Leveraging Structure in Software Merge: An Empirical Study

Georg Seibt, Florian Heck, Guilherme Cavalcanti, Paulo Borba, and Sven Apel

Abstract—Large-scale software development today relies heavily on version control systems facilitating distributed development of software projects. For the purpose of merging diverging versions of the code base, version control systems employ line-based merge algorithms, which are applicable to all text files. Structured merge algorithms have been proposed as an alternative to unstructured, line-based merging, with the goal of reducing the number of merge conflicts that have to be manually resolved by the developer. By leveraging the structure inherent in source code (i.e., by representing source code files in terms of abstract syntax trees instead of sequences of text lines), these algorithms are able to merge revisions in various situations (e.g., reordering of methods) that would cause conflicts when merged using an unstructured approach. However, merging abstract syntax trees is inherently more complex than merging sequences of text lines, which makes structured merge algorithms computationally more expensive than an unstructured merge. To reduce the runtime cost of structured merge algorithms, semistructured merge as well as combinations of different merge strategies were proposed. As such, we observe a range of increasingly structured merge algorithms, which feature different characteristics in terms of conflict resolution and runtime. The progressively increasing use of structure to avoid merge conflicts or to automate conflict resolution raises a number of questions: How is the correctness of the code resulting from a merge affected when employing structured merge algorithms? Which algorithm strikes the best balance between runtime, conflict resolution potential, and correctness of the merge result? For the first time, we evaluate a whole range of merge algorithms (from unstructured over semistructured to structured as well as combinations) by replaying merge commits in a controlled setting. We employ the test suite of the projects in question as an oracle for the correctness of the resulting code, triangulated by a thorough manual analysis. Using 7727 merge commits from 10 open-source projects, we find that combined strategies appear to be the best of both worlds: They resolve as many conflicts as structured merge at a significantly lower runtime per merge commit. Notably, structured merge strategies do cause more test failures, however, the increase is small.

Index Terms—JDIME, version control systems, software merge, structured merge, semistructured merge

1 INTRODUCTION

VERSION control systems are commonplace in modern software development. With tools such as GIT and SVN, developers are able to keep track of changes made to source code artifacts and independently develop separate (or overlapping) aspects of a software project. Branching is used by platforms such as GITHUB to implement sophisticated workflows resulting in coordinated merges of independent development streams, thereby driving the development of the project as a whole.

To automate the merge of software revisions (e.g., the tips of two branches), version control systems offer a number of merge algorithms. The state of the art in practice is the use of unstructured algorithms, which consider source code artifacts to be merely sequences of text lines. Unstructured merge is popular due to its simplicity and its applicability to all kinds of text files, irrespective of the kind of content. Another strength is the superior scalability of these algorithms. For the most part, the performance of unstructured merge, in terms of the number and kind of conflicts produced when merging revisions, is satisfactory in practice [1]. On the downside, when conflicts arise, sometimes they are trivial to developers who are familiar with the semantics of the language. For example, unstructured merge is not able to recognize reordering of methods, a refactoring that makes no semantic difference in languages like JAVA. This shows the downside of the generality of unstructured merge: The rigid structure of source code, which could be used to resolve merge conflicts, is ignored.

Researchers have developed various structured merge algorithms, which leverage the structure of the source code being merged with the goal of reducing the number of conflicts. The improved conflict resolution capability however comes at a cost. On the one hand, to implement a structured merge algorithm, one has to be able to parse the programming language in which the code is written, fixing the kind of source code artifacts the algorithm can be applied to. On the other hand, structured merge, particularly tree matching, is computationally far more complex than merging sequences of text lines. This leads to a significantly higher runtime in practice. While unstructured merge typically takes, at most, a few seconds to run, a structured merge of a complex AST easily takes several minutes to complete. To mitigate the runtime cost of a structured merge while retaining the conflict resolution potential, previous work introduced *combined*, adaptive strategies, which apply less sophisticated merge algorithms first before moving on to a structured merge in the case of conflicts. For example, Leßenich et al. introduced *structured merge with autotuning*, which applies unstructured merge and, in the case of con-

• *G. Seibt and F. Heck are with the University of Passau, Passau, Germany.*
• *P. Borba is with the Federal University of Pernambuco, Recife, Brazil.*
• *G. Cavalcanti is with the Federal Institute of Alagoas, Penedo, Brazil.*
• *S. Apel is with Saarland University, Saarland Informatics Campus, Saarbrücken, Germany.*

Manuscript received TBD; revised TBD.

flict, escalates to a fully structured merge [2].

Previous work has provided data about runtime and conflict characteristics of merge strategies in isolation, however, the full spectrum of possibilities was never studied in a controlled, integrated environment. Without comprehensive data derived from real-world merge scenarios, the adoption of structured merge algorithms outside of academia remains limited. Our assumption is that, while the number of conflicts produced by structured merge is lower, there is the possibility of incorrectly resolving conflicts that should not have been merged. Therefore, we examine the correctness of the merged code by employing the test suite of the projects as an oracle to answer a previously open question:

> *Does the intelligent, structure-aware resolution of merge conflicts performed by structured merge algorithms come at a cost to the correctness of the produced code? That is to say, does structured merge resolve too many merge conflicts?*

We pose corresponding research questions concerning the influence of "structuredness" on the correctness of conflict free code as well as the number and characteristics (i.e., size) of conflicts and the runtime of the algorithms.

In this paper, we work with JDIME,[1] a structured merge tool for JAVA, which contains implementations of unstructured merge, structured merge based on merging abstract syntax trees, and semistructured merge, in which code above the level of method declarations is merged structurally and method bodies are merged in an unstructured way. JDIME also provides implementations of combined strategies. For our study, we implemented a measurement framework that identifies merge commits and replays them using JDIME as a merge driver within the GIT version control system. We can thereby guarantee that, for every merge commit, the merge algorithm (via JDIME) is presented with the same set of files for a fair comparison.

We perform an empirical study using 7727 merge commits from 10 open-source JAVA projects gathered from GITHUB to both replicate results concerning runtime and conflicts for unstructured, semistructured and structured merge algorithms and extend the body of knowledge by adding results for combined strategies. We found that simple strategies follow a predictable pattern in terms of runtime: As in previous studies, runtime increases with the complexity of the merge algorithm, most significantly for fully structured merge [2]. The same applies to conflicts: Structured merge can resolve the most conflicts, semistructured merge falls between unstructured and structured algorithms. The results for combined strategies show that their runtime is mostly determined by their least complex strategy (recall that most merge scenarios can be resolved by unstructured merge). Their conflict resolution potential however is quite surprising: In some cases, combined strategies outperform fully structured merge, which is due to resolving cases that can not be handled by unstructured merge as well as those that structured merge produces conflicts for (but unstructured does not). The runtime overhead of combined strategies per merge scenario is far less severe than is the case for structured merge, whereas combined strategies are able to resolve, at least, as many conflicts as structured merge.

As a notable result, the influence of the use of structured merge algorithms on the correctness of the code (as determined by a test suite and a thorough manual analysis) is minimal. Increasing the complexity of the merge algorithm will increase the number of build and test failures, but the increase is small enough to be acceptable in practice.

In summary, we make the following contributions:

- We integrate unstructured, semistructured and structured merge algorithms as well as combined strategies into one controlled framework.
- We perform an empirical study on 7727 merge commits from 10 open-source projects gathering data about runtime, conflict characteristics, and correctness of the merge result for all strategies.
- Our study confirms previous results about conflict resolution and runtime performance of various structured merge approaches.
- Our results show that, while the lower number of conflicts produced by more complex merge strategies comes at a runtime cost, that can be mitigated using combined strategies.
- Most notably, we found that employing structured merge algorithms to automatically resolve conflicts does not result in a large increase in test failures indicating incorrect code. That is to say, combined strategies are an attractive compromise between runtime and conflict resolution potential, while their increase in test failures remains small enough to make them viable in practice. They retain the low runtime of unstructured merge for most scenarios while resolving, at least, as many conflicts as structured merge in cases where unstructured merge fails.

This article synthesizes and expands on a number of previous papers: Semistructured merge has been proposed by Apel et al. [3]. For this article, we re-implemented the semistructured merge algorithm for JAVA on top of JDIME. Structured merge and, in particular, its combination with unstructured merge has been introduced and implemented by Apel et al. for JDIME [4]. We expand on this work by implementing more combined strategies. Apel et al. [3], Leßenich et al. [2], and Cavalcanti et al. [5] compared different subsets of merge strategies. We improve upon these previous studies by comparing all basic strategies as well as all combined strategies in a controlled setting. In addition to merge conflicts and merge time, we analyze for all strategies the correctness of the merged code, which provides insights into the tension between reducing the number of conflicts and still producing correct code. To determine correctness, we rely on the test suite of our subject systems and additionally perform a thorough manual analysis.

We provide raw data and instructions necessary for replication via our supplementary Website.[2]

2 BACKGROUND

In this section, we provide an overview of the merge strategies used in our study. To illustrate unstructured, semistruc-

1. https://github.com/se-sic/jdime/

2. https://se-sic.github.io/sism-supp/

tured, and structured merge, we introduce a running example used throughout the section. We provide an overview of each approach, a description of the relevant algorithms, and then we use the running example to describe how the algorithm behaves in practice.

2.1 Running Example

We use Figure 1 as a running example throughout the paper to illustrate and contrast the different merging strategies. The running example consists of the Java class Counter, which, in the BASE revision, contains a field for storing the value of the counter and a method get for retrieving it.

Now suppose two programmers independently extend this class. To support this kind of development, version control systems provide the concept of branches in which independent sets of changes can be applied to a base version of the source code. In Figure 1, two branches are split off from the BASE revision resulting in the revisions LEFT and RIGHT.

Both branches add the visibility modifier private to field num, but they use different styles. The LEFT revision adds the modifier on a separate line, whereas RIGHT follows the usual convention and adds the modifier on the same line as the field. Furthermore, a method called inc is added in both revisions, which is supposed to increase the value of the counter by one while printing a corresponding log message. The complication here is that the implementations and placement of the method differ between LEFT and RIGHT. Method inc is implemented properly in the LEFT revision, where the developer chose to place it above the get method. The RIGHT revision, however, contains a faulty implementation of inc, in which the counter is actually increased by two instead of one and the formatting of the log message is non-standard (there are spaces before the argument of println). In revision RIGHT, the new method was placed below get.

At some point, the two branches are merged with the aim of generating a consolidated version of class Counter. This merge between the three revisions LEFT and RIGHT based on their common ancestor BASE is known as a *three-way merge* [1]. In practice, most merges are performed using unstructured algorithms (see Section 2.2) as implemented in version control systems such as GIT. Algorithms that exploit source code structure to automatically resolve merge conflicts are introduced in Sections 2.3 and 2.4.

2.2 Unstructured Merge

2.2.1 Overview

The approach that is most widely used in software merge tools is unstructured merge. It appears in widely used UNIX tools such as DIFF and MERGE and is used in the most popular version control systems, chiefly GIT and SVN. While the implementation is complex, the idea is straight-forward: The algorithm considers software artifacts simply as a sequence of text lines. An algorithm inspired by the Longest Common Subsequence problem [6] is used to identify changed blocks in the lines that make up the versions to be merged. The algorithm walks through the changed blocks and applies a set of merge rules to either accept changes into the merge output or flag conflicts where the correct change to accept can not be decided.

As such, the algorithm is applicable to any file that can be interpreted as a sequence of text lines. This property and the very high runtime efficiency are the major selling points for unstructured merge. There is, however, a downside. The granularity of the algorithm is at the level of text lines; the fact that it is source code that is being merged is not taken into account. This leads to weaknesses in conflict resolution since the structure of the source code is not considered. In their survey on software merging, Mens et al. conjecture that up to 90 % of merge scenarios may be resolved using unstructured merge, while the remaining 10 % require more complex solutions such as structured merge [1].

2.2.2 Example

Figure 1 (bottom left) shows the result of an unstructured three-way-merge between the revisions LEFT, BASE, and RIGHT. The unstructured merge algorithm notices a conflicting change to Line 2 of BASE between the LEFT and RIGHT revision. As the algorithm is unable to recognize that both versions are semantically equivalent, it reports a conflict starting on Line 2 and closing on Line 7 of the unstructured merge result. The second issue with the merge result is not marked by a conflict, arguably making it more severe: Unstructured merge includes both versions of method inc, so the code would not compile.

2.2.3 Algorithm

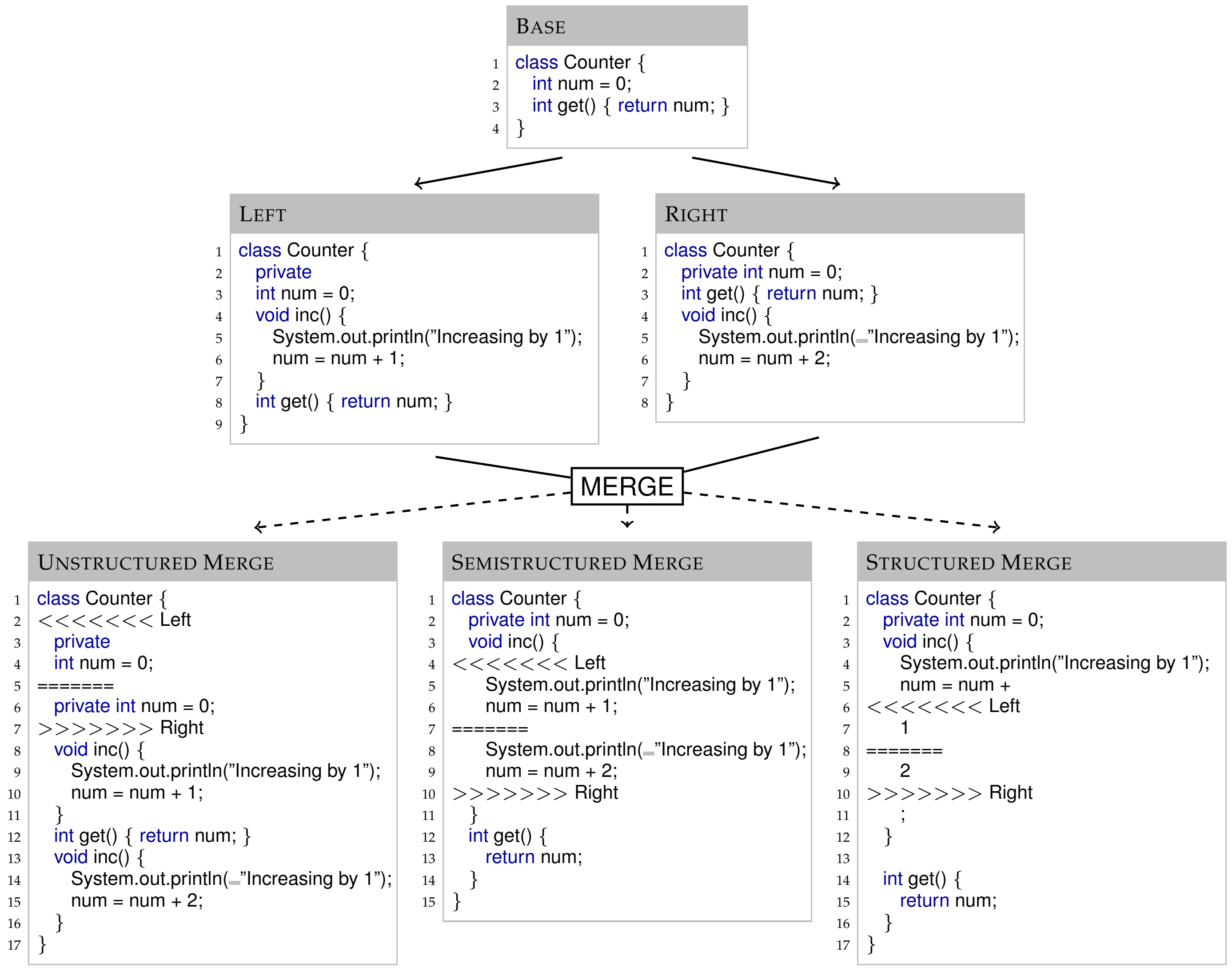
The output of unstructured merge shown in Figure 1 was produced by the merge algorithm included in the GIT version control system.

The actual implementation of the merge algorithm in GIT is out of scope of this paper as it involves a range of specialized optimizations. However, the algorithm is based on DIFF3, which is simpler and captures still the essence of unstructured merge [7]. Unstructured merge in DIFF3 consists of two steps: First, three versions (sequences of text lines in the case of source code files), one of which is considered the base version, are passed to the algorithm. The DIFF algorithm is called twice to identify the *longest common subsequences* between the changed versions and the base version. These matchings between versions and their base are then overlaid to form a sequence of *chunks*, either *stable* (all versions agree) or *unstable* (at least one of the versions differs from the base). Finally, the changes made in each chunk are examined and, if possible, merged. Chunks in which only one version applies changes to the base are merged. However, if two versions make inconsistent changes to the base, the algorithm reports a conflict.

2.3 Structured Merge

2.3.1 Overview

Structured merge leverages the rigid structure of source code with the goal of resolving merge conflicts. Westfechtel and Buffenbarger were pioneers of this field; they proposed the use of structural information about the code, derived from its syntax rules, in the merge algorithm [8], [9]. Specifically, source code is represented as an abstract syntax tree (AST) instead of a sequence of lines. Merging ASTs consists



```
class Counter {
  int num = 0;
  int get() { return num; }
}
```

```
class Counter {
  private
  int num = 0;
  void inc() {
    System.out.println(”Increasing by 1”);
    num = num + 1;
  }
  int get() { return num; }
}
```

```
class Counter {
  private int num = 0;
  int get() { return num; }
  void inc() {
    System.out.println(▬”Increasing by 1”);
    num = num + 2;
  }
}
```

```
class Counter {
<<<<<<< Left
  private
  int num = 0;
=======
  private int num = 0;
>>>>>>> Right
  void inc() {
    System.out.println(”Increasing by 1”);
    num = num + 1;
  }
  int get() { return num; }
  void inc() {
    System.out.println(▬”Increasing by 1”);
    num = num + 2;
  }
}
```

```
class Counter {
  private int num = 0;
  void inc() {
<<<<<<< Left
    System.out.println(”Increasing by 1”);
    num = num + 1;
=======
    System.out.println(▬”Increasing by 1”);
    num = num + 2;
>>>>>>> Right
  }
  int get() {
    return num;
  }
}
```

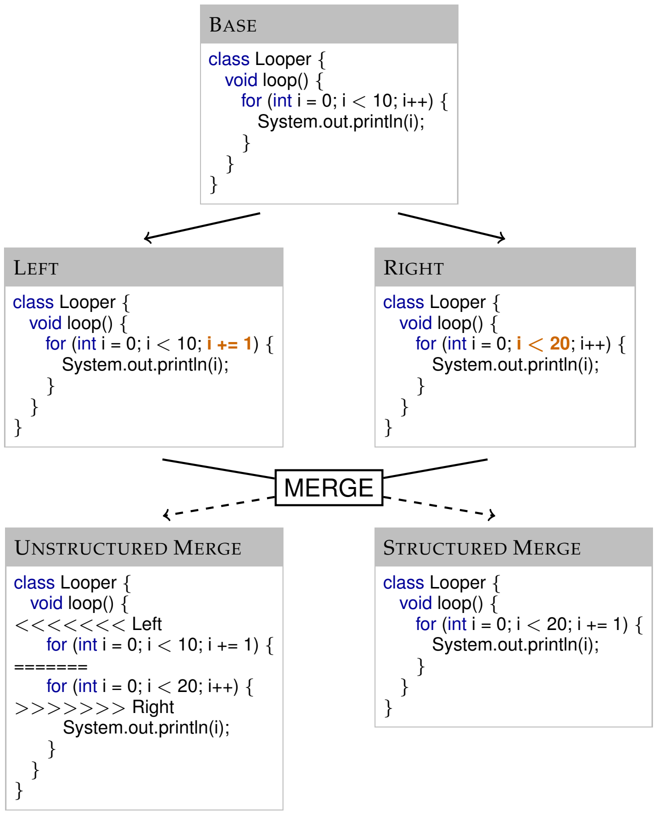
```
class Counter {
  private int num = 0;
  void inc() {
    System.out.println(”Increasing by 1”);
    num = num +
<<<<<<< Left
    1
=======
    2
>>>>>>> Right
    ;
  }

  int get() {
    return num;
  }
}
```

Fig. 1: The merge scenario used as a running example throughout the paper. The code of revision BASE is inconsistently modified in revisions LEFT and RIGHT. Performing an unstructured, semistructured and structured three-way merge of these revisions results in code containing different kinds of conflicts.

of two steps: First, the nodes of the trees are matched to establish equality relationships between pairs of nodes from two different revisions. In three-way merge, the matching algorithm would match the revisions LEFT and RIGHT with the BASE revision as well as the LEFT and RIGHT revisions with each other. Second, the revisions LEFT and RIGHT are merged to form the AST representing the merge result. To construct the merged AST, the algorithm walks the two ASTs to be merged in lockstep and applies three-way merge rules [1] to decide whether to include nodes, delete them, or flag conflicts.

As this process exploits a variety of properties of the language being merged (e.g., that it is safe to permute methods, Figure 1), a range of situations that are undecidable for an unstructured algorithm become trivial to resolve [3]. Another example in which knowledge about language structure is useful and could be exploited by a structured merge tool is the merging of loops: A `for` loop in Java consists of a head and the associated body, with the head being made up of three distinct parts. These parts are usually located on the same line but are represented by distinct subtrees in an AST. Figure 2 shows a merge scenario in which the LEFT revision increments the iteration variable of the loop using an alternative style, which affects the third part of the loop head. The RIGHT revision modifies the bounds of the loop which entails changing the second part of the loop head. Unstructured merge produces one conflict consisting of the line which contains the loop head. Structured merge is able to avoid the conflict due to the structure of the AST representing the loop. The changes occurred in separate subtrees of the AST. Note that, while it is safe in our example, accepting both changes to the head of the loop may cause the parts of the body that depend on the iteration variable to work incorrectly, thereby causing a test failure (in the best case) instead of a merge conflict. A further benefit of structured merge is that formatting is not present in an AST and as such has no influence on the merge



BASE
```
class Looper {
  void loop() {
    for (int i = 0; i < 10; i++) {
      System.out.println(i);
    }
  }
}
```

LEFT
```
class Looper {
  void loop() {
    for (int i = 0; i < 10; i += 1) {
      System.out.println(i);
    }
  }
}
```

RIGHT
```
class Looper {
  void loop() {
    for (int i = 0; i < 20; i++) {
      System.out.println(i);
    }
  }
}
```

UNSTRUCTURED MERGE
```
class Looper {
  void loop() {
<<<<<<< Left
    for (int i = 0; i < 10; i += 1) {
=======
    for (int i = 0; i < 20; i++) {
>>>>>>> Right
      System.out.println(i);
    }
  }
}
```

STRUCTURED MERGE
```
class Looper {
  void loop() {
    for (int i = 0; i < 20; i += 1) {
      System.out.println(i);
    }
  }
}
```

Fig. 2: Unstructured vs. structured merge of a simple for loop. In the LEFT version, the iteration variable is incremented using an alternative style while the RIGHT version increases the bounds of the iteration. Unstructured merge produces a conflict as the changes happen in the same line. Structured merge is able to avoid the conflict since the changes are represented by distinct subtrees of the AST.

process.[3]

The increased ability to resolve conflicts comes at a cost. Structured merge relies on being able to transform source code into a structured representation. In practice, this means that one has to implement (or use) an appropriate parser for every language to be supported. For widely used languages such as Java, these parsers already exist and require only minor modification to be usable in a merge tool. Any parser being considered for structured merge will, at least, have to support programmatic construction of ASTs and be extensible enough to add the concept of a conflict into the AST structure and the pretty-printing algorithm.

A more severe problem of structured merge is the runtime complexity of the underlying matching and merging algorithms, which work on trees making them inherently more complex than their counterparts in unstructured merging, which consider only collections of lines. For example, to maximize precision, the matching step of the algorithm would need to handle both shifted subtrees as well as renamed nodes in the ASTs [10]. This corresponds to solving the Tree Amalgamation Problem and the Maximum Common Embedded Subtree Problem, both of which are known to be $\mathcal{NP}$ hard [11], [12]. Although structured algorithms can be adjusted to reduce their complexity (e.g., by decreasing the granularity of the data structures) even these compromises result in polynomial or even exponential algorithms. This necessitates that matchings are constrained. For example, in JDIME, only nodes at the same level in the respective ASTs can match.

2.3.2 Example

The exemplary result presented in Figure 1 demonstrates several of the properties of structured merge discussed so far. Where unstructured merge produced a conflict due to inconsistent formatting of the "private int num = 0;" statement, a structured merge does not produce this conflict. The subtrees representing the field declaration were fully matched and thus included as merged code in the result. This involves normalizing the code formatting to whatever the AST library in use produces when converting an AST into code, which may not be desirable in all circumstances.[4]

Furthermore, there is only one method inc in the merged code as opposed to the two methods produced by unstructured merge. Likewise, the AST matching algorithm was able to match most of the AST representing method get in the LEFT version of the code with the one in the RIGHT version. Note that the System.out.println() statements in both versions were matched due to formatting being irrelevant as mentioned above. As the order of method declarations is irrelevant in Java, the algorithm chose to include the inc method above the get method. Structured merge thereby produces valid Java (except for the conflict) whereas unstructured merge duplicates the method inc leading to broken code even if all conflicts are resolved.

The code resulting from structured merge is not free of conflicts, though. The implementation of method inc in the RIGHT revision actually increases the counter by 2 instead of 1 (which the LEFT version does). This leads to the only conflict produced by structured merge. Note, however, that the conflict is more fine-grained than conflicts produced by unstructured merging. It only includes the part of the statement that actually differs between the revisions, the number being added to num. Conflicts in the unstructured case will always consist of whole lines as that is the "unit" being worked with.

2.3.3 Algorithm

Structured merge consists of two steps: calculating matchings between the input ASTs and constructing a merged AST based on the matchings. The complexity of the approach lies in the first step. For maximum precision, the aforementioned $\mathcal{NP}$-hard problems (Tree Amalgamation and Maximum Common Embedded Subtree) would have to be solved. To reduce the computational complexity, a restricted matching algorithm is used. Instead of searching for matchings anywhere in the opposing AST, the algorithm proceeds top-down and only considers pairs of nodes that

3. Unstructured merge tools such as GIT can often be configured to ignore whitespace changes, etc. These options are rarely enabled by default, though. In principle, unstructured merge does not distinguish between the kinds of text it merges.

4. A sophisticated structured merge tool might augment the AST to include formatting information, which could be used when pretty-printing but ignored when matching ASTs.

Algorithm 1 AST Matching (level-wise, early return)

```
1:  function Match(Node $L$, Node $R$)
2:      if $L \neq R$ then
3:          return 0                                  ▷ Nodes do not match, early return
4:      end if
5:      $cs_L \leftarrow$ children of $L$
6:      $cs_R \leftarrow$ children of $R$
7:      if IsOrdered($cs_L$) $\vee$ IsOrdered($cs_R$) then
8:          return OrderedMatching($L$, $R$)          ▷ Considering order
9:      else
10:         return UnorderedMatching($L$, $R$)        ▷ Ignoring order
11:     end if
12: end function
```

lie at the same level. This corresponds to the Maximum Common Subtree Problem [12].

Algorithm 1 shows the top-down matching algorithm as implemented in JDIME. First, the nodes L and R are checked for equality, which is defined in terms of their type and, depending on the kind of AST node, other attributes, such as the name of a declared method or the value of an integer literal. Crucially, the children of AST nodes are not considered when equality of nodes is determined. If the nodes are not equal, the algorithm stops and reports that they do not match.

If the two nodes are equal, their children are matched using one of two standard methods: the Longest Common Subsequence Algorithm for ordered children and the Hungarian method for unordered matches [6], [13]. Both methods will recursively call the matching function shown in Algorithm 1.

After calculating the matches, an appropriate merge algorithm for the children of the nodes L and R is selected depending on whether both lists of children are considered ordered or unordered. This determination is based on JAVA'S syntax rules (e.g., children representing statements in a method are ordered, whereas children representing method declarations in a class are not). For the ordered case, Algorithm 2 is applied. The algorithm for the unordered case follows a similar, less complex, pattern, since we can just process the children of L first, and then the children of R. JDIME performs additional steps in the unordered merge algorithm to retain the original ordering as much as possible.

Function OrderedMerge accepts the nodes L and R and the node $target$, whose list of children is to be constructed by merging the children of L and R. The algorithm is initialized by selecting the first child of L and R respectively using function Next, which returns the next element from the given list. The main loop of the algorithm proceeds merging elements from the lists of children until the end of one of them is reached. Merging proceeds as follows: First, OrderedMerge calculates a number of indicators based on the matchings that were previously calculated. The algorithm implemented in JDIME uses 5 such indicators. Table 1 enumerates the relevant indicator configurations. All omitted cases are considered invalid and would result in an error. These indicators are then used to select appropriate merge operations.

The rows in Table 1 denote changes that occurred in the versions being merged. For example, the first row indicates that c_L and c_R match and are also present in the base AST

Algorithm 2 OrderedMerge

```
1:  function OrderedMerge(Node $L$, Node $R$, Node $t$)
2:      $cs_L \leftarrow$ children of $L$
3:      $cs_R \leftarrow$ children of $R$
4:      $c_L$, $done_L \leftarrow$ Next($cs_L$)
5:      $c_R$, $done_R \leftarrow$ Next($cs_R$)
6:      while $\neg done_L \wedge \neg done_R$ do              ▷ Run until one list is consumed
7:          $ind \leftarrow$ Indicators($c_L$, $c_R$)             ▷ Calculate indicators
8:          $ops \leftarrow$ ChooseOperations($ind$)           ▷ Operations from Table 1
9:          $move_L$, $move_R \leftarrow$ ApplyAll($ops$, $c_L$, $c_R$, $t$)
10:         if $move_L$ then                               ▷ Move on to the next left child
11:             $c_L$, $done_L \leftarrow$ Next($cs_L$)
12:         end if
13:         if $move_R$ then                               ▷ Move on to the next right child
14:             $c_R$, $done_R \leftarrow$ Next($cs_R$)
15:         end if
16:     end while
17:     while $\neg done_L$ do                       ▷ Process any remaining left children
18:         $ind \leftarrow$ Indicators($c_L$)
19:         $op \leftarrow$ ChooseOperation($ind$)
20:         Apply($op$, $c_L$, $t$)
21:         $c_L$, $done_L \leftarrow$ Next($cs_L$)
22:     end while
23:     while $\neg done_R$ do                      ▷ Process any remaining right children
24:         $ind \leftarrow$ Indicators($c_R$)
25:         $op \leftarrow$ ChooseOperation($ind$)
26:         Apply($op$, $c_R$, $t$)
27:         $c_R$, $done_R \leftarrow$ Next($cs_R$)
28:     end while
29: end function
```

(i.e., that part of the AST was not changed). The third row refers to the situation that, while c_R was present in the base AST, it is missing from the left AST, which is interpreted as the left revision deleting c_R. However, if there are changes detected anywhere in the subtree under c_R, we report a conflict (between that change and the deletion) instead.

The selection of merge operations is performed in ChooseOperations, which produces one or more instances of the operations Merge(n, m), Addition(n, m), Deletion(n, m), and Conflict(n, m). Operations represent actions to be taken to construct the merged AST. Operation Addition(c_L, t), for example, adds c_L (the whole subtree) as a child to t, thereby "consuming" c_L. Whether c_L or c_R was consumed is returned from function ApplyAll to OrderedMerge. This information is used to decide whether the next child from the children of the left or right node (or both) should be selected for the next iteration.

After one of the children lists is consumed, the remaining nodes on the other side of the merge are merged. This proceeds generally in the same way the main loop is implemented. The algorithm finishes once both child lists are consumed. The target nodes children represent the merge result of the children lists of L and R.

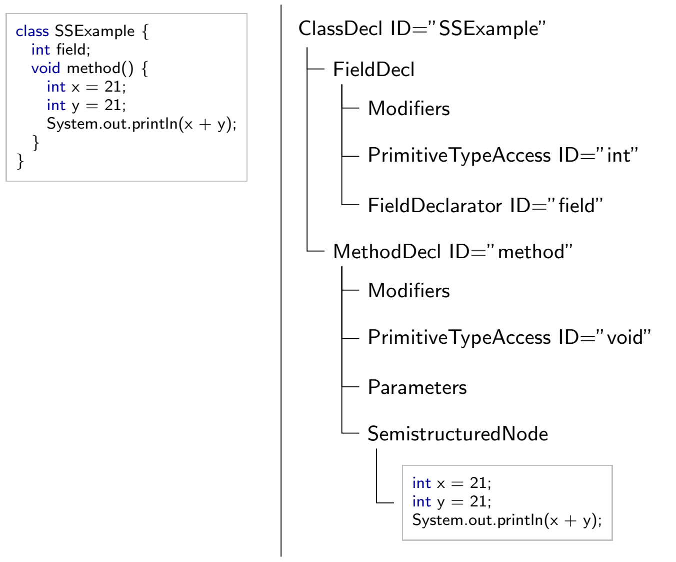
2.4 Semistructured Merge

2.4.1 Overview

Semistructured merge targets the middle ground between unstructured and structured merge algorithms. It is aimed at resolving as many of the conflicts that structured merging would resolve as possible while speeding up the merge procedure by using unstructured merging for parts of the source code. The key idea is that the parser producing the AST is interrupted when a certain level (e.g., method declarations) is reached. The parser is extended with a

TABLE 1: The indicator lookup table for OrderedMerge. Both c_L and c_R represent the children of L and R being merged in Algorithm 2; l, r and b are placeholders for any (other) node from the left, base, and right AST. The column $c_L \leftrightarrow c_R$ indicates whether a bidirectional matching between the nodes c_L and c_R was found. The remaining indicators $c_L \rightarrow r$, $c_L \rightarrow b$, $c_R \rightarrow l$, and $c_R \rightarrow b$ denote a match from c_L or c_R to any node in an opposing tree. A checkmark (✓) indicates that a match is present, empty otherwise. The remaining columns show the actions to be taken if the corresponding configuration of indicators is encountered. All action operate on the children of the newly created node t (see Algorithm 2).

$c_L \leftrightarrow c_R$	$c_L \rightarrow r$	$c_L \rightarrow b$	$c_R \rightarrow l$	$c_R \rightarrow b$	Result	If left subtree changed.	If right subtree changed.	If both subtrees changed.
✓	✓	✓	✓	✓	Merge(c_L, c_R)			
✓	✓		✓		Merge(c_L, c_R)			
	✓	✓		✓	Deletion(c_R, t)		Conflict$(c_R, none)$	
	✓	✓			Addition(c_R, t)			
	✓			✓	Deletion(c_R, t)		Conflict$(c_R, none)$	
	✓				Addition(c_R, t)			
		✓	✓	✓	Deletion(c_L, t)	Conflict$(c_L, none)$		
		✓	✓		Deletion(c_L, t)	Conflict$(c_L, none)$		
		✓		✓	Deletion(c_R, t), Deletion(c_L, t)	Conflict$(c_L, none)$	Conflict$(none, c_R)$	Conflict(c_L, c_R)
		✓			Addition(c_R, t), Deletion(c_L, t)	Conflict$(c_L, none)$		
			✓	✓	Addition(c_L, t)			
			✓		Addition(c_L, t)			
				✓	Addition(c_L, t), Deletion(c_R, t)		Conflict$(none, c_R)$	
					Conflict(c_L, c_R)			



```
class SSExample {
   int field;
   void method() {
      int x = 21;
      int y = 21;
      System.out.println(x + y);
   }
}
```

```
int x = 21;
int y = 21;
System.out.println(x + y);
```

Fig. 3: The semistructured AST for class SSExample (left). Everything above the method body (class and field declarations as well as the method signature) are represented in a structured manner. A node SemistructuredNode holds the method body by representing its code as text.

new kind AST node that represents the remaining source code (e.g., the method bodies) as blocks of text. Figure 3 shows an example of a semistructured AST. Semistructured merge could be configured to stop at any level (e.g., class bodies, method bodies, bodies of loops or conditional expressions). In JDIME, semistructured merge uses structured merge for everything above and including the level of method declarations and unstructured merge for the bodies of methods and constructors, which has been successful in practical settings [3]. This leads to a significant speedup compared to fully structured merge, while retaining the ability to recognize and correctly match reordered methods and fields. In other words, the usual structured algorithms are used to merge the ASTs, but when leaves representing text blocks (e.g., method bodies) are matched or merged, the algorithm delegates to an unstructured strategy. The AST resulting from the merge will then contain text block nodes representing the results of unstructured merging between blocks of text. These sections of code will be included in the code resulting from the transformation of the surrounding AST.

2.4.2 *Example*

The example of semistructured merge shown in Figure 1 was produced by considering method bodies as text and everything above as AST. As such, a semistructured merge algorithm also avoids the conflict introduced by inconsistent formatting of the visibility modifier of field num. The field declaration takes place above the level of method bodies and was therefore merged structurally.

Method declarations themselves (meaning the AST nodes representing the method signature) are also merged structurally. This enables semistructured merge to avoid duplicating method inc as is the case in unstructured merge. The method body, however, is merged using unstructured merge, which produces one large conflict over the whole body of method inc. The conflict between Line 7 of the LEFT revision and Line 8 of the RIGHT is unavoidable. It is also present (though smaller) in the code produced by structured merge (the literals "1" and "2" conflict). The conflict between Lines 6 and 7, on the other hand, is specific to unstructured merge. The lines only differ in that the RIGHT revision adds two spaces in before the argument to System.out.println. Unstructured merge does not recognize that this difference is syntactically irrelevant. The idea of semistructured merge is to retain as much of the conflict resolution potential of structured merge while reducing its runtime cost significantly.

2.5 Combining Strategies

To alleviate a major disadvantage of structured merging—its often impractical runtime—prior work has proposed to

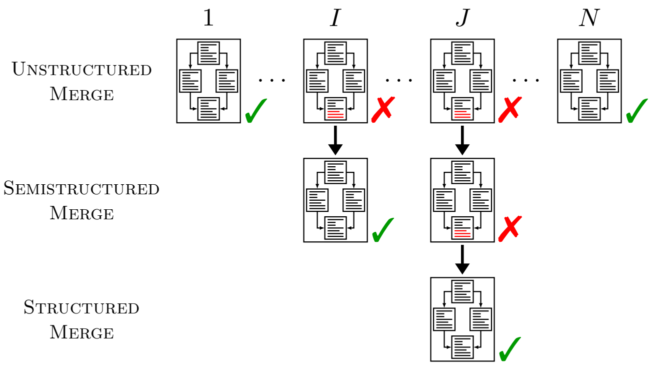


Fig. 4: Merging multiple files (from 1 to N) using a combined strategy. In the first round using unstructured merge, files I and J are merged with conflicts. Semistructured merge is able to merge file I without conflicts but file J still conflicts. Finally, structured merge is able to resolve all remaining conflicts.

combine strategies sequentially [2]. The idea is straightforward: a merge tool employs merge strategies of lower complexity first and only moves on to more complex algorithms if the faster strategies resulted in conflicts. For example: JDIME provides the ability to combine arbitrary merge strategies into a metastrategy. One might set up the strategy *unstructured→semistructured→structured*, which will employ unstructured merging first, in the case of conflicts move on to semistructured merging, and only then escalate to the expensive structured strategy. Previous work by Leßenich et al. has employed the combined strategy *unstructured→structured* and found that it meets expectations by striking a good balance between runtime and conflict resolution [2].

JDIME provides three basic merge strategies, namely unstructured (*US*), semistructured (*SS*), and structured (*S*), which we study in individually. In addition, we analyze all combinations of these strategies in which the expected runtime increases and no strategy appears twice. This leaves us with four strategies, *US→SS*, *SS→S*, *US→S*, and *US→SS→S*.

Figure 4 shows a merge scenario in which N files are merged. At first, the merge for all files is performed using the unstructured strategy. This provides the best runtime and will succeed without producing conflicts in most cases [2]. Let's say that the merge succeeds for all but the files having the index I and J. In the second "round" of the merge, the files I and J are passed to the semistructured strategy, which merges parts of the source code structurally, while still using an unstructured approach for method bodies. This succeeds for file I, while still producing conflicts for J. In the last round of the merge, the file J is successfully merged using the fully structured approach. If any file were to still contain conflicts after the structured strategy was tried, these conflicts would remain for the user to resolve.

3 EMPIRICAL STUDY

To compare unstructured, semistructured and structured merging as well as combinations thereof, we conducted an empirical study on 7727 merge commits from 10 projects. All data, case studies, scripts, and further plots are available on our supplementary Website.[5] Next, we describe the research questions, setup, subjects, and results of our study.

3.1 Research Questions

While using more complex merge algorithms will naturally result in higher runtime, the extent and proportion of the increase is unclear. We measure the runtime of all simple strategies (unstructured, semistructured and structured) by re-merging real-world merge scenarios from the version history of our subject systems (see Section 3.4). Additionally, the use of combined strategies promises to decrease the runtime, while resolving as many conflicts as possible by using simpler strategies first before escalating to more complex strategies in the case of conflicts.

RQ$_1$ What is the runtime cost of using progressively more complex merge strategies?

Apart from the runtime of a given strategy, its effectiveness in terms of conflict resolution is an important measure. We expect the number of conflicts to decrease as the complexity of the merge strategy increases.

RQ$_2$ How does using progressively more complex merge strategies affect the number of conflicts resulting from the merge?

To learn about the characteristics of the conflicts produced by the various merge strategies, we additionally measure the size of the conflicts in terms of lines of code.

RQ$_3$ How does using progressively more complex merge strategies affect the size of the conflicts resulting from the merge?

Previous work has studied merge strategies in terms of the number and size of the conflicts they produce. This study examines the subject in a more granular way. When a merge strategy produces code that does not contain conflicts, we check it for both build- and test failures. The former is recorded when conflict-free code does not compile. In addition to these failures occurring in the build phase, this study considers failures in the testing phase, that is, code which does compile but results in incorrect behavior of the program. We use the test suite of the subject project under consideration as an oracle to detect incorrect behavior, and we evaluate whether reducing the number of conflicts results in more failures in either the build phase or incorrect behavior indicated by failing tests in the testing phase.

RQ$_4$ How does using progressively more complex merge strategies affect the number of build and test failures?

To triangulate the findings about failures in the build and testing phase and increase confidence in the validity of this approach, we perform a manual analysis on conflict resolutions that resulted in the test suite passing without failures.

3.2 Experiment Design

Based on our research questions, we consider the dependent and independent variables shown in Table 2 in our

5. https://se-sic.github.io/sism-supp/

experiment design. To measure our dependent variables, we use established tools such as JDIME. Furthermore, we have implemented a measurement framework, MERGEPROFILER, which we will discuss in Section 3.3.

3.2.1 Independent Variables

We keep all configuration options of the involved tools constant except for the merge scenario and the merge strategy used to perform the merge. These two independent variables are managed by our measurement framework, which orchestrates our measurements for all merge commits and strategies under test.

3.2.2 Dependent Variables

To answer **RQ**$_1$ we measure the dependent variable *Runtime*. We use JDIME for executing the merges and enable its statistics mode to collect runtime and other metrics. Runtime measurements are performed 5 times on the same hardware. We calculate both the mean and median runtime of the 5 runs and, for every strategy, the mean of means and median of medians over all merge commits. Since the standard deviations of these measures are large, the median is used predominantly to remove extreme values.

For research questions **RQ**$_2$ and **RQ**$_3$, we instruct JDIME to measure the number of conflicts as well as their size in terms of lines of code. When a merge scenario consists of more than one file, we sum up these metrics over all files with conflicts. To hone in on the commits that are actually of interest for answering **RQ**$_2$ and **RQ**$_3$, we only consider commits, for which at least one of our merge strategies produced conflicts, for answering these research questions.

RQ$_4$ is answered by attempting to build the code resulting from a merge and, if successful, running the test suite of the project.

3.3 Measurement Setup

In our study, we use two key tools. First, we use JDIME to perform the merges between files. JDIME contains implementations of all merge strategies under test and is additionally able to combine merge strategies by applying them in succession until one does not produce conflicts. JDIME is able to collect a range of statistics, which we use to measure our dependent variables. We use MERGEPROFILER to vary the independent variables, prepare merge scenarios, invoke JDIME, and subsequently collect the resulting statistics. Furthermore, MERGEPROFILER is responsible for executing the test suite of the project being evaluated and recording the states of all tests in it. MERGEPROFILER uses the build tool of the project for executing builds and running the test suite. Whenever possible, we call the build tool programmatically using the appropriate libraries and use the provided APIs for determining whether the build was successful and, if it was, the state of the test suite. To perform merges, we injected JDIME into GIT as a merge driver.[6]

Taking one "measurement" consequently means choosing a merge commit from the history of a project and a merge strategy from the ones available in JDIME, using the strategy to merge the ancestors of the merge commit, and examining the resulting code. We call this "re-merging" a merge commit.

Our study was executed using a cluster of machines. Each job had exclusive access to one node of our cluster and consisted of analyzing one merge commit. The machines used for runtime measurements were equipped with two Intel Xeon E5-2650v2 CPUs, 128 GiB RAM, and SATA SSD storage. For our analysis, we restricted the JVM Heap to 30 GB of RAM.

To check whether the code produced by re-merging a merge commit contains any merge conflicts or results in any failures when building or testing the project, we collect a number of metrics for every merge. We group these metrics into three categories:

TABLE 2: Experiment variables of the study.

Variable	Description
Independent variables	
Merge commit	A commit with two ancestors from the history of one of the subject systems.
Merge strategy	The merge strategy used to remerge the merge scenario.
Dependent variables	
Runtime	The runtime of the merge strategy.
Dependent variables: Conflicts	
Number of conflicts	The number of conflicts reported by the merge tool (JDIME).
Number of files containing conflicts	The number of files containing conflicts after remerging.
Size of the conflicts	The accumulated size of the conflicts in a merge scenario in terms of lines of code.
Dependent variable: Build	
Build status	Whether the code can be successfully built by its build tool.
Dependent variables: Test suite	
State of tests	The state of every individual test in the test suite, i.e. its name and whether it passed, failed or ended in some other state.
State of test suite	The aggregated state of the test suite (e.g., "*Test Suite Passed*") when all tests passed.

3.3.1 Build and Test Metrics

Ideally, given the code base resulting from re-merging a merge commit from a subject system, we would, if there are no conflicts and the build is successful, run the test suite via the build tool that was used in the project and collect the names of the tests that were run and their respective outcomes. Taking into account imperfect merge algorithms, build setups, and the like, scenarios that do not result in this information are possible.

Figure 5 shows an example Sankey[7] plot summarizing the outcomes of all merge commit re-merges using unstructured merge. The example plot shows that, for 17 of 82 merge commits, something other than a successful run of the test suite happened. We group these remaining merge commits into

6. https://git-scm.com/docs/gitattributes#_defining_a_custom_merge_driver

7. https://en.wikipedia.org/wiki/Sankey_diagram

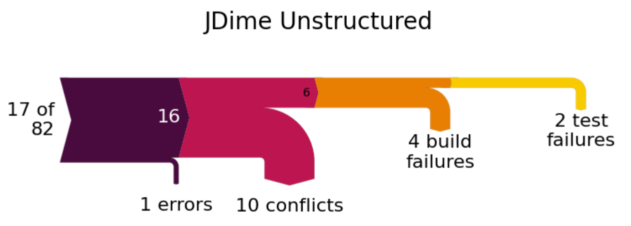


Fig. 5: An example Sankey plot for the project COMMONS-MATH.

a number of categories which are visualized as sequential, interlocking arrows in the Sankey plot.

A frequent outcome of our study was that all tests passed after a merge commit was re-merged using one of the strategies. We do not show these merge commits in the Sankey plot as it would make the other possible outcomes hard to discern visually. These omitted commits are tagged as "*Test Suite Passed*" commits.

All attempts at running the test suite which failed due to an error are tagged as "*Error*". An error may occur at any stage of the evaluation process though we minimized the number of commits for which this was an issue. Sources of errors include internal errors in our tooling (very rare), broken build scripts, missing dependencies, timeouts while running the test suite (most frequent, we applied a one hour time limit), and out-of-memory errors during the testing phase (we restricted our analysis jobs to 30 GB of RAM). In the example in Figure 5, 1 commit falls in this category.

Next, the plot shows the number of commits that are tagged as "*Conflicts*" after the merge strategy produced conflicts re-merging it. We do not proceed further for these commits and assume that the code can not be built. Instead, we collect the metrics described in Section 3.3.2. In Figure 5, 10 fall in this category.

If no conflicts were detected, we attempt to build the project by performing appropriate actions using the build system of the project and examining the resulting output. When we determine that the build failed, the merge commit is tagged as "*Build Failure*". No further metrics are collected. This was the case for 4 of 17 commits in Figure 5.

Finally, the test suite is executed using the build system of the project, and the names and states of all the tests that ran are recorded. If any of the tests fail, we tag the merge commit as "*Test Failure*". In Figure 5, 2 of 17 commits had failing tests.

There are some corner cases that may occur such as there being no tests or all tests being skipped, but they only occur in a minimal number of cases, which do not influence our results. Note that we filter out some scenarios in which we expect failures, e.g., we ignore a build failure after re-merging if both parent commits also resulted in a build failure. The same goes for a test failure if the test failed in both parent commits. We assume that the failure is to be expected in these cases and not due to the merge strategy that was applied.

3.3.2 Conflict Metrics

When remerging a merge commit using JDIME, we collect a number of metrics about any conflicts produced during the merge. MERGEPROFILER configures GIT in such a way that it only merges using JDIME when Java source code files are merged. JDIME, being a structured merge tool, is language specific. As such, only conflicts in these files are considered.

We record how many files contain conflicts and the number of conflicts (i.e., the number of conflict markers) per merge commit and additionally, for every conflict, we record its size in terms of lines of code. The number of lines in conflict consist of the lines on the left side of the conflict in addition to the lines on the right side. Lines consisting of only white spaces and those which are commented out are not counted.

Note that we perform filtering of conflicts consisting only of whitespace or comments because these elements are not present in an AST. Including these conflicts would put unstructured merging at a disadvantage, as these conflicts do not arise in structured merge.

3.3.3 Runtime Metrics

Whenever a merge is performed, JDIME collects runtime statistics appropriate for the strategy in use. The JDIME code is instrumented such that only the merge procedure is measured as opposed to the whole JDIME invocation.

In the case of unstructured merge, this means measuring the runtime of the JNA[8] call to the native LIBGIT2 library that performs the merge. For the structured strategy, the execution time of the AST parsing, matching and merging implemented in JDIME is measured.

As the semistructured strategy performs both calls to the unstructured merger (for method bodies) and uses the AST merge algorithm present in JDIME, its runtime is measured by combining measurements of both code paths. All combined strategies will return the sum of the measurements performed by their constituting basic strategies.

3.3.4 Manual Analysis

In addition to relying only on automatically obtained quantitative data on the number of failures in any of the phases, we performed a manual analysis on relevant merge commits. Specifically, we selected those merge commits for which the re-merge using one of our merge strategies, T_0, produced conflicts, whereas other strategies, $T_{1\dots n}$, did not produce conflicts and, instead, resulted in a passing test suite. These commits have the potential to exhibit the worst-case scenario: An incorrect conflict resolution that was, potentially, due to the affected code not being tested, missed by the test suite. While all other conflict resolutions also have the potential to be incorrect, they resulted in a test suite failure, meaning that the test suite is a proper oracle for correctness.

For every file F, for which T_0 produced conflicts, we archived the resolved versions of that file produced by the strategies $T_{1\dots n}$. We then manually analyzed all files F containing conflicts and their resolution for that dataset.

First, we classified the conflicts in F following the methodology of Cavalcanti et al. [14]: We assign the conflicting files to one of two categories: *True positives* — conflicts where the changes actually interfere, and *false positives* —

8. https://github.com/java-native-access/jna/

conflicts where the changes do not interfere. Ideally, the conflicts of the latter category should be resolved automatically by a merge strategy.

For this first step, we extracted the versions of F from the ancestors of the merge commit, and, if present, from the corresponding merge base. We made our determination on whether the conflict was a true or false positive using traditional three-way visual diff tools. This analysis of 92 files was performed by two of the authors. Where their determinations disagreed, a consensus was reached by discussing the specific conflict in question.

After classifying the conflicts in this manner, we looked at the conflict resolutions produced by $T_{1...n}$. To cut down the number of cases that needed to be analyzed, we grouped conflict resolutions into resolution groups, the members of which are semantically (i.e., behavior wise) equal. Since semantic equality can not be easily determined automatically, we employ syntactic equality instead. Clearly, when two conflict resolutions contain the same code, they behave the same.

The grouping was performed using JDIME'S matching algorithms. When the ASTs of two resolved versions of F matched completely, we assigned them to the same resolution group, as semantic equality follows from syntactic equality. Likewise, we compared one representative of every resolution group with the version of F that was committed in the original merge commit and recorded whether there were syntactic differences.

As a last step, for every resolution group, we manually determined whether the conflict resolution was correct, or whether the test suite missed a semantically invalid resolution. We made this determination for every resolution group, however we paid special attention to those groups that were not syntactically equivalent to the merge commit. In these cases we additionally determined what the syntactic difference was and whether it could indicate an incorrect conflict resolution. Our supplementary website contains a step-by-step example of this process.

Figure 6 shows our process for selecting the conflict resolutions for manual analysis. The basic set of our search were 56 relevant commits as defined above. From these commits, we extracted 92 files for which a merge strategy T_0 produced conflicts. For every file containing conflicts, we found the resolved version of that file produced by some other merge strategy $T_{1...n}$. This left us with 441 conflict resolutions to analyze. We grouped these resolved versions of F by syntactic equality, and analyzed the 105 resolution groups.

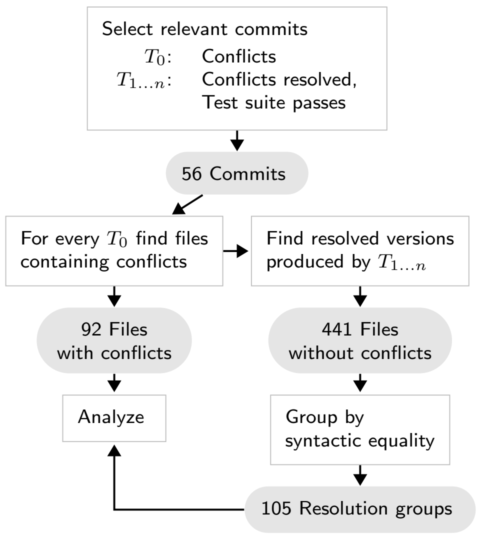
3.4 Subject Systems

Table 3 shows the 10 subject projects that we chose for our study. All projects are available open source on GITHUB and use the GIT version control system. This enables us to examine the entire history of the project and extract all merge scenarios. Furthermore, the projects all use the MAVEN build system, which is used by MERGEPROFILER to execute builds and run tests.

We selected our subject systems by querying the GITHUB search API for Java projects ranked by their popularity (as determined by the number of watchers). Additionally, we considered projects that are known to the authors from previous studies.[9] From this set of projects, we selected those that use MAVEN by examining them for the presence of a pom.xml file in the root folder of the repository. This left us with a set of projects small enough for manual analysis, during which we excluded projects that had too few (less than 100) merge commits or did not have a useful test suite. To determine the latter, we examined a sample of merge commits from the history of the project and executed the test suite manually. We looked at whether unit tests where consistently used throughout the history of the project and excluded those projects where testing was only a recent addition. Furthermore, we manually examined the test suite and judged whether significant portions of the project were covered. We also recorded the runtime of the test suite and, to keep the runtime of the whole analysis feasible, excluded projects which took longer than one hour to test.

The remaining 10 projects range from 109 merge commits to 1829 and cover a variety of application domains:

- COMMONS-MATH is a library extending the Java java.math package by additional mathematical and statistical functions. It is developed by the Apache Software



Select relevant commits
T_0: Conflicts
$T_{1...n}$: Conflicts resolved, Test suite passes
56 Commits
For every T_0 find files containing conflicts
Find resolved versions produced by $T_{1...n}$
92 Files with conflicts
441 Files without conflicts
Analyze
Group by syntactic equality
105 Resolution groups

Fig. 6: The process for selecting conflict resolutions for manual analysis.

TABLE 3: Overview of our subject projects in terms of their number of lines of code, contributors, (merge) commits, and tests.

Project	KLOC	Contributors	Commits	Merge Commits	Tests
COMMONS-MATH	174	44	7428	109	5526
DROPWIZARD	51	364	6788	1021	12
FASTJSON	170	158	3783	453	4646
GHPRB-PLUGIN	8	127	1661	338	77
GITHUB-API	15	156	1381	194	144
JAVAPARSER	158	154	6219	1664	1479
JEDIS	32	174	3214	366	420
OKHTTP	37	235	4708	1829	2244
ONTOP	144	32	11 176	1453	46
OPENMRS-CORE	120	368	9085	1152	4022

9. https://twiki.cin.ufpe.br/twiki/bin/view/SPG/SampleSystems?sortcol=3;table=1;up=0#sorted_table

Foundation.

- DROPWIZARD is a framework for building RESTful Web services. The project was created by Coda Hale (Yammer) and maintained by the Dropwizard team.
- FASTJSON is a library for serialization and deserialization of Java objects to and from JSON. Fastjson is developed by the Fastjson Develop Team at Alibaba.
- GHPRB-PLUGIN is the GITHUB pull request builder plugin for the build automation server JENKINS. It provides access to GitHub pull requests (through the GITHUB API) on Jenkins. The plugin is developed by the community behind the Jenkins project.
- GITHUB-API is a library for accessing the GITHUB API in an object-oriented way such that it is more familiar to Java programmers. It maps most functions of the REST API to JAVA objects and is developed by an active development community.
- JAVAPARSER is a library enabling its users to parse JAVA code to an AST, operate on the AST (e.g., perform refactorings) and generate Java code from a given AST. The project is maintained by Danny van Bruggen and developed on GITHUB.
- JEDIS is the JAVA client for the database server REDIS. The library provides access to REDIS servers using patterns familiar to Java programmers. It is maintained by Jonathan Leibiusky on GITHUB.
- OKHTTP is frequently used in Android Apps as an HTTP and HTTP2 client library. Its aim is to provide efficient and stable communication with a Web server. The library is maintained by Square, Inc.
- ONTOP is a framework providing an interface between the graph-based query language for RDF data SPARQL and relational databases. It was developed by the "Knowledge Representation meets Databases" research group at the Free University of Bozen.
- OPENMRS-CORE is part of the OPENMRS medical records system. The library provides the base API and the code of the web application. The code is developed by the non-profit OpenMRS, Inc.

Table 3 provides several metrics about the subject projects, which we used in determining whether they are suitable for the study. The line count was taken from the latest merge commit in the history of the project (see Appendix A) and determined using SLOCCOUNT[10] and represents the number of lines of JAVA code. Similar to the line count, the number of tests was determined by our tooling using the latest merge commit of the repository for which the test suite could be executed. Both the number of commits and the number of merge commits can be determined using GIT. Note that 852 merge commits listed in the table were not usable for our study (e.g., did not compile, did not have tests) and were filtered out, leaving us with a total of 7727 merge commits. The number of contributors was determined by querying the GITHUB API.

10. https://dwheeler.com/sloccount/

3.5 Results

3.5.1 *RQ*$_1$ (Runtime)

To evaluate the runtime cost of using more sophisticated merge algorithms, we collected runtime statistics reported by JDIME for the unstructured (*US*), semistructured (*SS*), and structured (*S*) merge strategies, for all merge scenarios (i.e., triples of files being merged) occurring in our subject systems. In addition, we profiled the combined strategies *SS*→*S*, *US*→*SS*, *US*→*S*, and *US*→*SS*→*S* in the same way.

TABLE 4: The results of our runtime analysis. Mean and median runtimes, as well as the corresponding standard deviations, are given in milliseconds.

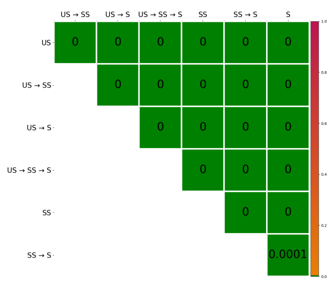
Strategy	Mean	Median	SD Mean	SD Median
US	93	22	336	342
US→*SS*	5145	171	22 293	22 303
US→*S*	8557	174	35 230	35 234
US→*SS*→*S*	8981	169	41 016	40 961
SS	17 825	2657	60 987	60 892
SS→*S*	22 955	4513	70 709	70 547
S	26 665	5589	73 981	73 791

Table 4 gives the mean and median runtime over all merge commits. Additionally, we show the standard deviation of the mean and median runtime.

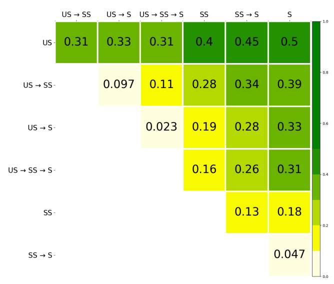
Intuitively, the expectation is that more complex merge strategies should come with a correspondingly higher runtime. Our findings confirm this expectation. We found that the unstructured strategy was fastest with a median runtime of 22 ms and a standard deviation of 342 ms. The median runtime sharply increases to 2657 ms (standard deviation of 60 892 ms) when using the semistructured strategy and a further 111 % to 5589 ms (standard deviation of 73 791 ms) when employing fully structured merging. Ranking the strategies by their median runtime therefore places unstructured merging first, semistructured second, and structured merging third. Given the standard deviations, the mean runtime of the strategies is much higher than the median. However, using the mean for ranking gives the same result as using the median.

Furthermore, we found that the runtime of a combined strategy is mostly dependent on the first strategy in the sequence of applications. The combined strategies having *US* as their first component all display a similar median runtime at around 170 ms, however their standard deviations differ substantially, *US*→*SS* having 22 303 ms, *US*→*S* 35 234 ms, and *US*→*SS*→*S* 40 961 ms. Amongst them, the *US*→*SS*→*S* strategy was the fastest, followed by *US*→*SS* and *US*→*S*. This finding confirms that, using a faster strategy first to "filter out" merge scenarios that are trivially merged without conflicts, works as intended. The last combined strategy, *SS*→*S*, was found to have a median runtime of 4513 ms putting it between its components *SS* (2657 ms) and *S* (5589 ms). Our supplementary Website contains a breakdown of the component runtimes for the combined strategies.

Figure 7 shows the results of statistical tests performed on the runtimes of our strategies. First, we performed a Wilcoxon signed-rank test paired on commits [15]. For this



(a) The results of the Wilcoxon signed-rank test rounded to four significant digits.



(b) The effect sizes calculated using Cohen's d.

Fig. 7: The results of our statistical tests performed on the runtimes of our strategies.

test, the null hypothesis is that the runtime differences have a median of zero and the alternative hypothesis is that the strategy of the row has a lower runtime than the strategy of the column. Due to the repeated statistical tests, we apply a Bonferroni correction to our original significance level of 0.05, which results in an adjusted level of 0.002 38 [16]. All individual tests gave a p-value below this level, confirming that the observed increase in the runtime is statistically significant. Furthermore, we calculated the effect sizes using Cohen's d and found that they increase with the difference in mean and median runtime between the strategy of the row and the strategy of the column [17].

Note that, since we ran every merge commit 5 times, one could use the mean or median of these runs for the above calculations. Figure 7 shows the values for the mean, the results for the median are virtually identical.

Combining the rankings, we find that *US* is the fastest, followed by the combined strategies having *US* as their first component, then followed by *SS*, *SS→S*, and finally *S* (fully structured merging).

3.5.2 **RQ$_2$** (Number of Conflicts)

Table 5 shows the results obtained from re-merging the merge commits found in our subject systems using the strategies under test. For **RQ$_2$**, the results regarding merge conflicts are relevant; they show the number of merge commits for which re-merging with the respective strategy produced merge conflicts. As commits may be discarded before being checked for merge conflicts (e.g., if there is an error when attempting to re-merge them), the table also contains a column holding the percentage of all possible commits that were found to contain conflicts.

The first row ("Commit") of the table shows the state of the code as it was committed to the repository. The second row of the table shows the results of applying unstructured merge using JDIME. We found 467 (6.55 %) commits that produced merge conflicts. As expected, the number of conflicting merges drops when applying semistructured (6.02 %) and structured (5.32 %) merging. Looking at the combined strategies, we can see that their performance in terms of merge conflicts is determined by the most complex strategy in the sequence. *US→S*, *US→SS→S* and *SS→S* perform, at least, as well as the purely structured strategy, while *US→SS* shows a similar performance as the semistructured strategy. Comparing *US* and *US→SS→S*, we observe the largest decrease in the number of merge commits producing conflicts: 29.98 %.

Figure 8 shows the conflict statistics on a per-commit basis. For every merge commit and strategy, we provide the number of conflicts (top left) and the number of files containing conflicts (top right). To calculate the given p-values, we performed a Wilcoxon signed-rank test paired on commits of adjacent strategies with the null hypothesis being that the measurement differences have a median of zero and the alternative hypothesis being that the left strategy is greater than the right. To compensate for the repeated statistical tests, we apply a Bonferroni correction to our original significance level of 0.05, which results in an adjusted level of 0.0083. The p-values given in Figure 8 are bold if they are significant considering our adjusted significance level.

Strategies that do not produce a conflict for a given commit were recorded with a 0. Commits for which no strategy resulted in a conflict were not included. The strategies are given in descending order of their total number of conflicts.

Except for the reductions from *US→SS* to *S* and from *US→S* to *US→SS→S*, all observed reductions are statistically significant. In terms of the number of files containing conflicts, the picture is even clearer. Here, only the p-value for *US→SS* to *S* is above our adjusted significance level. Before applying a Bonferroni correction, this reduction was also statistically significant.

The median values for all strategies are rather similar, so we calculated the effect sizes using Cohen's d. We find that, for both conflicts per commit and conflicting files per commit, all d-values for the adjacent pairs of strategies in Figure 8 are low (less than 0.08). Highlighting one pair of strategies not adjacent in Figure 8, for the conflicting files of *US* and *US→SS→S*, we observe small effect sizes of >0.2. Practically, this means that, for the majority of merge scenarios, the choice of the merge strategy has only a small influence on the number of conflicts, but it is the outliers that matter in practice. We provide all Cohen's d values in Appendix B.

Figure 8 does not show outliers[11], and for good reason: While all strategies show a similar median value for both conflicts per commit and conflicting files per commit, there is a large number of outliers that deviate far from the median value. Figure 9 only shows the outliers which were omitted from Figure 8. Note the difference in the scale of the y-axis. Notably, employing more structure yields fewer merge commits with large numbers of conflicts left over for the developer to resolve, which makes indeed a difference in practice.

Our results clearly show that using more complex merge strategies leads to a statistically significant decrease (up to 30 %) in the number of conflicting merge commits.

TABLE 5: The results for all merge strategies. We record the number of merge commits found to fall into the categories described in Section 3.3.1 and the percentage of the potential number of commits that could have fallen into that category.

	Error	%	of	Merge Conflict	%	of	Build Failure	%	of	Test Failure	%	of	Passed
Commit	50	0.65	7727	0	0.00	7677	186	2.42	7677	40	0.53	7491	7451
JDIME *US*	595	7.70	7727	467	6.55	7132	119	1.79	6665	31	0.47	6546	6515
JDIME *SS*	603	7.80	7727	429	6.02	7124	125	1.87	6695	33	0.50	6570	6537
JDIME *US*→*SS*	606	7.84	7727	394	5.53	7121	132	1.96	6727	41	0.62	6595	6554
JDIME *S*	600	7.76	7727	379	5.32	7127	130	1.93	6748	35	0.53	6618	6583
JDIME *SS*→*S*	605	7.83	7727	362	5.08	7122	134	1.98	6760	36	0.54	6626	6590
JDIME *US*→*S*	603	7.80	7727	333	4.67	7124	140	2.06	6791	39	0.59	6651	6612
JDIME *US*→*SS*→*S*	608	7.87	7727	327	4.59	7119	142	2.09	6792	47	0.71	6650	6603

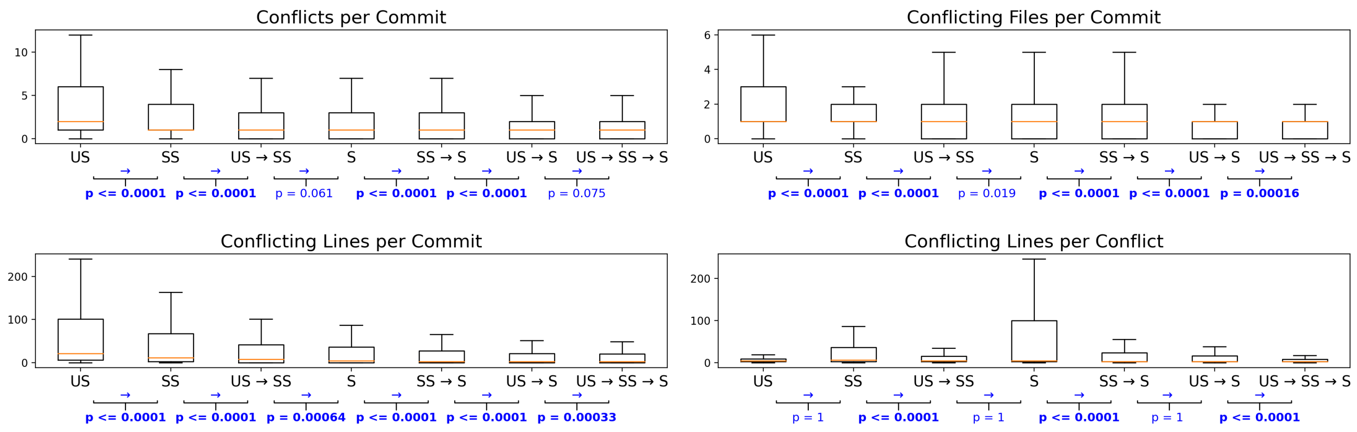


Fig. 8: Conflicts statistics on a per-commit basis, omitting outliers. The top-left plot shows the number of conflicts (one file might contain multiple conflicts), the top right shows the number of files containing conflicts, the bottom left shows the sum of the number of lines in the left and right sides of all conflicts aggregated per merge commit, and the bottom right shows the number of lines in individual conflicts without aggregation.

3.5.3 ***RQ***$_3$ *(Size of Conflicts)*

In addition to the number of conflicts and conflicting files per commit, we determined the size of the reported conflicts in terms of lines of code. Figure 8 shows that the number of conflicting lines per commit follows the same pattern as the number of conflicts per commit. With leveraging progressively more structure, the number of conflicting lines per commit decreases significantly. In fact, whereas the reduction in conflicts per commit was not significant between *US*→*SS* to *S* and *US*→*S* to *US*→*SS*→*S*, the reduction in the number of lines of code is.

For the bottom right plot, where the samples are conflicts instead of merge commits, we performed the Mann-Whitney rank test as conflicts cannot be paired. We find that the potential to produce very large conflicts is higher, the more structure the merge algorithm exploits. Fully structured merge produces the largest conflicts, however, it also reports the fewest.

The effect sizes for the conflicting lines per commit (considering adjacent pairs of strategies from Figure 8) follow a similar pattern as those observed for the conflicts per commit and conflicting files per commit. Except for *US* to *SS* (0.16) and *US*→*SS* to *S* (0.1) they are below 0.08. For the conflicting lines per conflict, d-values are small, but with some notable exceptions: *US* to *SS* with $d = 0.41$ and *US*→*SS* to *S* with $d = 0.23$.

Figure 9 shows the outliers that were omitted from Figure 8. Of particular interest is comparing *US* with *SS* and *S*. The latter two strategies show more extreme outliers for conflicting lines per commit and, in particular, for conflicting lines per conflict.

Section 3.6 discusses an example showing that this is likely due to the heuristics employed in tree matching algorithms of structured merge. Namely, when classes are renamed, a node very close to the root of the AST changes and, since level-wise matching is employed by JDIME, the renamed class is not matched with the other revision (or BASE). As such, when there are also changes to the body of the class, JDIME reports a large conflict over the whole class. Still, this potential to produce conflicts consisting of many lines is, overall, outweighed by the reduction in the number of conflicts, with structured merge producing significantly less conflicting lines per commit than *SS* and *US*.

Examining the size of the conflicts, we find that the reduction in the number of conflicts reported by structured merge outweighs its potential to produce very large conflicts.

11. For outlier detection, we use the PYTHON library PINGOUIN, specifically the method madmedianrule, which is based on the MAD-median rule [18].

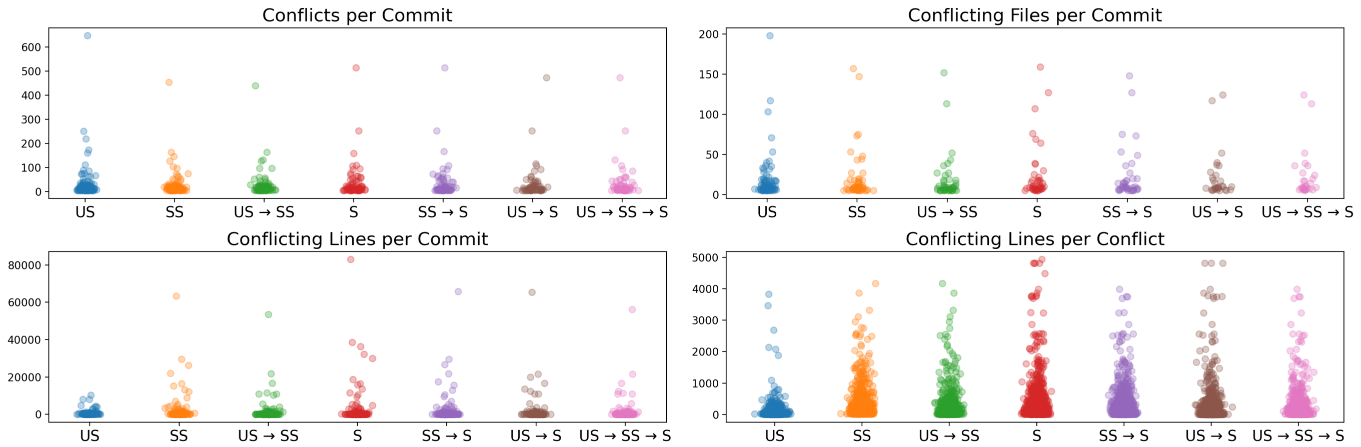


Fig. 9: Conflict statistics on a per-commit basis for outliers only. This plot shows statistics on the outliers that Figure 8 omitted.

3.5.4 **RQ**$_4$ (Number of Build and Test Failures)

The central question of this study was how the number of build- and test failures is affected when using increasingly complex merge strategies. In addition to the raw data in Table 5, Figure 10 provides Sankey plots visualizing the number of commits that were found to fall into one of the categories introduced in Section 3.3.1 (*"Error"*, *"Conflicts"*, *"Build Failure"*, *"Test Failure"*, and *"Test Suite Passed"*). The results for **RQ**$_2$ show that the percentage of conflicting scenarios decreases with the complexity of the merge strategy. At the same time, our results show only a slight increase in the percentage of build failures across the strategies. That is, the complexity of the merge strategy does not lead to a major increase in failures in the build phase. Amongst the simple strategies, the unstructured JDIME mode has the lowest percentage of build failures with 1.79 % whereas the structured mode has the highest percentage (1.93 %), a difference of only 0.14 percentage points. The combined strategies are similarly close, with *US→SS* lowest at 1.96 % and *US→SS→S* highest (2.09 %).

We also observe a slight increase in test failures when using more complex strategies. While unstructured merge produces test failures in 0.47 % of the merges, the semistructured strategy gives 0.5 % and the structured strategy 0.53 %. Using combined strategies again gives a slight increase in test failures over the simple strategies. Counterintuitively, the *US→SS→S* strategy produces the highest number of test failures (0.71 %). The other combined strategies are not far behind with between 0.62 % and 0.54 %.

Overall, we observe a small increase in test failures (12.76 % *US* to *S*) and a minor increase in build failures (7.82 % *US* to *S*) amongst the strategies when the complexity of the strategy increases.

3.5.5 Manual Analysis

For our manual analysis, we selected merge commits from our subjects systems for which one strategy produced conflicts, while other strategies resolved these conflicts with the test suite passing subsequently. We found 92 instances in which a file, being part of the re-merge of one such merge commit, contained conflicts after one of our merge strategies was applied. Of these conflicting files, we found 91 false positives and 1 true positive.

Amongst the false positives, one pattern is particularly common: The unstructured merge strategies (which includes merging method bodies for semistructured merge) frequently report conflicts between changes to adjacent lines in the source code. For example, LEFT changes a line containing the signature of a method, while RIGHT changes an annotation directly above that line. In these cases, strategies exploiting structure in their merge algorithm resolve the conflict, since the changes were made to different parts of the AST and do not interfere.

Structured strategies produce false positives as well. Here, the conflicts are mainly caused by incorrect matching of ASTs. For example, when a method is renamed in LEFT while RIGHT changes that methods body, JDIME will produce a deletion–change conflict, since the renaming is detected as a deletion of the original method (which was changed by RIGHT) and an addition of a new method. Similarly, when both versions add a method, each to the same class, these methods may erroneously be matched and, if they are not exactly equal, will produce a conflict where they differ. These invalid matchings do not occur in fully unstructured merge or when the changes are on the method level in semistructured merge.

We did find 1 true positive as well, namely commit `53740555`[12] of FASTJSON. In this case, both versions being merged add the same file, which does not exist in the merge base, with different content. As there is no base version of the file, unstructured merge produces conflicts where the versions differ. Since these changes are in white spaces and comments only, structured approaches do not produce these conflicts. We classified this as a true positive since, without a base, there is no safe comparison. However, the resolution produced by the structured approaches is syntactically equivalent to the merge commit.

For the conflicts classified as either true or false positives, we examined the files in which conflicts were resolved

12. https://github.com/alibaba/fastjson/commit/53740555a27a599b42d6e8efc3b893066faa66c9

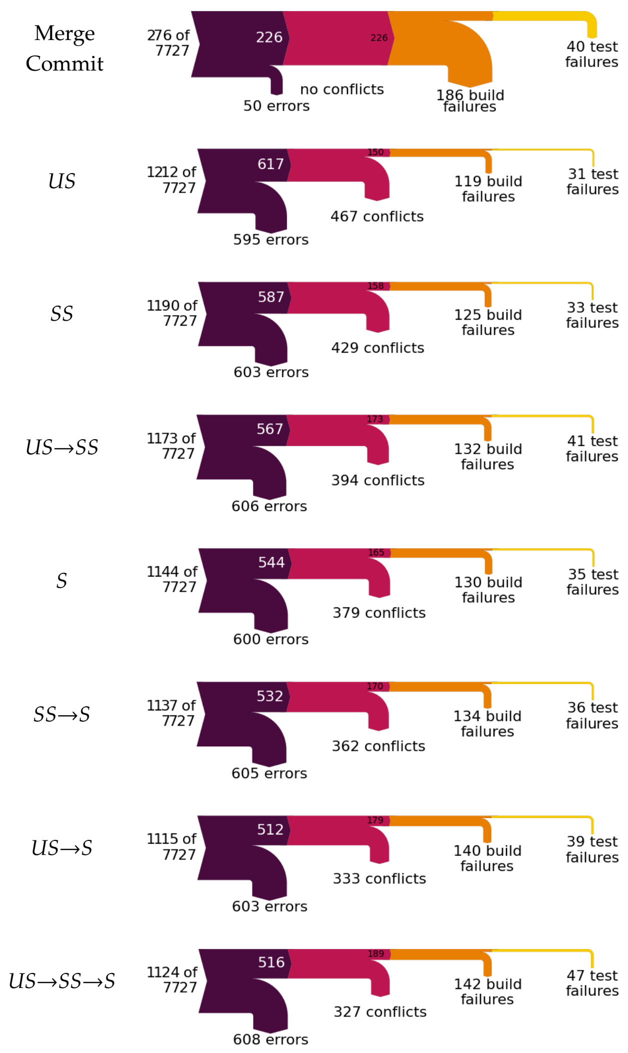


Fig. 10: Sankey plots showing the number of merge commits found to fall into the categories described in Section 3.3.1 depending on the merge strategy used to remerge them. The merge commits of all subject systems were aggregated for this plot. Note that we do not show commits for which the test suite passed to make the plots legible. The number of excluded commits is different for every strategy and the plots are scaled to be of similar size.

by a strategy and determined whether the resolution was correct or not. Recall that we grouped these resolved files by syntactic equality as determined by JDIME. Overall, we examined 105 such groups. Of these groups, 51 were syntactically equal to the resolved version of the conflicting file that was committed as part of the re-merged merge commit. We found all of these resolutions to be correct.

Furthermore, 45 were not syntactically equivalent to the merge commit's version of the file. However, upon manual inspection, we found the syntactic differences to be irrelevant to the functionality of the program in all these cases. The most common pattern here was that, when the code included `if` statements without braces, semistructured merge would keep these statements (as it produces an unstructured merge for method bodies), while structured merge, after pretty-printing the AST, added braces to the `if` statement. As such, the resolved file produced by the structured merge is, at AST level, different from the merge commit. Functionally, however, they are the same.

In 9 cases, we found that the conflict resolution was not correct. These cases occurred in two merge commits, `88ef35c4`[13] of ONTOP and `9aa175e5`[14] of JAVAPARSER. In each case, we observed the same, rather complicated pattern: The file was moved or renamed and the package statement or name of the class in the file was adjusted accordingly. At the same time, the class was changed and these changes only occurred at the method level. In each case, the move was detected by GIT, and the appropriate files from every revision were merged, but the changes made to them, caused conflicts when applying an unstructured merge.

Fully structured merge failed due to the change in the package statement. This change means that the AST of the renamed class does not match with the base revision, whereas the AST of the other revision (where the class was not renamed) does. This is interpreted as a renamed class being added, while the class with the original name is deleted. However, since the deleted class was also changed compared to the base version, this causes a deletion–change conflict. Remarkably, semistructured merge does not exhibit this behavior and instead produces no conflicts but uses the code of the version in which the class was renamed. Upon further examination, we found that this behavior is due to a limitation in semistructured merge. Recall that, to implement semistructured merge, it replaces method bodies by text block nodes in the AST. The structured merge algorithm is then applied to the ASTs. To make it merge the text block nodes, these match by definition, regardless of content, while taking into account their position in the AST, of course. The merge is then performed by the unstructured algorithm and may produce conflicts. However, in the specific case above, this means that the AST of the class with the original name is fully matched with the base revision. The changes in the method bodies are not visible to the matcher. This means that there is no deletion–change conflict, since the deleted class is not changed as far as the structured merge algorithm is concerned. As such, the renamed version of the class is accepted as the result of the merge, discarding the changes made in the other version's method bodies.

While we determined that the resulting code was still correct, as was indicated by the test suite, we marked these cases as incorrect conflict resolutions. Discarding changes made by one version is clearly a problem in semistructured merge. As the combination of changes required for triggering these incorrect conflict resolutions occurs only infrequently, this problem is irrelevant for the overall results. All occurrences, where incorrectly resolved conflicts could have been missed by the test suite, were covered by our manual analysis. A full breakdown of the conflict resolutions that were examined can be found on our supplementary Website.

13. https://github.com/ontop/ontop/commit/88ef35c421b2cabe882243f498833731981b9f82
14. https://github.com/javaparser/javaparser/commit/9aa175e5b6af1a2d8ae601514556ffd88893d94e

3.6 Discussion

In our study we replayed a large number of merges and measured the performance of both simple merge strategies, including simple strategies (i.e., unstructured, semistructured and structured) and combined strategies made up of simple ones. The goal was to compare the strategies for the first time in a controlled setting. Specifically, we considered the runtime of the strategies, the number of conflicts they produced, and we use the test suite to assess the correctness of the code they produced.

We were able to confirm previous results about the runtime performance and the number of conflicts produced by the merge strategies [2], [3]. Simple strategies were studied before, and our controlled setting showed the same sharp increase in runtime as one moves from unstructured over semistructured to a fully structured approach. Combined strategies have been developed with the assumption in mind that most files in a merge scenario can usually be merged without conflicts by an unstructured algorithm. The expectation is that this should lead to a significantly lower runtime (e.g., *US*→*S* as compared to *S*). While there is a small overhead to combining strategies, our data confirm that this assumption holds, at least, for the projects we studied. Going beyond previous work, we developed and evaluated a number of novel combinations (*US*→*SS*, *SS*→*S*, . . .).

We build upon a series of studies that analyzed merge conflicts using a subset of our merge strategies. First, Cavalcanti et al. [19], which replicated the study of Apel et al. [3], analyzed the difference between unstructured and semistructured merge using a different sample than the original study. The authors found a far superior reduction when using semistructured merge in the number of reported conflicts when compared to unstructured merge. Furthermore, Cavalcanti et al. [14], guided by the observed differences in the behavior of these two strategies, compares unstructured and semistructured merge not only in terms of the number of reported conflicts, but also in terms conflicts incorrectly reported by one strategy but not by the other (false positives), and conflicts correctly reported by one strategy but missed by the other (false negatives). They found evidence that semistructured merge, again, reduces the number of conflicts, but also reduces false positives. However, no evidence was found that semistructured reduces false negatives.

In summary, previous work found that more complex algorithms produce less conflicts. Our data reflect that as well and additionally shows that combining strategies perform as intended.

While the observed reductions in conflict metrics are statistically significant, the effect sizes are small. This could be interpreted as that, in practice, the choice of merge strategy does not matter. After all, the median values of our conflict metrics are very similar across all strategies. But, looking at the extremes, for instance, for conflicts per commit (see Figure 9), suggests a different interpretation: While the merge strategies perform similarly for the majority of conflicting merge commits, more structured approaches are able to reduce the number of extreme outliers which are, after all, real merge scenarios with a high number of conflicts (and not statistical noise that needs to be canceled out).

Combined strategies, especially *US*→*SS*→*S* and *US*→*S*, are actually performing better than simply employing structured merging. At first, this appeared counterintuitive to us since this would imply unstructured merge being able to successfully merge a set of files for which structured (or semistructured) merge would produce conflicts. However, Cavalcanti et al. discuss the difference between semistructured and structure merge in their 2019 paper showing that the cases for which *SS* fails are not a subset of the cases for which *US* produces conflicts [5].

We randomly selected and manually examined several merge commits in which *US* produced less conflicts than *S*. In these cases, fully structured merge will perform worse than combined strategies that include unstructured merge.[15] We found situations in which structured merge, due to the unavoidable heuristics employed in the algorithm (e.g., level-wise matching), produced unnecessary conflicts. For example, in our subject system DROPWIZARD, commit d86d7a7e,[16] the package structure of the project was substantially reworked. Most files involved in the merge can be successfully merged using both unstructured as well as structured merge. However, unstructured merge reports conflicts for class DropwizardServiceRuleTest whereas structured merge flags conflicts in both DropwizardServiceRule and DropwizardServiceRuleTest.

Looking at the classes, we found that unstructured merge actually reports two conflicts in DropwizardServiceRuleTest. The first conflict involves the package statement (which was only changed in one version) and the following import statements, which were changed inconsistently in both versions. The second conflict is also due to inconsistent changes to large parts of the class body in both versions. In the second file, the versions, considering the BASE version, make no inconsistent changes and can therefore be merged without conflicts using unstructured merge.

Structured merge flags conflicts (over the whole file) for both DropwizardServiceRule and DropwizardServiceRuleTest. To explain this behavior, the AST structure used in JDIME has to be considered. In the AST, the package statement is actually not a node, it is an attribute of node ClassDecl, which is the root node of a class declaration. As such, when the package statement is changed, the ClassDecl nodes of the ASTs obtained from the LEFT and RIGHT versions do not match. Normally, this conflict would be resolved using the matchings with the BASE version. However, since one (actually both) of the subtrees of the ClassDecl nodes were also changed compared to the BASE version, JDIME can not accept one of the ClassDecl nodes (along with their whole subtree) as the change and therefore the ClassDecl nodes are in conflict. Thus, the unstructured merge algorithm wins out in one of the classes while it still produces conflicts, likely due to clustering granularity of change blocks, in the other.

We performed a more systematic manual analysis of merge conflict resolutions in Sections 3.3.4 and 3.5.5. Look-

15. Analogously, when semistructured merge resolves conflicts that structured merge does not, *SS*→*S* can outperform *S*.

16. https://github.com/dropwizard/dropwizard/commit/d86d7a7e67d871dc6035c916601cfaca49ad056f

ing at these conflicts and their resolutions, we find similar patterns as were described above. Unstructured merge will often produce (false positive) conflicts due to adjacent changes, which are clustered by GIT. While structured merge is able to resolve these conflicts, it struggles in different areas where unstructured merge does not. Specifically, when classes are renamed and changed at the same time, this will lead to large conflicts across entire class definitions. Using more sophisticated matching techniques in structured merge, such as a lookahead [10], would help resolving the conflicts that structured merge reports.

Summarizing our observations, combined strategies are looking very enticing. While unstructured merge remains the fastest strategy, the combined strategies are not far off. Their runtime should still be acceptable for integrating them into the coding workflow especially when one considers that they produce significantly fewer conflicts than unstructured merge. Especially *US*→*SS*→*S* is a promising strategy. For most projects, this strategy shows an acceptable increase over unstructured merging (22 ms median runtime for *US*, 169 ms for *US*→*SS*→*S*) compared to the substantial increase incurred by employing fully structured merging (5589 ms median runtime). Additionally, it performs, at least, as well as the structured strategy in terms of conflicts; for some projects it even produces fewer conflicts than the structured strategy.[17]

It is important to note that all strategies, except for *US*, show a large standard deviation in their median runtime, suggesting that runtime spikes are an issue for structured merge. Recall that, even when changes are small, the runtime of structured merge increases with the complexity of the AST (i.e., large classes incur a substantial runtime penalty). Combined strategies that include *US* as one of their components appear to be able to mitigate this unfavorable behavior.

A major contribution of this work is the additional data obtained by running the test suite of the subject projects after replaying a merge commit. Using a merge strategy, that is able to resolve merge conflicts automatically that otherwise would require manual intervention, is not of much use if the merged code does not work correctly. In general, we found that using more complex strategies leads to more build and test failures. Crucially, combined strategies further increase this number, i.e. *US*→*SS*→*S* appears to cause more test failures than simple structured merging. Apparently the ways in which the strategies produce conflict-free but incorrect code get worse in combined strategies, leading to the observed increase in build and test failures. Still, with the increase in test failures being quite small, from 0.53 % for the structured strategy to 0.71 % for the combined strategy *US*→*SS*→*S*, we still believe that it is a viable option. Assuming the reasons for incorrect code being produced by *US*, *SS*, and *S* are different, this also presents three separate avenues of improvement for the combined strategy.

While the number of build and test failures increases, so does the number of cases in which the merged code passes the test suite. Essentially, some resolved conflicts reappear as build or test failures, indicating that the conflict resolution may not have been correct. Others do not cause build or test failures, instead contributing to the number of commits for which the test suite passed. We see this as an indicator for the correctness of the merge result. Note that we also performed a manual analysis on merge conflict resolutions to increase confidence in their correctness. For a description of that analysis please refer to Section 3.3.4 and Section 3.5.5 for the results.

Finally, what does this mean for developers? Using anything other than unstructured merge in day-to-day software development is unusual, to say the least. So, why should developers adopt novel strategies which might have an increased runtime and pose the risk of introducing superfluous conflicts or test failures? We believe that a combined strategy of unstructured and structured approaches is feasible in practice. In terms of runtime, it remains an open question what exactly constitutes an acceptable runtime, however, the runtime overhead of a combined strategy in which the unstructured component does not produce conflicts is small, and the runtime of unstructured merge is clearly acceptable for developers. When a structured strategy is employed, the runtime increases significantly. However, this is only necessary in a few cases and has the benefit of resolving additional conflicts. The conflicts that are produced by more complex strategies have the potential of being larger, which, intuitively, might be undesirable behavior. However, in this case, a merge tool might fall back to the conflicts produced by unstructured merge if that is preferable to the developer. Thus, we envision structured merge to be a part of a merge tool that is responsive to the users preferences, as well as takes into account the conflicts produced by its constituent strategies and the state of the project's test suite. This is the ultimate consequence of the original aim of structured merge: Using as much information as is available to resolve conflicts.

3.7 Threats to Validity

3.7.1 Internal Validity

Our analysis uses the test suites present in our subject systems as an indicator for the correctness of the code. While we did take care to select subject systems with appropriate test suites, it remains a threat to internal validity that the test suite may be insufficient to detect some semantically invalid merge conflict resolutions. Since it is unclear how to automatically determine whether a merge conflict resolution was sufficiently tested (test coverage alone is not sufficient), we chose to perform a manual analysis of conflict resolutions. Bearing in mind the results presented in Section 3.5.5, we are confident that this threat is sufficiently mitigated.

Another threat to internal validity is that our results may have been influenced by hidden variables other than the combination of merge strategy and the scenario to be merged. To mitigate the effects of confounding variables when measuring the runtime of our merge strategies, we ensured that the computations were always performed on the same hardware with exclusive access. Additionally, we performed runtime measurements 5 times to reduce the influence of various factors affecting runtime that are inherent in modern hardware and software such as the Java Virtual Machine. We then calculated both the mean and

17. Our supplementary website contains a table showing the total number of merge commits for which a combined strategy outperforms structured merge.

median of our measurements and additionally examined the standard deviation for both measures. When calculating the number and size of conflicts, we employ a well-tested parsing algorithm implemented in JDIME and as such can largely rule out confounding variables. Similarly, the state of the test suite is determined by the build tool. We merely parse its output or, depending on the build tool, interact with it using its API. To rule out flaky tests (e.g., tests whose outcome is based on some random factor), we ran each test suite multiple times and filtered out the tests that did not always give the same result.

A potential threat to internal validity arises from the fact that that three of our subject systems make up a significant portion of merge scenarios. These three projects however already cover a good range of domains (AST parser, HTTP query library, relational DB query library). The threat is also mitigated by the fact that we consider all merge commits to be in one big pool for our analysis.

Overall, the comparatively large number and diverse origin of the merge scenarios we looked at should serve to rule out variables, including differences in software development approach, programming style and experience, as well as the differences between application domains.

3.7.2 External Validity

Care has to be taken when generalizing the results of this work beyond our corpus. Due to the language-specific nature of structured merge algorithms, all our subject systems are written in Java. Semistructured and structured merging may produce more (or less) merge, build, or test failures when implemented for another language. As such, our findings may not be applicable to any language, but they should apply to languages having a similar syntactic structure for which our merge algorithms could be ported (e.g., C++, C#, PYTHON).

Furthermore, we could only access the history of open-source projects from GITHUB. Collaboration practices may differ on other platforms and for closed-source development. In general, the level of sophistication in software development techniques may influence the kinds of merge scenarios to which the merge algorithms are applied. We selected subject systems of various sizes and levels of activity from a range of domains, but how they compare to professionally developed closed-source projects remains an open question for the community as a whole.

4 RELATED WORK

Merging software artifacts is a central task in software development. As a consequence, merge conflicts arise as a common side effect of concurrent development, impairing productivity and compromising quality. Thus, a number of researchers propose development tools and strategies to better support these collaborative development environments.

The state-of-practice is adopting unstructured, line-based tools to merge source code artifacts. Structured merge tools have been proposed as an alternative to unstructured merging with the goal of reducing the number of reported merge conflicts. These tools achieve that by leveraging the structure inherent in source code. Westfechtel [8] and Buffenbarger [9] have pioneered in proposing structured merge algorithms which incorporate context-free and context-sensitive structures.

In this study, we employ JDIME, a structured merge tool proposed by Apel et al. [4]. This tool is also capable of tuning the merging process on-line by switching between unstructured and structured merge, depending on the presence of conflicts.

JDIME was used by Leßenich et al. [2] to study "auto-tuning", which is a precursor of the combined strategies studied in this paper. The auto-tuning approach applies unstructured merge first, and, in case of conflicts, falls back to structured merge. This is, in concept, equivalent to our strategy *US→S*. However, the concrete implementation in JDIME has evolved since this prior work. While our study considers more merge strategies and far more merge scenarios from different subject systems, our results are in line with Leßenich et al. [2]: More structured approaches produce less conflicts, mostly at a finer granularity, with some outliers (e.g., when classes are renamed and changed in two versions) being much larger than in unstructured merge. In terms of runtime vs. conflict resolution performance, our combined approaches are just as promising as the original auto-tuning approach. Our study introduces the *US→SS→S* strategy, which improves over auto-tuning by more gradually ramping up the use of structure in the merge algorithm. Additionally, we employ the test suite of our subject systems, and a thorough manual analysis, as an oracle for correctness of merge conflict resolutions.

Leßenich et al. [10] attempt to improve JDIME by employing a syntax specific look-ahead to detect renamings of declarations and shifted code. They demonstrate that their solution can significantly improve matching precision in 28 % while maintaining performance.

Zhu et al. [20] built, on top of JDIME, another structured merge tool (so-called AUTOMERGE) that matches nodes based on an adjustable *quality function*. Their goal is to find a set of matching nodes that maximizes the *quality function*, preventing the matching of logically unrelated nodes, and, as consequence, unnecessary conflicts. They found that AUTOMERGE was able to reduce the number of reported conflicts by 63 % when compared to original *JDime*, being only 17 % slower. Besides, they found that about 99 % of the results yielded by AUTOMERGE exactly correspond to original developers' result, compared to 93 % from JDIME.

A major limitation of these structured tools, in relation to the state-of-practice unstructured merge, is the runtime complexity of the underlying matching and merging algorithms since it works by merging trees instead of lines of text. So, a semistructured strategy has been proposed by Apel et al. [3] aiming to be the middle ground between unstructured and structured merge algorithms. By working on simplified trees, semistructured merge attempts to resolve as many of the conflicts that structured merging would resolve as possible while speeding up the merge procedure by using unstructured merging for parts of the source code (e.g., to merge body of method declarations). In their work, Apel et al. presented a semistructured merge algorithm that is, in concept, similar to semistructured merge in JDIME. The implementation in JDIME is tightly integrated into the regular structured merge algorithm, restricting it to merging

JAVA code only. This tight integration leads to consistent behavior across the merge strategies that JDIME supports. While there is no overlap in subject systems between this paper and Apel et al. [3], in terms of semistructured merge, our finding align well: Overall, semistructured merge reduces the number of conflicts compared to unstructured merge, however, it struggles in the presence of renaming changes.

Prior work [3], [19] provide evidence that semistructured merge, similarly to structured merge, is able to reduce the number of reported conflicts in relation to traditional unstructured merge. Cavalcanti et al. [14] go further and provide evidence that the number of unnecessary conflicts (false positives) is significantly reduced when using semistructured merge. However, they do not find evidence that semistructured merge misses fewer actual conflicts (false negatives). Similar findings, but in a lesser extent, is observed by Trindade et al. [21] when investigating semistructured merge in JAVASCRIPT systems.

All these previous studies, however, evaluate merge strategies in isolation. In this work we synthesize and expand these studies by exploiting the full spectrum of possibilities: we combine and evaluate all the combinations of unstructured, semistructured, and structured merge in a controlled, integrated environment. We find that combined strategies perform at least as well in terms of conflicts as simple strategies.

Semantic strategies have been impractical for a long time. Horwitz et al. [22] were the first to propose an algorithm for merging program versions without semantic conflicts for a very simple assignment-based programming language. This original work was later extended to handle procedure calls [23] and to identify semantics-preserving transformations [24]. Berzins [25] proposed a general approach by providing a language-independent definition of semantic merging with the use of a generalization of the use of traditional denotation semantics. In recent effort towards semantic merge feasibility, Sousa et al. [26] proposed SAFEMERGE, a semantic tool that checks whether a merged program does not introduce new unwanted behavior. They achieve that by combining lightweight dependence analysis for shared program fragments and precise relational reasoning for the modifications. They found that the proposed approach can identify behavioral issues in problematic merges that are generated by unstructured tools.

Researchers have also investigated the frequency and impact of merge conflicts, and their associated causes. They all conclude that conflicts are frequent. For example, Kasi and Sarma [27] and Brun et al. [28] reproduce merge scenarios from different GitHub projects with the purpose of measuring the frequency of merge scenarios that resulted in conflicts. These studies show average conflicting scenarios rates for merge conflicts of 14 %, and 17 % respectively. Zimmerman [29] conducted a similar analysis reproducing merges from CVS (a centralized version control system) projects instead. Perry et al. [30] made an observational case study to analyze the effect of concurrent changes on a large-scale industrial software system. They reported that, although 90 % of the files could be merged without problems, the degree of parallel changes is high – merge conflicts involved between 2 to up to 16 parallel changes. Our work complements these studies bringing evidence of conflict frequency with the use of different merge strategies, and also the combination of merge strategies. We find that the number of conflicts drops as the complexity of the merge strategy increases. Surprisingly, combined strategies are able to resolve more conflicts than simple strategies.

There are also studies that analyze different technical and organizational aspects that might have an impact on the occurrence of conflicts, and characteristic of conflicts. Cataldo and Herbsleb [31] presented an empirical analysis of a large-scale project where they examined the impact that software architecture characteristics, and organizational factors have on the number of conflicts. They concluded that architecture related factors such as the nature and the quantity of component dependencies, as well as organizational factors such as the geographic dispersion of development teams, can lead to a higher rate of issues when merging. Menezes et al. [32] analyze merge scenarios from open source Java projects to investigate the nature of merge conflicts in terms of what conflicts look like, what kinds of conflicts occur, how developers fix them, how conflicts relate to each other, and more. Based on their results, they argue that it is difficult to envision a single generic merge strategy that can automatically resolve all possible conflicts, because the diversity in conflicts is simply too large. Our work synthesizes and expands on a number of previous papers on merge strategies, so it might help practitioners in deciding which merge strategy, or combination of strategies, to adopt. Accioly et al. [33] derive a catalog of conflict patterns expressed in terms of the structure of code changes that lead to merge conflicts. Their results show that most conflicts occur because developers independently edit the same or consecutive lines of the same method. However, the probability of creating a merge conflict is approximately the same when editing methods, class fields, and modifier lists. They noticed that the most part of conflicting merge scenarios, and merge conflicts, involve more than two developers. Also, that copying and pasting pieces of code, or even entire files, across different repositories is a common practice and cause of conflicts. Similarly, Yuzuki et al. [34] investigate how conflicts on method declarations are resolved on open source Java projects. They found that the most part of them is resolved by adopting one of the versions, then discarding the other. All these findings about conflicts characteristics might be adapted by a merge tool as strategies for resolving conflicts.

Finally, assessing how often merging results in build or test issues, which can be seen as a consequence of a fault merging process, partially motivated a couple of studies [27], [28], [35]. Kasi and Sarma [27], for instance, report merges that result in build issues occurring in ranges between 2 % to 15 %, while Brun et al. [28] describe both build and test issues ranging around 33 %. Comparing merge strategies, however, is not the focus of these studies, they are actually based on traditional unstructured merge. Cavalcanti et al. [5] investigate the frequency of build and test issues with semistructured and structured merge strategies, and they found that structured merge causes more build and test failures than semistructured merge. We complement these prior work by investigating build or test issues with unstructured, semistructured, structure merge and the combination of these strategies. Our results show that, while there is an increase in build- and test failures, the increase is not substantial.

5 Conclusion

Resolving merge conflicts manually remains a major pain-point for software developers [29], [36]. While unstructured merging (which considers source code as a sequence of lines) is applicable to all source code artifacts, structured merging (which considers the syntactic structure of the code) may resolve additional conflicts automatically by exploiting language-specific information. We consider both semistructured and fully structured, AST-based approaches to software merging. Additionally, strategies which represent an adaptive combination of simple strategies are tested (e.g., unstructured merging followed by structured merging if there are conflicts).

Conventional wisdom is that the more structure and language-specific details a merge algorithm considers, the fewer conflicts it produces at a considerable runtime cost. We showed that this assumption holds in practice by applying the merge strategies to 10 open-source projects having in total 7727 merge commits. We observed merge conflicts for 6.69 % of the merge commits for the unstructured strategy while the structured strategy produced conflicts in 5.32 %. The semistructured strategy's result was in between with 6.02 %. Combined strategies performed even better than simple strategies, the best being the combination of all three simple strategies (in increasing order of complexity) with 4.59 % of merge commits being re-merged with conflicts. As such, combined strategies proved to be a very useful addition to the set of available strategies. They appear to combine the best of both worlds, reducing runtime while being able to resolve, at least, as many conflicts as a fully structured merge.

A major contribution of this study is that, in addition to the number of conflicts and execution time, the correctness of the merge result, as determined by the test suite of the project and a thorough manual analysis, was also considered. The key issue was whether, in an effort to resolve as many conflicts as possible automatically, structured merge algorithms produce code that is conflict free but no longer works as intended. In our study, we found that, while the number of build and test failures does increase when employing the semistructured and structured merge strategies, the increase is small enough for these algorithms to be viable in practice.

References

[1] T. Mens, "A State-of-the-art Survey on Software Merging," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 5, pp. 449–462, 2002.

[2] O. Leßenich, S. Apel, and C. Lengauer, "Balancing Precision and Performance in Structured Merge," *Automated Software Engineering*, vol. 22, no. 3, pp. 367–397, 2015.

[3] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured Merge: Rethinking Merge in Revision Control Systems," in *Proc. Europ. Software Engineering Conf. and the Int. Symposium Foundations of Software Engineering (ESEC/FSE)*. ACM, 2011, pp. 190–200.

[4] S. Apel, O. Leßenich, and C. Lengauer, "Structured Merge With Auto-tuning: Balancing Precision and Performance," in *Proc. Int. Conf. Automated Software Engineering (ASE)*. ACM, 2012, pp. 120–129.

[5] G. Cavalcanti, P. Borba, G. Seibt, and S. Apel, "The Impact of Structure on Software Merging: Semistructured Versus Structured Merge," in *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1002–1013.

[6] L. Bergroth, H. Hakonen, and T. Raita, "A Survey of Longest Common Subsequence Algorithms," in *Proc. Int. Symposium String Processing and Information Retrieval (SPIRE)*. IEEE, 2000, pp. 39–48.

[7] S. Khanna, K. Kunal, and B. C. Pierce, "A Formal Investigation of Diff3," in *Proc. Int. Conf. Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Springer, 2007, pp. 485–496.

[8] B. Westfechtel, "Structure-oriented Merging of Revisions of Software Documents," in *Proc. Int. Workshop Software Configuration Management (SCM)*. ACM, 1991, pp. 68–79.

[9] J. Buffenbarger, "Syntactic Software Merging," in *Proc. Int. Workshop Software Configuration Management (SCM)*, 1995, pp. 153–172.

[10] O. Leßenich, S. Apel, C. Kästner, G. Seibt, and J. Siegmund, "Renaming and Shifted Code in Structured Merging: Looking Ahead for Precision and Performance," in *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE, 2017, pp. 543–553.

[11] S. Böcker, D. Bryant, A. W. Dress, and M. A. Steel, "Algorithmic Aspects of Tree Amalgamation," *Journal on Algorithms*, vol. 37, no. 2, pp. 522–537, 2000.

[12] K. Zhang and T. Jiang, "Some MAX SNP-hard Results Concerning Unordered Labeled Trees," *Information Processing Letters*, vol. 49, no. 5, pp. 249–254, 1994.

[13] H. W. Kuhn, "The Hungarian Method for the Assignment Problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1–2, pp. 83–97, 1955.

[14] G. Cavalcanti, P. Borba, and P. R. G. Accioly, "Evaluating and Improving Semistructured Merge," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. OOPSLA, pp. 59:1–59:27, 2017.

[15] F. Wilcoxon, "Individual Comparisons of Grouped Data by Ranking Methods," *Journal of Economic Entomology*, vol. 39, no. 2, pp. 269–270, 1946.

[16] R. A. Armstrong, "When to use the Bonferroni Correction," *Ophthalmic & Physiological Optics*, vol. 34, no. 5, pp. 502–508, 2014.

[17] J.-C. Goulet-Pelletier and D. Cousineau, "A Review of Effect Sizes and their Confidence Intervals, Part I: The Cohen's d Family," *The Quantitative Methods for Psychology*, vol. 14, no. 4, pp. 242–265, 2018.

[18] P. Hall and A. Welsh, "Limit Theorems for the Median Deviation," *Annals of the Institute of Statistical Mathematics*, vol. 37, no. 1, pp. 27–36, 1985.

[19] G. Cavalcanti, P. Accioly, and P. Borba, "Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment," in *Proc. Int. Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 2015, pp. 267–276.

[20] F. Zhu, F. He, and Q. Yu, "Enhancing Precision of Structured Merge by Proper Tree Matching," in *Proc. Int. Conf. Software Engineering (ICSE): Companion Proceedings*. IEEE / ACM, 2019, pp. 286–287.

[21] A. Trindade, P. Borba, G. Cavalcanti, and S. Soares, "Semistructured Merge in Javascript Systems," in *Proc. Int. Conf. Automated Software Engineering (ASE)*. ACM, 2019, pp. 1014–1025.

[22] S. Horwitz, J. F. Prins, and T. W. Reps, "Integrating Noninterfering Versions of Programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 11, no. 3, pp. 345–387, 1989.

[23] D. W. Binkley, S. Horwitz, and T. W. Reps, "Program Integration for Languages with Procedure Calls," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 4, no. 1, pp. 3–35, 1995.

[24] W. Yang, S. Horwitz, and T. W. Reps, "A Program Integration Algorithm That Accommodates Semantics-preserving Transformations," in *Proc. Symposium Software Development Environments (SDE)*. ACM, 1990, pp. 133–143.

[25] V. Berzins, "Software Merge: Semantics of Combining Changes to Programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 6, pp. 1875–1903, 1994.

[26] M. Sousa, I. Dillig, and S. K. Lahiri, "Verified Three-way Program Merge," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, no. OOPSLA, pp. 165:1–165:29, 2018.

[27] B. K. Kasi and A. Sarma, "Cassandra: Proactive Conflict Minimization Through Optimized Task Scheduling," in *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2013, pp. 732–741.

[28] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive Detection of Collaboration Conflicts," in *Proc. Europ. Software*

Engineering Conf. and the Int. Symposium Foundations of Software Engineering (ESEC/FSE). ACM, 2011, pp. 168–178.

[29] T. Zimmermann, "Mining Workspace Updates in CVS," in *Proc. Int. Workshop Mining Software Repositories (MSR)*. IEEE, 2007, p. 11.

[30] D. E. Perry, H. P. Siy, and L. G. Votta, "Parallel Changes in Large-scale Software Development: An Observational Case Study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 3, pp. 308–337, 2001.

[31] M. Cataldo and J. D. Herbsleb, "Factors Leading to Integration Failures in Global Feature-oriented Development: An Empirical Analysis," in *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2011, pp. 161–170.

[32] G. G. L. Menezes, L. G. P. Murta, M. O. Barros, and A. Van Der Hoek, "On the Nature of Merge Conflicts: a Study of 2,731 Open Source Java Projects Hosted by Github," *IEEE Transactions on Software Engineering (TSE)*, 2018.

[33] P. R. G. Accioly, P. Borba, and G. Cavalcanti, "Understanding Semi-structured Merge Conflict Characteristics in Open-source Java Projects," *Empirical Software Engineering (EMSE)*, vol. 23, no. 4, pp. 2051–2085, 2018.

[34] R. Yuzuki, H. Hata, and K. Matsumoto, "How We Resolve Conflict: An Empirical Study of Method-level Conflict Resolution," in *Proc. Int. Workshop on Software Analytics (SWAN)* . IEEE, 2015, pp. 21–24.

[35] P. Accioly, P. Borba, L. Silva, and G. Cavalcanti, "Analyzing Conflict Predictors in Open Source Java Projects," in *Proc. Int. Conf. Mining Software Repositories (MSR)*. ACM, 2018, pp. 576–586.

[36] H. C. Estler, M. Nordio, C. A. Furia, and B. Meyer, "Awareness and Merge Conflicts in Distributed Software Development," in *Proc. Int. Conf. Global Software Engineering (ICGSE)*, 2014, pp. 26–35.