A Characterization Study of Merge Conflicts in Java Projects

BOWEN SHEN, Virginia Tech, United States
MUHAMMAD ALI GULZAR, Virginia Tech, United States
FEI HE, Tsinghua University, China
NA MENG, Virginia Tech, United States

In collaborative software development, programmers create software branches to add features and fix bugs tentatively, and then merge branches to integrate edits. When edits from different branches textually overlap (i.e., *textual conflicts*) or lead to compilation and runtime errors (i.e., *build and test conflicts*), it is challenging for developers to remove such conflicts. Prior work proposed tools to detect and solve conflicts. They investigate how conflicts relate to code smells and the software development process. However, many questions are still not fully investigated, such as what types of conflicts exist in real-world applications and how developers or tools handle them. For this paper, we used automated textual merge, compilation, and testing to reveal 3 types of conflicts in 208 open-source repositories: textual conflicts, build conflicts (i.e., conflicts causing build errors), and test conflicts (i.e., conflicts triggering test failures). We manually inspected 538 conflicts and their resolutions to characterize merge conflicts from different angles.

Our analysis revealed three interesting phenomena. First, higher-order conflicts (i.e., build and test conflicts) are harder to detect and resolve, while existing tools mainly focus on textual conflicts. Second, developers manually resolved most higher-order conflicts by applying similar edits to multiple program locations; their conflict resolutions share common editing patterns implying great opportunities for future tool design. Third, developers resolved 64% of true textual conflicts by keeping complete edits from either a left or right branch. Unlike prior studies, our research for the first time thoroughly characterizes three types of conflicts, with a special focus on higher-order conflicts and limitations of existing tool design. Our work will shed light on future research of software merge.

CCS Concepts: • **General and reference** → **Empirical studies**; • **Software and its engineering** → *Collaboration in software development*; Software maintenance tools.

Additional Key Words and Phrases: empirical, software merge, conflict detection, conflict resolution

1 INTRODUCTION

"Integration Hell" refers to the scenarios where developers integrate or *merge* big chunks of code changes from software branches right before delivering a software product [20]. In practice, this integration process is rarely smooth and seamless due to *conflicts*, which can take developers hours or even days to debug so that branches can finally merge [12]. To avoid "Integration Hell", an increasing number of developers use Continuous Integration (CI) to integrate code frequently (e.g., once a day) and to verify each integration via automated build (i.e., compilation) and testing [54, 57]. Nevertheless, CI practices do not eliminate the challenges posed by merge conflicts. Developers still rely on the merge feature of version control systems (e.g., git-merge [25]) to automatically (1) integrate branches and (2) reveal conflicts that require manual resolution. *Such text-based merge*

Authors' addresses: Bowen Shen, bowenshe@vt.edu, Virginia Tech, Blacksburg, Virginia, United States; Muhammad Ali Gulzar, gulzar@vt.edu, Virginia Tech, Blacksburg, Virginia, United States; Fei He, hefei@mail.tsinghua.edu.cn, Tsinghua University, Beijing, China; Na Meng, nm8247@vt.edu, Virginia Tech, Blacksburg, Virginia, United States.

1049-331X/2022/7-ART $15.00
https://doi.org/10.1145/3546944

usually produces numerous false positives and false negatives. For example, when two branches reformat the same line in divergent ways (e.g., add vs. delete a whitespace), git-merge reports a *textual conflict* even though such a conflict is unimportant and poses no syntactic or semantic difference. Meanwhile, when two branches edit different lines modifying the program semantics in conflicting ways, git-merge silently applies both edits without reporting any conflict.

Background. In prior literature, many tools were proposed to improve text-based merge [27, 28, 30, 39, 46, 59]. For instance, FSTMerge [28] models program entities (e.g., classes and methods) as unordered tree nodes, matches entities based on their signatures, and uses text-based merge to integrate edits inside matched entities. JDime [27, 39] extends FSTMerge by modeling both program entities and statements in its tree representation. It applies tree matching and amalgamation algorithms to integrate edits. Given textually conflicting edits, AutoMerge [59] enumerates all possible combinations of the edit operations from both branches and recommends alternative conflict resolutions. However, all of these tools only compare edits applied to the same program entity; they do not check whether the co-application of edits to different entities can trigger any compilation or testing error. Crystal [30] overcomes this limitation by building and testing tentatively merged code to reveal ***higher-order conflicts*** (i.e., build conflicts and test conflicts).

Motivation. Despite the existence of diverse tools, some fundamental research questions (RQs) are yet to be explored, including

- RQ1: How were the three types of conflicts (i.e., textual, build, and test conflicts) introduced in real-world applications?
- RQ2: What are developers' resolution strategies for different types of conflicts?
- RQ3: What characteristics of conflicts are overlooked by existing tool design?

Exploring these questions is important for two reasons. First, by contrasting the characteristics of merge conflicts with the focus of existing merge tools, we can reveal the critical aspects of conflicts overlooked by such tools. Second, by examining how conflicts were introduced and resolved by developers, we can motivate new tools to address conflicts by mimicking developers' practices or bringing humans into the loop.

Methodology. To investigate the RQs, we conducted a comprehensive characterization study of merge conflicts. A typical **merging scenario** in software version history involves four program commits: base version $\boldsymbol{b}$, left branch $\boldsymbol{l}$, right branch $\boldsymbol{r}$, and developers' merge result $\boldsymbol{m}$ (see Fig. 2). To crawl for merging scenarios in version history, we searched for any commit that has two parent commits. If a commit c has two parent commits, we consider c to be developers' merge result m, treat the first parent commit as l, and regard the second parent as r. To identify the common base version b, we applied the command "`git merge-base`" to the two parent commits.

As shown in Fig. 1, we took a two-phase approach to extract and analyze merge conflicts in Java open-source software repositories. In Phase I, starting from an existing merging scenario, we applied git-merge, automated build, and automated testing in sequence. With git-merge, we tentatively generated a text-based merged version $\boldsymbol{A_m}$ from l and r (see Fig. 2). We then built A_m and tested it with developers' predefined test cases. When any of these steps failed, we concluded l and r to have textual, build, or test conflicts. For deeper analysis, in Phase II, we manually inspected a sample set of scenarios with textual conflicts and all scenarios with build or test conflicts. For each of these scenarios, we compared the five program versions involved—b, l, r, m, and A_m—to comprehend the root cause and resolution of conflict(s).

Our Results. In our study, we mined the software version history of 208 popular open-source Java projects, and found 117,218 merging scenarios (i.e., any code commit with two parent commits) in those repositories. With the two-phase methodology mentioned above, we identified 15,886 merging scenarios with textual conflicts, 79 scenarios with build conflicts, and 33 scenarios with test conflicts. Due to the huge number of revealed textual

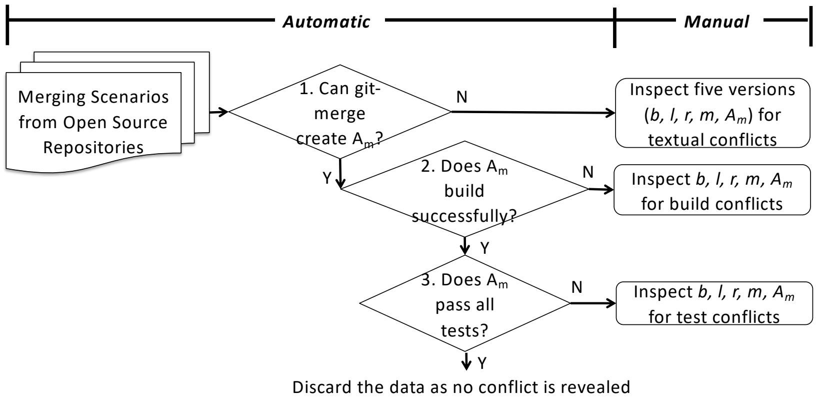
Automatic
Manual
Merging Scenarios from Open Source Repositories
1. Can git-merge create A_m?
N
Inspect five versions (b, l, r, m, A_m) for textual conflicts
Y
2. Does A_m build successfully?
N
Inspect b, l, r, m, A_m for build conflicts
Y
3. Does A_m pass all tests?
N
Inspect b, l, r, m, A_m for test conflicts
Y
Discard the data as no conflict is revealed

Fig. 1. Workflow of our hybrid approach

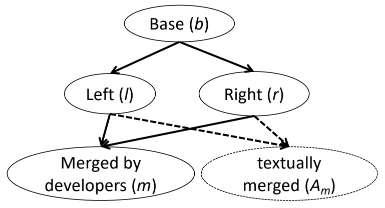

Fig. 2. Software versions related to a merging scenario

conflicts, we randomly picked 385 conflicts from distinct merging scenarios for manual inspection. We also manually analyzed all revealed build and test conflicts to characterize the root cause and developers' conflict resolutions. The major findings are as below:

- **How were conflicts introduced?** 65 out of 385 inspected textual conflicts are false positives because l and r edit adjacent lines instead of the same lines. 18 of the 65 false positives are located in non-Java files (e.g., pom.xml). Build conflicts occurred when the co-application of edits from l and r broke any def-use link between program elements (e.g., classes or libraries). For instance, when l updates a method from `foo()` to `bar()` and r adds a call `foo()`, the co-application of both edits can break a def-use link as it leads to a mismatch between the use (i.e., call) and def (i.e., declaration) of `foo()`. 85% (39/46) of test conflicts happened, as the co-application of edits broke def-use links between elements or led to mismatches between test oracles and software implementation.
- **How did developers manually resolve merge conflicts?** Within the 320 true textual conflicts, developers resolved most conflicts (i.e., 206) by exclusively keeping changes from one branch. However, developers resolved most of the higher-order conflicts by combining all edits from both branches with additional edits. The additional edits solve build or test errors by repairing broken def-use links or fixing mismatches between implementation and tests. Such edits present systematic editing patterns that modify similar code in similar ways.
- **What conflicts cannot be handled by current tools?** Existing tools detect textual conflicts in non-Java software artifacts (e.g., readme.txt) with relatively low precision (72%) and thus unable to resolve true textual conflicts fully automatically. When textual conflicts exist and A_m could not be generated, neither compilation nor testing is applicable for conflict detection. Even if A_m is available, compilation and testing can only reveal the symptoms (e.g., errors or failures) triggered by higher-order conflicts instead of the precise root causes. There is insufficient tool support for the detection and resolution of higher-order conflicts.

The insights from this study enlighten future software merge tools and suggest new research directions in related areas like systematic editing and change recommendation. The program and data presented in this work are publicly available at https://figshare.com/s/c174e1ffd2ad02b15211.

2 BACKGROUND

This section defines terminology used in this paper (Section 2.1), and introduces existing tools (Section 2.2).

Table 1. Exemplar merge conflicts

Changes in *l*	**Changes in *r***
(a) Textual conflict	
`- private int a=4;`	`- private int a=4;`
`+ private int a=40;`	**`+ private int a=20;`**
(b) Build conflict	
`  public void foo() {`	`  public class C {`
`+   C.m();`	`-   public static void m(){...}`
`  ... }`	**`+  public static void m(int p){...}}`**
(c) Test conflict	
`  y = foo(x);`	`- y = foo(x);`
	`+ y = foo(x) + 1;`
`- if(y < 13){...}`	`  if(y < 13){...}`
`+ if(++y < 13){...}`	
`  else error()`	`  else error()`

Table 2. Overview of current merge tools

	Detection	**Resolution**
Textual conflicts	git-merge, FSTMerge, JDime, IntelliMerge, AutoMerge, Crystal, WeCode, CloudStudio	FSTMerge, JDime, IntelliMerge, AutoMerge
Build conflicts	Crystal, WeCode, CloudStudio	-
Test conflicts	Crystal, WeCode, SafeMerge	-

2.1 Terminology

When developers merge two branches (e.g., *l* and *r*) in a software repository, it may lead to three types of conflicts.

1. Textual Conflicts exist when *l* and *r* edit the same line in divergent ways. As illustrated by Table 1 (a), since *l* and *r* update the same statement with conflicting values (e.g., `40 vs. 20`), there is a textual conflict between the branches.

2. Build Conflicts occur when the edits of *l* and *r* do not conflict textually, but their co-application triggers a *compilation* or *build* error. In Table 1 (b), when *l* adds an invocation to `C.m()` and *r* simultaneously updates the signature of `C.m()`, the edit integration causes an unresolved method reference.

3. Test Conflicts occur when the edits of *l* and *r* do not conflict textually and do not cause any compilation issue, but the co-application of both edits triggers a *runtime error* or *test error*. For example, in Table 1 (c), although the edits in *l* and *r* separately increment `y` by `1`, applying both edits will increment the variable by `2` and thus fail the related tests.

2.2 Existing Tools for Software Merge

Various tools were proposed to detect or resolve merge conflicts [21, 27, 28, 30–32, 36, 39, 46, 55, 56, 59]. For brevity, in this paper, we focus our discussion on the state-of-the-art tools proposed in the past ten years.

FSTMerge [21, 28, 32] is also known as **semistructured merge**. It constructs abstract syntax trees (ASTs) of programming languages (i.e., Java, C#, and Python), and converts each AST to a *program structure tree* by removing the AST subtrees of program statements. Using program structure trees, FSTMerge matches nodes between *l* and *r* purely based on the class or method signatures. It then integrates the edits inside matching nodes via textual merge. By modeling and comparing Java classes and methods, FSTMerge could align code better than textual merge and thus reports conflicts with better accuracy [28, 32].

JDime [27] is also known as **structured merge**. Similar to FSTMerge, JDime matches Java methods and classes based on program structure trees. However, unlike FSTMerge, JDime creates ASTs and conducts tree-based merge instead of text-based merge for each matching pair. A recent study shows that such structured merge could report conflicts more precisely than semistructured merge [33], mainly because structured merge observes the syntactic structures when matching program statements.

AutoMerge [59] is similar to JDime, in the sense that it also detects conflicts based on AST comparison. However, AutoMerge goes beyond reporting conflicts and attempts to resolve conflicts by proposing alternative strategies to merge *l* and *r*, with each strategy integrating some edits from both branches.

Similar to FSTMerge, IntelliMerge [55] also matches program elements (e.g., Java classes, methods, and fields) based on the syntactic structures. For each matching pair, IntelliMerge integrates the edits inside matching nodes via textual merge. However, unlike FSTMerge, IntelliMerge detects refactoring edits (e.g., method renaming) applied to software branches and considers those edits when matching program elements. In this way, IntelliMerge reports conflicts with higher precision and higher recall than FSTMerge [55].

SafeMerge [56] takes in four program versions related to a merging scenario: b, l, r, and m and statically infers the relational postconditions of distinct versions to model program semantics. Afterwards, SafeMerge compares all postconditions to decide whether l and r are *free of conflicts* (i.e., m does not introduce new unwanted behaviors). Like the tools mentioned above, SafeMerge only examines edits applied to the same program entity for potential conflicts. It does not relate edits applied to distinct program entities to reveal more conflicts.

Crystal [30, 31] takes three steps to reveal different kinds of conflicts. As with our Phase I, Crystal first applies textual merge to l and r to reveal any textual conflict. Next, it exploits automatic build and testing to reveal any higher-order conflict.

WeCode [36] is similar to Crystal. It continuously merges the committed and uncommitted changes in software branches to raise developers' awareness of potential conflicts even before program edits are all committed, and branches get merged.

CloudStudio [34] is similar to Crystal and WeCode, in the sense that they are all awareness systems. These systems notify developers of the potential conflicts their branches can have with the branches of other developers. As a web-based IDE, CloudStudio monitors developers' coding activities, and reports textual conflicts when two developers modify the same line. It also reports build conflicts by trying to merge simultaneously applied edits, and to compile the merged version.

To facilitate understanding, we visualize the task categories covered by each tool in Table 2. This table reflects the solution domain for software merge conflicts. However, the table does not characterize the problem domain (i.e., the conflicts exist in reality), neither does it map a tool's applicability to different characteristics of conflicts. Without such mapping, it is hard to tell (1) what kind of conflicts are well captured by the methodologies implemented by current tools and (2) what new methodologies are still needed. **Therefore, our study intends to systematically characterize merge conflicts and their resolutions, and to compare them against the application scope of existing techniques, while shedding light on future research directions**.

3 STUDY APPROACH

Our approach has two phases (see Fig. 1). Phase I uses automatic approaches to discover conflicts (Section 3.1). Phase II relies on manual inspection to comprehend the revealed conflicts and developers' resolutions (Section 3.2).

3.1 Phase I: Automatic Detection of Conflicts

To ensure the representativeness of our study, we ranked Java projects on GitHub based on their popularity (i.e., star counts). We cloned repositories for the top 1,000 projects as our initial dataset. We then refined the dataset with two heuristics. First, we only kept the projects that can be built with Maven [22], Ant [24], or Gradle [19], because these three build tools are popularly used and we will rely on them to build and test each naïvely merged version A_m. In particular, Gradle projects typically contain build files named *.gradle; Maven and Ant projects often have build files separately named as pom.xml and build.xml. We recognized the projects that can be built with Maven, Ant, or Gradle based on the existence of corresponding build files. Second, we removed tutorial projects as they are not real Java applications and may not show real-world merging scenarios. Namely, if the readme file of any project contains some description like "this tutorial project serves for learning purposes", we removed the project.

Consequently, our refined dataset comprises 208 repositories. Among the 208 repositories, we identified 117,218 merging scenarios by searching for any commit with two parent/predecessor commits. In each identified merging scenario, we regard the first and second parent commits as l and r separately. We treat their common child and ancestor commits as m and b. Fig. 3 shows the distribution of projects based on their merging-scenario counts. According to this figure, 3 projects contain no more than 2 merging scenarios, while 2 projects contain more

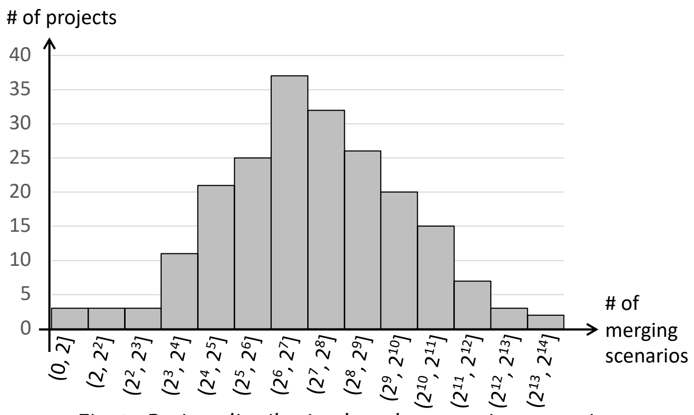

Fig. 3. Project distribution based on merging scenarios

Table 3. The merging scenarios with conflicts

		Textual	**Build**	**Test**
Phase I	**# of merging scenarios**	15,886	79	33
	# of projects	183	37	22
	Max # of scenarios per repository	4,172	6	4
Phase II	**# of inspected conflicts**	385	107	46

than 2^{13} or 8,192 scenarios. Among the 14 count intervals, $(2^6, 2^7]$ corresponds to the most projects (i.e., 37). The median count per project is 131, which means that merging scenarios exist widely in the subject software repositories.

Although some of the 208 repositories have few merging scenarios (e.g., less than 10) or are maintained by single developers, we did not further refine this dataset for two reasons. First, if the developers of some repositories do not create or merge branches very often, our dataset can cover the merging practices by those developers to be representative. Second, if some developers eagerly create and merge branches even though they are the solo contributors of their projects, we believe it also important to cover the merging scenarios in their repositories. Because different projects may define test files in different ways, we leveraged the regular expression "*Test*.java" to search for test files in the latest versions of 208 repositories. We found test files in 191 repositories; only 17 repositories have no test file. These numbers indicate that test files popularly exist in the subject repositories; they are usually available when testing is required to reveal test conflicts.

As illustrated in Fig. 1, in Phase I, we process each merging scenario by taking three steps sequentially. In *Step 1*, we apply git-merge to l and r to generate a text-based merged version A_m. If this trial fails, git-merge reports all textual conflicts, and we record that scenario. Otherwise, if both l and r build smoothly and we successfully generate A_m then, in *Step 2*, we attempt to build A_m. If the attempt fails, we log all build errors and label the merging scenario to have build conflicts. Otherwise, if A_m builds successfully and both l and r pass developers' test cases, then in *Step 3*, we further execute the successfully built version of A_m with developers' test suite. If the program fails any test, we log all runtime errors and label the scenario to have test conflicts; otherwise, we skip the merging scenario as no conflict is revealed. By applying such a three-step method, we marked 15,886 merging scenarios with textual conflicts, 79 scenarios with build conflicts, and 33 scenarios with test conflicts. As shown in Table 3, these three types of scenarios are separately from 183, 37, and 22 repositories. We found at most 4,172, 6, and 4 merging scenarios related to individual conflicts per repository. All reported numbers imply the prevalence of merge conflicts. Textual conflicts were reported more often than the other two types of conflicts.

3.2 Phase II: Manual Inspection

We inspected the merge conflicts to investigate the three research questions mentioned in Section 1. Due to a large number of merging scenarios with textual conflicts (i.e., 15,886), it is infeasible to analyze each conflict manually. We thus decided to sample 385 textual conflicts to reduce manual effort while ensuring the representativeness of our observations. In particular, when 15,886 merging scenarios contain in total thousands or millions of textual conflicts, 385 is a statistically significant sample size with 95% confidence level and ±5% confidence interval [23, 41]. To construct our sample set, we randomly picked one or more merging scenarios in each of the

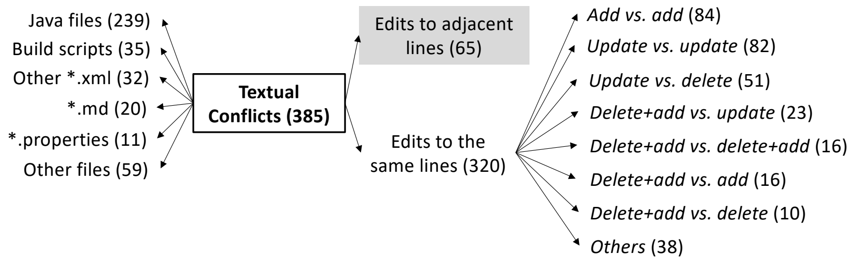

Fig. 4. Characterization of root causes for textual conflicts

183 repositories and examined one of the reported textual conflicts for each scenario. To manually analyze every sampled conflict, we studied five related program versions: b, l, r, m, and A_m.

We inspected all build conflicts without sampling because of only a few such scenarios (i.e., 79). Different from git-merge, build tools report compilation errors but never pinpoint the conflicting edits. To locate and understand those edits that correspond to each compilation error, we checked five program versions (i.e., b, l, r, m, and A_m) and inspected edits applied to distinct locations. If the co-application of certain edits from l and r is responsible for the error, we identified *the minimum set of involved edits* as the root cause of a build conflict. Similarly, we inspected all scenarios labeled with test conflicts (i.e., 33). Automated testing reports runtime errors but does not locate any conflicting edits. To reveal those edits, we inspected all five program versions to compare the semantics and identified *the minimum set of responsible edits*.

With more details, to locate the root cause of a higher-order conflict, we checked whether developers' merged version m had any build or test error. If both A_m and m had build or test errors, we could not decide how those errors were introduced or resolved. In such scenarios, we checked three more commits after m in the software history. If none of these additional commits resolved the build or test errors introduced earlier, we concluded that the merging scenario had some unknown build/test conflicts. Otherwise, if m or any of the later commits being checked had zero build/test error, we compared A_m with that commit to locate conflicts.

To ensure the quality of our manual inspection, two authors independently examined the sampled textual conflicts and all merging scenarios with build or test failures. The authors compared their description on root causes and resolutions for all conflicts and extensively discussed any disagreement until reaching a consensus. As shown in Table 3, we sampled 385 textual conflicts, and manually located 107 build conflicts as well as 46 test conflicts in the 112 (i.e., 79+33) examined scenarios. Notice that one merging scenario can have one or more higher-order conflicts. Thus, the total number of higher-order conflicts (i.e., 107+46) is larger than the total number of examined scenarios (i.e., 112). We report our empirical findings for different conflict types in Sections 4–6.

4 STUDY RESULTS ON TEXTUAL CONFLICTS

This section presents our analysis of the 385 textual conflicts to answer each research question.

4.1 RQ1: Root Causes of Conflicts

We studied three characterisitics of each conflict: the edited files, relative edit locations, and edit types (see Fig. 4). In terms of edited files, the 385 textual conflicts can be classified into 6 categories based on where they arise: 239 conflicts occur in Java code; 35 conflicts exist in build scripts (i.e., build.xml, pom.xml, and *.gradle); 32 conflicts are found in other XML files (e.g., plugin.xml and web.xml); 20 conflicts are in MARKDOWN files (i.e., *.md), 11 conflicts are located in property files (i.e., *.properties), and 59 conflicts happen in other files (e.g., code written in other programming languages). Most conflicts occur in Java code, resonating with the fact that we inspected merging scenarios in Java projects only, and developers mainly modified code via commits.

In terms of relative edit locations, we discovered two types of conflicts: (i) in 65 conflicts, the edits from l and r were applied to adjacent instead of same lines; (ii) in the other 320 conflicts, the edits from different branches commonly cover at least one line. Table 4 presents an exemplar conflict of type (i) in (a), and conflict examples of type (ii) in (b)–(i). Conflicts in the first category are false positives generated by git-merge, because the edits do not overlap and can be co-applied theoretically. These conflicts do not satisfy the definition of textual conflicts mentioned in Section 2.1. Among the 65 false positives, there are 18 instances located in non-Java files (i.e., build scripts and other files) and 47 instances in Java files. In the following exploration, we focused our analysis on the 320 true conflicts.

In terms of edit types, we further classified the true conflicts into eight categories: (1) add vs. add, (2) update vs. update, (3) update vs. delete, (4) delete+add vs. update, (5) delete+add vs. delete+add, (6) delete+add vs. add, (7) delete+add vs. delete, and (8) others. Given an added line and a deleted line, if they have at least 50% of string similarity, our manual inspection considers them to present an update operation; otherwise, our manual inspection treats them as two separate edit operations: one insertion and one deletion. ***Add vs. add*** means l and r insert divergent content at the same location, as shown by the example in Table 4 (b). ***Update vs. update*** means l and r update the same line(s) in divergent ways (e.g., Table 4 (c)). ***Update vs. delete*** means between l and r, one branch updates some text while the other deletes the same text. As shown in Table 4 (d), l deletes a Java file while r updates content of that file. These three edit types take up the majority of 320 conflicts (i.e., 68%).

Delete+add vs. update and ***delete+add vs. delete+add*** are similar to the edit type *update vs. update*, as these three types all involve simultaneous line addition and deletion in both branches. In our research, we differentiate *update* from *deletion+add* based on the observed textual similarity. Namely, if in a branch the inserted lines are similar to deleted ones, we label the edits *update*; otherwise, the label is *delete+add*. In Table 4, (e)–(f) present examples for both types. Additionally, ***delete+add vs. add*** means that one branch deletes and adds lines while the other branch inserts lines to the same edit location. As shown in Table 4 (g), l deletes two imports and adds another import, and r adds two imports. Because the edit locations by both branches overlap (i.e., the last import addition), git-merge reports a conflict. ***Delete+add vs. delete*** means that one branch deletes and adds lines while the other branch deletes lines; the two branches have overlapping edit locations (see Table 4 (h)). ***Others*** include miscellaneous conflicts. For example, as shown in Table 4 (i), l updates a line and adds extra lines to implement new logic; r replaces two statements with three semantically similar statements. We consider such conflicts as *update+add vs. update* and put them into the miscellaneous category.

We defined the eight categories mainly due to the larger proportion of the first seven types of conflicts. *Others* include all smaller types of edits, each of which covers less than 10 conflicts.

RQ1 answer summary: *Among the inspected 385 textual conflicts, 146 conflicts exist in non-Java files; 65 conflicts are false positives by git-merge; 166 true conflicts are caused by divergent insertions or updates between branches (i.e., add vs. add, and update vs. update).*

4.2 RQ2: Resolutions of Conflicts

We observed that developers may react to a given conflict in one of the following eight ways:

- **L**: Keep all edits from l.
- **R**: Keep all edits from r.
- **M**: Apply new edits that are not from l or r.
- **L+R**: Keep edits from both sides.
- **L+M**: Keep edits from l and apply extra edits as needed. Compared with L, L+M may keep some instead of all edits from l and/or add new edits.
- **R+M**: Keep the edits from r and apply extra edits as needed. Compared with R, R+M may keep some instead of all edits from r and/or add new edits.

Table 4. Nine exemplar conflicts reported by git-merge

(a) A reported conflict due to edits to adjacent lines [1]	

Changes in ***l***:
  import java.io.IOException;
  import java.io.InputStream;
  import java.util.List;
  import java.util.Map;
- import java.util.concurrent.Executors;
	Changes in ***r***:
	  import java.io.IOException;
	- import java.io.InputStream;
	- import java.util.List;
	- import java.util.Map;
	  import java.util.concurrent.Executors;

(b) A conflict of type *add vs. add* [14]	

Changes in ***l***:
  import org.apache.dubbo.common.URL;
+ import org.apache.dubbo.common.utils.StringUtils;
	Changes in ***r***:
	  import org.apache.dubbo.common.URL;
	+ import org.apache.dubbo.common.Version;

(c) A conflict of type *update vs. update* [3]	
Changes in ***l***: <artifactId>jackson-jaxrs </art ifactId> - <version>1.8.1</version> **+ <version>1.9.7</version>** <scope>test</scope>	Changes in ***r***: <artifactId>jackson-jaxrs </art ifactId> - <version>1.8.1</version> **+ <version>1.9.3</version>** <scope>test</scope>
(d) A conflict of type *update vs. delete* [15]	
Changes in ***l***: Delete the whole file TestTryCatchNoMove.java	Changes in ***r***: Update content of the file TestTryCatchNoMove.java.
(e) A conflict of type *delete+add vs. update* [13]	
Changes in ***l***: Delete and add many lines in a Java method of Server-Connection.java. The edited area overlaps with that of *r*.	Changes in ***r***: - if(tableKey.equalsIgnoreCase(table) && itemConfig!=null){ **+ if((tableKey.equalsIgnoreCase(table) &&** **itemConfig!=null) || tableKey.equals("*")){**
(f) A conflict of type *delete+add vs. delete+add* [7]	
Changes in ***l***: Delete 40 lines and insert 3 lines in mycat.xml. The edited region overlaps with that of *r*.	Changes in ***r***: Delete 29 lines and add 1 line in mycat.xml.
(g) A conflict of type *delete + add vs. add* [2]	

Changes in ***l***:
 	
- import java.util.Collection;
- import java.util.ConcurrentModificationException;
+ import java.util.Iterator;
  import java.util.Map;
	Changes in ***r***:
	+ import java.util.ArrayList;
	  import java.util.Collection;
	  import java.util.ConcurrentModificationException;
	+ import java.util.List;
	  import java.util.Map;

(h) A conflict of type *delete+add vs. delete* [5]	
Changes in ***l***: - import azkaban.executor.ExecutableFlow.FailureAction; - import azkaban.executor.ExecutableFlow.Status; **+ import azkaban.executor.ExecutionOptions;**	Changes in ***r***: import azkaban.executor.ExecutableFlow.FailureAction; - import azkaban.executor.ExecutableFlow.Status;
(i) A conflict of type *others* [8]	
Changes in ***l***: this.addConnectionExecutor = ...; - this.closeConnectionExecutor = createThreadPoolExecu- tor(4, ...); **+ this.closeConnectionExecutor = createThreadPoolExecu-** **tor(1 + (config.getMaximumPoolSize() / 2), ...);** **+ if (config.getMetricsTrackerFactory() != null) { ...**	Changes in ***r***: - this.addConnectionExecutor = ...; - this.closeConnectionExecutor = createThreadPoolExecu- tor(4, ...); **+ ThreadFactory threadFactory = ...;** **+ this.addConnectionExecutor = ...;** **+ this.closeConnectionExecutor = createThreadPoolExecu-** **tor(1 + config.getMaximumPoolSize() / 2, ...);**

- **L+R+M**: Keep edits from sides and apply extra edits as needed. Compared with L+R, L+R+M may keep fewer edits from both sides and/or add new edits.
- **X**: Do not resolve the conflict.

For the 320 true conflicts in our sample set, we present the distribution of developers' resolution strategies in Table 5. As shown in the table, developers resolved 64% of conflicts (206/320) by taking all edits from either *l* or *r* (i.e., L or R). They resolved 20% of conflicts (64/320) by including all edits from both branches (i.e., L+R). Table 6 presents such an example. Developers handled 8% of conflicts (25/320) by applying additional changes after integrating all edits from both branches (i.e., L+R+M). They seldom used the other resolution strategies.

Table 5. Developers' resolutions for the 320 true conflicts

Edit Types	L	R	M	L+R	L+M	R+M	L+R+M	X
Add vs. add	19	17	-	40	1	2	4	1
Update vs. update	42	23	4	5	1	-	6	1
Update vs. delete	35	12	1	-	1	2	-	-
Delete+add vs. update	8	7	-	2	1	-	5	-
Delete+add vs. delete+add	7	4	-	1	-	2	2	-
Delete+add vs. add	2	3	1	7	1	-	1	1
Delete+add vs. delete	5	1	-	2	1	-	1	-
Others	11	10	-	7	3	1	6	-
Total	**129**	**77**	**6**	**64**	**9**	**7**	**25**	**3**

"-" indicates "zero-entry"

Table 6. A textual conflict resolved via L+R [16]

Changes in ***l***:
```
- @Override public String[] ..."*:file:*.aar", "*:file:*. jmod");}
+ @Override public String[] ..."*:file:*.aar", "*:file:*. jmod", "*:file:*.kar");}
```

Changes in ***r***:
```
- @Override public String[] ..."*:file:*.aar", "*:file:*. jmod");}
+ @Override public String[] ..."*:file:*.aar");}
```

Changes in ***m***:
```
- @Override public String[] ..."*:file:*.aar", "*:file:*. jmod");}
+ @Override public String[] ..."*:file:*.aar", "*:file:*. kar");}
```

For the 206 conflicts that developers resolved by keeping all edits from one branch only, we further compared the involved edits to understand how developers chose one branch over the other. Interestingly, we found three insights. First, among the 36 *add vs. add* instances resolved via either L or R, there are 14 conflicts reported because the content inserted by one branch fully contains the one inserted by the other. Developers resolved 12 of the 14 conflicts by taking the branch with more insertions. There are seven conflicts triggered when branches insert semantically equivalent but textually different content, mainly due to formatting or code comments. Developers resolved these seven conflicts by (probably randomly) picking one branch. Second, among the 65 *update vs. update* instances resolved via either L or R, there are 30 conflicts caused by divergent updates to the library version. Developers resolved 26 conflicts by taking the branch that introduced a higher version number. In the conflict shown in Table 4 (b), developers kept the edit from *l* probably because `1.9.7 > 1.9.3`. Third, among the 47 *update vs. delete* cases resolved via L or R, developers handled 36 cases by keeping delete operations but resolved 11 cases by preserving updates. It means that developers are more likely to delete the edited content for resolution given an *update vs. delete* conflict. These observations imply that developers adopt strategic resolutions in certain circumstances depending on the types and content of conflicting edits.

RQ2 answer summary: *Developers resolved 206 true conflicts via either L or R. Their decisions of choosing one branch over the other seem predictable in at least 74 (i.e., 12 + 26 + 36) of these instances.*

4.3 RQ3: Discussion on Existing Tool Design

Limitations and Opportunities of Conflict Detectors. Table 2 lists eight tools that detect textual conflicts. Three tools (i.e., git-merge, Crystal, and WeCode) have the false-positive issues grayed in Fig. 4, as they all conduct text-based merge. CloudStudio seems to be able to detect textual conflicts accurately, as it monitors developers' coding activities. However, because it needs to eagerly detect and report textual conflicts whenever two developers touch the same line, CloudStudio's performance can scale poorly with the number of users concurrently accessing the server. Prior work [28, 32, 33, 55] show that the other four tools (i.e., FSTMerge, IntelliMerge, JDime, and AutoMerge) can overcome the limitation of text-based merge, because they all observe

Table 7. A pilot study that applies four tools to tentatively resolve eight different kinds of textual conflicts

Conflict Type	FSTMerge	JDime	IntelliMerge	AutoMerge
Add vs. add	Succeed	Succeed	Succeed	Succeed
Update vs. update	Fail	Fail	Fail	Fail
Update vs. delete	Fail	Fail	Fail	Fail
Delete+add vs. update	Fail	Fail	Fail	Fail
Delete+add vs. delete+add	Fail	Fail	Succeed	Fail
Delete+add vs. add	Fail	Fail	Fail	Succeed
Delete+add vs. delete	Fail	Succeed	Fail	Succeed
Others	Fail	Fail	Fail	Fail

program syntax and report conflicts only if the same tree/graph nodes are edited divergently between l and r. For simplicity, we use **syntax-based merge** to refer to the methodology shared by the four tools. Because syntax-based tools were designed to analyze Java files only, they are unlikely to overcome the false-positive issues in non-Java files (e.g., build scripts and readme files).

We conducted a pilot study by applying FSTMerge, IntelliMerge, JDime, and AutoMerge to 10 randomly picked merging scenarios to validate our hypothesis. Among these scenarios, we sampled three false-positive and seven true conflicts in non-Java files reported by git-merge. Our study shows that none of the four tools can properly handle non-Java files, let alone overcome the false-positive issues. Given the three versions (b, l, and r) of each scenario, FSTMerge either outputs nothing or generates incorrectly merged files with no conflict found. IntelliMerge prompts that it cannot find any file to merge and both JDime and AutoMerge generate error messages to explain that the tools only accept Java files. Our pilot study confirms that syntax-based tools do not address the false-positive issues in non-Java files. As we observed a considerable portion—12% (18/146)—of the conflicts revealed by textual merge in non-Java files to be false positives, we believe that it is still important to create new tools that better detect textual conflicts in such files.

Limitations and Opportunities of Conflict Resolvers. Table 2 lists four tools that can resolve textual conflicts. To assess the resolution capabilities of distinct tools, we conducted a pilot study by constructing a dataset of eight randomly sampled textual conflicts, with each conflict corresponding to one edit type mentioned in Table 5. As shown in Table 7, after applying tools to the eight textual conflicts, we found all tools to be able to successfully resolve the conflict of type *add vs. add*. Additionally, JDime resolved one more type of conflict successfully: *delete+add vs. delete*; IntelliMerge successfully resolved another type of conflict: *delete+add vs. delete+add*; AutoMerge managed to resolve two more types of conflicts: *delete+add vs. add* and *delete+add vs. delete*. Notice that our main focus is the characterization of various conflicts instead of comparison between tools, and it is understandable that different tools have separate implementation issues. Therefore, in this section, we conducted a pilot study to qualitatively measure existing tools' resolution capabilities, instead of performing a large-scale study to quantitatively measure the applicability or resolution quality of all tools. We believe that the detailed empirical comparison between tools is worth thorough investigation, although it is not the main focus of this paper.

Although in Section 4.2 we observed developers' preferences when they choose one branch over the other, we are unaware of any tool built that predicts or suggests such resolution strategies. Our pilot study show that the tools do not resolve conflicts as developers did in most cases. Future tools have great opportunities to propose better candidate merged versions by predicting and automating more resolution strategies based on scenario characterization.

RQ3 answer summary: *Despite the necessity observed in this study, existing tools cannot precisely detect textual conflicts in non-Java files (e.g., build scripts and other files). We observed typical strategies and preferences in developers' conflict resolution practices, which shows promise for future tools that suggest resolutions based on scenario characterization.*

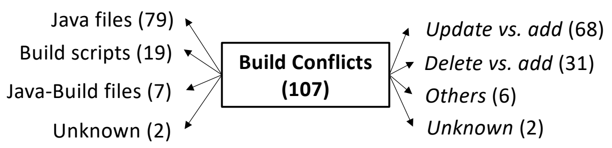

Fig. 5. Characterization for build conflicts

Table 8. Developers' resolutions for build conflicts

Edit Types	L	R	M	L+R	L+M	R+M	L+R+M	X
Update vs. add	2	8	-	-	-	-	58	-
Delete vs. add	3	11	-	-	-	-	17	-
Others	-	3	-	-	-	-	3	-
Total	**5**	**22**	-	-	-	-	**78**	-

5 STUDY RESULTS ON BUILD CONFLICTS

This section presents our analysis of the 107 build conflicts.

5.1 RQ1: Root Causes of Build Conflicts

We characterized each inspected build conflict from two perspectives: the edited files and edit types (see Fig. 5). We did not classify conflicts based on relative edit locations because no conflicting edits overlap textually. In other words, each pair of conflicting edits were smoothly integrated into A_m by git-merge. In terms of edited files, 107 build conflicts can be classified into 4 categories. 79 conflicts exist among Java edits (e.g., Table 9 (a)); 19 conflicts occur among edits to build scripts (e.g., Table 9 (c)); 7 conflicts are due to simultaneous edits in Java code and build scripts (e.g., Table 9 (d)); and 2 conflicts are caused by unknown reasons. All these conflicts are true positives because they all trigger compilation errors. We classified the conflicts into four categories in terms of edit types: (1) *update vs. add*, (2) *delete vs. add*, (3) *others*, and (4) *unknown*.

5.1.1 Update vs. Add. These 68 conflicts can be further classified into 4 subcategories: declaration-reference, super-sub, version-version, and dependency-code. We defined these subcategories based on the content and semantic dependencies of conflicting edits. Table 9 presents an example for each subcategory, and we will explain all of them in detail below.

(a) Declaration-reference When l (or r) updates the declaration of a program entity and r (or l) adds references to the original declaration, there is a mismatch between referencers and the referencee. For example, in Table 9 (a), r updates a field name `EPHEMERAL` to `DISTRO`, while l adds a reference to `EPHEMERAL`. The integration of both edits causes a build error *"cannot find symbol: variable EPHEMERAL"*.

(b) Super-sub There are scenarios where one branch adds a type reference via inheritance (i.e., `A extends B`) or implementation (i.e., `A implements B`), and the other branch updates a method of the super or sub type. The edit co-application makes method signatures inconsistent between two Java types. For instance, in Table 9 (b), r revises an interface `IndexDAO` to declare a new method `getTaskLogs(...)`, while l defines a class `ElasticSearch5DAO` to implement the original interface. Because the defined class does not implement the newly added method, the compiler outputs an error *"ElasticSearch5DAO is not abstract and does not override abstract method getTaskLogs(String) in IndexDAO"*.

(c) Version-version In some merging scenarios, one branch updates the version number of a defined artifact in build scripts while the other branch adds one or more references to the original artifact version. As shown in Table 9 (c), l adds a reference to version `0.2.0` of artifact `spring-cloud-alibaba`; however, r updates the artifact's version from `0.2.0` to `0.2.0.BUILD-SNAPSHOT`. When both edits are applied, the compiler or build system produces an error *"Could not find artifact org.springframework. cloud:spring-cloud-alibaba:pom:0.2.0"*.

(d) Dependency-code There are cases where one branch updates a library dependency in build scripts while the other branch adds code that accesses APIs only supported by the original library. In Table 9 (d), l upgrades artifact `elasticsearch` and r adds code to call `Bucket.getKeyAsText()`, which is only supported by the old library. The co-application of both edits triggers a compilation error *"cannot find symbol: method getKeyAsText()"*.

Table 9. Four representative build conflicts of the category *Update vs. Add* (Section 5.1.1)

(a) declaration-reference [17]

Changes in l:
In `ServerListManager.java`,
```
+ Loggers.EPHEMERAL.debug("check distro heartbeat.");
```

Changes in r:
In `Loggers.java`,
```
- public static final Logger EPHEMERAL = LoggerFactory.getLogger("com.alibaba.nacos.naming.ephemeral");
+ public static final Logger DISTRO = LoggerFactory.getLogger("com.alibaba.nacos.naming.distro");
```

Additional edits in m for conflict resolution:
In `ServerListManager.java`, use `Loggers.SRV_LOG` to replace the nonexistent field `Loggers.EPHEMERAL`.

(b) super-sub [9]

Changes in l:
Add `ElasticSearch5DAO.java`,
```
+ public class ElasticSearch5DAO implements IndexDAO {...}
```

Changes in r:
In `IndexDAO.java`,
```
+ public List<TaskExecLog> getTaskLogs(String taskId);
```

Additional edits in m for conflict resolution:
In `ElasticSearch5DAO.java`, add code to override the newly declared method interface `getTaskLogs(...)`.

(c) version-version [11]

Changes in l:
In `spring-cloud-alibaba-sentinel-datasource/pom.xml`,
```
+ <parent>
+   <groupId>org.springframework.cloud</groupId>
+   <artifactId>spring-cloud-alibaba</artifactId>
+   <version>0.2.0</version>
+ </parent>
```

Changes in r:
In `pom.xml`,
```
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-alibaba</artifactId>
- <version>0.2.0</version>
+ <version>0.2.0.BUILD-SNAPSHOT</version>
```

Additional edits in m for conflict resolution:
In `spring-cloud-alibaba-sentinel-datasource/pom.xml`, update the version number to `0.2.0.BUILD-SNAPSHOT` to refer to the updated parent artifact.

(d) dependency-code [6]

Changes in l:
In `pom.xml`,
```
 <groupId>org.elasticsearch</groupId>
 <artifactId>elasticsearch</artifactId>
- <version>1.6.0</version>
+ <version>2.0.0</version>
```

Changes in r:
In `CSVResultsExtractor.java`,
```
+ for (MultiBucketsAggregation.Bucket bucket : buckets) {
+   String key = bucket.getKeyAsText().string();
```

Additional edits in m for conflict resolution:
In `CSVResultsExtractor.java`, replace the function call `getKeyAsText()` with `getKeyAsString()`.

5.1.2 *Delete vs. Add.*

These 31 conflicts can be further put into 2 subcategories: declaration-reference and dependency-code. The first subcategory represents scenarios where l (or r) deletes an entity declaration and r (or l) adds one or more references to that entity. In Table 10, r deletes an entire file `DubboTransportedMetadata.java` and thus removes the class definition for `DubboTransportedMetadata`. Meanwhile, l adds an `import`-declaration for the class. Consequently, the combination of both edits leads to an error "*cannot find symbol: class DubboTransportedMetadata*".

The second subcategory means one branch deletes the library dependency in build scripts, and the other branch adds references to APIs solely provided by that library. These two subcategories are similar to the subcategories (a) and (d) of *update vs. add*, but different in terms of edit types.

Table 10. A representative build conflict of the category *Delete vs. Add* (Section 5.1.2)

Changes in ***l***:
In `DubboGatewayServlet.java`,
```
+ import org.springframework.cloud.alibaba.dubbo.metadata.DubboTransportedMetadata;
```

Changes in ***r***:
Delete the entire file `DubboTransportedMetadata.java`,

5.1.3 Others. Six conflicts were introduced for miscellaneous reasons. Four of them are *add vs. add* conflicts. For example, as shown in Table 11, when both branches add declarations of the same field `required`, integrating those edits can trigger a build error like "*variable required is already defined in class CodegenParameter*". The other two instances are *update vs. update* conflicts. Namely, when *l* and *r* update different code regions of the same method, the edit integration accidentally introduces the usage of undefined variables.

Table 11. A representative build conflict of the category *Others* (Section 5.1.3)

Changes in ***l***:
In `CodegenParameter.java`,
```
- public Boolean hasMore = null, isContainer = null, secondaryParam = null;
+ public Boolean hasMore = null, isContainer = null, secondaryParam = null, required = null;
```

Changes in ***r***:
In `CodegenParameter.java`,
```
+ public boolean required;
```

5.1.4 Unknown. For two conflicts, we did not figure out how the edits from *l* and *r* conflict with each other. The reported errors are all about missing packages. For instance, one error is "*package org.springframework.mock.web does not exist*". Although this seems to be a library configuration issue, we could not locate the root cause or developers' manual resolution.

Summary. We notice an interesting common characteristic among the 105 conflicts with known reasons. When developers define and use program elements (e.g., classes or libraries) in software, they should always observe two constraints regarding def-use links. First, no element should be defined twice. Second, any used element should correspond to a defined element. If the edit integration between branches breaks def-use links by violating any constraint, build systems report errors, and those integrated edits result in build conflicts.

RQ1 answer summary: *Within the 107 conflicts studied, at least 105 conflicts are due to the co-applied edits that break def-use links in Java files and/or build scripts; 99 conflicts can be explained with two typical combinations of edit types: update vs. add, and delete vs. add.*

5.2 RQ2: Resolutions of Build Conflicts

For the 105 conflicts with identified root causes, we further studied developers' manual resolutions. Table 8 presents the resolution distribution. As shown in the table, developers resolved 78 conflicts via L+R+M. When the co-application of edits from branches triggers any build error, developers usually applied adaptive changes to glue those edits. For instance, to resolve the conflict shown in Table 9 (b), developers kept edits from both sides and inserted code to `ElasticSearch5DAO` in order to implement the newly added method interface `getTaskLogs(...)`. Additionally, developers adopted L and R to separately resolve 5 and 22 conflicts.

Interestingly, the resolution distribution shown in Table 8 is quite different from developers' resolutions to textual conflicts (see Table 5). Most build conflicts were resolved via L+R+M instead of L or R. One possible reason to explain this contrast is that build conflicts are between edits applied to distinct program locations, while textual conflicts are between divergent edits to the same location. Although it is hard to co-apply the edits that textually overlap, it is easier to co-apply the edits whose locations are different. Thus, it is more favorable for developers to keep edits from both sides when resolving build conflicts and make adaptive changes as needed.

Although the additional edits M vary with merging scenarios, we observed two important commonalities. First, the edits were always applied to remedy broken def-use links. For simplicity, we name such edits for link-repair purposes as **adaptive changes**. Second, for many scenarios, the adaptive edits similar to M were already applied in either l or r. Take Table 9 (c) as an example. When developers updated the artifact's version number from `0.2.0` to `0.2.0.BUILD-SNAPSHOT` in r, they also revised build scripts to consistently update all version references in order to use the new artifact version. We use **consistent edits** or **systematic edits** to refer to the similar edits repetitively applied to multiple locations to address the similar or identical coding issues in those locations. In Table 9 (c), we observed developers to apply adaptive changes to resolve all build errors triggered by the version upgrade in r. These adaptive changes are very similar to the additional edits M because both sets of edits intend to fix the same kind of errors.

Among the 78 conflicts resolved via L+R+M, we saw 64 conflicts (82%) to have M consistent with (i.e., similar to) the adaptive edits applied in one branch. These highly consistent edits indicate great opportunities for automatic conflict resolution.

RQ2 answer summary: *We analyzed in total 107 build conflicts. Among the 105 build conflicts with identified root causes, 78 conflicts were resolved via L+R+M. The resolution edits M are often consistent with the adaptive changes applied in one branch for program compilability.*

5.3 RQ3: Discussion on Existing Tool Design

Limitations of Conflict Detectors. Table 2 shows three detectors for build conflicts— Crystal, WeCode, and CloudStudio. All of them rely on build systems to reveal build conflicts. However, based on our experience, the common methodology of these tools has three limitations.

First, the builder-based or compiler-based detection of conflicts is ineffective in reporting build conflicts when textual conflicts exist between l and r. Actually, in our procedural of manual inspection for textual conflicts, we noticed that build conflicts can coexist with textual conflicts in unmergeable versions by git-merge. For such scenarios, neither Crystal nor WeCode reports any build conflict, as they require users to remove all textual conflicts first and produce a merged version A_m. Second, given a merged version, builders or build systems may not report all compilation errors in one run. Namely, the initially revealed errors can prevent builders from detecting other errors. Thus, it can be time-consuming for developers to recognize all build errors through repetitive compilation and program revision. Our study did not change any program or fix any build error. Instead, we focused on the initial build errors. As a result, our analysis can miss some build errors hidden by the initially reported ones. It is possible that in the 79 merging scenarios with build conflicts detected, there are more than 107 build conflicts between branches. Third, given a build error, developers have to identify the conflicting edits manually. When l and r contain many edits, developers may find it challenging to locate the root cause manually. In our manual analysis, even though we spent lots of time investigating the root causes for every reported build error, there are still two errors for which we cannot locate the conflicting edits. Existing tools do not help developers/users reason about conflicts.

Future Opportunities of Conflict Detectors. As mentioned in Section 5.2, build conflicts are mainly about the broken def-use links induced by software merge. The resolution edits always repair broken links, and those edits are often inferable from relevant adaptive changes in either l or r. Based on these observations, we envision a promising conflict detector to replace the usage of compilers with static program analysis. Specifically, the tool can contrast all edits separately applied in l and r, reason about the def-use links between edited program elements, and report conflicts whenever a def-use constraint between elements is violated. For example, for the

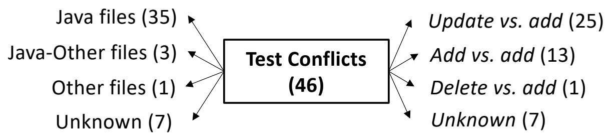

Fig. 6. Characterization for test conflicts

Table 12. Developers' resolutions for test conflicts

Edit Types	L	R	M	L+R	L+M	R+M	L+R+M	X
Update vs. add	1	3	-	-	-	-	21	-
Add vs. add	-	13	-	-	-	-	-	-
Delete vs. add	-	1	-	-	-	-	-	-
Total	**1**	**17**	-	-	-	-	**21**	-

merging scenario shown in Table 9 (a), a future tool can scan for any updated field in either branch (e.g., r) and then check whether the other branch adds any reference to the original field before the update. If so, a broken def-use link is detected, and a conflict is reported accordingly. Without using any builder or compiler, such static analyzers can overcome the three limitations mentioned above.

Limitations and Opportunities of Conflict Resolvers. We were not aware of any tool that can resolve build conflicts, so we conducted a pilot study by applying four syntax-based tools (i.e., FSTMerge, JDime, IntelliMerge, and AutoMerge) to five merging scenarios with known build conflicts. According to our experiments, JDime and AutoMerge are unable to detect or resolve build conflicts. In the scenarios when only Java files are involved in conflicts, both tools either naïvely integrate edits as git-merge does, or fail to process newly added Java files because they strictly require all three program versions (b, l, r) to be provided. In the scenarios when build scripts are also edited, both tools either fail to process non-Java files or output nothing. To sum up, JDime and AutoMerge are unable to detect and resolve build conflicts. Interestingly, when applied to the five scenarios, FSTMerge and IntelliMerge successfully handled one scenario by adding an extra edit to fix the broken def-use link but failed in the other four. In the scenario where FSTMerge and IntelliMerge resolved a conflict, l removes an import from a Java file, and r adds a reference to the originally imported class in the same file. The tools fixed the broken def-use link by adding back the removed import. Our pilot study shows that we lack systematic tool support for the resolution of build conflicts.

Based on our conflict characterization, we see great opportunities to create conflict resolvers. Specifically, a promising approach can infer systematic editing patterns from the adaptive changes applied in either l or r and customize the inferred patterns to resolve conflicts in A_m. Our insight is that *if developers resolve compilation errors in either branch for any edit that breaks def-use links, they are likely to resolve the same compilation errors in A_m in similar ways for that merged-in edit.* Take Table 9 (a) as an example. Given the conflicting edits between branches, a future tool can focus on the field-update edit and mine r for any adaptive edits related to that update. Suppose an adaptive edits were applied in r to remedy broken links between the defined fields and field accesses (i.e., by updating all added field accesses to refer to `DISTRO`). In that case, the future tool can similarly apply such edits to A_m to resolve conflicts.

RQ3 answer summary: *Existing detectors of build conflicts have quite limited applicability, efficiency, and effectiveness. There is almost no tool support to automatically resolve build conflicts.*

6 STUDY RESULTS ON TEST CONFLICTS

This section presents our analysis of the 46 test conflicts.

6.1 RQ1: Root Causes of Test Conflicts

We characterized test conflicts from two perspectives: the edited files and edit types. As shown in Fig. 6, we classified conflicts into four categories based on edited files: 35 conflicts exist among Java edits (e.g., Table 13 (b)); 3 conflicts are among edits to Java and other files (i.e., .xml or .groovy); 1 conflict is among non-Java files (e.g., Table 13 (a)); and 7 conflicts were caused by unknown reasons. All these conflicts involve simultaneous edits to

Table 13. Two representative test conflicts

(a) An exemplar conflict of update vs. add [10]
Changes in ***l***: In `KotlinJsonAdapter.kt`, code implementation is updated
Changes in ***r***: In `KotlinJsonAdapterTest.kt`, a new test is added to invoke functions defined in `KotlinJsonAdapter.kt`
Additional edits in ***m*** for conflict resolution: In `KotlinJsonAdapterTest.kt`, an assertion is revised to match the updated implementation logic of invoked functions.
(b) An exemplar conflict of add vs. add [4]
Changes in ***l***: In `test/.../SampleObjects.java`, developers defined a new method `public long getId(){...}`
Changes in ***r***: In `test/.../SampleObjects.java`, developers also defined the same method `public long getId(){...}` in the same way.
Additional edits in ***m*** for conflict resolution: In `test/.../SampleObjects.java`, keep the defined method from *r*

test oracles and/or software implementation. Compared with Fig. 5, we have more test conflicts due to unknown reasons (i.e., 7 vs. 2). This is because build errors usually report the program elements that break def-use links, which helped us manually identify root causes. On the other hand, test errors only report the locations where abnormal program behaviors are observed. These locations may be far away from the places where program states initially become erroneous. Without project-specific domain knowledge and dynamic analysis tools, it is hard for us to manually reason about all root causes.

We classified conflicts into four categories in terms of edit types: (1) *update vs. add*, (2) *add vs. add*, (3) *delete vs. add*, and (4) *unknown. Update vs. add* has four subcategories: declaration-reference, super-sub, dependency-code, and implementation-oracle. The first three subcategories are similar to the subcategories (a), (b), and (d) mentioned in Section 5, and they all break def-use links. However, these conflicts trigger test errors instead of build errors because part of the conflicting edits were applied to test code, which is only compiled in the testing instead of the build phase.

The subcategory *implementation-oracle* means that a conflict happens if one branch changes program implementation, while the other branch adds test code whose oracle matches the original implementation. For instance, in Table 13 (a), *l* updates the implementation of `KotlineJsonAdapter` and *r* adds a test case to `KotlineJsonAdapterTest` in order to test the original implementation. The co-application of both edits triggers an assertion error "*Expecting message: <'Non-null value 'a' was null at $'> but was: <'Non-null value 'a' was null at $.a'>*". It indicates that there is a semantic mismatch between the code implementation and test oracle.

Add vs. add means that *l* and *r* add the same method at distinct program locations of the same file, causing one method to have duplicated definitions. Take Table 13 (b) as an example. Because both branches simultaneously define the same method `getId()` in a file inside the test folder, the co-application of both edits triggers an error "*method getId() is already defined in class Employee*". Notice that **such duplicated definition errors were introduced by git-merge, as text-based merge naïvely combines the edits simultaneously applied to distinct program locations by branches**. These errors are only reported by automated testing instead of software build because the edits were applied to test files.

The single *delete vs. add* conflict was introduced when one branch deletes a class, and the other branch adds a test class to refer to that deleted class. This subcategory is similar to the *delete vs. add* category mentioned in Section 5. The only difference is that the added reference exists in a test class, which is compiled in the testing instead of the build phase and thus triggers test errors.

Summary. We noticed three interesting phenomena. First, many of the observed test errors are compilation errors in test cases. They are revealed in the testing phase just because the test cases get complied within this

Table 14. Comparison between the adaptive changes in l and the extra edits M in manual resolution

Adaptive changes in ***l***:
```
    jsonAdapter.fromJson("{\"a\":null}")
    fail()
  } catch (expected: JsonDataException) {
-   assertThat(expected).hasMessage("Non-null value 'a' was null at \$")
+   assertThat(expected).hasMessage("Non-null value 'a' was null at \$.a") }
```

Extra edits M in ***m***:
```
    jsonAdapter.fromJson("{\"a\":\"hello\"}")
    fail()
  } catch (expected: JsonDataException) {
-   assertThat(expected).hasMessage("Non-null value 'a' was null at \$")
+   assertThat(expected).hasMessage("Non-null value 'a' was null at \$.a") }
```

phase. We currently count the conflicts triggering these errors as test conflicts simply because the compiler-based detection of build conflicts is insufficient and produces false negatives. Second, the add-add conflicts are false negatives of the text-based merge. Git-merge naïvely introduced them into A_m without comparing the edits co-applied to different locations for conflict detection or resolution. Third, among the inspected 46 test conflicts, we observed 11 related to mismatches between code implementation and test oracles.

RQ1 answer summary: *We analyzed in total 46 test conflicts, and managed to locate the root causes for 39 conflicts. All of these 39 conflicts are due to the edit integration that either breaks def-use links, repetitively adds new code, or causes mismatches between implementation and oracles.*

6.2 RQ2: Resolutions of Test Conflicts

For the 39 conflicts with revealed root causes, we further classified the corresponding manual resolutions. As shown in Table 12, developers resolved 21 conflicts via L+R+M, and all these conflicts belong to the *update vs. add* category. For instance, to resolve the conflict shown in Table 13 (a), developers kept edits from branches and revised the test oracle to match with the output of the updated implementation. Additionally, developers resolved 13 conflicts via R, all of which belong to the *add vs. add* category. For instance, to resolve the conflict listed in Table 13 (b), developers kept the right version because the methods repetitively added by branches cannot coexist in the same program context. We believe that the developers prefer L+R+M over other resolution strategies.

Similar to what we found among resolutions to build conflicts, the additional edits M applied to resolve test conflicts have two important characteristics. First, all edits were applied to fix def-use links or match test oracle with code implementation. For simplicity, we also refer to such edits with **adaptive changes**. Second, they were usually consistent with the adaptive edits applied in one branch. Take Table 13 (a) as an example. When developers updated the implementation of `KotlinJsonAdapter` in l, they also adapted assertions (i.e., test oracles) in `KotlinJsonAdapterTest` to ensure test success. If we compare their adaptive changes in l and the additional edits in m (see Table 14), the two change-sets are identical albeit distinct program context. Among the 21 conflicts resolved via L+R+M, 12 conflicts have M similar or identical to the adaptive edits in one branch.

RQ2 answer summary: *When developers resolved test conflicts, they usually prefer L+R+M over other strategies; and the extra edits M seem predictable from adaptive edits applied in one branch.*

6.3 RQ3: Discussion on Existing Tool Design

Limitations of Conflict Detectors. Table 2 lists three detectors for test conflicts. Both Crystal and WeCode rely on automated testing to reveal test conflicts so that they can cover all test conflicts mentioned in Section 6.1. However, according to our experience, the common methodology of both tools suffers from five limitations. First, when test conflicts coexist with other types of conflicts, automated testing is infeasible. Developers have to resolve the other two kinds of conflicts before getting a chance to run code with test cases and explore test conflicts. During our manual analysis of textual conflict scenarios, we observed test conflicts to coexist with textual ones. For such scenarios, neither Crystal nor WeCode reveals any test conflict. Second, when multiple test conflicts coexist, the runtime errors triggered by some conflicts can stop program execution and prevent testing from detecting other conflicts. In other words, testing-based conflict detection can be slow and may require lots of test runs. Third, it is time-consuming for developers to write test cases and ensure sufficient coverage of testing. Fourth, both tools can only reveal the symptoms (i.e., runtime errors) instead of root causes for any test conflict. They do not provide further assistance in conflict localization. Fifth, there are flaky tests that pass and fail nondeterministically in different program runs. For such scenarios, automated testing cannot effectively reveal symptoms of conflicts, and we did not include such scenarios in our dataset.

SafeMerge is designed to statically reason about program semantics to overcome all limitations of testing-based conflict detection. However, due to its complex modeling for program semantics, SafeMerge cannot relate edits applied to distinct Java methods, neither can it analyze edits applied to non-code files. Therefore, SafeMerge cannot reveal any test conflict caused by the edits simultaneously applied to distinct program elements (i.e., files or classes). Our study found all 39 conflicts to be introduced by simultaneous edits to different program elements, limiting the usefulness of SafeMerge in real-world scenarios. In fact, we tried to apply SafeMerge to test conflicts but without success. Therefore, we were unable to conduct a pilot study to validate the tool's capability.

Opportunities of Conflict Detectors. SafeMerge demonstrates a promising way to detect test conflicts via static program analysis. However, SafeMerge is limited to intra-procedural analysis, while most conflicts we observed require inter-procedural analysis. The mismatch between the problem domain and the current solution indicates great opportunities for conflict detectors based on inter-procedural static program analysis.

Limitations and Opportunities of Conflict Resolvers. We were unaware of any tool that can resolve test conflicts, so we conducted a pilot study by applying 4 syntax-based tools (i.e., FSTMerge, JDime, IntelliMerge, and AutoMerge) to 15 merging scenarios with known test conflicts. Among the scenarios, 10 conflicts belong to *add vs. add*, which were wrongly introduced by git-merge and 5 conflicts belong to *update vs. add.* As syntax-based tools were designed to align Java methods between branches based on method signatures and implementation, they should be able to detect or even resolve the *add vs. add* conflicts. Unsurprisingly, our study shows that the four tools resolved all 10 conflicts successfully. Specifically, these tools all recognized the duplicated addition of the same methods in 10 scenarios and included single copies of those methods in the merged versions. In other words, our pilot study confirmed that JDime, IntelliMerge, FSTMerge, and AutoMerge could nicely address the *add vs. add* conflicts incorrectly introduced by git-merge. Meanwhile, no syntax-based tool can detect or resolve the other five test conflicts (*update vs. add*). This is because these tools do not model or compare any program semantics for co-applied edits.

There is no tool that automatically resolves semantic mismatches in test conflicts. Based on our conflict characterization, nevertheless, we see great research opportunities to create conflict resolvers. Similar to the future tool design described in Section 5.3, a promising approach can infer systematic editing patterns from the adaptive changes in l or r and customize the inferred patterns to resolve conflicts in A_m. Our insight is that *if developers resolve test errors in one branch for edits that either break def-use links or cause mismatches between code and tests, they are likely to resolve the same errors in A_m in similar ways.* Among the 21 conflicts resolved with L+R+M, this proposed approach will be able to resolve 12 conflicts.

RQ3 answer summary: *Existing detectors of test conflicts are insufficient in four aspects: applicability, effectiveness, efficiency, and program coverage. Currently, no resolver can fix the semantic mismatches between co-applied edits to remove test conflicts.*

7 OUR RECOMMENDATIONS

Based on our study, we would like to provide the following recommendations for future research.

Resolution Prediction for Textual Conflicts. According to our analysis, it seems too ambitious to build a tool that automatically resolves all conflicts because the additional edits M involved in some strategies are specific to projects and domains. However, it is promising to predict developers' resolution strategies, as the developers of different projects implicitly share decision-making patterns like preferring L and R over other strategies. Future tools can characterize merging scenarios from different perspectives (e.g., the file types, edit types, edit content, program contexts, authors, and timestamps) and train machine-learning models for strategy prediction. As developers resolve most conflicts by keeping all edits from one branch, a highly accurate prediction model can automatically resolve those conflicts with high success rates to reduce manual effort.

Detection of Higher-Order Conflicts. Existing tools mainly detect higher-order conflicts via automatic compilation and testing. Both compilation and testing heavily rely on the existence of merged versions and require intensive human-in-the-loop interactions before uncovering all conflicts. Therefore, the applicability and capability of such tools are not always satisfactory.

We recommend future tools to statically analyze and contrast software branches for conflict detection without requiring branches to be merged or compiled successfully. Specifically, a new approach can enumerate all situations where the naïve edit integration between branches can cause build or test errors and define patterns to represent those situations. For instance, conflicting situations can include (1) one branch renames or removes a program element (e.g., class, method, or field), while the other branch adds references to the original element; (2) one branch adds a method $m()$ to a Java interface, while the other branch defines a new class to implement the interface but does not define any method body for $m()$. With patterns defined for such conflicting situations, the new approach can search for pattern matches between l and r in any given merging scenario to detect conflicts. In this way, the future tool does not need automatic compilation or testing.

Resolution of Higher-Order Conflicts. In our study, we observed that most conflicts were resolved via L+R+M, and the additional edits M are often similar to the adaptive changes applied to one of the branches. The rationale behind the observed similarity is that *when developers apply certain edits to remove the build or test errors in either branch, they are likely to apply similar edits to remove the same build or test errors in the naïvely merged software A_m.*

Some systematic editing tools like Sydit [45] and Lase [44] can generalize abstract program transformations from concrete code change examples and repetitively apply similar edits to similar code snippets. We see promise in extending these tools for automatic conflict resolution. For instance, given edits from l and r producing a higher-order conflict, a future tool can extend Lase to search for relevant adaptive changes applied to one branch. The tool can then extract systematic editing patterns from those adaptive changes, customize patterns for particular program locations, and apply the customized edits.

8 RELATED WORK

The related work includes empirical studies and change recommendation systems.

8.1 Empirical Studies on Merge Conflicts

Several studies were conducted to characterize the relationship between merge conflicts and developers' coding activities [26, 35, 40, 43, 48**?**]. For instance, Leßenich et al. surveyed 41 developers and identified 7 potential indicators (e.g., # of changed files in both branches) for merge conflicts [40]. With a further empirical study of the indicators, the researchers found that none can predict the frequency of conflicts. Similarly, Owhadi-Kareshk et al. defined 9 features to characterize merging scenarios and trained a machine-learning model that predicts conflicts with 57%–68% accuracy [**?**]. Mahmoudi et al. observed that certain refactoring types (e.g., Extract

Method) are more related to merge conflicts [43]. These studies only focus on the relationship between textual conflicts and other coding features (e.g., refactorings). They do not characterize any root cause or resolution for conflicts, neither do they analyze higher-order conflicts.

Some other studies characterize the root causes and/or resolutions of *textual* conflicts [18, 29, 37, 50, 53, 58]. Specifically, Ji et al. studied the textual conflicts caused by Git Rebase—a practice to merge program changes by rewriting the evolution history [37]. Our research scope is different, as we focus on the conflicts revealed or caused by Git Merge–a more popularly used approach of merging program changes. Yuzuki et al. inspected hundreds of textual conflicts [58]. They observed that 44% of conflicts were caused by conflicting updates to the same line of code, and developers resolved 99% of conflicts by taking either the left- or right- version of code. Pan et al. studied 271 merging scenarios with textual conflicts inside the repository of Microsoft Edge [53]. The researchers clustered scenarios based on the file types, conflict sizes, conflict locations, and resolution patterns. They also showed the feasibility of using program synthesis to resolve merge conflicts. Our study revealed similar findings for textual conflicts, but our analysis scope is wider and deeper. Namely, we characterized the root causes and resolutions for higher-order conflicts; we also analyzed the limitation of existing merge tools based on our conflict characterization.

Nguyen et al. [50] studied the version history of four projects and analyzed all merging scenarios to assess (1) the integration and conflict rates between software branches and (2) the frequency of rollbacks for merge operations. The researchers found that (i) a higher integration rate of a project does not generate a higher unresolved conflict rate, (ii) developers rolled back merge operations for 5–33% of cases, and (iii) git-merge falsely reports many textual conflicts when concurrent edits are applied to adjacent code instead of same lines. Different from Nguyen et al., we characterized conflicts from new perspectives. For textual conflicts, in addition to relative edit locations, we characterized the edited files and edit types as root causes of conflicts. We also discussed all possible strategies developers may take when resolving conflicts. Furthermore, we conducted a similar characterization for both build and test conflicts.

Brindescu et al. [29] manually inspected 606 textual conflicts. They characterized merge conflicts in terms of the AST diff size, LOC diff size, and the number of authors. They reported that the vast majority of merge conflicts are resolved by the author of one of the branches. They identified three resolution strategies: SELECT ONE (i.e., keep edits from one branch), INTERLEAVE (i.e., keep edits from both sides), and ADAPTED (i.e., change existing edits and/or add new edits). Compared to Brindescu's work, our study characterizes textual conflicts in terms of the edited files, relative edit locations, and edit types. We did not measure the diff size, count authors, or compare the authors of branches with conflict resolvers. The three resolution strategies recognized by Brindescu et al. are roughly equivalent to our eight strategies mentioned in Section 4.2. For instance, SELECT ONE corresponds to L and R. However, our strategy classification is finer-grained. More importantly, we studied higher-order conflicts and reasoned about the limitations of existing tools based on characterized conflicts.

The study by Ghiotto et al. [18] is most relevant to our work. Ghiotto et al. focused on the textual conflicts found in 2,731 open-source Java projects hosted by GitHub. Seeded by the manual analysis of the history of five projects, their automated analysis of all 2,731 projects (1) characterizes the merge conflicts in terms of the number of chunks, size, and programming language constructs involved, (2) classifies the manual resolution strategies that developers use to address these merge conflicts, and (3) analyzes the relationships between various characteristics of the merge conflicts and the chosen resolution strategies. The researchers also provided recommendations for future merge techniques based on their study.

We got inspired by Ghiotto's work, but our study is unique in two aspects. First, in addition to textual conflicts, we also mined, inspected, and characterized higher-order conflicts. Second, our recommendations complement the suggestions by Ghiotto et al.. Two of our research recommendations are related to higher-order conflicts. Without characterizing such conflicts or exploring the relevant tools, no prior work recognizes the research opportunities as our study does.

8.2 Change Recommendation Systems

Based on the insight that similar code is likely to be changed similarly, researchers proposed tools to recommend code changes or facilitate systematic program editing [38, 42, 44, 45, 47, 49, 51]. With more details, simultaneous editing enables developers to simultaneously edit multiple preselected code fragments in the same way [47]. While a developer interactively demonstrates the edit operations in one fragment, the tool replicates the lexical edits (e.g., copy a line) to other fragments. CP-Miner identifies code clones (i.e., similar code snippets) and detects copy-paste-related bugs if clones have inconsistent identifiers or context mappings [42]. Given two or more similarly changed code examples, Lase extracts the common edit operations, infers a general program transformation, and leverages the transformation to locate code for similar edits [44].

Although both software merge and change recommendation systems are active research areas, our study is the first piece of work that methodically connects the two areas. Our study uncovers various scenarios where developers repetitively applied similar edits to resolve conflicts. It enlightens future research to improve change recommendations, such that conflict-specific systematic edits are automatically created and applied.

9 THREATS TO VALIDITY

Threats to External Validity: Our study is based on the 538 conflicts extracted from 208 Java project repositories on GitHub. The characterization of conflicts and the observed resolutions may not generalize well to other conflicts, other projects, other programming languages, or other hosting platforms (e.g., Bitbucket). Our data collection started from the 208 popular open-source Java projects in GitHub, but we only found 79 build conflicts and 33 test conflicts. Although we made our best effort by spending one year in exploring all merging scenarios in those repositories, the relatively small number of higher-order conflicts are the best dataset we can create at this point, mainly due to the limitations of existing tool support. Consequently, the relative low numbers can also limit the generalizability of our empirical findings. In the future, we plan to mitigate this threat by creating and using better tools to reveal higher-order conflicts more effectively.

As with prior work [18, 29, 50, 58], we relied on the intuitive pattern "one child commit following two parent commits" to recognize merging scenarios in software version history. However, it is possible that some developers may merge software branches without using the command `git merge` or producing the commits matching our pattern. They may simply copy and paste code from branches, manually integrate the edits of distinct branches, and then discard some branches. As such non-typical software merge practices leave no obvious footprint in software version history, we were unable to identify or study such merging scenarios. Therefore, our empirical findings may not generalize well to the non-typical merging scenarios.

Threats to Construct Validity: Although we tried our best to manually inspect the sampled textual conflicts and all revealed build/test conflicts, it is possible that our manual analysis is subject to human bias and restricted by our domain knowledge. To alleviate the problem, two authors independently examined each sampled textual conflict, every reported build or test error in the merged software A_m, and related resolution edits applied by the developers. The two authors actively discussed the instances they disagreed upon until reaching a consensus. The root causes of some observed compilation/test errors are not manually located, mainly because we are not familiar with the project-specific software context. In the future, as we gain more knowledge of merge conflicts and create better tools to reveal the root causes automatically, we can further improve the quality of empirical findings.

Threats to Internal Validity: We adopted compilation and testing to reveal build and test errors and then manually analyzed merging scenarios to locate conflicts responsible for those errors. Both methodologies (i.e., compiler-based and testing-based conflict detection) have various limitations (see Sections 5.3 and 6.3). For instance, compiler-based method is inapplicable when build conflicts coexist with textual conflicts. This is because the existence of textual conflicts prevents `git merge` from creating a merged version A_m, making it impossible for a build process to tentatively build A_m and to detect compilation errors. Similarly, testing-based method is inapplicable when test conflicts coexist with textual (or build) conflicts. Additionally, testing-based method only reports test failures or runtime errors; it does not directly pinpoint the conflicting edits responsible for any abnormal program behavior. Therefore, the number of higher-order conflicts reported in our study can be lower than the actual number of conflicts contained by the studied projects. In the future, we would like to overcome this limitation by creating better approaches for conflict detection.

Our current method automatically detects test conflicts only when (1) both l and r compile and pass all tests, and (2) A_m compiles but fails one or more tests. Given the fact that this method only revealed 33 test conflicts in our study, people may be tempted to relax the filtering condition (2), and only require A_m to partially instead of fully compile but fail certain tests. We thought about using this alternative method to enlarge the dataset of test conflicts, and believed the method to not work for two reasons. First, we currently rely on Maven/Ant/Gradle to compile and test projects. All build tools have their internal predefined workflows. In a typical workflow, the test phase follows the build phase, and the test phase only starts when the build phase completes successfully. We cannot manipulate this built-in process to run tests on partially compiled code. Even with the straightforward and standard built-in process, we have already spent tremendous time and effort on all studied merging scenarios. It is hard to imagine how much more time and effort we need to put in order to customize the workflows for distinct projects. Second, when partial compilation is successful and test failures occur, it can be even harder to manually diagnose the root cause of test conflicts. This is because a test failure may be due to incomplete compilation or merged edits. It can be more time-consuming to triage the root causes of test failures produced by partially compiled code. Meanwhile, we may lack the domain knowledge to always correctly triage the root causes.

10 CONCLUSION

Prior empirical studies showed that merge conflicts frequently occur, and conflict resolution is important but challenging. This study comprehensively studied three kinds of conflicts and their resolutions and characterized the conflicts that existing tools cannot handle. Unlike prior studies that focus on textual conflicts, our study is wider and deeper for two reasons. First, we examined higher-order conflicts in addition to textual conflicts, as (1) higher-order conflicts are harder to detect and resolve and (2) few tools are available to handle those conflicts. Second, by comparing the approach design or methodologies against characteristics of real-world conflicts, we identified the limitations of current approaches and suggested future research to overcome those limitations. Our study intends to (1) explore the gap between real conflicts and current tool support and (2) suggest future research to close the gap. No prior work shares the same goals.

Our study provides multiple insights. First, developers usually resolved textual conflicts by keeping all edits from one branch. Our empirical study characterizes the scenarios where a future tool can accurately predict developers' decision-making to select one branch over the other. Second, current tools mainly rely on two methods to uncover higher-order conflicts: compilation and testing. However, these tools are limited as (1) neither method is applicable when textual conflicts exist between branches, and (2) both methods require heavy human involvement to locate all conflicts. Our research shows that higher-order conflicts present typical commonalities (e.g., broken def-use links and test-implementation inconsistency), which future conflict detectors can leverage. Third, developers usually resolved higher-order conflicts by applying similar edits to similar code locations. By automating such practices, future tools can resolve higher-order conflicts.

ACKNOWLEDGMENT

We thank anonymous reviewers for their valuable comments on our earlier version of the paper. This work was supported by NSF-1845446.

REFERENCES

[1] 2010. Merge branch 'master' of github.com:ning/async-http-client. https://github.com/AsyncHttpClient/async-http-client/commit/4999b8dd4278f9cae0a1d278c00f067e8df5c12e.

[2] 2012. close hooks. https://github.com/eclipse-vertx/vert.x/commit/48972cf7c810802eeb88bb0babbabc56f34023d5.

[3] 2012. Merge branch 'master' of https://wenshao@github.com/AlibabaTech /fastjson.git. https://github.com/alibaba/fastjson/commit/ee3728b727c52d9090919f7791e1b0d738e1ea6f.

[4] 2013. Merge branch 'master-issue1115' of github.com:enesakar/hazelcast into master. https://github.com/hazelcast/hazelcast/commit/dc13b1efc99bb0fb086e817492c9ed196836b33b.

[5] 2013. Merge branch 'master' of github.com:azkaban/azkaban2 into pipelineui. https://github.com/azkaban/azkaban/commit/4816d7435f6d0ba469bf20e68852ccc056c9435d.

[6] 2015. Merge commit 'b2fb76cdacfbf2ce2c2aab8e3e4e87bfcae292a4' into elastic2.0. https://github.com/NLPchina/elasticsearch-sql/commit/10f0159963f9d84f13f32b37490222059c6a9397.

[7] 2015. Merge remote-tracking branch 'origin/master'. https://github.com/MyCATApache/Mycat-Server/commit/56e17e66e0fb7f94ebf7a97d0a1ac0407bb07478.

[8] 2016. Merge branch 'dev' of https://github.com/nitincchauhan/HikariCP into dev. https://github.com/brettwooldridge/HikariCP/commit/68a52f143194e3d2fc7a14a005c64a58aca0d5a7.

[9] 2017. Merge branch 'dev' of https://github.com/Netflix/conductor into dev. https://github.com/Netflix/conductor/commit/4fd6ff1a6cc4979b9fc7832c464615281f5734c9.

[10] 2018. Merge pull request #511 from square/eric.non-null. https://github.com/square/moshi/commit/eb24a235682eacdfb7bbfd7444ff6a2bca65ef76.

[11] 2018. Merge remote-tracking branch 'upstream/master'. https://github.com/alibaba/spring-cloud-alibaba/commit/ff2c3906734e917594994c1dd85d010e2d121a69.

[12] 2018. Which manufacturer updates its phones fastest? Android Oreo edition. https://www.androidauthority.com/android-oreo-fastest-manufacturers-update-874788/.

[13] 2019. Merge branch '1.6.6-druid' into zzw-druid. https://github.com/MyCATApache/Mycat-Server/commit/c42db00282577e9524a55ccef8889a3dff713518.

[14] 2019. Merge branch '2.7.0-release'. https://github.com/apache/dubbo/commit/68fa9c189d5a260260fb411a7ad7e8e035475b8a.

[15] 2019. Merge branch 'master' into type-inference-wip. https://github.com/skylot/jadx/commit/e1f4955286f73a6ca4525fc27b11fd02106f7f08.

[16] 2019. Merge branch 'master' of https://github.com/java-decompiler/jd-gui. https://github.com/java-decompiler/jd-gui/commit/2ae4688ea65a52a7921c36e3b993b502c2d34fee.

[17] 2019. Merge pull request #1552 from nkorange/hotfix_log_optimization. https://github.com/alibaba/nacos/commit/9b2367eb91dbd909ef4fd0ceeddfd9762d348d02.

[18] 2020. On the Nature of Merge Conflicts: A Study of 2,731 Open Source Java Projects Hosted by GitHub. *IEEE Transactions on Software Engineering* 46, 8 (2020), 892–915. https://doi.org/10.1109/TSE.2018.2871083

[19] 2021. Gradle. https://gradle.org.

[20] 2021. Integration Hell. https://www.solutionsiq.com/agile-glossary/integration-hell/.

[21] 2021. jFSTMerge. https://github.com/guilhermejccavalcanti/jFSTMerge.

[22] 2021. Maven. https://maven.apache.org.

[23] 2021. Sample size calculator. https://www.surveymonkey.com/mp/sample-size-calculator/.

[24] Last visited 07/18/19. Ant. https://ant.apache.org.

[25] Last visited 07/26/2019. Git Merge. https://git-scm.com/docs/git-merge.

[26] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma. 2017. An Empirical Examination of the Relationship between Code Smells and Merge Conflicts. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 58–67. https://doi.org/10.1109/ESEM.2017.12

[27] Sven Apel, Olaf Lessenich, and Christian Lengauer. 2012. Structured Merge with Auto-tuning: Balancing Precision and Performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE 2012)*. ACM, New York, NY, USA, 120–129. https://doi.org/10.1145/2351676.2351694

[28] Sven Apel, Jorg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kastner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/2025113.2025141

[29] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. 2020. An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering* 25, 1 (2020), 562–590. https://doi.org/10.1007/s10664-019-09735-4

[30] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 168–178. https://doi.org/10.1145/2025113.2025139

[31] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. 2013. Early Detection of Collaboration Conflicts and Risks. *IEEE Transactions on Software Engineering* 39, 10 (Oct 2013), 1358–1375. https://doi.org/10.1109/TSE.2013.28

[32] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and Improving Semistructured Merge. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 59 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133883

[33] Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. 2019. The Impact of Structure on Software Merging: Semistructured versus Structured Merge. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 1002–1013. https://doi.org/10.1109/ASE.2019.00097

[34] H. Christian Estler, Martin Nordio, Carlo A. Furia, and Bertrand Meyer. 2013. Unifying Configuration Management with Merge Conflict Detection and Awareness Systems. In *2013 22nd Australian Software Engineering Conference*. 201–210. https://doi.org/10.1109/ASWEC.2013.32

[35] H. C. Estler, M. Nordio, C. A. Furia, and B. Meyer. 2014. Awareness and Merge Conflicts in Distributed Software Development. In *2014 IEEE 9th International Conference on Global Software Engineering*. 26–35. https://doi.org/10.1109/ICGSE.2014.17

[36] Mário Luís Guimarães and António Rito Silva. 2012. Improving Early Detection of Software Merge Conflicts. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 342–352. http://dl.acm.org/citation.cfm?id=2337223.2337264

[37] Tao Ji, Liqian Chen, Xin Yi, and Xiaoguang Mao. 2020. Understanding Merge Conflicts and Resolutions in Git Rebases. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 70–80. https://doi.org/10.1109/ISSRE5003.2020.00016

[38] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based Detection of Clone-related Bugs. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering.*

[39] Olaf Leßenich, Sven Apel, and Christian Lengauer. 2014. Balancing precision and performance in structured merge. *Automated Software Engineering* 22 (2014), 367–397.

[40] Olaf Leßenich, Janet Siegmund, Sven Apel, Christian K´astner, and Claus Hunsen. 2018. Indicators for Merge Conflicts in the Wild: Survey and Empirical Study. *Automated Software Engg.* 25, 2 (June 2018), 279–313. https://doi.org/10.1007/s10515-017-0227-0

[41] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 249–260. https://doi.org/10.1109/ICSME.2017.46

[42] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code.. In *OSDI*. 289–302.

[43] M. Mahmoudi, S. Nadi, and N. Tsantalis. 2019. Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 151–162. https://doi.org/10.1109/SANER.2019.8668012

[44] Na Meng, Miryung Kim, and Kathryn McKinley. 2013. LASE: Locating and Applying Systematic Edits. In *ICSE*. 10.

[45] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic Editing: Generating Program Transformations from an Example. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 329–342. https://doi.org/10.1145/1993498.1993537

[46] T. Mens. 2002. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering* 28, 5 (2002), 449–462. https://doi.org/10.1109/TSE.2002.1000449

[47] Robert C. Miller and Brad A. Myers. 2001. Interactive Simultaneous Editing of Multiple Text Regions. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 161–174.

[48] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. 2018. The life-cycle of merge conflicts: processes, barriers, and strategies. *Empirical Software Engineering* (2018), 1–44.

[49] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A graph-based approach to API usage adaptation. 302–321.

[50] Hoai Le Nguyen and Claudia-Lavinia Ignat. 2018. An Analysis of Merge Conflicts and Resolutions in Git-Based Open Source Projects. *Computer Supported Cooperative Work (CSCW)* 27, 3 (01 Dec 2018), 741–765. https://doi.org/10.1007/s10606-018-9323-3

[51] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Clone-Aware Configuration Management. In *ASE*. 123–134. https://doi.org/10.1109/ASE.2009.90

[52]]Owhadi-Kareshk2019 Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. [n. d.]. Predicting Merge Conflicts in Collaborative Software Development. https://arxiv.org/pdf/1907.06274.pdf.

[53] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu K. Lahiri, and Mike Kaufman. 2021. Can Program Synthesis be Used to Learn Merge Conflict Resolutions? An Empirical Analysis. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. 785–796. https://doi.org/10.1109/ICSE43902.2021.00077

[54] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous Deployment at Facebook and OANDA. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/2889160.2889223

[55] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. 2019. IntelliMerge: A Refactoring-Aware Software Merging Technique. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 170 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360596

[56] Marcelo Sousa, Isil Dillig, and Shuvendu Lahiri. 2018. Verified Three-Way Program Merge. In *Object-Oriented Programming, Systems, Languages & Applications Conference (OOPSLA 2018)*. ACM. https://www.microsoft.com/en-us/research/publication/verified-three-way-program-merge/

[57] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 805–816. https://doi.org/10.1145/2786805.2786850

[58] R. Yuzuki, H. Hata, and K. Matsumoto. 2015. How we resolve conflict: an empirical study of method-level conflict resolution. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*. 21–24. https://doi.org/10.1109/SWAN.2015.7070484

[59] Fengmin Zhu and Fei He. 2018. Conflict Resolution for Structured Merge via Version Space Algebra. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 166 (Oct. 2018), 25 pages. https://doi.org/10.1145/3276536