

RESEARCH ARTICLE

Build Conflicts in the Wild

Léuson da Silva | Paulo Borba | Arthur Pires

Informatics Center,
Federal University of Pernambuco,
Recife, Brazil.

Correspondence

Leuson Da Silva
Federal University of Pernambuco,
Recife, Brazil.
Email: lmps2@cin.ufpe.br

Summary

When collaborating, developers often create and change software artifacts without being fully aware of team members' work. While such independence is essential for increasing development productivity, it might also result in conflicts when integrating developers' contributions. To better understand the conflicts 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 their Travis build logs, and check whether the 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 merge scenario contributors. Based on our catalogue of conflict causes, awareness tools could alert developers about the risk of conflict situations. Program repair tools could benefit from our catalogue of resolution patterns to automatically fix conflicts; we illustrate that with a proof of concept implementation of a tool that fixes 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,¹ 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}. These integration conflicts have been observed even when using advanced merge tools^{5,6,7,8,9} that avoid common spurious merge conflicts reported by the 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-ins¹⁴. 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 complexity²².

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}; 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*

¹Projects involving non-small teams might face more code integration problems as more developers may simultaneously change the same files. However, small teams might face these problems as well.

in a larger context, as we do here analyzing more than 450 open-source projects. Furthermore, previous studies do 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. These projects 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*).² To make sure the integrated changes cause the breakage, 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 presented in Section 4 and related threats discussed in Section 6. 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 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 integrated changes do not cause the failure, 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 another developer's changes. 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 the project state to a previous commit. We have also observed that most fix commits are authored by one of the merge scenarios' contributors.

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 and applies fixes for some build conflict causes reported in our catalogue. 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.³ 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. Nevertheless, 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).

²In 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*.

³Build ID sanity/quickml/53571613, Merge Commit 62a2190.

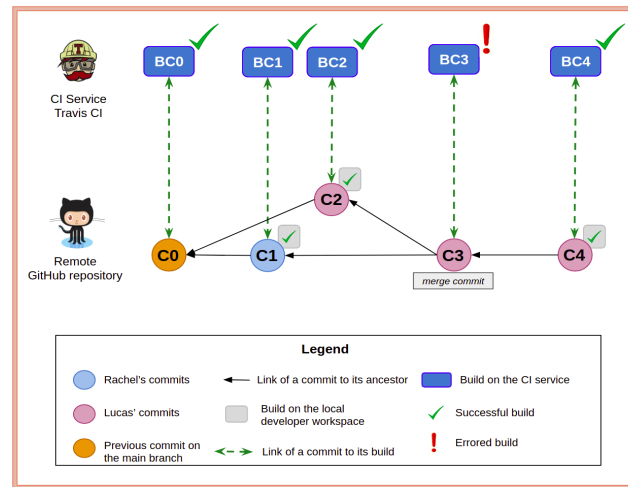


FIGURE 1 Build conflicts in practice. On the bottom, blue and pink circles represent commits done by Rachel and Lucas, respectively. In contrast, the orange circle represents the previous commit in the main branch before the described merge scenario. Above some developers' commits, gray squares represent the build processes done locally on the developers' workspace (✓, for successful builds). Black arrows link commits to their ancestors. Finally, on the top, green arrows associate commits from the remote repository with their build processes (blue boxes) on the Continuous Integration (CI) service. Above each build, we present its status (✓ and !, for successful and errored builds, respectively). For simplicity, we do not present commits done on developers' private repositories.

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 unintentionally interfere^{7,29} with the changes of the other. Trying to confirm the interference, Lucas checks the broken build log. He observes that a method call for `ignoreAttributeAtNodeProbability`, in the `StaticBuilders` class, is the source of a compilation error because its declaration is missing 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`. He 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's changes to her private repository, merging them, and making sure she can successfully build and test the integrated code. Adopting this strategy would have avoided leaving the main repository in an inconsistent state, but would not avoid the conflict nor the effort to resolve it. The developer would have to stop her work, understand the problem, and find and fix the conflict. 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 contribution model would avoid the inconsistent state, and help to earlier detect the conflict, but would not necessarily avoid it. In the discussed merge scenario, we illustrate the occurrence of a single build conflict. However, a merge scenario may have several conflicts caused by different changes, requiring specific treatment for each one.

Our motivating example presents a specific context in which build conflicts may occur. However, there are other possible contexts 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. Although the discussed build conflict appears when different branches of a single project are integrated, these conflicts are not exclusive to integrations involving a single project. Sung et al.³⁰ report build conflicts also occur when different projects are integrated. In their study, conflicts occur when developers pull changes from the main project (Chromium) into another project (Edge). Since both involved projects are in conflicting different states, build conflicts are reported.

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 PR changes. In case the repository adopts CI services like Travis CI, the broken build would be reported on the GitHub PR page, and the conflict could be easily detected. Thus, the integrator would deal with the conflict or request changes to the contributors until the problem is fixed. Otherwise, the PR could be accepted, polluting the main repository. Someone may argue that PR integration could be prioritized aiming to reduce the chances of conflicting integrations³¹. However, even in such circumstances, build conflicts might be earlier detected but not avoided. 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 activity³⁰.

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}. Kasi and Sarma⁴ report that build conflict rates substantially vary across projects, ranging from 2% to 15%. Brun et al.³ find a similar range: 1% to 10%. However, this is not 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 it 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. Suppose our scripts are not able to check that due to the limited set of patterns they recognize. In that case, 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 of Kasi and Sarma⁴ and Brun et al.³ do not bring information about conflict causes. They do not investigate the *structure* of changes that cause build conflicts. We go further by investigating and classifying these causes. Seo et al.³² 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 unexpectedly 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 conflict causes, and identify the most common causes.

Identifying conflict causes might help developers to reduce conflict numbers, but most likely will not eliminate them³³. 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

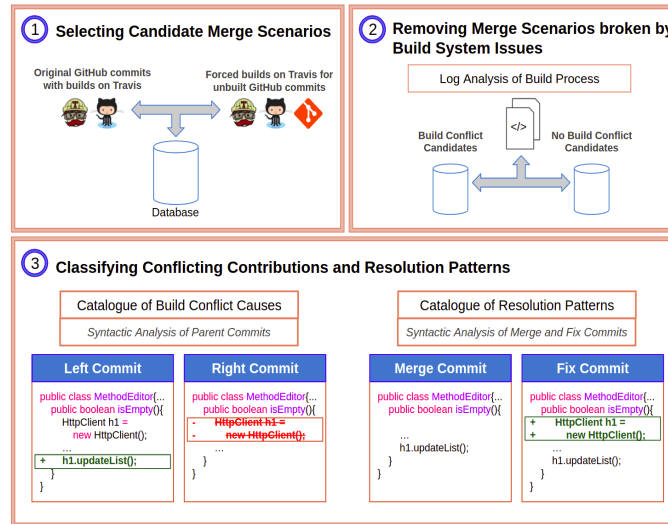


FIGURE 2 Study setup composed of three steps. The first step (Section 3.1) mines project repositories. Step two (Section 3.2) filters merge scenarios and yield build conflict candidates. Finally, step three (Section 3.3) classifies build conflict causes and resolution patterns.

also important to understand whether conflicts are fixed by both contributors and integrators, especially because integrators might have a harder time fixing them. When we say contributor changes, we mean changes performed before integration, while integrator changes are done during the integration.⁴ It is also possible that contributors play the role of integrators when integrating their contributions. Thus, our third research question, and a complementary related question, is 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 of Kasi and Sarma⁴ and Brun et al.³, 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 online³⁵ to support replications.

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.

⁴This is related but not strictly the same as an alternative terminology³⁴ that considers contributors as commit authors, and integrators as project members that inspect and integrate changes into the project's main development line. In this terminology, contributors might also play the role of integrators when integrating their contributions.

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

Project Sample

As our experiment relies on the analysis of source code and build status information, we opt for GitHub projects that use Travis CI for continuous integration for service popularity reasons. 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 Maven³⁶ 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 used in the studies of Munaiah et al.³⁷ and Beller et al.³⁸, which include a large number of selected and active open-source projects covering different languages and domains. For each project, the datasets also inform whether the project adopts any continuous integration service. We then select Java projects that satisfy the following criteria: (i) presence of the `.travis.yml` file in the root directory of the latest revision⁵ 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 the use of Maven), and absence of Gradle³⁹ configuration files, which could represent early use of Maven, but later migration to Gradle, demanding a different build log parser; and finally, (iv) the project should have at least one merge scenario considering the history interval we were analyzing. We discuss the threats regarding our choices in Section 6. This way, we ensure only Java projects that use Maven and Travis were selected^{27,28}. As a result, we select a sample of 451 projects.

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 online³⁵.

Merge Scenario Candidates

For each selected project, our scripts locally clone the project and select⁶ 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 merge scenarios that are Travis ready. We also avoid selecting *fast-forward* situations^{40, 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 API⁷ 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. We select or discard a merge scenario based on its final build status. So, if a build is composed of many *jobs* (build processes with a specific list of steps) and not all jobs have the same final status, we do not check individual job status to reach a decision. We consider the final status reported by Travis that takes into consideration all valid build jobs. To avoid bias, we also discard duplicate merge scenarios, which involve different merge commit hashes but have the same parents. Otherwise, the same merge scenario would be evaluated twice and introduce noise in our results as we would report duplicate conflicts caused by the same changes. As a result, 4252 merge scenarios are classified (third row in Table 1).

⁵As in August 2018.

⁶We use the `git log -merges` command with an extra parameter specifying the initial date to drive the search.

⁷<https://developer.github.com/v3/>

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 applied 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

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 investigation is needed because the broken build status might have been caused by a number of reasons 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 is 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 analyze, 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. The remaining 1% of the cases are not classified, as their logs were empty or without information that let us classify the cause; it may happen because old Maven versions do not report logs with complete information. 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. To measure these false negatives, we would need to build merge commits, probably updating the adopted Maven version, and expect to have access to a log with complete information. However, our scripts present a high precision when selecting or removing merge scenarios during this analysis. The third column of 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 the second column.

To filter out scenarios with build breakages caused by build system issues, we discard scenarios with broken merge commit builds whose logs have one of the error messages listed in rows *Environment Resource* and *Dependency* of 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 with specific error messages. However, even after discarding breakages caused by build system issues, these messages' occurrences do not imply conflict occurrences since the integrated changes might not have caused the error. For example, the breakage might have been inherited from one of the parent commits instead of conflicting

contributions. 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 online³⁵.

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 integrated changes caused the build breakage. So, we parse the build log and check whether the logged error messages are related to the changes integrated into the merge commit. To capture the integrated changes, we use GumTree⁴¹ to compute syntactic *diffs* among the *merge commit*, its *parents*, and the associated *base*⁸ commit. When our scripts, due to the limited set of recognized patterns supported by them, 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.⁹ So, after confirming the presence of a “cannot find...” error message (see Table 2, third column, fourth row) 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 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, fourth 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. 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 opt for not automating these cases because they rarely occur in our sample; for example, in our study, a removed build dependency causes a conflict only in one scenario. 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. Automatically detecting conflicts caused by *Incompatible Types* using GumTree *diffs* might introduce false positives in our results, as the *diffs* treat object types as strings. Suppose a project has two classes with the same name but on different packages, GumTree treats them as one single type. So our scripts could not differentiate them. Besides that, for cases caused by external dependencies, we could not compare different jar files using GumTree. In the same way, to automatically detect conflicts caused by *Project Rules*, we would need to work with XML and Java files as this cause involves style changes. However, we can not track style changes on GumTree *diffs*. 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 into 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.¹⁰ 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. If so, it 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. Our scripts now check again whether the merge commit parents present a superior status (*failed* or *passed*), and if the merge commit does not contain

⁸The latest common ancestor to the parent commits.

⁹Conflicts 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.

¹⁰The same developer might be playing both roles in the same merge scenario, but that is not necessarily the case.

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 online³⁵.

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 statuses indicate that the source code could at least be compiled. Suppose a possible fix commit under analysis does not present a superior build status. In that case, we move for the next commit performed after the previous commit under analysis until we find a proper commit with a superior build status. We adopt this strategy because a developer might have created a few conflicts before she is actually successful in fixing the conflict.

Our scripts automatically analyze the fix commits, extracting common resolution patterns that appear in the third column of Table 4. Similar 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 resolution 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. Similar 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 are 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. As explained in previous sections, the first question 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.¹¹ 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.

To better contextualize that, we observe in Table 1 that roughly 7.5% of the merge commit builds are broken in our sample (4252 out of 57065 merge scenarios). 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 build 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 repository²¹. 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. Accioly et al.⁸ observe that when applying improved merge tools on merge scenarios, merge conflicts are reported involving duplicated declaration methods. So traditional tools would merge the code without reporting a

¹¹Each scenario might have a number of conflicts caused by different changes and associated with different error messages in the build log of the scenario merge commit.

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

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, our 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 processes, and also of non integration related breakages. As these two aspects widely vary across the analyzed projects, we miss existing integration conflicts. Sung et al.³⁰ also investigate build conflicts, but when changes of different projects are integrated; developers face build conflicts when pulling changes from a project into another. Analyzing a project during three months, they report the occurrence of 398 build conflicts. Their results bring evidence build conflicts occur involving different projects when these problems do not reach the main development line.

In the end, 37 projects 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 developers' 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). Despite the low number of projects with conflicts, we do not observe exceptional characteristics that could justify the conflict occurrence in these projects.

As we comment in our motivating example in Section 2, our results 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 catalogue of build conflicts. Table 3 shows these six 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 classes (112 occurrences), corresponding to 73% of all *Unavailable Symbol* occurrences. Missing methods (22) and variables or

parameters (20) come next, with the missing declaration and the dangling reference possibly associated with 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 reinforce 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 that 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. Nevertheless, we have also observed similar problems when developers change files or program elements that are not directly related.

For example, in the Ontop project,¹² 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 `composeFunctions` added to the interface.

Another example occurs in the project PAC4J.¹³ 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* annotations 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.¹⁴ 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 annotation *override* above the method declaration could not be resolved. This case occurs because one parent 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 methods, each parent commit adds its method declaration in a class resulting in one single declaration. However, 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`.¹⁵ 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 his 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`.¹⁶ 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

¹²Build ID: ontop/ontop/59371438 – Merge Commit: c626206

¹³Build ID: pac4j/pac4j/291027337 – Merge Commit: e18dd85

¹⁴Build ID: ontop/ontop/83689234 – Merge Commit: 0f62121

¹⁵Build ID: tinkerpops/blueprints/267833702 – Merge Commit: 5a25e3a

¹⁶Build ID: perwendel/spark/229805514 – Merge Commit: 3fd18a9

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 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 correctly indicates the mentioned problem using type checking. If this analysis is remotely 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`.¹⁷ 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 observed inspecting these reported classes, the developer might explore the history changes performed by the parent commits and then verify 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 static analysis tools caused by *Project Rules* (third column, seventh row in Table 3). These failures appear during the so called ASAT phase of the Travis build process.¹⁸ 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.¹⁹ For instance, in the project HDIV,²⁰ one developer updates the license header file, while the other developer adds two new classes (`HTTPSessionCache` and `SimpleCacheKey`). As the newly added classes have the old header style, the verifications identify inconsistencies caused by build conflicts. We decide to include *Project Rules*' causes in our catalogue as the conflicts break the build process and involve conflicting contributions. However, as reported in Table 3, we classify them as *Other Static Analysis* as this cause does not involve static semantic aspects like the other causes.

Our catalogue 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 it was potentially hard to resolve; as it occurred in an auxiliary branch, developers ignored the changes in the branch and moved on to 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).²¹ Third, as our scripts try to build fix commits not built yet, environment problems may 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 analyze fixes for just part of the conflicts: 148 fixes out of all build conflicts. Only part of these conflict fixes is automatically analyzed by our scripts (44), while the remaining cases (104) are manually analyzed. We adopt this manual analysis as some patterns could not be checked using GumTree diffs, or rarely occurred; for example, in *Unavailable Symbol* conflicts caused by a missing class, they can be fixed by importing the whole package where the missing class is declared. However, there is no way to ensure the missing class is declared in that imported package by using GumTree diffs. The identified resolution patterns, together with their frequencies, appear in the fourth column of Table 4.

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 the project state to a previous commit. This practice is adopted in

¹⁷Build ID: NLPchina/elasticsearch-sql/99402562 – Merge Commit: eb5fe5f

¹⁸Travis Documentation: The Build Lifecycle

¹⁹Broken builds caused by errors during automated static analysis are classified as *errored* by Travis.

²⁰Build ID: hdiv/hdiv/126140680 – Merge Commit: c620070

²¹As we select our sample using previous studies' datasets, we may have selected projects that were active during the execution of these previous studies but no longer during our study.

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

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 that class. After the integration, the import could not be resolved, and the build conflict is reported. 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 a change in classes' location does not cause a conflict, 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, the reference for the missing symbol `builderWithHighestTrackableLatencyMillis` in the request of the `Activator` class was updated to the new method signature `builder`.²² 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* `deadline` (local variable), in the `GzipSource` class, was just removed.²³

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`.²⁴ 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. Regarding this conflict involving duplicated test cases, someone may argue that the developers want to test the same piece of code twice. If so, the integrator could rename one of the duplicated test cases and keep both tests. However, this observation is not applied to duplicated methods, when both developers add the same method and a specific call for that method. Renaming one of these methods and keeping both would introduce duplicated code in the class, which is not a good practice.

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

²²Build ID `datastax/java-driver/144600040`, Fix Build ID `datastax/java-driver/144856728`.

²³Build ID `square/okhttp/19399475`, Build Fix ID `square/okhttp/19404300`.

²⁴Build ID `tinkerpop/blueprints/267833702`, Build Fix ID `tinkerpop/blueprints/10120795`

is done by adding a call to the method `toString`, which now returns a `String` as expected by the parent contribution changes.²⁵ 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`.²⁶ 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 remote repository.

In other cases, adopted resolution patterns are not made in source code but on 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).²⁷ As we previously discuss (see Section 4.2), in this work, we do not focus on conflicts 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 that pattern). In this context, contributors may play an integrator role by integrating their contributions and applying changes when required. We could also observe this situation in other development models; for example, in pull-based development, when integrators require contributors to perform changes. So contributors would fix the build conflicts until the pull request is stable and can be accepted. Otherwise, considering the integrator is not aware of these conflicts, he can accept the PR polluting the repository.

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 online³⁵. 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 online³⁵.

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 Sarma⁴ 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 reaches 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. We must say all these conflict causes are intrinsic changes that were introduced by specific changes in the source code⁴². Even conflicts caused by external dependencies, they occur because a developer changes the external dependency version.

In our study, we investigate build conflicts aspects not targeted by the related work of Kasi and Sarma⁴ and Brun et al.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.³² discuss build errors in general, not relating them to integration conflicts, they report

²⁵Build ID NLPchina/elasticsearch-sql/99402562, Build Fix ID NLPchina/elasticsearch-sql/99402657

²⁶Build ID perwendel/spark/229805514, Build Fix ID perwendel/spark/403188038

²⁷Build ID vavr-io/vavr/58945430, Build Fix ID vavr-io/vavr/58958884.

a related finding: *Unavailable Symbol* is the most frequent cause of build error in their sample. It reveals the common mistake of removing or renaming declarations, but possibly not updating all references to the new identifiers. Assistive tools like Palantír¹⁰ 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).

Build conflicts caused by *Duplicated Declaration* have also been observed as semistructured merge conflicts in previous studies of Cavalcanti et al.⁷ and Accioly et al.⁸. It supports our assumption that build conflicts frequently occur in private repositories, but not so much in public repositories of projects with automated build processes 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 assignment of activities based on the chance of interference among contributions. Rocha et al.⁴³ 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 in some situations, conflicting contributions will be required to be executed simultaneously, and consequently, build conflicts will arise, impacting the team productivity and the software quality. Van der Veen et al.³¹ propose prioritizing pull request (PR) integration aiming to reduce the chances of conflicting integrations. However, prioritizing PR integration at this point would not avoid the occurrence of build conflicts as the contributions are already implemented. So the conflicts could be earlier reported but not avoided.

Although most source code related to build breakages we observe 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 change the merge integration result before committing it. For example, in project Traccar, both parents change overlapping lines of the class `WebServer`.²⁸ While left parent adds the new method `initRestApi`, right also adds the method `initConsole`. During the integration attempt, the merge tool reports a merge conflict treated by the integrator. As a result, the method declaration `initRestApi` is removed, and consequently, a reference for this method could not be satisfied, breaking the build process.

This kind of breakage 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 merge⁷ 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, the integrator's changes during the integration are responsible for the broken build. For example, in project DSpace, the parent contributions do not conflict in Java files.²⁹ Even though, the broken build is caused by an *Unavailable Symbol* (`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 is changed, and a new code is added. Despite the changes in class `DSpaceItemRepository` introduced by the right parent, the integrator is responsible for removing class `DSpaceSetSpecFilter` breaking the build.

As our build classification criteria are 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,³⁰ 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 Palantír¹⁰ 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 alert the second developer that another developer has introduced that method signature. Our catalogue of build conflict causes could support assistive tools that consider developers' workspace as source information for predicting conflicts. That tool should be aware of changes in dependent classes independent of when changes are performed.

Program repair tools⁴⁴ 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

²⁸Build ID traccar/traccar/232042524, Merge Commit ca06e8d.

²⁹Build ID DSpace/DSpace/263379307, Merge Commit 049eb50.

³⁰Failures revealed after testing the integrated code.

for an *Unavailable Symbol* (missing method). 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 assumption might also reflect often adopted practices of requiring contributors to successfully integrate code before sending contributions to the main repository; for example, in a pull-based development model, it is expected that integrators require changes to contributors until the pull request is stable and can be integrated.

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, based on the conflict types already supported, we believe the tool could have automatically fixed 21% of the build conflicts in our sample if used by the respective development teams. All cases of the currently supported conflict types are similar, so the way the tool deals with them is satisfactory; for example, in all cases of *Duplicated Declaration* involving methods, these methods share the same method signature and body. So removing any of these duplicated methods fixes the conflict without semantically impacting the resulting code. Next, we present how the tool handles 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 ³¹, 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 a new reference for 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 occurs only in merge scenarios. As the merge commit owns the changes performed during a merge scenario, merge commits are required as input for our proposed tool. In the following sections, we explain how the particularities of build conflicts are reflected in the way the tool works.

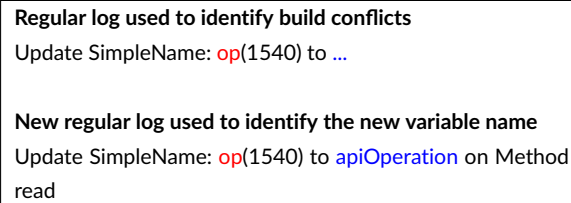
5.3.1 | Fault Localization

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, 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.

The tool adopts our scripts analyzing syntactic information of parent contributions to ensure the missing variable is an actual build conflict. 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. Similarly, analyzing only the right parent 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, one method's changes do not impact the other method. 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. This tool could also monitor the local Git repository to trigger the fault localization process for each new merge commit.

³¹Build ID swagger-api/swagger-core/65086450, Merge Commit 657b64b.



Regular log used to identify build conflicts
Update SimpleName: **op**(1540) to ...

New regular log used to identify the new variable name
Update SimpleName: **op**(1540) to **apiOperation** on Method read

FIGURE 3 Syntactic diff used to identify the new variable name

5.3.2 | Patch Creation

The tool further analyses the integrated code contributions to better understand the changes that lead to the missing variable declaration issue and create a patch. During this process, the tool must be aware that 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. As a solution, the tool proposes to update the dangling variable references to use the new variable name. In case of deleting the missing variable declaration, the tool proposes a solution that consists of reintroducing the deleted declaration.

The identification that the variable `op` is renamed, it is carried on by one of our study scripts, as explained before (see Section 3.3). For that, the scripts analyze the syntactic diff between the base and each parent commits (see the top of Figure 3). As the tool now must know the new name of the missing variable, we extend the previously mentioned scripts to obtain the required extra information (see bottom of Figure 3).

5.3.3 | Patch Validation

In this last step, the tool applies the required changes and then confirms that the build conflict is appropriately solved with developers. For that, the tool informs the build conflict cause, 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 remain. If no problem is detected, the tool creates 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 ³⁵.

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 a greater impact due to their wider visibility, it is worth studying them even if they do not frequently occur in projects that use continuous integration. Studying private developers' repositories would, nevertheless, be also important. Zhao et al.⁴⁵ have also experienced that. Their study observes a decreasing trend in builds with errors caused by missing classes and dependencies after projects adopt Travis.

We manually analyzed some reported build conflicts concerning the accuracy of our scripts, like the examples we discuss in this study, confirming they are true positives. Although we did not analyze all reported conflicts, they are detected using the same process. To detect the reported conflicts, we adopt 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 syntactic 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 manually analyzed cases, some interferences among contributions may not be verified. Instead of assuming these cases as build conflicts and introducing possible false positives in our results, we conservatively categorize them as broken builds caused by post-integration changes.

Regarding how our prototype works, it may adopt different solutions based on each conflict cause. Errors caused by *Unavailable Symbol* of variables are related to missing variable declarations in a specific scope. Depending on the circumstances that caused the missing declaration, the tool may adopt different fixes. If a variable declaration is removed, the tool recommends the reintroduction of the missing declaration. In the 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 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^{4,3,46,7}, we might have missed code integration scenarios, and conflicts, that reach public repositories but do not result in merge commits. This limitation might occur when using git commands such as *rebase*, *squash*, *cherry-pick*, or under smart kinds of fast-forward performed by git merge. It 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 analyzed the risky cases and confirmed that no imprecision occurred. At most, a conflict was classified as a post-integration change that led to a broken build. For resolution patterns, we analyze the syntactic differences applied to the fix commit. In both cases, we perform manual analyses to classify conflicts and resolution patterns. From all conflicts, 27% of them demanded manual analysis, while resolution patterns required manual analysis in 77% of all cases. Both analyses were rather simple and syntactic, and our scripts supported some steps 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. When we look for a specific class method in the diff, 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. Suppose both parents introduce two methods with the same name but different signatures, and in the merge commit, there are two methods with the same signature. In that case, the conflict is motivated by the integrator changes and not by contributors. 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 another researcher verified the procedures, 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 groups different projects regarding the domain, size, and the number of collaborators, most of them 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 with problems that cause the build breakage. After a pattern is identified, we perform additional analysis as explained in Section 3, to ensure a conflict causes the problem. For a new sample, different patterns can arise, demanding script adaptation to handle them. Build systems with less informative log reports, CI services, and tools with less accessible information could hinder replication. Besides that, we are not aware of how our choices could affect the results.

7 | RELATED WORK

Empirical studies about build conflicts provide evidence of their occurrence in practice. However, none of them investigate the *causes* and *resolution patterns* for build conflicts as we do here. Early studies of Kasi and Sarma⁴ and Brun et al.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 project's 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. As better explained earlier, many conflicts do not reach main public repositories as analyzed here. Once developers have this support locally, it is expected they sent their individual contributions to the main repository without problems. Thus, the main repository will always be 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 locally build 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. In their studies, Kasi and Sarma⁴ and Brun et al.³ do not clearly explain if they check the build process status of merge parent commits. Similarly, 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 in Java projects.

Sung et al.³⁰ investigate build conflicts, but when changes of different projects are integrated. In this way, conflicts arise when developers change their project (Edge) and then pull changes from the main project (Chromium) to sync their project with the main development line (*upstream*). To detect conflicts, they manually analyze commits of a C++ project for three months and classify conflicts based on their fixes. We also detect conflicts by a manual analysis in our study, but most of our conflicts are automatically detected (73%); for the conflicts detected by manual analysis, our scripts also provide information to guide the analysis and mitigate eventual errors. Although they investigate conflicts in a C++ project and adopt a different terminology, all conflict causes reported by them are mapped in our catalogue. They also report *Unavailable Symbol* is the most recurrent build conflict cause (67%), while the remaining causes present distributions in conformity with our findings. Furthermore, we report uncovered causes not reported by them like *Duplicated Declaration*, *Unimplemented Method*, and *Project Rules*.

Just like us, they also evaluate the feasibility of automatically fixing build conflicts for two causes (*Unavailable Symbol* and *Incompatible Types*). In contrast, our prototype supports three causes (*Unavailable Symbol*, *Unimplemented Method*, and *Duplicated Declaration*). For that, they propose a prototype tool called *MrgBldBrkFixer*; for each conflict detected, the tool analyzes the files involved in the conflict using GumTree⁴¹, suggests a fix for the error, and updates the required files with the fix. Although both prototypes follow a similar methodology, our prototype adopts a process completely automated, while *MrgBldBrkFixer* requires some manual steps done by the developer. For a build conflict caused by a variable rename, when using *MrgBldBrkFixer*, the developer must manually inform the missing variable name and the file where that variable should be available. Our prototype gets this information automatically based on the scripts used to run our study.

Several studies investigate the causes of errors in the build process^{32,33,47,26,25,24}, 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.³² 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 them (*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 importantly, 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 the associated fix.

Some studies have investigated build processes under the perspective of non-deterministic commit^{48,49,50}. In these cases, different build processes associated with a single commit present different results. It may occur 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 presents a broken status, we further analyze the contributions to ensure the conflict build occurrence. So the broken build status is just a starting 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, Vassallo et al.²⁵ group errors based on the build process phase that the error occurs. However, they do not observe if conflicting contributions among developers caused the build error. The studies identify general problem categories, but they do not go further and analyze each category; for example, compilation problems are classified as a category, but there are no specific causes. They also do not focus on the resolution patterns adopted to fix these broken builds. Kerzazi et al.³³ present a study investigating the impact of broken builds on software development. For that, they perform a qualitative and quantitative study. As one of our findings, the qualitative study reports that the primary cause of broken builds is missing classes (*Unavailable Symbol*). However, they do not investigate other causes like we do here and 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.

Finally, Ghaleb et al.⁵¹ investigate general build breakage causes in Travis CI builds. For that, they adopt a similar methodology to our work, as they analyze GitHub projects linked to Travis CI. Initially, they perform a manual analysis with selected builds to study the breakages' causes, extract error messages and categorize the causes. Next, they create heuristics to automate this process. As a result, they report environment factors cause 33% of the breakages. From all environment breakages, 79% are caused by *Internal CI errors* or *Issues related to external services and exceeding limits*. Although we do not focus here on studying environmental causes of broken builds, we use the mentioned causes to filter out merge scenarios associated with this kind of broken build (see Table 2 in Section 3). Their findings support our claims that we correctly classify build breakages and discard them when necessary.

Vassallo et al.⁵² present BART, a tool that relates to our build conflict repair tool. Initially, the authors investigate how developers could more easily understand broken build reports. They propose a tool that summarizes the causes of Maven projects' broken builds and suggests possible solutions linking online external resources. Regarding the summarized information of broken builds, both tools adopt a similar approach; for instance, for a broken build caused by a compilation problem, both tools present, when available, the main cause of the breakage, the class, and the line where the breakage takes place. However, the tools adopt different approaches when supporting developers. While BART suggests hints to fix the build breakage based on the build log and possibly searching for solutions online in discussion forums, our tool directly suggests fixes for the breakage. If the developer accepts the suggestion, the fix is automatically performed without further human intervention.

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 study, 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 for Java projects, 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 conflict 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 the 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 and bring ideas of new tools for helping developers treat such problems. As future work, we would like to evaluate private repositories and 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.

Acknowledgements

We thank Marcelo d'Amorim, Maurício Aniche, and Jacob Krueger for the quite pertinent suggestions that contributed to improving this work. We also would like to thank Samuel Brasileiro, Tales Tomaz, and Luís Santos for providing support on our prototype tool. This work is partially supported by INES (www.ines.org.br), CNPq grant 465614/2014-0, CAPES grant 88887.136410/2017-00, and FACEPE grants APQ-0399-1.03/17, IBPG-0692-1.03/17, and PRONEX APQ/0388-1.03/14.

References

1. Perry DE, Siy HP, Votta LG. Parallel changes in large-scale software development: an observational case study. *ACM Transactions on Software Engineering and Methodology* 2001; 10(3): 308–337. doi: 10.1145/383876.383878
2. Zimmermann T. Mining workspace updates in CVS. In: *International Workshop on Mining Software Repositories*. IEEE. ; 2007: 11–11
3. Brun Y, Holmes R, Ernst MD, Notkin D. Proactive detection of collaboration conflicts. In: *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering*. ACM. ; 2011: 168–178
4. Kasi BK, Sarma A. Cassandra: Proactive conflict minimization through optimized task scheduling. In: *International Conference on Software Engineering*. IEEE. ; 2013: 732–741
5. Apel S, Liebig J, Brandl B, Lengauer C, Kästner C. Semistructured merge: rethinking merge in revision control systems. In: *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering*. ACM. ; 2011: 190–200
6. Apel S, Leßenich O, Lengauer C. Structured merge with auto-tuning: balancing precision and performance. In: *International Conference on Automated Software Engineering*. ACM. ; 2012: 120–129
7. Cavalcanti G, Borba P, Accioly P. Evaluating and improving semistructured merge. *ACM on Programming Languages* 2017; 1(OOPSLA): 1–27. doi: 10.1145/3133883
8. Accioly P, Borba P, Cavalcanti G. Understanding semi-structured merge conflict characteristics in open-source java projects. *Empirical Software Engineering* 2018; 23(4): 2051–2085. doi: 10.1007/s10664-017-9586-1
9. Cavalcanti G, Borba P, Seibt G, Apel S. The impact of structure on software merging: semistructured versus structured merge. In: *International Conference on Automated Software Engineering*. IEEE. ; 2019: 1002–1013
10. Sarma A, Redmiles DF, Van Der Hoek A. Palantir: Early detection of development conflicts arising from parallel code changes. *Transactions on Software Engineering* 2011; 38(4): 889–908. doi: 10.1109/TSE.2011.64
11. Bird C, Zimmermann T. Assessing the value of branches with what-if analysis. In: *International Symposium on the Foundations of Software Engineering*. ACM. ; 2012: 1–11
12. McKee S, Nelson N, Sarma A, Dig D. Software practitioner perspectives on merge conflicts and resolutions. In: *International Conference on Software Maintenance and Evolution*. IEEE. ; 2017: 467–478
13. Grinter RE. Supporting articulation work using software configuration management systems. *Computer Supported Cooperative Work* 1996; 5(4): 447–465.
14. De Souza CR, Redmiles D, Dourish P. " Breaking the code", moving between private and public work in collaborative software development. In: *International ACM SIGGROUP Conference on Supporting Group Work*. ACM. ; 2003: 105–114

15. Adams B, McIntosh S. Modern release engineering in a nutshell—why researchers should care. In: International Conference on Software Analysis, Evolution, and Reengineering. IEEE. ; 2016: 78–90
16. Henderson F. Software engineering at google. *arXiv preprint arXiv:1702.01715* 2017.
17. Potvin R, Levenberg J. Why Google stores billions of lines of code in a single repository. *Communications of the ACM* 2016; 59(7): 78–87. doi: 10.1145/2854146
18. Bass L, Weber I, Zhu L. *DevOps: A Software Architect's Perspective* . 2015.
19. Fowler M. Feature Toggle. <https://martinfowler.com/bliki/FeatureToggle.html>; 2017. Accessed: January 2020.
20. Hodgson P. 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/>; 2017. Accessed: January 2020.
21. Fowler M. Feature Branch. <https://martinfowler.com/bliki/FeatureBranch.html>; 2009. Accessed: January 2020.
22. Hodgson P. Feature Toggles (aka Feature Flags). <https://martinfowler.com/articles/feature-toggles.html>; 2017. Accessed: January 2020.
23. Cavalcanti G, Accioly P, Borba P. Assessing semistructured merge in version control systems: A replicated experiment. In: International Symposium on Empirical Software Engineering and Measurement. IEEE. ; 2015: 1–10
24. Hassan AE, Zhang K. Using decision trees to predict the certification result of a build. In: International Conference on Automated Software Engineering. IEEE. ; 2006: 189–198
25. Vassallo C, Schermann G, Zampetti F, et al. A tale of CI build failures: An open source and a financial organization perspective. In: International Conference on Software Maintenance and Evolution. IEEE. ; 2017: 183–193
26. Rausch T, Hummer W, Leitner P, Schulte S. 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: 345–355
27. Hilton M, Tunnell T, Huang K, Marinov D, Dig D. Usage, costs, and benefits of continuous integration in open-source projects. In: International Conference on Automated Software Engineering. IEEE. ; 2016: 426–437.
28. Vasilescu B, Van Schuylenburg S, Wulms J, Serebrenik A, Brand v. dMG. Continuous integration in a social-coding world: Empirical evidence from GitHub. In: International Conference on Software Maintenance and Evolution. IEEE. ; 2014: 401–405
29. Shao D, Khurshid S, Perry DE. Evaluation of semantic interference detection in parallel changes: an exploratory experiment. In: International Conference on Software Maintenance. IEEE. ; 2007: 74–83
30. Sung C, Lahiri SK, Kaufman M, Choudhury P, Wang C. 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: 172–181
31. Van Der Veen E, Gousios G, Zaidman A. Automatically prioritizing pull requests. In: Mining Software Repositories. IEEE. ; 2015: 357–361
32. Seo H, Sadowski C, Elbaum S, Aftandilian E, Bowdidge R. Programmers' build errors: a case study (at google). In: International Conference on Software Engineering. ACM. ; 2014: 724–734
33. Kerzazi N, Khomh F, Adams B. Why do automated builds break? an empirical study. In: International Conference on Software Maintenance and Evolution. IEEE. ; 2014: 41–50
34. Gousios G, Zaidman A, Storey MA, Van Deursen A. Work practices and challenges in pull-based development: The integrator's perspective. In: International Conference on Software Engineering. IEEE. ; 2015: 358–368
35. Da Silva L, Borba P, Pires A. Online Appendix. <https://spgroup.github.io/papers/build-conflicts.html>; . Accessed: March 2021.
36. Maven. URL:<https://maven.apache.org>; . Accessed: January 2020.
37. Munaiah N, Kroh S, Cabrey C, Nagappan M. Curating github for engineered software projects. *Empirical Software Engineering* 2017; 22(6): 3219–3253. doi: 10.1007/s10664-017-9512-6

38. Beller M, Gousios G, Zaidman A. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In: International Conference on Mining Software Repositories. IEEE. ; 2017: 447–450
39. Gradle. <https://gradle.org/>; . Accessed: January 2020.
40. Chacon S, Straub B. *Pro git* . 2014.
41. Falleri JR, Morandat F, Blanc X, Martinez M, Monperrus M. Fine-grained and accurate source code differencing. In: International Conference on Automated Software Engineering. ACM. ; 2014: 313–324
42. Rodríguez-Pérez G, Robles G, Serebrenik A, Zaidman A, Germán DM, Gonzalez-Barahona JM. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering* 2020; 25(2): 1294–1340. doi: 10.1007/s10664-019-09781-y
43. Rocha T, Borba P, Santos JP. Using acceptance tests to predict files changed by programming tasks. *Journal of Systems and Software* 2019; 154: 176–195. doi: 10.1016/j.jss.2019.04.060
44. Le Goues C, Forrest S, Weimer W. Current challenges in automatic software repair. *Software quality journal* 2013; 21(3): 421–443. doi: 10.1007/s11219-013-9208-0
45. Zhao Y, Serebrenik A, Zhou Y, Filkov V, Vasilescu B. The impact of continuous integration on other software development practices: a large-scale empirical study. In: International Conference on Automated Software Engineering. IEEE. ; 2017: 60–71
46. Leßenich O, Siegmund J, Apel S, Kästner C, Hunsen C. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering* 2018; 25(2): 279–313. doi: 10.1007/s10515-017-0227-0
47. Beller M, Gousios G, Zaidman A. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In: International Conference on Mining Software Repositories. IEEE. ; 2017: 356–367
48. Labuschagne A, Inozemtseva L, Holmes R. 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: 821–830
49. Herzig K, Nagappan N. Empirically detecting false test alarms using association rules. In: International Conference on Software Engineering. IEEE. ; 2015: 39–48
50. Elbaum S, Rothermel G, Penix J. Techniques for improving regression testing in continuous integration development environments. In: ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM. ; 2014: 235–245
51. Ghaleb TA, Costa dDA, Zou Y, Hassan AE. Studying the Impact of Noises in Build Breakage Data. *IEEE Transactions on Software Engineering* 2019. doi: 10.1109/TSE.2019.2941880
52. Vassallo C, Proksch S, Zemp T, Gall HC. Every build you break: Developer-oriented assistance for build failure resolution. *Empirical Software Engineering* 2019; 1–40. doi: 10.1007/s10664-019-09765-y
53. Beck K. *Extreme programming explained: embrace change* . 2000.
54. Biehl JT, Czerwinski M, Smith G, Robertson GG. FASTDash: a visual dashboard for fostering awareness in software teams. In: Conference on Human factors in Computing Systems. ACM. ; 2007: 1313–1322
55. Brun Y, Holmes R, Ernst MD, Notkin D. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering* 2013; 39(10): 1358–1375. doi: 10.1109/TSE.2013.28
56. Duvall PM. *Continuous integration* . 2007.
57. Miller A. A hundred days of continuous integration. In: Agile 2008 Conference. IEEE. ; 2008: 289–293
58. Muylaert W, De Roover C. Prevalence of botched code integrations. In: International Conference on Mining Software Repositories. IEEE. ; 2017: 503–506
59. Nagappan M, Zimmermann T, Bird C. Diversity in software engineering research. In: Joint Meeting on Foundations of Software Engineering. ACM. ; 2013: 466–476

60. Vasilescu B, Yu Y, Wang H, Devanbu P, Filkov V. Quality and productivity outcomes relating to continuous integration in GitHub. In: Joint Meeting on Foundations of Software Engineering. ACM. ; 2015: 805–816

How to cite this article: Da Silva L., Borba P., Pires A. (2021), Build Conflicts in the Wild, *Softw: Pract Exper.*, 2021;00:1–24.