The Journal of Systems and Software 214 (2024) 112084

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

ConflictBench: A benchmark to evaluate software merge tools☆

Bowen Shen, Na Meng *

Virginia Polytechnic Institute and State University, Blacksburg VA 24060, USA

ARTICLE INFO

Dataset link: https://github.com/UBOWENVT/ConflictBench

Keywords:
Empirical
Software merge
Conflict
Benchmark

ABSTRACT

In collaborative software development, programmers create branches for simultaneous program editing, and merge branches to integrate edits. When branches divergently edit the same text, the edits conflict and cannot get co-applied. Tools were built to automatically merge software branches, to detect conflicts, and to resolve conflicts along the way. However, there is no third-party benchmark or metric to comprehensively evaluate or compare those tools.

This paper presents ConflictBench, our novel benchmark to evaluate software merge tools. ConflictBench consists of 180 merging scenarios extracted from 180 open-source Java projects. For each scenario, we sampled a conflicting chunk (i.e., conflict) reported by git-merge. Because git-merge sometimes wrongly reports conflicts, with our manual inspection, we labeled 136 of the 180 chunks as true conflicts, and 44 chunks as false conflicts. To facilitate tool evaluation, we also defined a systematic method of manual analysis toanalyze all program versions involved in each merging scenario, and to summarize the root causes as well as developers' resolution strategies. We further defined three novel metrics to evaluate merge tools. By applying five state-of-the-art tools to ConflictBench, we observed that ConflictBench is effective to characterize different tools. It helps reveal limitations of existing tools and sheds light on future research.

1. Introduction

In collaborative software development, programmers can create software branches for tentative feature addition and bug fixing. Periodically, they may *merge* (i.e., integrate) the program changes from distinct branches to release software with new features or bug fixes. Unfortunately, such a merge process is not always smooth due to the existence of *conflicts*. Here, **conflict** means that when branches edit the same text in divergent ways, the edits are incompatible with each other and cannot get co-applied to the same version of software (Git merge conflicts, 2023). For instance, as shown in Fig. 1, *l*-branch deletes a `for`-loop while *r*-branch updates the loop header. The two sets of edits cannot get applied simultaneously, so they conflict. Ghiotto et al. (2020) showed that 8%–21% of the merge trials in 5 open-source projects fail due to conflicts.

Version control systems (e.g., git) provide the basic tools (e.g., git-merge (Git - git-merge, 2023)) to merge branches and detect conflicts. Such tools treat programs as plain text and merge edits line-by-line. However, because they neglect the domain knowledge of programming languages and have implementation flaws, prior work pinpoints that these tools are weak in expressing differences and handling conflicts (Mens, 2002; Apel et al., 2011; Shen et al., 2019). To overcome the limitations of basic merge tools, researchers proposed new tools that observe the Java program syntax, to improve over the basic tools when merging Java programs (Apel et al., 2011, 2012; Zhu and He, 2018; Shen et al., 2019). Although the syntax-based merge tools have different approach design, little is known about the empirical comparison among those tools. When creating a software merge tool, researchers typically construct their own dataset to evaluate their own tool. No third-party benchmark is available to comprehensively evaluate all merge tools, and no systematic investigation has been done to compare those tools.

We believe that it is always important to define a third-party benchmark of software merge data for two reasons. First, by evaluating merge tools on the same third-party benchmark, researchers can empirically compare tools in a fair manner. Second, by revealing the limitations of existing tools, the benchmark can help reveal new directions for future tool design and implementation. Therefore, for this paper, we created a benchmark of real-world software merge conflict data. Before constructing the benchmark, we conducted a literature review for existing merge tools (Apel et al., 2011, 2012; Zhu and He, 2018; Shen et al., 2019, 2023; Zhu et al., 2022) and empirical studies on merge techniques (Cavalcanti et al., 2015, 2017, 2019; Seibt et al., 2022). We observed and discussed how merge tools were evaluated, identifying the following requirements that a good benchmark should satisfy:

☆ Editor: Shane McIntosh.

* Corresponding author.
E-mail addresses: bowenshe@vt.edu (B. Shen), nm8247@vt.edu (N. Meng).

https://doi.org/10.1016/j.jss.2024.112084
Received 17 July 2023; Received in revised form 18 April 2024; Accepted 24 April 2024
Available online 27 April 2024

- **Diversity:** It should cover a wide range of scenarios where merge happens, so that the dataset is representative.
- **True Conflicts:** It should include true conflicts between branch edits, to assess whether merge tools can identify the conflicts when two branches edit the same text in different ways.
- **False Conflicts:** It should include false conflicts, to assess whether a merge tool wrongly reports conflicts when the branches do not edit the same text simultaneously.
- **Conflict Resolutions:** It should include developers' resolutions to reported conflicts, to evaluate whether the tool-generated resolutions match human-crafted ones.

To satisfy all requirements mentioned above, we created our benchmark by crawling 208 popular open-source Java repositories (Shen et al., 2023). For each repository, we randomly sampled a commit that attempts to merge software branches via git-merge, and manually inspected the conflicts reported by git-merge to pick one satisfying our selection criteria (see Section 3). After including all picked conflicts into our dataset, we formulated our benchmark named ConflictBench. Among the 180 conflicts it contains, there are 136 true conflicts and 44 false ones. To facilitate tool comparison, we also classified conflicts based on the types of branch edits, the types of edited files, and developers' resolution strategies.

We applied five state-of-the-art merge tools to ConflictBench, to check whether our benchmark is effective in characterizing tools' effectiveness and in revealing differences between tools. The tools include KDiff3 (Eibl, 2007), FSTMerge (jFSTMerge, 2021; Apel et al., 2011), JDime (Apel et al., 2012), IntelliMerge (Shen et al., 2019), and AutoMerge (Zhu and He, 2018). We observed the following interesting phenomena in our experiments. KDiff3 has wider applicability than the other tools. JDime reported conflicts with the highest precision (92%), while AutoMerge reported the fewest conflicts (i.e., 17). KDiff3 achieved the highest resolution desirability (83%), meaning that the majority of merged versions it produces match developers' hand-crafted versions.

In this paper, we made the following research contributions:

- We defined a novel systematic method to classify merge-conflict data, and applied that method to manually create a benchmark of merge-conflict data named ConflictBench. This benchmark includes 180 merging scenarios with labeled true/false conflicts, types of branch edits, types of edited files, and developers' resolution strategies. No prior work characterizes conflicts in such a comprehensive and rigorous way.
- We defined three novel metrics to evaluate software merge tools: tool applicability, detection precision, and resolution desirability.
- We comprehensively evaluated five state-of-the-art software merge tools using ConflictBench, and observed interesting phenomena in terms of tool applicability, conflict-detection precision, and conflict-resolution desirability. No prior work does such an empirical evaluation of these tools or presents the novel findings we have.

In the following part of our paper, we will introduce the technical background (Section 2), our methodology of benchmark creation (Section 3), the ConflictBench dataset (Section 4), and experiments with ConflictBench (Section 5).

2. Background

This section first clarifies the terms used in software merge (Section 2.1). It then describes the three-way merge approach implemented by textual merge tools (e.g., git-merge), and tree-based software merge tools (Sections 2.2 and 2.3). Finally, it introduces the studies conducted to assess software merge tools, limitations of current studies, and our research motivation (Section 2.4).

Changes in local (*l*) branch	Changes in remote (*r*) branch
- for (BlockNode b : blocks) {	- for (BlockNode b : blocks) {
- if (b.predecessors().isEmpty())	+ for (BlockNode b : mth.getBlocks()) {
- {...}	if (b.predecessors().isEmpty())
- }	{...}
	}

Fig. 1. An exemplar merge conflict.

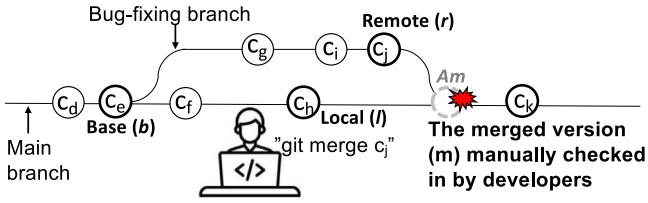

Fig. 2. Text-based merge tools (e.g., git-merge) can be used to merge software branches and reveal conflicts.

2.1. Terminology

In software repositories, basic merge tools (e.g., git-merge) can be used to tentatively merge software branches, and to detect conflicts in this process. Fig. 2 illustrates such a merge process. In this figure, the two horizontal lines visualize two software branches in a repository: the main branch and bug-fixing branch. Each node (except *Am*) represents a program commit checked in. When a developer uses the command "`git merge` c_j" to incorporate changes from the named commit c_j to the current commit c_h, git-merge treats the developer's commit c_h as **local version (*l*)**, considers the named commit c_j as **remote version (*r*)**, locates the common ancestor (i.e., predecessor) of both commits as **base version (*b*)**, in order to conduct three-way merge (see Section 2.2) and produce an **automatically merged version** (***Am***).

If git-merge detects no conflict, *Am* shows the merged software that incorporates all edits. Otherwise, if conflicts are detected, *Am* also denotes conflicts with specialized marks. For instance, Fig. 3 shows a conflict reported by git-merge, due to the divergent updates applied by *l* and *r*; git-merge generates a **conflicting chunk** in A_m to mark the conflicting edits. The tool uses "≪≪≪≪<" and "=======" to mark the unique edits from *l*; it also uses "=======" and "≫≫≫≫>" to mark the unique edits from *r*. Depending on their needs, developers may further edit *Am* before committing a final merged version *m* to the repository. Therefore, *m* may be identical to or different from *Am*. We use **merging scenarios** to refer to developers' merge trials. In software repositories, a typical merging scenario involves five program versions: the base *b*, the local *l*, the remote *r*, the automatically merged version *Am*, and developers' merged version *m*.

2.2. Line-based three-way merge

Given three program versions (i.e., base *b*, local *l*, remote *r*), the three-way merge approach in basic tools takes two steps to produce the automatically merged version *Am*. First, it locates the textual difference between *l* and *b* and that between *r* and *b*, treating them as line-based edits by individual branches. Second, if *l* and *r* apply identical edits, the common edits are applied once to produce *Am*; if only one version applies edits to a particular line, the edits are integrated into *Am*; otherwise, conflicts are reported as the branches apply divergent edits to the same line(s).

Line-based three-way merge has two major limitations: (1) content misalignment and (2) insufficient capabilities of combining edits. First, when it incorrectly aligns lines from different versions, the reported conflicts can be wrong. For instance, Fig. 4(a) is a wrongly reported conflict for a real-world merging scenario (Merge branch, 2014), where

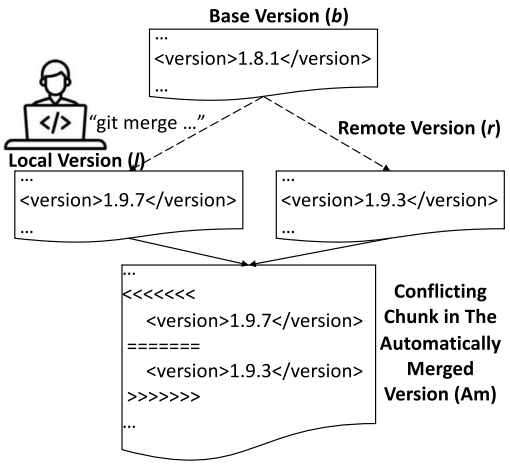

Fig. 3. A conflict reported by git-merge.

l deletes two lines and *r* updates a third line. Although the edits manipulate different lines and have no conflict, a typical line-based merge tool git-merge incorrectly aligns them and reports a conflict. Second, when branches simultaneously edit distinct parts of the same line, line-based merge cannot integrate those edits. As shown in Fig. 4(b), *l* updates the parameter list of a declared method and *r* updates the method name. Line-based merge considers both edits applied to the same text and thus reports a conflict. These limitations motivated current research of creating better merge tools.

2.3. Tree-based or syntax-based software merge tools

Various tree-based merge tools (Apel et al., 2011, 2012; Zhu and He, 2018; Shen et al., 2019) were recently created to overcome the two limitations mentioned above for line-based merge tools. For instance, FSTMerge (Apel et al., 2011) and IntelliMerge (Shen et al., 2019) parse Java programs, and create simplified parsing trees or graphs to represent the parent–child relationship between program entities (i.e., classes, methods, and fields). By matching entities across versions based on entity names and/or code content, both tools attempt to avoid entity-level misalignment. JDime (Apel et al., 2012) and AutoMerge (Zhu and He, 2018) create abstract syntax trees (ASTs) to model all syntactic constructs in Java code; they both compare and integrate branch edits based on tree-node matching to overcome the two limitations mentioned above. However, it is unknown how well these state-of-the-art tools overcome the limitations of line-based merge tools or how they compare with each other. Although some researchers conducted experiments to assess merge tools' capabilities of conflict detection (Cavalcanti et al., 2015, 2017, 2019; Seibt et al., 2022), they did not leverage any benchmark that labels true/false conflicts to measure the precision of tools' conflict detection capability. Also, existing studies do not leverage any benchmark that labels developers' manual conflict resolution strategies, to assess the desirability of tool-generated conflict resolutions.

2.4. Current assessments of software merge tools

A few studies were done to empirically compare software merge tools (Cavalcanti et al., 2015, 2017, 2019; Seibt et al., 2022). Specifically, Cavalcanti et al. (2015) applied a line-based tool and a semistructured merge tool to the same set of subject programs, to compare the number of conflicts reported. However, they did not analyze conflict reports to characterize false alarms. Another study by Cavalcanti et al. (2017) conducts pattern matching, to estimate the number of scenarios where FSTMerge and KDiff3 (a line-based merge tool similar to git-merge) can have false positives or false negatives. As described in the paper, the pattern-matching approach is designed to overestimate the additional false positives and false negatives of FSTMerge, while underestimating the additional false positives and false negatives of KDiff3. Thus, the estimates cannot precisely quantify the effectiveness comparison between tools. Two studies (Cavalcanti et al., 2019; Seibt et al., 2022) involve the manual inspection of 54–92 merging scenarios, to explore the false positive and false negative issues in conflicts detected by variants of the same tool or two distinct tools.

Existing studies have three limitations. First, they do not often examine the precision of the conflict detection capability for merge tools; the only two studies that inspected false positives and false negatives so far (Cavalcanti et al., 2019; Seibt et al., 2022) focused on a relatively small set of merging scenarios (i.e., 54–92 scenarios). We believe it necessary to characterize the limitations of current tools with more rigor by defining more metrics and using well-labeled datasets, so that researchers can come up with better tools in the future to detect or resolve conflicts more effectively. Second, no existing study examines how well the tool-generated conflict resolutions match developers' resolutions. If tools always generate resolutions differently from developers, developers cannot fully trust tool-generated merge results. Third, each of the current studies compared two or three approaches/tools simultaneously. We believe it necessary to compare more tools on the same dataset, to better understand the advantages or disadvantages of different tool implementations and merge algorithms.

The datasets used by existing studies only record the merging commits in various open-source projects. They do not label any merging scenario to have true/false conflicts, neither do they summarize or label developers' manual conflict resolution strategies. Thus, these datasets do not quite help researchers assess the precision of conflict detectors, neither do they facilitate people to evaluate the desirability of conflict resolutions proposed by merge tools.

The limitations of current studies and datasets motivated us to create a large-scale labeled dataset for the evaluation of software merge tools. The ground-truth labels should include merging scenarios with true conflicts as well as false conflicts, scenario characterizations in terms of branch-edit types or file types, and the merged versions produced by developers for those scenarios. We envision with such a benchmark, researchers and tool developers can better characterize the strengths and weaknesses of various merge tools, characterize the scenarios where certain tools succeed or fail, create better tools, and fairly compare tools in terms of conflict detection as well as resolution.

3. Methodology

This section first introduces our process of benchmark creation (Section 3.1), and then explains our manual analysis—the most important and challenging part in the whole process (Section 3.2).

3.1. The whole process of benchmark creation

As mentioned in Section 1, we considered four requirements when building the benchmark: diversity, true conflicts, false conflicts, and conflict resolution. To satisfy all requirements, we started our benchmark-construction process with the 208 open-source Java repositories mentioned by prior work (Shen et al., 2023). Shen et al. (2023) recently created a dataset of 208 open-source repositories that have merge conflicts. Specifically, the researchers ranked Java projects on GitHub based on their popularity (i.e., star counts), and cloned repositories for the top 1000 projects as their initial dataset. They then refined the dataset with two heuristics. First, they only kept the projects that can be built with Maven, Ant, or Gradle to ensure high software quality and program executability. Second, they removed tutorial projects as those projects are not real Java applications and may not show real-world merging scenarios. By crawling these repositories, we intended to quickly locate and extract diverse merging scenarios.

(a) A wrongly reported conflict due to the incorrect alignment of lines

Changes in local *l*:	Conflict reported by git-merge:
- import rx.operators.OperationAll; - import rx.operators.OperationAny; import rx.operators.OperationAsObservable; **Changes in remote *r*:** import rx.operators.OperationAll; import rx.operators.OperationAny; - import rx.operators.OperationAsObservable; + import rx.operators.OperatorAsObservable;	<<<<<<< import rx.operators.OperationAsObservable; ======= import rx.operators.OperationAll; import rx.operators.OperationAny; import rx.operators.OperatorAsObservable; >>>>>>>

(b) A conflict reported due to insufficient capabilities of combining edits

Changes in local *l*:	Conflict reported by git-merge:

- public void onError(HDI hdi, CI ci, T t) {
+ public void onError(HDI hdi, T t) {
 	
Changes in right *r*:
- public void onError(HDI hdi, CI ci, T t) {
+ public void onChunkError(HDI hdi, CI ci, T t) {
	<<<<<<<
	public void onError(HDI hdi, T t) {
	=======
	public void onChunkError(HDI hdi, CI ci, T t) {
	>>>>>>>

Fig. 4. Two exemplar conflicts to show limitations of line-based three-way merge.

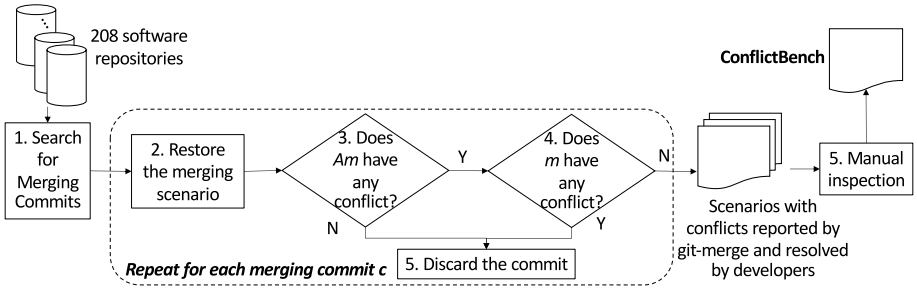

Fig. 5. Our crawling process takes five steps to identify conflict samples.

Table 1
Statistics of the 208 open-source repositories.

	#of Stars	Repository lifetime	# of Contributors	# of Commits	Program size (kB)
Minimum	2,225	4 yrs 8 mos	1	5	77
Maximum	71,169	14 yrs 10 mos	403	45,983	5,147,485
Median	6,046	9 yrs 7 mos	61	1,550	16,355

To investigate the representativeness of program data from Shen et al.'s work (Shen et al., 2023), after downloading the 208 repositories from GitHub, we measured 3 aspects of the projects: project popularity (i.e., number of stars), project maturity (i.e., lifetime of repositories), and level of development activity (e.g., number of contributors, number of commits, and program size). As of November 2023, all projects have been very popular, receiving 2225 – 71,169 stars. As shown in Table 1, these projects also seem mature, with their lifetime spanning between 4 years 8 months and 14 years 10 months. Each project has 1–403 contributors. The number of commits also varies a lot, in the range [5, 45,983]. The dataset has programs of very small sizes (e.g., 77 kB) and programs with very big sizes (e.g., 5,147,485 kB). The diversity of numeric measurements we obtained for individual projects implies the representativeness of this dataset.

We further clustered projects based on the number of conflicting merging scenarios they contain. As shown in Fig. 6, 25 of the projects contain no merge conflict in their repositories, 19 project repositories contain single conflicting scenarios, and 164 project repositories have multiple conflicting scenarios. Apache Cassandra contains the largest number of conflicting merging scenarios: 4172. All these numbers motivated us to sample one conflicting merging scenario in each repository, because (1) the number of conflicting merging scenarios varies so much across projects, and (2) our sample dataset can cover diverse scenarios from more repositories when each repository has a scenario sampled.

For the repositories downloaded from GitHub, we took five steps to search for merging scenarios in each repository, and to sample conflict data in those scenarios (see Fig. 5). With more details, in each repository, if a program commit has two parent commits (predecessors), we name it a **merging commit**, and use it to retrieve or restore the five program versions (l, r, b, Am, m) related to a merging scenario. Namely, the two parent commits are used as local version l and remote version r; the common ancestor of the parents is treated as base version b; Am is restored as git-merge is applied to l and r; the merging commit is considered as developers' merged version m. In Step 3, if the automatically merged version Am shows no conflicting chunk, we discard the merging scenario as it contains no conflict-related data. Otherwise, in Step 4, we further check whether m has any conflicting chunk reported by git-merge. If so, we discard the merging scenario because it lacks information of developers' conflict resolutions. At the end of Step 4, we obtained refined sets of conflict-related merging scenarios in 182 of the 208 repositories. Note that 26 of the repositories were filtered out because they have no merging commit satisfying the requirements mentioned above.

In Step 5, we manually inspected the refined set of each repository, to find conflicts to include into ConflictBench. If multiple scenarios are found in a repository, we randomly sampled one scenario for further analysis. If the sampled scenario has multiple conflicts reported by git-merge, we went over the conflict reports in sequence until finding one that satisfies the following criteria.

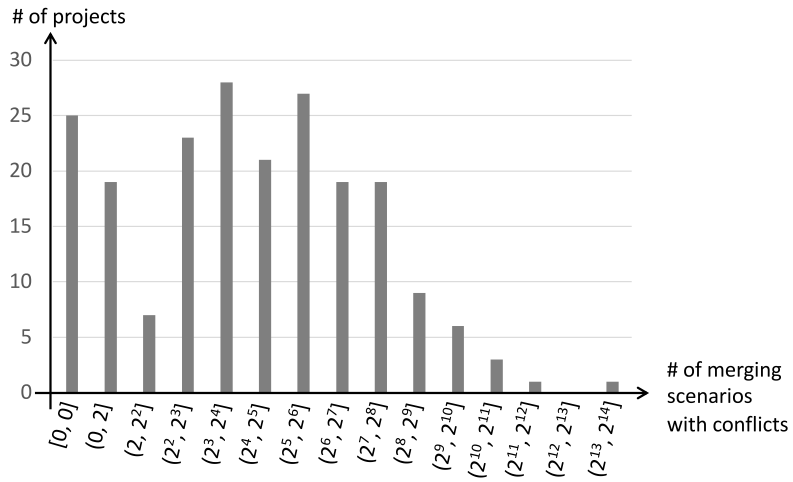

Fig. 6. Project distribution based on the number of merging scenarios with conflicts.

1. If both branches revise parts of the same file, we looked for the first conflicting chunk that involves no more than 20 lines of unique text from either branch.
2. If one branch or both branches edit the file as a whole (e.g., by deleting or moving it), the 20-line limit mentioned above does not apply. We simply included the conflict into our dataset.

We used the 20-line limit, because conflicts always become harder for comprehension and characterization when they involve more edits. Based on our experience so far, we are confident to properly analyze the conflicting chunks that involve no more than 20 lines of unique text by either branch.

At the end of this step, ConflictBench includes 180 conflicts from 180 of the repositories. We removed 2 repositories from the refined 182 repositories, because those 2 repositories have no conflict meeting the criteria mentioned above. For each conflict, we further labeled (1) whether it is a true conflict, (2) the types of edits applied by each branch, (3) the types of edited files, and (4) developers' resolution strategies. Section 3.2 details on our labeling procedure.

3.2. Details of our manual analysis

In our manual analysis, we took the open coding method to characterize and classify conflicts. Specifically, the two authors separately inspected some merge conflicts reported by git-merge, to come up with four initial classification methods: conflict classification based on the (1) relative positions of branch edits (i.e., applied to the same overlapping text regions or not), (2) edit type contrasts between branches, (3) types of edited files, and (4) developers' resolution strategies. For each merging scenario, we inspected five program versions: the base *b*, the local *l*, the remote *r*, the automatically merged version *Am*, and developers' merged version *m*. Afterwards, both authors independently inspected all conflicts, to manually characterize and label those conflicts. Finally, we compared the labels, to discuss and refine classification methods whenever divergence occurred. The discussion continued until we reached a consensus. After several iterations of discussion and reclassification, the authors settled down all categories and the classification labels.

3.2.1. Edit comprehension

To diagnose whether a reported conflict is true or false, we need to first comprehend the branch edits contributing to that conflict. Typically, git marks *l*- and *r*-edits with added lines (denoted with "+") and/or deleted lines (denoted with "−"), as shown in Fig. 4. However, such denotation is insufficient because it does not relate added with deleted lines to capture update operations. To facilitate the edit comparison between branches and conflict diagnosis, we tried to identify line updates in given branch edits. *If (1) a number of deleted lines are followed by the same number of added lines, and (2) each added line is similar or identical to the corresponding deleted line, then we interpret the edits as updates.* Here, to calculate the similarity of two given lines, we computed longest common character subsequence between those lines, and considered them similar if the subsequence contains at least 50% of characters from both lines. By detecting update operations, we can better align lines and edits across branches, to decide whether edits are applied to the same, overlapping, or different regions.

Additionally, when a branch revises multiple lines, we used the following criteria to characterize branch edits:

- If only one type of edits are applied by a branch, we use **I**, **D**, or **U** to label the branch edits, which letters separately denote insertion, deletion, and updates.
- If two types of edits are applied by a branch, we use **DI** (meaning deletion and insertion), **DU**, and **IU** to label the branch edits.
- When all three types of edits are applied, we simply use **DI** to label those edits for two reasons. First, the updated lines are closely related to surrounding added or deleted lines. Second, our 50%-threshold is imperfect and sometimes interpret similar edits in divergent ways.

With the labeling criteria mentioned above, we can characterize the edit types contributing to each reported conflict, classify samples accordingly to present the dataset diversity, and contrast the effectiveness of merge tools when they deal with different kinds of conflicts.

3.2.2. Conflict diagnosis

As git-merge may falsely report conflicts (see Section 2.2), we manually applied the following criteria in sequence to compare branch edits and to decide whether a sampled conflict is true or false:

- If branch edits manipulate different lines and there is no overlap between the manipulated lines, we label the reported conflict "False". For instance, Fig. 4(a) shows a false conflict, because the deleted lines in *l* and the updated line in *r* share no code in common.
- If branch edits are applied to overlapping or the same region, the reported conflict is labeled "True". Fig. 4(b) shows a true conflict, because *l* and *r* update the same line in divergent ways.
- If both branches insert distinct text at the same location (e.g., between two existing lines), we label the reported conflict "True". This is because even though both insertions can get applied simultaneously, it is hard to automatically decide the sequential order between them.

- If one branch inserts text between two existing lines, while the other branch deletes or updates both lines, we label the conflict "True". This is because if we consider the two lines as anchors to mark the insertion location, any update or removal of them both can make it very difficult to position insertion in the merged software.

3.2.3. Developers' resolutions

As with prior work (Shen et al., 2023), we adopted seven labels to describe developers' resolution strategies for true conflicts:

- **KL**: Keep all edits from *l*.
- **KR**: Keep all edits from *r*.
- **KL+KR**: Keep all edits from both branches.
- **ME**: Apply new manual edits, but no edit from *l* or *r*.
- **KL+ME**: Apply *l*-edits and new manual edits.
- **KR+ME**: Apply *r*-edits and new manual edits.
- **KL+KR+ME**: Apply edits from both branches, together with new manual edits.

The above-mentioned labels are self-explanatory, indicating distinct types of resolution strategies taken by developers. For instance, in a merging scenario, if all edits from *l* are included into developers' resolution and no edit from *r* is included, we use **KL** to summarize the resolution strategy. Nevertheless, there are still corner cases ambiguous to label, due to the similarity or relevance between branch edits. Thus, we further defined the following criteria to handle those complicated scenarios and to ensure consistent labeling:

- If branch edits are similar to each other while developers' merged version is identical to *l* (or *r*), we use **KL** (or **KR**) to summarize the resolution.
- If branch edits and the edits in developers' merged version (*m*) are all similar but not identical, we calculate the similarity between the edits in *m* and the edits by either branch. If the edits in *m* is more similar to *l*-edits, we label it **KL+ME**; otherwise, we use **KR+ME**.
- If *l* and *r* edit different parts of the same line and *m* applies both edits to the same line, we use **KL+KR** to label the resolution.

4. The dataset of conflictbench

In our dataset, we created a folder for each conflict sample. Inside that folder, we created four folders to keep track of separate versions of the single edited file. The four versions include *b*, *l*, *r*, and *m*; we denote the four folders with F_b, F_l, F_r, and F_m. If one version deletes the file, the corresponding folder is empty. We also created a spreadsheet to record additional information (e.g., classification labels and conflicting chunks) for all samples. Table 2 shows the 180 conflict samples included into ConflictBench. In this table, we classify conflicts based on the (1) truth or falsity of conflicts, (2) types of branch edits, and (3) types of edited files.

4.1. True or false conflicts

As shown in Table 2, 136 conflicts are labeled with "True" because branches apply divergent or incompatible edits to overlapping text regions; 44 conflicts are labeled as "False" because the branch edits are applied to non-overlapping regions, and they should have been co-applied automatically to the merged version. Notice that in our sampling process (see Section 3), we had no control over whether the selected conflicts are true or false. Therefore, the considerably large number of false conflicts (i.e., 44) implies (1) a lot of noises produced by git-merge when it reports conflicts, and (2) a significant improvement space for better merge tools.

Table 2
The 180 samples included in ConflictBench.

True? False?	Edit types	File types		Total per edit types
		Java	Non-Java	
True Conflicts	I vs. I	20	18	38
	U vs. U	14	16	30
	U vs. D	15	11	26
	DI vs. DI	6	3	9
	U vs. DI	5	3	8
	D vs. DI	2	2	4
	DI vs. DU	3	1	4
	IU vs. U	2	2	4
	IU vs. DI	3	1	4
	D vs. I	3	0	3
	DI vs. I	2	0	2
	D vs. D	0	1	1
	U vs. I	1	0	1
	U vs. DU	0	1	1
	D vs. DU	1	0	1
Subtotal		**77**	**59**	**136**
False Conflicts	I + U	7	1	8
	I + D	7	0	7
	I + DI	2	4	6
	I + N	4	1	5
	U + IU	2	1	3
	DI + IU	1	2	3
	DI + U	3	0	3
	U + U	1	1	2
	N + DI	0	2	2
	U + D	1	0	1
	D + DI	0	1	1
	D + D	1	0	1
	D + DU	0	1	1
	I + IU	0	1	1
Subtotal		**29**	**15**	**44**
Total		**106**	**74**	**180**

D means deletion. I means insertion. N means no edit from a branch contributing to the conflicting chunk. U means update.

4.2. Conflict classification based on edit types

Table 2 uses "vs." to contrast the edit types applied by individual branches for each true conflict, and uses "+" to compare the types of edits applied by separate branches for false conflicts. In addition to the edit types (i.e., **I**, **D**, **U**) mentioned in Section 3, this table also adopts **N** to mark two subcategories of false conflicts. Here **N** means that one of the branches contribute no edit at all to the reported conflicting chunk. As shown in Fig. 7, although *l* applies no edit while *r* deletes and inserts some text, git-merge wrongly interprets the scenario and falsely reports a conflict. By comparing edits across branches, we identified 15 subcategories in true conflicts. In particular, **I vs. I** is the largest subcategory of true conflicts, capturing scenarios where branches insert distinct content at the same location. We also identified 14 subcategories in false conflicts, with **I + U** as the largest one.

4.3. Conflict classification based on file types

Although all conflict samples are from Java repositories, we noticed that many of them reside in non-Java files. Thus, we also classified samples based on file types. As shown in Table 2, among the 136 true conflicts, there are 77 conflicts located in Java files and 59 ones located in non-Java files. Among the 44 false conflicts, 29 conflicts are from Java files while 15 conflicts are from non-Java files. In particular, the 74 non-Java files include 22 XML files, 13 Markdown documentation (.md), 10 Gradle files (.gradle), 7 property files (.properties), and 22 miscellaneous files. In our sampling process, we did not control what file to sample. Therefore, the conflict distribution among file types implies the diversity of our dataset.

Changes in local *l*: (no change)

```
 jacocoTestReport {
   reports {
     xml.enabled = true
     html.enabled = true
   }
```

Changes in remote *r*:

```
- jacocoTestReport {
-   reports {
-     xml.enabled = true
-     html.enabled = true
-   }
+ cobertura {
+   coverageFormats = ['html', 'xml']
```

Conflict reported by git-merge:

```
<<<<<<<
jacocoTestReport {
  reports {
    xml.enabled = true
    html.enabled = true
  }
=======
cobertura {
  coverageFormats = ['html', 'xml']
>>>>>>>
```

Fig. 7. A false conflict belonging to N + DI, where N means no edit from a branch contributing to the conflicting chunk and DI means a mixture of line deletion and insertion.

Table 3
Developers' resolutions to all conflict samples.

Strategy	# of true conflicts	# of false conflicts	Total
KL	61	11	72
KR	36	11	47
KL+KR	15	16	31
ME	3	1	4
KL+ME	2	1	3
KR+ME	5	0	5
KL+KR+ME	14	4	18
Sum	136	44	180

KL means "keep local version". KR means "keep remote version". KL+KR means "keep edits from both versions". ME means "apply manual edits". KL+ME means "apply local edits and new manual edits". KR+ME means "apply remote edits and new manual edits". KL+KR+ME means "apply edits from both branches and new manual edits".

4.4. Conflict classification based on resolution strategies

Table 3 presents our classification of conflict samples based on developers' resolution strategies. As shown in this table, most conflicts were resolved by either KL or KR, meaning that developers often resolve conflicts by keeping edits purely from one branch. KL+KR was adopted to resolve slightly more false conflicts than true ones (16 vs. 15), and 13 of the resolved true conflicts belong to I vs. I. The fourth most popular strategy is KL+KR+ME, which resolved 14 true conflicts and 4 false conflicts. The remaining three strategies (i.e., KL+ME, KR+ME, and KL+KR+ME) were applied a lot less often.

5. Experiment

Our overall research problem is how well ConflictBench helps characterize the effectiveness of existing software merge tools. To investigate this problem, we applied 5 state-of-the-art tools to the 180 merging scenarios and analyzed the tool results to explore 3 research questions (RQs):

- RQ1: How widely is a software merge tool applicable to merging scenarios?
- RQ2: When a software merge tool reports conflicts, how precise are those reports?
- RQ3: If a software merge tool can resolve conflicts, how well do those resolutions match developers' resolutions?

This section first introduces the five tools we adopted (Section 5.1). It then describes our experiment setting (Section 5.2), evaluation metrics (Section 5.3), and our experiment results for all three RQs (Section 5.4).

5.1. Five software merge tools

Among the papers recently published on the research topic of software merge, the following five tools have been frequently mentioned and used:

KDiff3 (Eibl, 2007) is another line-based software merge tool. According to its online handbook (Eibl, 2007), the tool outperforms other line-based merge tools by showing not only the changed lines, but also what has changed within these lines. Namely, it presents differences line-by-line and character-by-character. We downloaded the latest version compatible with Ubuntu 22.04 (i.e., the OS we used for our experiment)—1.9.5—via the command "apt install kdiff3".

FSTMerge (Apel et al., 2011; Cavalcanti et al., 2017; jFSTMerge, 2021), also referred to as semistructured merge, parses programs written in Java, C#, or Python. For each parsing tree, it generates a simplified program structure tree (PST). Within a PST, there is no low-level statement or expression node. Each inner node represents a high-level program structure (i.e., class, method, or field), and each leaf node represents the body implementation of a method or field. With PSTs, FSTMerge matches nodes between branches purely based on signatures. For each pair of matched nodes, it compares and integrates branch edits applied to the body implementation via textual merge (e.g., git-merge). We downloaded the latest version (commit ID: 81724157) of FSTMerge from its website (GitHub, 2017).

JDime (Apel et al., 2012), also referred to as structured merge, is similar to FSTMerge by parsing Java programs for tree structures. However, different from FSTMerge, JDime directly compares parsing trees to identify and integrate branch edits. Given the trees of *l* and *r*, JDime computes the largest common subtree, and adds matching information to those trees. It then takes the three trees enriched with matching information (i.e., trees of *b*, *l*, and *r*), to create a merged tree as result. We downloaded the version (commit ID: 63ffc342) of JDime from the tool's website (GitHub, 2019).

AutoMerge (Zhu and He, 2018) extends JDime, so it also merges software based on tree matching. While JDime does not attempt to resolve any conflict it reveals, AutoMerge was designed as an interactive approach to propose alternative resolutions for the detected conflicts. Each of the proposed resolutions may include some edits from a single branch or integrate edits from both branches. AutoMerge also has a mechanism of ranking alternative resolutions, so that the top-ranked resolution is very likely to satisfy developers. We downloaded the latest version (commit ID: 4e00b8ad) from the tool's website (thufv/automerge, 2018).

IntelliMerge (Shen et al., 2019) creates program element graphs (PEGs) to model program elements (e.g., Java classes, methods, and fields), as well as the relationship between elements (e.g., containment and access). It matches nodes based on the node content (e.g., method signature and its body implementation), as well as surrounding context

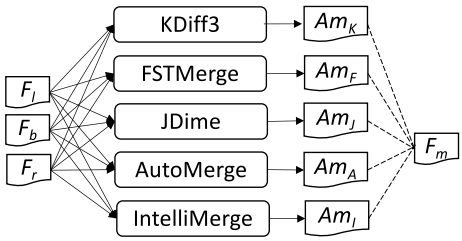

Fig. 8. The experiment settings. Here, F_l, F_b, F_r, and F_m are folders to separately hold programs of local, base, remote, and manually merged versions. Am_K, Am_F, Am_J, Am_A, and Am_I are automatically merged versions separately produced by KDiff3, FSTMerge, JDime, AutoMerge, and IntelliMerge.

(e.g., incoming and outgoing edges). Similar to FSTMerge, for each pair of matched nodes, IntelliMerge integrates branch edits via textual merge. However, different from all tools mentioned above, IntelliMerge can detect refactoring edits (e.g., method renaming) applied by either branch, and resolve the conflicts caused by those refactoring operations. We downloaded the latest version (commit ID: 1aa08901) of IntelliMerge from its website (GitHub, 2021).

5.2. Experiment setting

To use tools appropriately and compare them fairly, we downloaded the latest version (executable on our Ubuntu 22.04 desktop) of five tools—KDiff3, FSTMerge, JDime, AutoMerge, and IntelliMerge—and followed the usage instructions on the tool websites. Specifically, we used the following tool versions without fine-tuning any parameter or configuration: KDiff3-1.9.5, FSTMerge (commit ID: 81724157 as there is no tool release information), AutoMerge-1.0, JDime-0.5.0, and IntelliMerge-1.0.9. As shown in Fig. 8, for each merging scenario, we provided all tools three folders— F_b, F_l, F_r—which correspond to three program versions (b, l, r), so that each tool has sufficient information to detect conflicts and generate a merged version. If both branches revise the same file, the folders separately hold different versions of that file. If one branch removes a file, the corresponding folder is empty because the file does not exist.

After applying all tools to ConflictBench, we manually checked the tool results. If a tool reports nothing or throws runtime errors, we interpret the phenomena as indicators of tools' limitations. For each reported conflict, we examined whether ConflictBench labels it as a true or false conflict. For each resolved conflict, we checked whether the tool's resolution (i.e., merged version) is semantically equivalent to the human-crafted one. Notice that each tool may report or resolve multiple conflicts in one merging scenario. To ensure fair comparison among tools, we only focused on (1) the branch edits, conflicts, as well as resolutions labeled by ConflictBench, and (2) the tool-generated conflicts or resolutions for those labeled regions.

5.3. Metrics

We defined three metrics to evaluate software merge tools:

Tool Applicability (A) measures among all merging scenarios, for how many scenarios a tool can take inputs and produce outputs normally without throwing errors.

$$A = \frac{\#\text{ of scenarios processable}}{\text{Total }\#\text{ of labeled scenarios}} \tag{1}$$

A is within [0, 1]. "Processable" means given the three folders related to a merging scenario, a tool either reports a conflict or generates a merged version. The more scenarios a tool can process, the better applicability it has. All the following two metrics focus on the scenarios processable by individual tools.

Detection Precision (P) measures among all reported conflicts by a merge tool, how many of them are true conflicts:

$$P = \frac{\#\text{ of true conflicts}}{\text{Total }\#\text{ of reported conflicts}} \tag{2}$$

P varies within [0, 1]. Suppose that a tool reports A conflicts, while B of them are labeled as true conflicts in our dataset. Then the precision is B/A. Notice that given a merging scenario with true conflicts, if a tool does not report any conflict, the tool may overlook those conflicts or properly resolve all conflicts. Namely, missing true conflicts does not necessarily mean that software merge tools have poor capabilities of conflict detection. Thus, we decided not to assess tools by defining the metric detection recall, which calculates the percentage of true conflicts recalled by each tool.

Resolution Desirability (D) measures among all conflicts resolved by a tool, for how many of them the tool's resolutions match developers' resolutions. "Match" means a tool-generated merged version is semantically equivalent to the merged version developers hand-crafted.

$$D = \frac{\#\text{ of resolutions matching developers' resolutions}}{\text{Total }\#\text{ of resolutions generated by a tool}} \tag{3}$$

This metric also varies within [0, 1].

Different from prior work (Apel et al., 2011, 2012; Cavalcanti et al., 2015, 2019; Seibt et al., 2022), we intentionally avoid comparing tools solely based on the numbers of conflicts they report for two reasons. First, if a tool reports a large number of conflicts and many of which are false alarms (i.e., falsely reported conflicts), the tool is not reliable. Second, some tools can perfectly resolve conflicts and generate correctly merged software; even though these tools report few conflicts, we cannot conclude that these tools are bad conflict-detectors. We believe that conflict detection and resolution are two closely related capabilities of a merge tool, so we defined a metric for each of them. It is possible that some tools detect conflicts with high precision, but cannot resolve many conflicts. Meanwhile, some tools may detect conflicts with low precision, but is able to correctly merge branch edits that should have been reported as conflicts by other tools.

5.4. Results

Table 4 presents an overview of our experiment results with different tools. In the following subsections, we will describe and explain all results in detail.

5.4.1. Applicability comparison between tools

As shown in Table 4, KDiff3 has the highest applicability score—95%; FSTMerge has the second highest score—63%; the other tools have similar applicability (51%–52%). Specifically, given the 3 folders for each of the 180 samples, KDiff3 fails (i.e., either reports an error or problematically produces nothing) for 9 samples, 3 of which are in non-Java files and 6 samples are in Java files. All of these nine samples share the same conflicting pattern: one branch updates certain file while the other branch deletes that file (U vs. D). As our dataset includes in total 26 conflicts of the pattern U vs. D, KDiff3 can smoothly process the remaining 17 (i.e., 26 – 9) scenarios.

FSTMerge does not work normally for 67 of the samples; 20 samples are in Java files and 47 of the samples are located in non-Java files. In our dataset, there are more conflicts in Java files than in non-Java ones (106 vs. 74). Thus, FSTMerge is more likely to behave abnormally when dealing with non-Java files than when it processes Java files. Our observations mentioned above imply that both KDiff3 and FSTMerge have implementation flaws that hinder their applicability. AutoMerge reports errors for two kinds of conflicts:

1. The conflicts reside in non-Java files.
2. One branch edits a Java file, while the other branch manipulates the file system by removing that file or moving the entire file folder.

Table 4
Overview of experiment results.

Metrics	KDiff3	FSTMerge	JDime	AutoMerge	IntelliMerge
Tool Applicability (A)	95% (171/180)	63%(113/180)	52%(93/180)	52%(94/180)	51%(92/180)
Detection Precision (P)	84% (114/136)	74%(28/38)	92%(22/24)	88%(15/17)	79%(30/38)
Resolution Desirability (D)	83% (29/35)	44%(33/75)	68%(47/69)	64%(49/77)	54%(29/54)

AutoMerge works normally only when both branches apply edits inside Java files. It detects conflicts by parsing Java code and comparing parsing trees, so it was not designed to handle conflicts outside Java files. JDime and IntelliMerge share the same design limitation with AutoMerge. Additionally, both tools suffer from unknown implementation issues, which prevent them from properly handling one or two conflicts inside Java files. Specifically, JDime generates nothing for a true conflict, whose branch edits insert distinct content to the same location (I vs. I). IntelliMerge generates nothing for two conflicts. One of the conflicts is a true one, involving folder renaming by one branch and file insertion into the renamed folder by the other branch (U vs. I); the other conflict is a false one, where one branch inserts code and the other branch deletes code at a different location (I + D).

This experiment indicates that there is still considerable improvement space for four experimented tools (except KDiff3), so that they can become more applicable and usable.

> **Finding 1:** *KDiff3 and FSTMerge have better applicability among the five tools. Their applicability is limited by implementation issues, while the applicability of the other three tools (i.e., JDime, AutoMerge, and IntelliMerge) is mainly limited by approach design.*

5.4.2. The comparison of detection precision between tools

As shown in Table 4, JDime has the highest detection precision—92% (22/24); FSTMerge has the lowest detection precision—74% (28/38). The precision comparison among all tools is JDime > AutoMerge > KDiff3 > IntelliMerge > FSTMerge. The numerical comparison of true-conflict reporting among tools is KDiff3 > IntelliMerge > FSTMerge > JDime > AutoMerge. Namely, KDiff3 and AutoMerge separately reported the largest and smallest numbers of true conflicts (114 and 15). Similarly, the numerical comparison of false-conflict reporting among tools is KDiff3 > FSTMerge > IntelliMerge > JDime = AutoMerge. Once again, KDiff3 and AutoMerge separately reported the largest and smallest numbers of false conflicts (22 and 2), while JDime reported the same number of false conflicts as AutoMerge.

Three reasons can explain why KDiff3 reported the most true conflicts and most false conflicts. First, it is applicable to the most merging scenarios, to handle conflicts in both Java and non-Java files. Second, it treats source code as plain text and compares programs by ignoring the syntactic structures. Thus, it sometimes mismatches adjacent lines and falsely reports conflicts (see limitations of line-based merge tools mentioned in Section 2.2). Third, it does not map code changes to operations on AST nodes, so it cannot resolve conflicts when branches apply simultaneous edits to different AST nodes. Our comparison implies that if developers want to reveal as many merge conflicts as possible and have great tolerance of false-conflict reporting, they can consider using KDiff3. Otherwise, if developers have little tolerance of false-conflict reporting, they may consider JDime. FSTMerge does not seem to be a good option, as it did not report the largest number of conflicts while its precision rate is the lowest.

Table 5 presents the distribution of conflicts reported by each tool, among different subcategories based on edit types. According to this table, all tools except KDiff3 reported the most true conflicts of subcategory U vs. U for two reasons. First, this subcategory contains more conflicts than many other subcategories. Second, such conflicts are easy to detect but hard to resolve automatically. KDiff3 reported a lot more true conflicts of I vs. I than the other tools for two reasons. First, it is more applicable than other tools to merging scenarios. Second, when applicable, the other tools try to resolve conflicts while KDiff3 did not attempt to resolve any of the conflicts. In particular, the most popular resolution strategies applied by those tools for I vs. I conflicts include KL+KR and KL+KR+ME, meaning that those tools resolve conflicts by co-applying insertions from both branches. KDiff3, FSTMerge, and IntelliMerge reported the most false conflicts for subcategory I + U. Both JDime and AutoMerge reported a false conflict of I + U, and a false conflict of I + N.

Table 5
The conflicts reported by tools.

	Edit types	KDiff3	FSTMerge	JDime	AutoMerge	IntelliMerge
True Conflicts	I vs. I	34	3	3	2	4
	U vs. U	28	8	8	6	11
	U vs. D	17	6	2	1	1
	DI vs. DI	9	3	2	2	3
	U vs. DI	7	2	1	0	4
	D vs. DI	3	0	1	1	0
	DI vs. DU	4	2	0	0	3
	IU vs. U	3	2	2	0	2
	IU vs. DI	4	0	1	1	2
	D vs. I	2	1	1	1	0
	DI vs. I	1	1	1	1	0
	D vs. D	0	0	0	0	0
	U vs. I	0	0	0	0	0
	U vs. DU	1	0	0	0	0
	D vs. DU	1	0	0	0	0
Subtotal		**114**	**28**	**22**	**15**	**30**
False Conflicts	I + U	5	3	1	1	3
	I + DI	4	0	0	0	0
	I + D	2	1	0	0	1
	I + N	4	1	1	1	1
	U + IU	3	1	0	0	2
	DI + IU	2	1	0	0	0
	DI + U	0	2	0	0	1
	U + U	0	1	0	0	0
	N + DI	1	0	0	0	0
	U + D	0	0	0	0	0
	D + DI	0	0	0	0	0
	D + D	0	0	0	0	0
	D + DU	0	0	0	0	0
	I + IU	1	0	0	0	0
Subtotal		**22**	**10**	**2**	**2**	**8**
Total		**136**	**38**	**24**	**17**	**38**
Detection precision		**84%**	**74%**	**92%**	**88%**	**79%**

D means deletion. I means insertion. N means no edit from a branch contributing to the conflicting chunk. U means update.

All tools reported 9 conflicts in common, 8 of which are true conflicts and 1 is a false conflict. This implies that even if we try to combine all tools to report conflicts with a higher precision rate, there is still improvement space for future tools to detect conflicts more precisely.

> **Finding 2:** *JDime got the highest detection precision (92%), while FSTMerge acquired the lowest (74%). JDime and AutoMerge reported the smallest number of false conflicts (i.e., 2), while KDiff3 reported the largest number of false conflicts (i.e., 22). AutoMerge and KDiff3 separately reported the smallest and largest number of true conflicts (i.e., 15 and 114).*

Table 6
The comparison between tool-generated versions and developers' merged versions.

Resolution strategy	KDiff3		FSTMerge		JDime		AutoMerge		IntelliMerge	
	M(atched)	U(nmatched)	M	U	M	U	M	U	M	U
KL	4	4	19	15	19	7	19	9	11	12
KR	7	0	5	11	8	9	8	11	7	4
KL+KR	12	0	7	8	14	0	15	0	8	4
ME	1	0	0	3	0	1	0	1	0	1
KL+ME	0	0	1	0	1	1	1	1	1	1
KR+ME	2	0	0	0	1	0	2	0	1	0
KL+KR+ME	3	2	1	5	4	4	4	6	1	3
Total	29	6	33	42	47	22	49	28	29	25

KL means "keep local version". KR means "keep remote version". KL+KR means "keep edits from both versions". ME means "apply manual edits". KL+ME means "apply local edits and new manual edits". KR+ME means "apply remote edits and new manual edits". KL+KR+ME means "apply edits from both branches, and new manual edits".

5.4.3. The comparison of resolution desirability between tools

According to Table 4, KDiff3 acquired the highest resolution desirability—83%, while FSTMerge obtained the lowest resolution desirability—44%. Our measurement for AutoMerge (64%) is higher than that for IntelliMerge (54%) but lower than that for JDime (68%). AutoMerge generated the most resolutions—77, while KDiff3 produced the fewest resolutions—35.

Table 6 shows the breakdown of conflicts resolved by each tool, based on developers' resolution strategies. **M** means for a given conflict, the tool-generated version matches or is semantically equivalent to developers' version, while **U** means that the two versions are unmatched. According to this table, the versions produced by FSTMerge successfully match developers' merged versions for 33 cases, but fail to match for 42 cases. JDime worked much better than FSTMerge, as it generated many more matched versions than FSTMerge (47 vs. 33), but a lot fewer unmatched versions (22 vs. 42). Compared with JDime, AutoMerge created slightly more matched versions (49 vs. 47), and even more unmatched versions (28 vs. 22). Among all tools, IntelliMerge and KDiff3 produced the fewest versions matching developers' versions—29. However, as KDiff3 also produced the fewest versions not matching developers' versions—6, its desirability is the highest. AutoMerge has the largest number of resolved versions to semantically match developers' resolved versions (i.e., 49); FSTMerge has the largest number of unmatched resolutions (i.e., 42).

Our observations imply that if developers want to take full advantage of tools' conflict-resolution automation and have great tolerance for wrongly resolved conflicts (e.g., they rely on compilation or automatic testing to always capture wrong resolutions), they can consider using AutoMerge. Otherwise, if developers want to cautiously leverage tools' resolution capability and have little tolerance for wrongly resolved conflicts, they may use KDiff3. FSTMerge does not seem to be a good option either way, as it did not produce the most resolutions but generated the largest number of wrong resolutions.

Most of the conflicts resolved by each tool fall into the subcategories KL, KR, and KL+KR. This is because the majority of conflicts included in our dataset were manually resolved via KL, KR, and KL+KR. Most of the matched versions produced by each tool also fall into these three subcategories. To further investigate tools' resolution capabilities, we also created Table 7 to present the actual resolution strategies applied by each tool. By comparing Table 7 with Table 3, we observed that all tools' distributions of resolution strategies are different from the distribution of developers' resolutions. This implies that when resolving conflicts, existing tools give little consideration to developers' preferences for resolution strategies and future tools may observe developers' preferences to better merge software.

Specifically, among the studied tools, KDiff3 resolved the fewest cases (i.e., 35) but reported the most conflicts (i.e., 136). KDiff3 resolved 23 of the 35 conflicts via KL+KR, while developers resolved most conflicts via KL or KR. Within the merged versions KDiff3 created, six versions do not match developers' versions for two reasons. First, KDiff3 managed to integrate branch edits via KL+KR for five cases; however, developers resolved four of the conflicts via KL and resolved a fifth conflict via KL+KR+ME. Second, in a sixth conflicting scenario, although both KDiff3 and developers resolved conflicts via KL+KR+ME, the produced versions are different. In particular, one branch inserts a Java file to an existing folder/package, while the other branch renames that folder/package. Developers' resolution integrates both edits and further updates the package name inside that inserted Java file for change consistency; however, KDiff3's resolution only partially integrates branch edits by keeping two versions of the renamed folder.

Table 7
The comparison between resolution strategies applied by tools.

Resolution strategy	KDiff3	FSTMerge	JDime	AutoMerge	IntelliMerge
KL	4	9	9	9	10
KR	5	4	1	1	13
KL+KR	23	0	9	9	12
ME	0	14	2	2	0
KL+ME	0	21	14	16	3
KR+ME	1	6	6	7	4
KL+KR+ME	2	21	28	33	12
Total	35	75	69	77	54

KL means "keep local version". KR means "keep remote version". KL+KR means "keep edits from both versions". ME means "apply manual edits". KL+ME means "apply local edits and new manual edits". KR+ME means "apply remote edits and new manual edits". KL+KR+ME means "apply edits from both branches, and new manual edits".

FSTMerge resolved conflicts by mainly applying three strategies: ME, KL+ME, and KL+KR+ME; meanwhile, the three most popular resolution strategies manually applied by developers include KL, KR, and KL+KR. FSTMerge applies new edits in 62 of the 75 cases, although the majority of these extra edits are about unnecessary formatting changes (e.g., inserting or deleting empty lines, changing method declaration orders, or modifying code indentation). We also inspected the 42 cases incorrectly resolved by FSTMerge and summarized 3 major root causes. First, FSTMerge shows preferences to KL or KL+ME in nine cases where developers' resolutions are KR, KL+KR, or KL+KR+ME. Second, in 10 cases, FSTMerge wrongly created duplicates for edited lines. Third, in 11 cases, FSTMerge kept some lines that should be removed from merged versions.

JDime resolved 42 of the 69 conflicts via KL+ME or KL+KR+ME. We observed JDime to often resolve conflicts by applying extra edits on top of branch edits; most of the extra edits were about formatting changes. We inspected the 22 cases incorrectly resolved by JDime, and observed that JDime managed to integrate all branch edits into merged versions for 17 cases. However, developers resolved these 17 conflicts mainly via KL or KR. Namely, JDime created merged versions by integrating branch edits with its best effort, while developers do not always merge in as many branch edits as possible.

AutoMerge resolved 49 of the 77 conflicts via KL+ME or KL+KR+ME. It also applied unnecessary formatting changes when resolving conflicts. We observed that for all the 69 conflicts resolved by JDime, AutoMerge produced exactly the same merged versions as JDime. This

is because AutoMerge extended JDime, and shares most of its implementation with JDime. AutoMerge outperformed JDime by automatically resolving eight more conflicts, although six of those resolutions do not match developers' resolutions.

IntelliMerge resolved 37 of the 54 conflicts via KR, KL+KR, and KL+KR+ME. Within the merged versions it produced, 25 versions do not match developers' versions for 3 reasons. First, IntelliMerge shows preferences to KR in seven cases, where developers' resolutions are KL, KL+KR, KL+KR+ME, or ME. Second, in six cases, IntelliMerge applied KL+KR while developers resolved five of the cases via KL or KR. Third, in five cases, IntelliMerge kept some lines removed by one of the branches, while developers excluded those lines from their merged versions.

Our experiments also imply that if developers are flexible with the unexpected formatting changes introduced by software merge tools, they can use AutoMerge for its better resolution automation. However, if developers are picky about the coding style of generated merged versions and do not want programs reformatted by tools, they can think about using KDiff3 or IntelliMerge.

Finding 3: *AutoMerge resolved the most conflicts (i.e., 77) in our dataset, while KDiff3 resolved the fewest (i.e., 35). KDiff3 acquired the highest resolution desirability—83%, while FSTMerge got the lowest—44%.*

6. Threats to validity

Threats to external validity. We limited the size of manually inspected conflicting chunks. Namely, each conflicting chunk should have no more than 20 lines of unique lines from either branch. We used this 20-line limit, because conflicts always become harder for comprehension and characterization when they involve more edits. Based on our experience so far, we are confident to properly analyze the conflicting chunks that have no more than 20 lines of unique text by either branch. To examine the generalizability of our observations to larger conflict chunks, we also manually inspected another 10 randomly sampled chunks with more than 20 lines of unique text by either branch. We found that our major findings based on the smaller conflicting chunks generalize well to larger chunks. For instance, on this extra 10-sample dataset, we also observed that KDiff3 has the best applicability and reports the most conflicts; tree-based software merge tools generally resolved more conflicts than KDiff3, although the resolutions do not necessarily better match developers' manual resolutions.

Our study is based on the 180 conflicts extracted from 180 Java project repositories on GitHub. The characterization of conflicts, the observed resolutions, and our experiment results may not generalize well to other conflicts, other projects, other programming languages, or other hosting platforms (e.g., BitBucket). However, as this study involves a lot of manual analysis, we have spent a lot of time (i.e., over one year) creating the dataset, running experiments, analyzing data, and validating our manual inspection results. In the future, we plan to mitigate the issue by including more conflicts into ConflictBench.

Threats to construct validity. For benchmark creation and tool evaluation, we manually inspected (1) the sampled conflicts, (2) the merged versions hand-crafted by developers, and (3) the merged versions output by tools. It is possible that our manual analysis is subject to human bias and restricted by our domain knowledge, but our unscalable manual analysis is very important and irreplaceable to characterize conflicts and their resolutions. To alleviate the problem of manual analysis, both authors independently examined all data. They cross-checked each other's analysis results, and actively discussed the instances they disagreed upon until clarifying the labeling criteria and reaching a consensus. The authors inspected all data iteratively, to ensure that their analysis results are always consistent with the adjusted labeling criteria.

Threats to internal validity. Our heuristics to identify line updates may underestimate the number of update operations actually applied by developers. Specifically, we interpret edits as updates if (1) a number of deleted lines are followed by the same number of added lines, and (2) each added line is at least 50% similar to the corresponding deleted line. Based on (1), if one line is split into two lines, our manual analysis may interpret the change as one deleted line followed by two added lines, instead of a single update. Based on (2), if line A is updated to line B and B is not very similar to A (i.e., with less than 50% similarity), our manual analysis interprets the change as one deleted line followed by an added line, instead of an update.

Such misinterpretation can impact our conflict classification based on edit types, and may influence the number of conflicts falling into the categories related to "U" and "DI". For instance, some "DI vs. DI" cases can be relabeled as "U vs. U". However, based on our experience, only a few cases have such arguable labels. More importantly, such heuristics do not impact our labeling for true/false conflicts in different projects. No matter whether a line is treated to be deleted or updated, as long as two branches manipulate the same line(s) divergently, we consider them to conflict. Correspondingly, if two branches simultaneously manipulate different lines, we treat them to have no conflict.

7. Related work

The related work includes empirical studies on merge conflicts and on merge techniques.

7.1. Empirical studies on merge conflicts

Several studies characterize the relationship between merge conflicts and software maintenance (Estler et al., 2014; Ahmed et al., 2017; Leßenich et al., 2018; Mahmoudi et al., 2019; Owhadi-Kareshk et al., 2019). For instance, Estler et al. (2014) surveyed 105 student developers, and found that the lack of awareness (i.e., knowing "who's changing what") occurs more frequently than merge conflicts. Leßenich et al. (2018) surveyed 41 developers and identified 7 potential indicators (e.g., number of changed files in both branches) for merge conflicts. With further investigation of those indicators, the researchers found that none can predict the conflict frequency. Similarly, Owhadi-Kareshk et al. defined nine features (e.g., number of added and deleted lines in a branch) to characterize merging scenarios; they trained a machine-learning model that predicts conflicts with 57%–68% accuracy (Owhadi-Kareshk et al., 2019). Ahmed et al. (2017) studied how bad design (code smells) influences merge conflicts; they found that entities that are smelly are three times more likely to be involved in merge conflicts. Mahmoudi et al. (2019) studied the relationship between refactorings and merge conflicts; they observed that refactoring operations are involved in 22% of merge conflicts.

Similar to these studies, our research also constructs a dataset of merge conflicts and characterizes various aspects of those conflicts. However, it is different in two aspects. First, we characterized the conflicts reported by git-merge in terms of (1) the truth/falsity of reports, (2) types of branch edits, (3) types of edited files, and (4) types of resolution strategies. Second, we conducted an empirical study of five state-of-the-art merge tools by applying them to our dataset. No prior work does that.

Some studies analyze the conflicts reported by textual merge tools (e.g., git-merge) (Yuzuki et al., 2015; Nguyen and Ignat, 2018; Ghiotto et al., 2020; Brindescu et al., 2020; Pan et al., 2021; Shen et al., 2023). Specifically, Yuzuki et al. inspected hundreds of conflicts (Yuzuki et al., 2015). They observed that conflicting updates caused 44% of conflicts to the same line of code, and developers resolved 99% of conflicts by taking either the left- or right- version of code. Brindescu et al. (2020) manually characterized hundreds of conflicts in terms of the AST diff size, LOC diff size, and the number of authors. 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). Pan et al. (2021) explored the merge conflicts in Microsoft Edge; they classified those conflicts based on file types, conflict locations, conflict sizes, and conflict-resolution patterns. Driven by their empirical study, the researchers further investigated to use program synthesis for conflict resolution.

Nguyen and Ignat (2018) analyzed all merging scenarios in four project repositories. They found that git-merge falsely reports many textual conflicts, when concurrent edits are applied to adjacent code instead of same lines. Ghiotto et al. (2020) studied the textual conflicts found in 2731 open-source Java projects. They (1) characterized conflicts in terms of the number of chunks, size, and programming language constructs involved, (2) classified developers' resolution strategies, and (3) analyzed the relationship between conflict characteristics and resolution strategies. Shen et al. (2023) applied git-merge, automatic build, and automatic testing in sequence to reveal and study three kinds of conflicts: textual conflicts, build conflicts, and test conflicts. Build conflicts refer to the merging scenarios where branch edits get merged smoothly, but the merged versions have build errors. Test conflicts refer to the scenarios where branch edits can be merged, but the merged versions fail at least one test case.

None of the studies mentioned above define (1) a systematic way of classifying or characterizing textual conflicts, or (2) metrics to measure tool effectiveness. Also, they did not apply as many merge tools as we did to the same dataset for empirical comparison. As our dataset is similar to the datasets mentioned in prior work, it is natural to think about reusing existing datasets instead of creating a new one for tool evaluation. When we initially did our research, we actually tried to reuse the dataset by Shen et al. (2023), as that dataset characterizes conflicts in terms of the truth/falsity of conflicts, edit types, edit locations, and resolution strategies. Unfortunately, because some characteristic labels (e.g., types of branch edits) are not precise and the imprecision can jeopardize the rigor of tool evaluation, we spent lots of time defining and refining our criteria for conflict characterization, reanalyzing and refining the data based on our criteria, and validating analysis results via cross-checking between authors. Our detailed explanation in Section 3.2 justifies and clarifies our manual labeling criteria, all of which are our new contributions compared with prior work (Shen et al., 2023).

7.2. Empirical studies on software merge tools

A few studies were recently conducted to empirically compare different merge tools (Cavalcanti et al., 2015, 2017, 2019; Seibt et al., 2022). Specifically, Cavalcanti et al. (2015) created a dataset of 3266 merging scenarios, to compare FSTMerge with git-merge. They observed that among all scenarios, FSTMerge reported fewer conflicts in 1804 scenarios, git-merge reported fewer conflicts in 283 scenarios, and both tools reported the same number of conflicts in 1179 scenarios. However, this study does not analyze the false positives (wrongly reported conflicts) of either tool. Our empirical comparison between FSTMerge and KDiff3 (a line-based merge tool similar to git-merge) confirmed the finding that FSTMerge reported fewer conflicts than line-based merge tools. We identified two reasons to explain this observation. First, FSTMerge is less applicable; it could not handle some merging scenarios, let alone to detect conflicts in those scenarios. Second, FSTMerge resolved more conflicts.

To overcome the limitation of prior work (Cavalcanti et al., 2015), in a later study (Cavalcanti et al., 2017), the same research group summarized patterns to characterize the (a) scenarios where FSTMerge produces more false positives or false negatives than KDiff3 (a tool similar to git-merge), and (b) scenarios where KDiff3 produces more false positives or false negatives than FSTMerge. Afterwards, they wrote scripts to locate scenarios matching those patterns and compute overestimated numbers for (a), as well as underestimated numbers of (b). By comparing the estimates between tools, Cavacanti et al. concluded that the number of false positives is significantly reduced when using FSTMerge; FSTMerge's false positives are easier to analyze and resolve than those of KDiff3. However, such a tool comparison is not reliable for two reasons. First, the comparison between overestimates and underestimates does not precisely quantify the effectiveness difference between tools. Second, the summarized pattern set does not cover all scenarios where more false positives/negatives can occur. For instance, as mentioned in the paper, the researchers estimated the additional false positives of FSTMerge, by assuming those false positives to occur only when one branch renames an entity involved in the tool-reported conflict. Nevertheless, based on our experiment, FSTMerge sometimes reported more false positives when edits are simultaneously applied to adjacent instead of same lines, even though neither edit renames any entity.

Cavalcanti et al. (2019) compared FSTMerge with JDime. They found that (i) JDime reported more conflicts than FSTMerge (4793 vs. 4732); (ii) FSTMerge detected conflicts with more false positives and JDime got more false negatives. These two observations seem contradictory to each other because if (ii) is correct, (i) is probably incorrect and FSTMerge could have reported more conflicts than JDime. However, this study does not explore why JDime obtained more false negatives, neither does it explain whether the conflict resolution by either tool matches developers' manual resolution. Our study partially confirmed (ii) but refuted (i): We observed FSTMerge to report more instead of fewer conflicts than JDime; FSTMerge's detection precision is lower, meaning that the tool has more false positives.

Seibt et al. (2022) empirically compared three alternative merge algorithms implemented in JDime: unstructured merge (i.e., line-based merge), semistructured merge (i.e., the algorithm in FSTMerge), and structured merge (i.e., the default algorithm in JDime). Among the algorithms, the complexity comparison is "structured merge (S) > semistructured merge (SS) > unstructured merge (US)". Seibt et al. observed that using more complex merge strategies leads to a statistically significant decrease in the number of conflicting merges. The researchers also manually inspected 92 reported conflicts; they observed that (1) many of the false positives reported by US are due to the branch edits co-applied to adjacent lines of source code, and (2) S reported false positives mainly due to mismatches of AST nodes.

Our study confirms the numerical comparison of conflict reporting between unstructured merge, semistructured merge, and structured merge (Seibt et al., 2022): We also found JDime to report fewer conflicts than FSTMerge, and FSTMerge to report fewer conflicts than KDiff3. However, we believe that the reduction of conflict reports does not necessarily imply technical advancement. According to our study, FSTMerge reported fewer conflicts than KDiff3 due to its worse applicability and stronger resolution-generation capability, although more of FSTMerge's resolutions are undesirable than KDiff3's. JDime reported fewer conflicts than FSTMerge due to its worse applicability, more precise conflict detection, and better conflict-resolution capability. When choosing among candidate merge tools, users should not base their selection solely on the number of reported conflicts. Instead, they should assess tools' capabilities in different dimensions (i.e., applicability, detection precision, and resolution desirability) and find the one with the best trade-off fitting into their circumstances.

Different from all studies mentioned above, our work has more technical depth as it compares more tools in a more comprehensive way. Although existing studies mainly compare merge tools based on the number of conflicts they report, we chose not to do that. Our insight is that multiple factors can simultaneously influence the number of conflicts a tool can report: the tool's applicability, its precision of conflict detection, and the power of conflict resolution. Namely, the fact that a tool reports fewer conflicts does not necessarily mean that it is better. To tease apart the influence of different factors, we carefully formulated ConflictBench—a benchmark that includes the oracle for true/false conflicts residing in different kinds of files, and developers' resolutions. After applying five merge tools to ConflictBench, we analyzed the tools' outputs with high rigor and characterized scenarios where tools work differently.

8. Lessons learned

Our research provides insights on the evaluation of software merge tools, empirical comparison between merge tools, and future research directions.

Evaluation of Software Merge Tools: Although researchers created large-scale datasets to evaluate software merge tools, their evaluation mainly focuses on counting the number of conflicts reported by individual tools. The basic assumption is that the fewer conflicts reported by a tool, the better that tool is. However, we noticed that multiple factors contribute to the overall quality of a tool, including (1) the applicability of the tool, (2) the precision of conflict detection, (3) the correctness of tool-generated merged versions, and (4) the alignment of that tool's resolution preference with developers' resolution preference. None of these factors can be easily assessed via conflict counting, which requires an evaluation dataset to include more hand-crafted ground truth.

Empirical Comparison between Tools: Our experiments corroborate the finding by prior work that when handling conflicts in Java files, JDime and IntelliMerge outperform FSTMerge (Apel et al., 2012; Cavalcanti et al., 2019; Shen et al., 2019; Seibt et al., 2022). However, we noticed that the better effectiveness of JDime and IntelliMerge should not be only attributed to tool design. The implementation quality of tools matters a lot. Namely, FSTMerge contains implementation issues that jeopardize the tool's applicability, and its capability of conflict detection as well as resolution. In comparison, JDime and IntelliMerge have higher-quality implementations. Although AutoMerge was proposed to extend and improve over JDime, our evaluation does not witness the outperformance of AutoMerge. Our research empirically compared JDime, IntelliMerge, and AutoMerge for the first time; no prior work does that. These three tools complement each other in terms of design choices. JDime has better capabilities of conflict detection as well as resolution than both AutoMerge and IntelliMerge.

Future Research Directions on Software Merge: We recommend future researchers to empirically compare their new tools against several existing merge tools, using a variety of metrics instead of purely comparing the conflict counts. This is because existing tools make different design choices, to achieve different trade-offs between the tools' applicability and effectiveness of conflict handling. We recommend researchers to adopt ConflictBench, as its ground truth characterizes conflicts from different angles, and thus can facilitate deeper tool comparison. We also recommend future tools to resolve conflicts with consideration on developers' preferences, as we observed a gap between current tool-generated resolutions and human-crafted ones. One potential direction can be (1) training a machine-learning model to predict developers' preference given a conflict, (2) combining existing tools to generate alternative resolutions for that conflict, and (3) leveraging the model to select the best resolution.

Evaluation Extension to Projects in Other Programming Languages: Our research focuses on Java because the state-of-the-art tools predominantly address textual conflicts in Java programs. Some of our observations can generalize well to projects in other languages. For instance, KDiff3 is a line-based merge tool. It merges software without considering any program syntax; thus, our observations of KDiff3 generally hold regardless of the programming language in use. FSTMerge can handle Java, C#, and Python programs; therefore, our observations derived from Java programs can generalize to programs written in either of the other two languages. JDime, IntelliMerge, and AutoMerge are Java-specific, so they are inapplicable to other languages. Unless tool builders reimplement the same tools for other languages, it is hard to tell how well our observations for these tools generalize. Furthermore, some language-specific features can jeopardize the generalizability of our findings. For instance, Python has strict rules on indentation, while Java has none. Thus, even if some tools can correctly merge Java programs, their reimplementation may incorrectly merge Python programs due to formatting changes. To evaluate the tool effectiveness in projects of other languages, more effort is required to create new datasets and apply all usable tools to those datasets.

9. Conclusion

Software merge has been very important and challenging in software engineering practice, and many researchers proposed a variety of tools to help with the software merge process. We believe that many factors can contribute to the overall quality of a tool, so we argue that merge tools should be assessed in different perspectives with ground-truth datasets that characterize conflicts in a variety of ways.

To assess these factors and better compare the quality of different merge tools, we created a new benchmark named ConflictBench to label true/false conflicts, the types of branch edits contributing to those conflicts, the files where conflicts reside, and developers' conflict resolutions. We also defined three metrics for tool evaluation: tool applicability, detection precision, and resolution desirability. Furthermore, we experimented with five merge tools: KDiff3, FSTMerge, JDime, AutoMerge, and IntelliMerge, to demonstrate (1) the usage of our benchmark in tool evaluation and (2) ConflictBench's effectiveness of showing divergence in tools' effectiveness. Although our dataset is not as large as the datasets used by prior work, its construction requires for a lot more manual effort. Our dataset contains more valuable label information for deeper analysis; thus the evaluation based on ConflictBench can provide deeper insights in the pros as well as cons of existing tools. Our research will shed light on future research of software merge.

CRediT authorship contribution statement

Bowen Shen: Writing – original draft, Software, Methodology, Investigation, Formal analysis, Data curation. **Na Meng:** Writing – review & editing, Writing – original draft, Supervision, Resources, Project administration, Methodology, Funding acquisition, Formal analysis, Data curation, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Batory, Don	The University of Texas at Austin
He, Fei	Tsinghua University (China)
Kim, Miryung	University of California, Los Angeles
McKinley, Kathryn S.	Google Inc.
Perry, Dewayne E	The University of Texas at Austin
Muhammad Ali Gulzar	Virginia Tech
Shmatikov, Vitaly	Cornell Tech
Cai, Haipeng	Washington State University
Jaeger, Trent	The Pennsylvania State University
Mytkowicz, Todd	Microsoft Research Lab — Redmond
Servant, Francisco	Universidad de Málaga (Spain)
Tilevich, Eli	Virginia Tech
Wang, Xiaoyin	University of Texas at San Antonio
Yao, Daphne	Virginia Tech
Zhong, Hao	Shanghai Jiaotong University (China)

Data availability

Our program and data are available at https://github.com/UBOWENVT/ConflictBench.

Acknowledgment

This works was supported by NSF, USA-1845446 and NSF, USA-2006278.

References

Ahmed, I., Brindescu, C., Mannan, U.A., Jensen, C., Sarma, A., 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, pp. 58–67. http://dx.doi.org/10.1109/ESEM.2017.12.

Apel, S., Lessenich, O., Lengauer, C., 2012. Structured merge with auto-tuning: Balancing precision and performance. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. In: ASE 2012, ACM, New York, NY, USA, pp. 120–129. http://dx.doi.org/10.1145/2351676.2351694, URL http://doi.acm.org/10.1145/2351676.2351694.

Apel, S., Liebig, J., Brandl, B., Lengauer, C., Kastner, C., 2011. Semistructured merge: Rethinking merge in revision control systems. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. In: ESEC/FSE '11, ACM, New York, NY, USA, pp. 190–200. http://dx.doi.org/10.1145/2025113.2025141, URL http://doi.acm.org/10.1145/2025113.2025141.

Brindescu, C., Ahmed, I., Jensen, C., Sarma, A., 2020. An empirical investigation into merge conflicts and their effect on software quality. Empir. Softw. Eng. 25 (1), 562–590. http://dx.doi.org/10.1007/s10664-019-09735-4.

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

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

Cavalcanti, G., Borba, P., Seibt, G., Apel, S., 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. ASE '19, IEEE Press, pp. 1002–1013. http://dx.doi.org/10.1109/ASE.2019.00097.

Eibl, J., 2007. The KDiff3 Handbook, https://kdiff3.sourceforge.net/doc/index.html.

Estler, H.C., Nordio, M., Furia, C.A., Meyer, B., 2014. Awareness and merge conflicts in distributed software development. In: 2014 IEEE 9th International Conference on Global Software Engineering. IEEE, pp. 26–35.

Ghiotto, G., Murta, L., Barros, M., van der Hoek, A., 2020. On the nature of merge conflicts: A study of 2,731 open source java projects hosted by GitHub. IEEE Trans. Softw. Eng. 46 (8), 892–915. http://dx.doi.org/10.1109/TSE.2018.2871083.

2023. Git - git-merge. https://git-scm.com/docs/git-merge.

2023. Git merge conflicts. https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts.

2017. GitHub - joliebig/featurehouse: language independent software composition and merging. https://github.com/joliebig/featurehouse.

2019. GitHub - se-sic/jdime: syntactic merge tool for java. https://github.com/se-sic/jdime.

2021. GitHub - symbolk/IntelliMerge: A graph-based refactoring-aware three-way merging tool for java programs. https://github.com/Symbolk/IntelliMerge.

2021. jFSTMerge. https://github.com/guilhermejccavalcanti/jFSTMerge.

Leßenich, O., Siegmund, J., Apel, S., Kästner, C., Hunsen, C., 2018. Indicators for merge conflicts in the wild: survey and empirical study. Autom. Softw. Eng. 25 (2), 279–313.

Mahmoudi, M., Nadi, S., Tsantalis, N., 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, pp. 151–162. http://dx.doi.org/10.1109/SANER.2019.8668012.

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

2014. Merge branch 'OperatorAsObservable' of github.com:akarnokd/RxJava into merge-asObservable. https://github.com/ReactiveX/RxJava/commit/45c9dc85.

Nguyen, H.L., Ignat, C.-L., 2018. An analysis of merge conflicts and resolutions in git-based open source projects. Comput. Support. Coop. Work (CSCW) 27 (3), 741–765. http://dx.doi.org/10.1007/s10606-018-9323-3.

Owhadi-Kareshk, M., Nadi, S., Rubin, J., 2019. Predicting merge conflicts in collaborative software development. https://arxiv.org/pdf/1907.06274.pdf.

Pan, R., Le, V., Nagappan, N., Gulwani, S., Lahiri, S., Kaufman, M., 2021. Can program synthesis be used to learn merge conflict resolutions? An empirical analysis. In: Proceedings of the 43rd International Conference on Software Engineering. ICSE '21, IEEE Press, pp. 785–796. http://dx.doi.org/10.1109/ICSE43902.2021.00077.

Seibt, G., Heck, F., Cavalcanti, G., Borba, P., Apel, S., 2022. Leveraging structure in software merge: An empirical study. IEEE Trans. Softw. Eng. 48 (11), 4590–4610. http://dx.doi.org/10.1109/TSE.2021.3123143.

Shen, B., Gulzar, M.A., He, F., Meng, N., 2023. A characterization study of merge conflicts in java projects. ACM Trans. Softw. Eng. Methodol. 32 (2), http://dx.doi.org/10.1145/3546944.

Shen, B., Zhang, W., Zhao, H., Liang, G., Jin, Z., Wang, Q., 2019. IntelliMerge: A refactoring-aware software merging technique. Proc. ACM Program. Lang. 3 (OOPSLA), http://dx.doi.org/10.1145/3360596.

2018. Thufv/automerge: Resolve conflicts via version space algebra in structured merge. https://github.com/thufv/automerge.

Yuzuki, R., Hata, H., Matsumoto, K., 2015. How we resolve conflict: an empirical study of method-level conflict resolution. In: 2015 IEEE 1st International Workshop on Software Analytics. SWAN, pp. 21–24. http://dx.doi.org/10.1109/SWAN.2015.7070484.

Zhu, F., He, F., 2018. Conflict resolution for structured merge via version space algebra. Proc. ACM Program. Lang. 2 (OOPSLA), 166:1–166:25. http://dx.doi.org/10.1145/3276536, URL http://doi.acm.org/10.1145/3276536.

Zhu, F., Xie, X., Feng, D., Meng, N., He, F., 2022. Mastery: Shifted-code-aware structured merging. In: Dong, W., Talpin, J.-P. (Eds.), Dependable Software Engineering. Theories, Tools, and Applications. Springer Nature Switzerland, Cham, pp. 70–87.

Bowen Shen is a Ph.D. student in Software Engineering at the Department of Computer Science in the Virginia Tech University, USA. He obtained B.Sc. in Intelligence Science and Technology from Nankai University, China, and obtained M.Sc. in Computer Science in The State University of New York at Buffalo. His research interests include software engineering, with a special focus on the software merge conflicts in open-source Java projects.

Na Meng has been an Associate Professor in the Department of Computer Science at Virginia Tech since 2021. She received her B.E. in Software Engineering from Northeastern University (NEU) in China in 2006, and received her M.S. in Computer Science from Peking University in China in 2009. She obtained her Ph.D. in Computer Science from The University of Texas at Austin in 2014. She started working in Virginia Tech as an Assistant Professor in 2015. Her research interests include Software Engineering, Programming Languages, Software Security, and Artificial Intelligence. Dr. Meng's research has been supported by NSF and ONR.