic information of parent contributions. In our example, the local variable `op` was renamed as `apiOperation` by one of the developers. Analyzing just the contributions of the left parent would only observe the variable was renamed. In the same way, analyzing the right parent individually would just inform that a reference for the missing variable was added. Considering each analysis separately, none of them present enough information to ensure that updating the variable name would fix the problem. First, it is necessary to be aware of the scope where the variable was renamed. For instance, if two methods present the same local variable declaration, the changes in one method do not impact the other method. Just when analyzing both parents together the conflict can be confirmed, and then a fix can be recommended.

A more robust tool would perform exactly these substeps, but relying on local build information (instead of depending only on Travis CI) to check conflict occurrence, and possibly monitoring the local Git repository to trigger the fault localization process for each new merge commit.

5.3.2. Patch Creation

To create a patch, the tool further analyses the integrated code contributions to better understand the changes that lead to the missing variable declaration issue. Different changes to the variable declaration demand different resolution strategies. In our example, the local variable `op` was renamed as `apiOperation` by one of the developers. The solution the tool proposes is then to update the dangling variable references so that they use the new name. In case of deletion of the missing variable declaration, the tool proposes a solution that basically

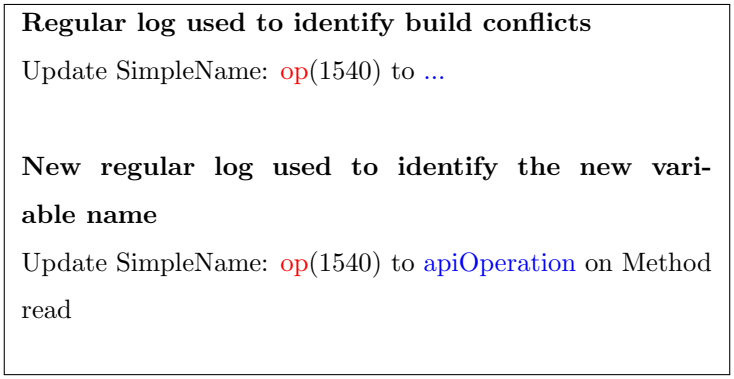

Figure 3: Syntactic diff used to identify the new variable name

consists in reintroducing the deleted declaration.

The identification that the variable *op* was renamed is carried on by one of our study scripts, as explained before (see Section 3.3). We basically analyze the result of the syntactic diff (see top of Figure 3). To identify the new variable name, our prototype extends the analysis process of the integrated code contributions to obtain the required extra information; it is also obtained by analyzing the result of the syntactic diff (see bottom of Figure 3).

5.3.3. Patch Validation

The expected behaviour in this step is to apply the changes, and then confirm with developers that the build conflict was appropriately solved. The tool basically informs the cause of the build conflict, presents the proposed fix, and asks the developer if she accepts the recommended fix. Once the developer accepts the recommendation, the tool applies the necessary changes and compiles the resulting source code to check if any compilation problems still remain. If no problem is detected, the tool created a new associated commit.

Our current implementation can also fix build conflicts caused by *Unimplemented Method*, *Duplicated Declaration* and *Unavailable Symbol Method*. Additional information about our prototype, and its source code, are available online [33].

6. Threats to Validity

Our empirical study leaves a set of validity threats that we now explain.

Construct Validity

As explained in earlier sections, given the nature of our sample, the conflict frequencies we observe should be interpreted in the limited scope of conflict occurrences that reach public repositories. As these occurrences might have greater impact due to their wider visibility, it is worth studying them even if they do not occur frequently in projects that use continuous integration; studying private developers repositories would, nevertheless, be also important. [43] have also experienced that. In their study, they observe a decreasing trend in builds with errors caused by missing classes and dependencies after projects adopt Travis.

Concerning the accuracy of our scripts, we can ensure our scripts are accurate, and our data do not have false positives. We achieve this accuracy adopting a more conservative approach than related work. For build conflicts in merge scenarios, we extract the causes responsible for the broken build. Next, we extract syntatic information of each parent contribution, and observe conflict occurrence checking if the error messages are caused by parent contributions. In case our scripts confirm the interference, we compute a new build conflict. As our scripts were created based on cases we manually analized, some interferences among contributions may not be verified. Instead of assuming these cases as build conflicts and introduce possible false positives, we conservative categorize them as broken builds caused by post integration changes.

Concerning the accuracy of our prototype, we also confirm it is accurate. Errors caused by *Unavailable Symbol* of variables are related to missing variable declarations in a specific scope. Depending of the circunstances that caused the missing declaration, the tool may adopt different fixes. If a variable declaration is removed, the tool recommends the reintroduction the missing declaration. In case of variable rename, our tool recommends the update of the dangling variable to its new name. Even if a developer changes the variable declaration

location in a specific scope, while the other adds a new variable reference, this action does not represent a conflict of *Unavailable Symbol*. The reference is still available. The error fixed by the prototype is related to the use of a variable in a specific scope without its declaration. However, if a developer changes the object type, a build conflict of *Incompatible Type* could arise. For this case, other fix patterns should be adopted considering the particularity of this conflict type that it is still not handled by our prototype.

Internal Validity

We miss conflicts that appear in merge scenarios that we could not build on Travis. Projects might specify a list of branches for creating builds on Travis. As all builds we create come from the master branch, if it is not in the list, no build process is started and we have to discard the potentially conflicting scenario. As in related work [4, 3, 44, 7], we might have missed code integration scenarios, and conflicts, that reach public repositories but do not result in merge commits. This might occur when using git commands such as *rebase*, *squash*, *cherry-pick*, or under smart kinds of fast-forward performed by git merge. This might also happen when stashing changes, pulling from the main repository, and then applying the stashed changes. Thus, our results are actually a lower bound for build conflicts.

Some of our scripts are limited in the sense that they search for conflicting contributions with partial information and resolution patterns. As discussed in Section 3.3, for some scenarios, this could bring imprecisions in the conflict confirmation process. So we manually analyze the risky cases and confirmed that no imprecision occurred, or at most that a conflict was classified as a post integration change that lead to a broken build. For resolution patterns, we analyze the syntactic differences applied into the fix commit . Only 27% of all build conflicts demanded manual analysis, while 77% of the resolution patterns were manually analyzed. Both analysis were rather simple and syntactic, and some steps were supported by our scripts as they inform the files to be analyzed reducing effort and the chances of human error. As explained in Section 3.2,

our scripts can parse a list of common build error messages. In case a build log presents an error message not in this list, the log is discarded. So our results actually represent a lower bound of build conflicts.

We use GumTree diff to analyze the contributions performed by developers and integrators. However, our approach has some limitations. For example, when we look for a specific method in the diff of a class, our search uses the method name instead of its signature. It represents a threat for *Duplicated Declaration* and *Unimplemented Method* since a class can have two methods with the same name but different signatures. When we try to classify build conflicts caused by contributor changes, we observe conflicting contributions looking for the method names in the syntactic diffs, which can lead us to a wrong conclusion. If both parents introduce two methods with the same name but different signature, and in the merge commit there are two methods with the same signature, the conflict is motivated by the integrator changes and not by contributor ones. To dimension the impact of this threat, we manually evaluate all cases of *Duplicated Declaration* methods revealing all cases were well classified.

During the study, some manual analysis was performed by one single person. Although the procedures were verified by another researcher, such analysis can introduce bias in the results as a false judgment would lead to inconsistencies. However, all manual analysis was supported by information from our scripts decreasing the chances of errors being done. As explained in Section 3.2, our scripts can parse a list of common build error messages. Regarding the accuracy of our scripts, they can parse a list of common build error messages. In case a build log presents an error message not in this list, the log is discarded. However, our scripts were able to classify 99% of all build commits

External Validity

Our results are specific to the context of open-source GitHub Java projects that use Maven and Travis CI. Although our sample of projects has a considerable degree of diversity concerning domain, size, and the number of collabora-

tors, most of the projects are small or medium-sized projects. In earlier sections, we motivate our choices. We also explain that results could be very different for dynamic languages; most build conflicts we discuss here would actually appear as test conflicts. We analyze build logs looking for specific message patterns associated to problems that cause the build breakage. After a pattern is identified, we perform additional analysis as explained in Section 3, to ensure the problem is caused by a conflict. For a new sample, different patterns can arise, demanding script adaptation to handle them. Build systems with less informative log reports, and CI services and tools with less accessible information, could hinder replication. Besides that, we are not aware how our choices could affect the results.

7. Related Work

Empirical studies about build conflicts provide evidence of their occurrence in practice, but none of them investigate the *causes* and *resolution patterns* for build *conflicts* as we do here. Early studies [4, 3] focus only on reporting conflict *frequencies* (the focus of RQ1), and use that as a motivation for the tools they propose. In total, these early studies analyze seven open-source GitHub projects, and consider that a merge scenario has a conflict if the corresponding merge commit build fails. We are more conservative because we additionally check whether the changes are related to the build error message. This way we do not consider as integration conflicts build breakages purely inherited from parents. They also replay build creation in their environment, which might introduce noise due to differences to the development environment used by project teams. For example, variations in the environment configuration, or an unavailable dependency, could break the build process, leading the authors to confirm a non existing conflict. Contrasting, we opt for collecting actual build logs created when developers contributed to the projects main repositories. At worst, we create builds in the same cloud environment.

Although we consider a much larger sample than the two mentioned early

studies, it is nevertheless biased by practices such as automated build support and continuous integration. Because of that, as better explained earlier, many conflicts do not reach main public repositories as analyzed here. Once developers have these support locally, it is expected they sent their individual contributions to the main repository without problems. Thus, the main repository will be always consistent. This phenomenon helps to further justify the much lower conflict frequencies we observe in our study. On the other hand, the mentioned previous studies only consider *clean merges scenarios* as valid subjects, while we also consider scenarios resulting from merge conflicts. To identify conflicts, they try to build locally the *clean merge scenarios* (merge scenarios without merge conflicts). Since they identify conflicts only based on the build process status of the merge commit, some false positives can be introduced. For example, the build process of a merge commit can fail due to previous changes performed in one of its parent commits. Previous studies [4, 3] do not clearly explain if they check the build process status of merge parent commits. In the same way, any variation in the environment configuration or an unavailable dependency could break the build process leading to results that are not entirely consistent with what actually occurred in practice. Finally, although the authors observe conflict frequency, they do not investigate conflict causes nor resolution patterns as we do here. We are not aware of previous studies that explore our research questions 2 and 3.

A number of studies investigate the causes of errors in the build process [31, 32, 45, 26, 25, 24], but none of them investigate whether these errors are caused by conflicting contributions, and are therefore related to collaboration or coordination breakdowns. Seo et al. [31] investigate build error causes, categories, and their related frequencies in general, not relating them to integration changes or conflicts as we do here. So they mostly explore the causes for individual developers changes that lead to a broken build, but do not study integration conflicts as motivated here. Although they analyze millions of builds from different projects, they consider projects from a single company. Concerning their error categories, four out of five can be mapped to our conflict causes. The only difference is

the fifth category that is related to *syntax* errors. We do not consider this kind of problem as a build conflict because, assuming the parents do not have syntactic errors, a syntactic problem in a merge commit can only result of changes performed by an integrator right after integration. Thus we consider this category only for build errors caused by post integration changes. We also have one conflict cause that has not been observed by [31] (*Project Rules*, see Table 3). The way they treat *Unavailable Symbol* is also different. While they consider all references to an unavailable element as a single problem, we further classify in three subcategories: *classes*, *methods*, and *variables*. This new perspective is necessary and valid when we think about repair tools and their approaches to fix them, as explained in Section 5.2. Just adding the missing elements might introduce inconsistencies in the project, resulting in situations such as having two versions of the same class, but in different packages. More important, they simply bring the frequencies of each build breakage cause, not performing further analysis to understand how many of these are actually caused by interfering contributions, nor understanding the structure of the changes that lead to the breakage and to the associated fix.

The other studies have the same limitation: they focus on build breakage, not integration conflicts. For example, [26, 25] explore the overall causes of broken builds. Consequently, they also do not go further and reveal the structure of the changes that lead to conflicts, or study resolution patterns, as we do here. Both studies group error categories based on the build process phase in which the error occurs. Although they consider compilation errors as causes for broken builds, they do not investigate further the errors grouping them into specific categories.

Some studies have investigated build process under the perspective of non-deterministic commit [46, 47, 48]. In these cases, different build process associated with a single commit present different results. It may occurs due to environment issues, semantic problems, or just by chance. Although previous studies identified these problems in test failures, they do not occur in our study because of the nature of build conflicts. For instance, even if a build process

present a broken status, we further analyze the contributions to ensure the conflict build occurrence. So the broken build status is just a start point of our analysis. If the contributions are not conflicting with each other, we do not consider this case as a build conflict.

While Raush et al. [26] list compilation problems as causes of broken builds, Vassalo et al. [25] group errors based on build process phase that the error occurs. However, they do not observe if the build error was caused by conflicting contributions among developers. The studies identify general problem categories, but they do not go further and analyze each category. For example, compilation problem is classified as a category, but there is not specific causes for this problem. They also do not focus on the resolution patterns adopted to fix these broken builds. Kerzazi et al. [32] present a study investigating the impact of broken builds on software development. For that, they perform a qualitative and quantitative study. Just like one of our findings, the qualitative study reports that the primary cause of broken builds is due to missing classes (*Unavailable Symbol*). However they do not investigate other causes like we do here, and also do not present a catalogue of causes. They investigate the costs of broken builds measuring the time spent to fix the broken build. We measure the cost of broken builds, but instead of checking the time spent, we evaluate the changes performed to fix the broken build. So they also do not present a catalogue of resolution patterns as we do here.

8. Conclusions

Although essential for non small teams, the independence provided by collaborative software development might result in conflicts, and their negative consequences on product quality and development productivity, when integrating developers changes. To better understand part of these conflicts— the ones revealed by failures when building integrated code— in this paper we investigate, through three research questions, their frequency, structure, and adopted resolution patterns, deriving novel catalogues of build conflicts. Only the first

question, related to conflict frequency, had been explored before, but in a much more restricted context.

Our conflict frequency results reveal build conflicts occur less frequently than reported by previous studies. However, we eliminate threats, which related studies are not aware. We analyze and justify that, highlighting the differences in the experiment design, and showing that our results are more conservative and complement previous results because of the focus on a rather different kind of sample. Besides the frequency results, we find and classify build conflicts causes, and their resolution patterns, deriving two novel conflict catalogues. We also find and analyze build failures caused by immediate post integration changes, which are often performed with the aim of fixing merge conflicts. Most build conflicts are caused by missing declarations removed or renamed by one developer but referenced by the changes of another developer. These conflicts are often resolved by updating the dangling reference. To resolve build conflicts developers often fix the integrated code, instead of completely discarding changes and conservatively restoring project state to a previous commit. Most fix commits are authored by one of the contributors involved in the merge scenario.

Based on the catalogue of build conflict causes derived from our study, awareness tools could alert developers about the risk of a number of conflict situations they currently do not support. Program repair tools could benefit from the derived catalogue of build conflict resolution patterns, and the study infrastructure, to automatically fix conflicts. We further explore this by developing a prototype program repair tool that helps developers to resolve some kinds of build conflicts we identified in our study. The current implementation could have automatically fixed 21% of the build conflicts in our sample, if used by the respective development teams.

As future work, we would like to propose further studies with different samples that consider alternative practical contexts. It would also be interesting to develop or improve assistive tools as discussed here. Our findings set up improvements on assistive tools aiming to avoid build conflicts, as also bring ideas of new tools for helping developers to treat such problems. As future work, we

would like to evaluate private repositories as also consider different subjects like better merge tools, CI services and build managers.

Collaborative software development brings some challenges when developers do their tasks, separately and simultaneously. When different contributions are integrated, conflicts arise impairing productivity. Conflicts are not only perceived during integration (merge conflicts), but also during the build process (build conflicts). In this study, we investigated the frequency, causes and resolution patterns adopted for build conflicts.

9. Acknowledgements

We thank Marcelo d'Amorim, Maurício Aniche, and Jacob Krueger for the quite pertinent suggestions that contributed to improve this work. We also would like to thank Samuel Brasileiro, Tales Tomaz, and Luís Santos for providing support on our protoype tool. Finally, for partially supporting this work, we would like to thank INES (National Software Engineering Institute), and the Brazilian research funding agencies CNPq (grant 309741/2013-0), FACEPE (IBPG-0692-1.03/17 and APQ/0388-1.03/14), and CAPES.

References

[1] D. E. Perry, H. P. Siy, L. G. Votta, Parallel changes in large-scale software development: an observational case study, ACM Transactions on Software Engineering and Methodology 10 (3) (2001) 308–337. doi:10.1145/383876.383878.

[2] T. Zimmermann, Mining workspace updates in cvs, in: International Workshop on Mining Software Repositories, IEEE, 2007, pp. 11–11. doi:10.1109/MSR.2007.22.

[3] Y. Brun, R. Holmes, M. D. Ernst, D. Notkin, Proactive detection of collaboration conflicts, in: ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering, ACM, 2011, pp. 168–178. doi:10.1145/2025113.2025139.

[4] B. K. Kasi, A. Sarma, Cassandra: Proactive conflict minimization through optimized task scheduling, in: International Conference on Software Engineering, IEEE, 2013, pp. 732–741. doi:10.1109/ICSE.2013.6606619.

[5] S. Apel, J. Liebig, B. Brandl, C. Lengauer, C. Kästner, Semistructured merge: rethinking merge in revision control systems, in: ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering, ACM, 2011, pp. 190–200. doi:10.1145/2025113.2025141.

[6] S. Apel, O. Leßenich, C. Lengauer, Structured merge with auto-tuning: balancing precision and performance, in: International Conference on Automated Software Engineering, ACM, 2012, pp. 120–129. doi:10.1145/2351676.2351694.

[7] G. Cavalcanti, P. Borba, P. Accioly, Evaluating and improving semistructured merge, ACM on Programming Languages 1 (OOPSLA) (2017) 1–27. doi:10.1145/3133883.

[8] P. Accioly, P. Borba, G. Cavalcanti, Understanding semi-structured merge conflict characteristics in open-source java projects, Empirical Software Engineering 23 (4) (2018) 2051–2085. doi:10.1007/s10664-017-9586-1.

[9] G. Cavalcanti, P. Borba, G. Seibt, S. Apel, The impact of structure on software merging: semistructured versus structured merge, in: International Conference on Automated Software Engineering, IEEE, 2019, pp. 1002–1013. doi:10.1109/ASE.2019.00097.

[10] A. Sarma, D. F. Redmiles, A. Van Der Hoek, Palantir: Early detection of development conflicts arising from parallel code changes, Transactions on Software Engineering 38 (4) (2011) 889–908. doi:10.1109/TSE.2011.64.

[11] C. Bird, T. Zimmermann, Assessing the value of branches with what-if analysis, in: International Symposium on the Foundations of Software Engineering, ACM, 2012, pp. 1–11. doi:10.1145/2393596.2393648.

[12] S. McKee, N. Nelson, A. Sarma, D. Dig, Software practitioner perspectives on merge conflicts and resolutions, in: International Conference on Software Maintenance and Evolution, IEEE, 2017, pp. 467–478. doi:10.1109/ICSME.2017.53.

[13] R. E. Grinter, Supporting articulation work using software configuration management systems, Computer Supported Cooperative Work 5 (4) (1996) 447–465.

[14] C. R. De Souza, D. Redmiles, P. Dourish, " breaking the code", moving between private and public work in collaborative software development, in: International ACM SIGGROUP Conference on Supporting Group Work, ACM, 2003, pp. 105–114. doi:10.1145/958160.958177.

[15] B. Adams, S. McIntosh, Modern release engineering in a nutshell–why researchers should care, in: International Conference on Software Analysis, Evolution, and Reengineering, IEEE, 2016, pp. 78–90. doi:10.1109/SANER.2016.108.

[16] F. Henderson, Software engineering at google, arXiv preprint arXiv:1702.01715.

[17] R. Potvin, J. Levenberg, Why google stores billions of lines of code in a single repository, Communications of the ACM 59 (7) (2016) 78–87. doi:10.1145/2854146.

[18] L. Bass, I. Weber, L. Zhu, DevOps: A Software Architect's Perspective, Addison-Wesley Professional, 2015.

[19] M. Fowler, Feature Toggle, https://martinfowler.com/bliki/FeatureToggle.html, Accessed: January 2020 (2017).

[20] P. Hodgson, Feature Branching vs. Feature Flags: What's the Right Tool for the Job?, https://devops.com/feature-branching-vs-feature-flags-whats-right-tool-job/, Accessed: January 2020 (2017).

[21] M. Fowler, Feature Branch, https://martinfowler.com/bliki/FeatureBranch.html, Accessed: January 2020 (2009).

[22] P. Hodgson, Feature Toggles (aka Feature Flags), https://martinfowler.com/articles/feature-toggles.html, Accessed: January 2020 (2017).

[23] G. Cavalcanti, P. Accioly, P. Borba, Assessing semistructured merge in version control systems: A replicated experiment, in: International Symposium on Empirical Software Engineering and Measurement, IEEE, 2015, pp. 1–10. doi:10.1109/ESEM.2015.7321191.

[24] A. E. Hassan, K. Zhang, Using decision trees to predict the certification result of a build, in: International Conference on Automated Software Engineering, IEEE, 2006, pp. 189–198. doi:10.1109/ASE.2006.72.

[25] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, S. Panichella, A tale of ci build failures: An open source and a financial organization perspective, in: International Conference on Software Maintenance and Evolution, IEEE, 2017, pp. 183–193. doi:10.1109/ICSME.2017.67.

[26] T. Rausch, W. Hummer, P. Leitner, S. Schulte, An empirical analysis of build failures in the continuous integration workflows of java-based open-source software, in: International Conference on Mining Software Repositories, IEEE, 2017, pp. 345–355. doi:10.1109/MSR.2017.54.

[27] M. Hilton, T. Tunnell, K. Huang, D. Marinov, D. Dig, Usage, costs, and benefits of continuous integration in open-source projects, in: International Conference on Automated Software Engineering, IEEE, 2016, pp. 426–437.

[28] B. Vasilescu, S. Van Schuylenburg, J. Wulms, A. Serebrenik, M. G. van den Brand, Continuous integration in a social-coding world: Empirical evidence from github, in: International Conference on Software Maintenance and Evolution, IEEE, 2014, pp. 401–405. doi:10.1109/ICSME.2014.62.

[29] D. Shao, S. Khurshid, D. E. Perry, Evaluation of semantic interference detection in parallel changes: an exploratory experiment, in: International Conference on Software Maintenance, IEEE, 2007, pp. 74–83. doi:10.1109/ICSM.2007.4362620.

[30] C. Sung, S. K. Lahiri, M. Kaufman, P. Choudhury, C. Wang, Towards understanding and fixing upstream merge induced conflicts in divergent forks: an industrial case study, in: International Conference on Software Engineering: Software Engineering in Practice, ACM, 2020, pp. 172–181. doi:10.1145/3377813.3381362.

[31] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, R. Bowdidge, Programmers' build errors: a case study (at google), in: International Conference on Software Engineering, ACM, 2014, pp. 724–734. doi:10.1145/2568225.2568255.

[32] N. Kerzazi, F. Khomh, B. Adams, Why do automated builds break? an empirical study, in: International Conference on Software Maintenance and Evolution, IEEE, 2014, pp. 41–50. doi:10.1109/ICSME.2014.26.

[33] L. Da Silva, P. Borba, A. Pires, Online Appendix, https://build-test-conflicts.github.io/build-conflicts/main.html, Accessed: December 2020.

[34] Maven, URL:https://maven.apache.org, Accessed: January 2020.

[35] N. Munaiah, S. Kroh, C. Cabrey, M. Nagappan, Curating github for engineered software projects, Empirical Software Engineering 22 (6) (2017) 3219–3253. doi:10.1007/s10664-017-9512-6.

[36] M. Beller, G. Gousios, A. Zaidman, Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration, in: International Conference on Mining Software Repositories, IEEE, 2017, pp. 447–450. doi:10.1109/MSR.2017.24.

[37] Gradle, https://gradle.org/, Accessed: January 2020.

[38] M. Nagappan, T. Zimmermann, C. Bird, Diversity in software engineering research, in: Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 466–476. doi:10.1145/2491411.2491415.

[39] S. Chacon, B. Straub, Pro git, Apress, 2014.

[40] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus, Fine-grained and accurate source code differencing, in: International Conference on Automated Software Engineering, ACM, 2014, pp. 313–324. doi:10.1145/2642937.2642982.

[41] T. Rocha, P. Borba, J. P. Santos, Using acceptance tests to predict files changed by programming tasks, Journal of Systems and Software 154 (2019) 176–195. doi:10.1016/j.jss.2019.04.060.

[42] C. Le Goues, S. Forrest, W. Weimer, Current challenges in automatic software repair, Software quality journal 21 (3) (2013) 421–443. doi:10.1007/s11219-013-9208-0.

[43] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, B. Vasilescu, The impact of continuous integration on other software development practices: a large-scale empirical study, in: International Conference on Automated Software Engineering, IEEE, 2017, pp. 60–71. doi:10.1109/ASE.2017.8115619.

[44] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, C. Hunsen, Indicators for merge conflicts in the wild: survey and empirical study, Automated Software Engineering 25 (2) (2018) 279–313. doi:10.1007/s10515-017-0227-0.

[45] M. Beller, G. Gousios, A. Zaidman, Oops, my tests broke the build: An explorative analysis of travis ci with github, in: International Conference on Mining Software Repositories, IEEE, 2017, pp. 356–367. doi:10.1109/MSR.2017.62.

[46] A. Labuschagne, L. Inozemtseva, R. Holmes, Measuring the cost of regression testing in practice: a study of java projects using continuous integration, in: Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 821–830. doi:10.1145/3106237.3106288.

[47] K. Herzig, N. Nagappan, Empirically detecting false test alarms using association rules, in: International Conference on Software Engineering, IEEE, 2015, pp. 39–48. doi:10.1109/ICSE.2015.133.

[48] S. Elbaum, G. Rothermel, J. Penix, Techniques for improving regression testing in continuous integration development environments, in: ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 235–245. doi:10.1145/2635868.2635910.