Mastery: Shifted-Code-Aware Structured Merging

Fengmin Zhu[1,4][0000−0003−4219−0837], Xingyu Xie[1][0000−0002−2220−7294], Dongyu Feng[1][0000−0002−8515−3975], Na Meng[5], and Fei He(✉)[1,2,3][0000−0002−4266−875X]

[1] School of Software, Tsinghua University
[2] Key Laboratory for Information System Security, MoE, China
[3] Beijing National Research Center for Information Science and Technology
[4] Max Planck Institute for Software Systems
[5] Virginia Tech
hefei@tsinghua.edu.cn

Abstract. Three-way merging is an essential infrastructure in version control systems. While the traditional line-based textual methods are efficient, syntax-based structured approaches have shown advantages in enhancing merge accuracy. Prior structured merging approaches visit abstract syntax trees in a top-down manner, which is hard to detect and merge shifted code in the general sense. This paper presents a novel methodology combining a top-down and a bottom-up visit of abstract syntax trees, which manipulates shifted code effectively and elegantly. This merge algorithm is order-preserving and linear-time. Compared with four representative merge tools in 40,533 real-world merge scenarios, our approach achieves the highest merge accuracy and 2.4 x as fast as a state-of-the-art structured merge tool.

Keywords: Version control systems · Three-way merging · Structured merging · Shifted code.

1 Introduction

Thanks to the wide application of version control systems such as Git and SVN, *three-way merging* has become an indispensable task in contemporary software development. A *three-way merge scenario* $(base, left, right)$ consists of three versions of a program, where the two *variants left* and *right* are both evolved independently, possibly by different developers, from their ancestor *base*. A three-way merge algorithm integrates the changes made by the variants and produces a merged version called a *target*. When the two variants (i.e., branches) introduce changes that are contradicted, according to *three-way merge principles*, a

Early revisions of this work were done when F. Zhu was in Tsinghua University.
F. Zhu and X. Xie contributed equally.

conflict shall be reported, leaving the developers to manually resolve them. The three-way merge principles conservatively describe whether the changes could be correctly integrated.

Unstructured merge is a mature merging approach that regards programs as a sequence of lines of plain text. Since the context-free syntax is neglected, merge accuracy is yet unsatisfying—studies [17,16] have shown the presence of *false conflicts* (i.e., the conflicts that should have been avoided), which increases the user burden of manual resolution. To enhance merge accuracy, *structured merge*, representing programs as *abstract syntax trees* (ASTs), has gained significant research interest in recent decades [16,15,2,1,5,23,24]. A structured merge algorithm takes a set of mappings between different program versions as input and computes a merged version as output. The mappings are obtained by AST differencing (also known as AST matching) algorithms [7,9,6].

To compute a target AST, a structured merge algorithm needs to traverse the input ASTs (i.e., *base*, *left*, and *right*). Prior approaches [16,15,2,1,5,23,24] all use a *top-down* order, which is quite natural and intuitive as it follows the structure of ASTs. Such a top-down AST comparison is usually restricted to be *level-wise*—only AST nodes at the same level (or depth) get compared, which makes it hard to detect if one piece of code is shifted into another, namely *shifted code* [14]. To identify shifted code in a top-down manner, one could search for the *largest common embedded subtree*. This problem, however, is known to be $\mathcal{NP}$-hard and difficult to approximate for general cases [22]. A more scalable approach is to employ *syntax-aware looking ahead* matching [14], but: looking ahead is only enabled for a few types of AST nodes; the maximum looking-ahead distance is short for efficiency considerations. Their work focuses on the AST matching problem; how to correctly merge shifted code remains an issue.

Thinking oppositely, we find a *bottom-up* traversing order a better option. The key to detecting shifted code is to allow node mappings across AST levels, which is natural and easier via a bottom-up manner. Meanwhile, top-down merging cannot handle across-level mappings, which means bottom-up merging is needed as the follow-up of the matching phase. Sometimes, the bottom-up visit *alone* incurs redundant computations. As an extreme example, if *left* is the same as *base*, meaning no changes are introduced on *left*, then by three-way merge principles, *right* introduces unique changes and should be the target version—there is no need to further inspect any of their descendants as in a bottom-up manner. We fix this issue by bringing in top-down merging.

Combining a top-down pruning pass and a bottom-up pass, we present a novel three-way structured merge algorithm, where the trivial merge scenarios are processed in the former pass, and other nontrivial merge scenarios, which may involve shifted code, are carefully operated in the latter pass. Because our algorithm is non-backtracking, the time complexity is linear.

Like in JDime [16], we distinguish if the children of an AST node list can be "safely permuted" (i.e., the permutation preserves semantics). If not, the list is called *ordered* (e.g., a sequence of statements) and a good merge algorithm should preserve the original occurrence order of the children in the merged ver-

sion, which we call *order-preservation*. We reduce this problem into computing a topological sort of a directed graph (that encodes required constraints), which is solvable by linear-time graph algorithms such as Kahn's algorithm [13].

We implemented our approach as a structured merge tool called Mastery[6]. To measure its usability and practicality in real scenarios, we extract 40,533 merge scenarios from 78 open-source Java projects. We identified that shifted code occurs in 38.54% merge scenarios, and conduct experimental comparisons with four representative tools: JDime (structured), jFSTMerge (tree-based semistructured), IntelliMerge (graph-based semistructured), and GitMerge (unstructured). Our results show: (1) Mastery achieves the highest merge accuracy of 82.26%; (2) Mastery reports the fewest 9.09% conflicts and the fewest 6,791 conflict blocks, excluding radical IntelliMerge; (3) Mastery is about 2.4× as fast as JDime, and about 1.3× as fast as jFSTMerge. Our tool and evaluation data are publicly available: https://github.com/thufv/mastery/.

To sum up, this paper makes the following contributions:

- We present a novel structured merge algorithm that visits ASTs in both a top-down and a bottom-up manner. The top-down pruning pass avoids a mass of redundant computations. The bottom-up pass makes it possible to handle shifted code elegantly and efficiently.
- We show that the proposed merging algorithm is linear-time and the ordered merging algorithm is order-preserving.
- We conduct comprehensive experiments on real-world merge scenarios. Results show that Mastery is competitive with state-of-the-art merge tools in the aspects of merge accuracy, the number of conflicts, and efficiency.

2 Preliminary

AST nodes In structured merging, programs are represented as *abstract syntax trees* (ASTs), parsed from source files. An AST is a labeled rooted tree with four types of nodes, each annotated with the name of its production rule in the grammar, called its *label* (*lbl*).

$$
\begin{array}{rll}
\text{Node } v ::= & \mathsf{Leaf}(lbl, x) & \text{(leaf)} \\
\mid & \mathsf{Ctor}_k(lbl, v_1, \ldots, v_k) & (k\text{-ary constructor}) \\
\mid & \mathsf{UList}(lbl, \{v_1, \ldots, v_n\}) & \text{(unordered list)} \\
\mid & \mathsf{OList}(lbl, [v_1, \ldots, v_n]) & \text{(ordered list)}
\end{array}
$$

A k-ary *constructor node* has exactly k children as its arguments. For instance, an if-statement—consisting of a Boolean condition, a true branch, and a false branch—is represented as a 3-ary constructor node. An arbitrary number of children is allowed in a *list node*, which is further divided into *unordered*—children can be safely permuted—and *ordered* (the opposite). For instance, a class member declaration list is unordered, while a statement list is ordered. In case of merge conflicts, we introduce conflicting nodes in target ASTs.

[6] Merging abstract syntax trees in a reasonable way.

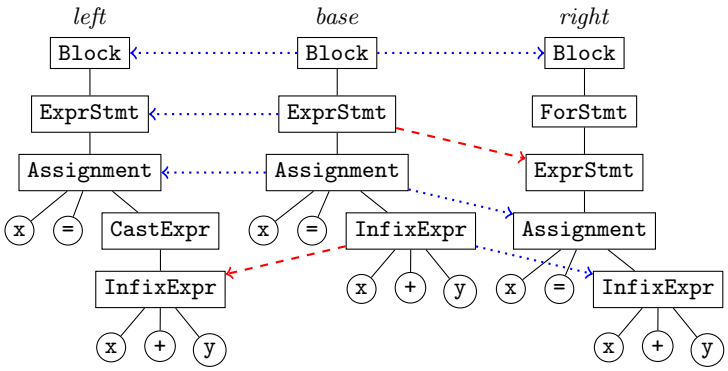

Fig. 1: A merge scenario with shifted code. (Shifted mappings are depicted as dashed arrows.)

Table 1: Three-way merge principles (dual cases omitted).

Type	Version *base*	Version *left*	Version *right*	Target T	Explanation
1 Node	e	e	e'	e'	left-change
2 Node	e	e_L	e_R	conflict	inconsistent change
3 List	$e \in$ *base*	$e \notin$ *left*	$e \in$ *right*	$e \notin T$	left-deletion
4 List	$e \notin$ *base*	$e \in$ *left*	$e \notin$ *right*	$e \in T$ or conflict	left-insertion

AST Matching A merging algorithm relies on a set of mappings between different versions of the programs to compute the merged version. The set of mappings between two ASTs T_1 and T_2 are represented by a *matching set* $\mathcal{M} = \{(u_i, v_i)\}_i$, where each pair (u_i, v_i) consists of two nodes $u_i \in T_1$ and $v_i \in T_2$. The mappings shall be *injective*—two nodes cannot be matched to the same node on the other AST simultaneously. Moreover, the matched nodes shall have the same label.

Definition 1 (Shifted code). *Given two mappings $(u', v'), (u, v) \in \mathcal{M}$, if u is a child of u', whereas v is not a child (i.e., direct descendant) but a later descendant of v', then (u, v) is said a* shifted *mapping. Meanwhile, the code fragment corresponding to the subtree of v is called a* shifted code.

For example, on the merge scenario shown in Fig. 1: in *left*, the code fragment of `InfixExpr` is a shifted code and is shifted into a `CastExpr`; in *right*, the code fragment of `ExprStmt` is a shifted code and is shifted into a `ForStmt`.

Three-way Merge Principles A three-way merge algorithm must abide by a couple of principles, as presented in Table 1. To avoid repetition, the dual cases of rows 1, 3, and 4 are not displayed. The first two rules are applicable for all types of nodes. If a node is modified by exactly one of the variants, then the change is unique and itself gives the target (row 1). If a node is concurrently modified by both variants inconsistently, a conflict is reported, as the algorithm has no

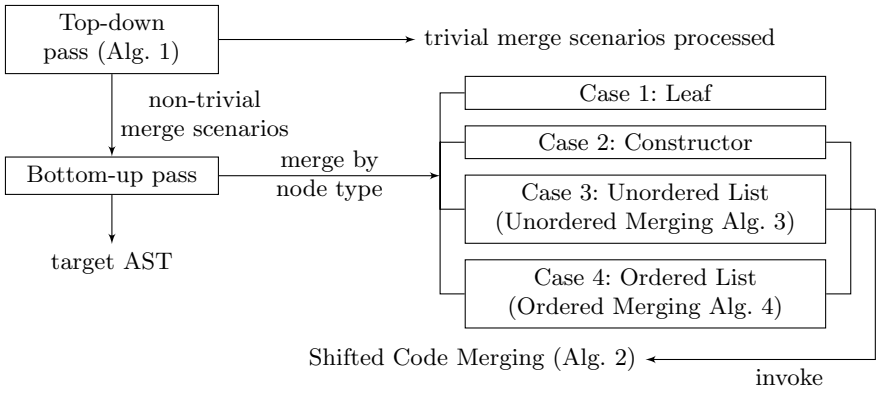

Fig. 2: High-level workflow of our merge algorithm.

adequate information to decide which one to take (row 2). The last two rules are applicable for only (ordered and unordered) list nodes. If an element of *base* presents in exactly one of the variants, then it is regarded as being removed and will be excluded from the target list (row 3). In contrast, if a new node is introduced in exactly one of the variants, it will be inserted into the target list (row 4). For ordered lists, conflicts may occur if the insertion position is ambiguous.

3 Merge Algorithm

Given a three-way merge scenario $(base, left, right)$, our merge algorithm accepts two matching sets (obtained by an AST matching algorithm) as input—$\mathcal{M}_L$ the matches between *base* and *left*, and $\mathcal{M}_R$ the matches between *base* and *right*. The algorithm generates a new tree, namely a *target AST*, as the merge result.

3.1 Algorithm Overview

Merging is performed on the matched nodes only, as unmatched nodes are assumed to have no relation. In the case of three-way merging, the two variants should match the base version. Formally, a merge scenario (b, l, r) is said *proper* if $(b, l) \in \mathcal{M}_L$ and $(b, r) \in \mathcal{M}_R$. The merge algorithm only needs to manipulate proper merge scenarios. Non-proper merge scenarios can be safely omitted because they are regarded as deletions and thus do not appear in the target.

Fig. 2 presents the high-level workflow of our merge algorithm. It consists of a *top-down* pass followed by a *bottom-up* one. In the top-down pass (see § 3.2 for details), input ASTs get traversed in pre-order, and any *trivial* merge scenario—any two of the three versions are equal—is processed immediately. Meanwhile, we collect other *non-trivial* proper merge scenarios in the list S.

In the bottom-up pass, the merge scenarios in S get processed in the reverse order, i.e., in post-order. Since matched nodes have the same label, the three

Algorithm 1: Top-down pruning pass

```
1 Function TopDownVisit(b: Node):
2     S ← [];
3     if ∃l ∈ left, r ∈ right : (b, l) ∈ M_L ∧ (b, r) ∈ M_R then
4         if b = r then R(b) ← l; return [];
5         if b = l or l = r then R(b) ← r; return [];
6         S += (b, l, r);
7     foreach child c of node b do
8         S ++= TopDownVisit(c);
9     return S;
```

versions in a proper merge scenario must be homogeneous—they must all be leaf nodes, constructor nodes, unordered list nodes, or ordered list nodes.

The first case, all nodes are leaf nodes, is the base case of our merge algorithm. This case is straightforward by three-way merge principles: We either take the only-changed variant as the target (by row 1 of Table 1) or report a conflict due to the inconsistent changes (by row 2).

The other cases are recursive cases where sub-scenarios need to be merged recursively. Merging constructor nodes of the same label and arity gives a constructor node of that label and arity too, and each child node is recursively merged from the sub-scenarios formed by the children at the corresponding index. Merging list nodes gives list nodes too, and it contains elements recursively merged from certain sub-scenarios drawn from the elements in the input lists (see § 3.4 and § 3.5 for details).

A challenging problem in solving the recursive cases is that: a sub-scenario (b, l, r) may *not be proper*, say b and l do not match. Even though it is rational to assume b matches *some descendant* of l (or else we simply report a conflict), which happens when l has shifted code. We encode this condition as a *relevant-to* relation: u is relevant to v, written $u \simeq v$, iff there exists a descendant $w \in v$ such that u matches w. With this notion, the assumption we make on a sub-scenario (b, l, r) is given by $b \simeq l \wedge b \simeq r$. Merging such a sub-scenario requires us to take shifted code into account. We will present this algorithm in § 3.3.

The merge result of the merge scenario (b, l, r) is recorded in a map $\mathcal{R}$ so that the algorithm can query it later on demand. Instead of using the entire merge scenario (b, l, r) as the index (or key) for the map $\mathcal{R}$, realizing that any node b of *base* appears in at most one merge scenario (by the injectivity of the matching sets), we simply use b as the index. In the end, the target AST of the top-most merge scenario $(\mathit{base}, \mathit{left}, \mathit{right})$ is obtained by querying $\mathcal{R}(\mathit{base})$.

3.2 Top-Down Pruning Pass

In the top-down pass, we visit *base* in a descendant recursive manner by invoking `TopDownVisit` (Alg. 1). This function returns all non-trivial merge scenarios that

Algorithm 2: Shifted Code Merging

```
1 Function IssueShifted(b: Node, l: Node, r: Node):
2     let l', r' be nodes s.t. (b, l') ∈ M_L, (b, r') ∈ M_R;
3     if l' = l ∧ r' = r then return R(b);
4     if l' ≠ l ∧ r' = r then return l[R(b)/l'];
5     if r' ≠ r ∧ l' = l then return r[R(b)/r'];
6     if l[R(b)/l'] = r[R(b)/r'] then return l[R(b)/l'];
7     return Conflict(l, r);
```

need processing later in the bottom-up pass. We use a list S to collect them. For short, we use two notations: $S \mathrel{+}= e$ for appending an element e to S, and $S \mathrel{++}= S'$ for appending all elements in S' to S. Upon traversing, if any trivial merge scenario is encountered, we immediately store the target AST in $\mathcal{R}$ and prune any further visit of its sub-scenarios by returning an empty list (lines 4 – 6). Otherwise, we proceed to collect merge scenarios recursively (lines 8 – 9).

3.3 Shifted Code Merging

Shifted code may exhibit in any type of node (except leaf node) in merge scenarios. Alg. 2 presents a unified algorithm for dealing with shifted code. It requires $b \simeq l \wedge b \simeq r$. Merging is performed according to where the shifted code involves:

(no shifting) If (b, l, r) is proper, then we simply query $\mathcal{R}$ (line 3).
(left-shifting) If b matches r but not l, then there exists a l' such that it is shifted into l. To integrate this shifting, we first make a copy of l and replace l' with the merge result of (b, l', r') i.e., $\mathcal{R}(b)$ (line 4). The notation $u[w/v]$ gives an updated tree by replacing a subtree v with w on u.
(right-shifting) Line 5 is symmetric to the above case.
(consistent-shifting) If both variants involve shifted code, the only circumstance we can safely merge is when they yield the same result (line 6).
(inconsistent-shifting) Otherwise, report a conflict (line 7).

Consider the example in Fig. 1. When merging the merge scenario consisting of the three `Assignments`, `CastExpr` from *left*, `InfixExpr` from *base* and `InfixExpr` from *right* form the arguments b, l, r in Alg. 2. The result is computed by taking the subtree of `CastExpr` and replacing its subtree of `InfixExpr` with the target one (by line 4). In merging the top-most merge scenario, the result is computed from the subtree of `ForStmt` by replacing its child `ExprStmt` with the target of `ExprStmts` (by line 5). In this way, the shifted changes made by the two variants are integrated.

3.4 Unordered Merging

Let B, L, and R respectively be the set of elements of three unordered list nodes that form a merge scenario. The goal of unordered merging is to compute a set

Algorithm 3: Unordered merge

```
1  Function Unordered(B: Set, L: Set, R: Set):
2      T ← ∅;
3      foreach b ∈ B do
4          if ∃l ∈ L, r ∈ R : b ≃ l ∧ b ≃ r then
5              T ← T ∪ {IssueShifted(b, l, r)};
6              mark l, r as "visited";
7          else if ∃l ∈ L : b ≃ l then // right case is symmetric
8              if b ≠ l then T ← T ∪ {Conflict(l, ε)};
9              mark l as "visited";
10     T ← T ∪ {e | e ∈ L ∪ R, e is not "visited"};
11     return T;
```

of elements T—without worrying about the order—that should appear in the target list. These elements are classified as follows:

(shifting) If an element $b \in B$ satisfies $b \simeq l \wedge b \simeq r$ for some $l \in L$ and $r \in R$, then the merge result is obtained by invoking `IssueShifted(`b, l, r`)`.
(left/right-insertion) If an element of L or R is not related to any element of B, then by row 4 of Table 1 it is an insertion.
(left/right-deletion-change conflict) If an element b satisfies, for example (dual case is similar), $b \simeq r$ for some $r \in R$, then it is a left-deletion (thus not included in T) when $b = r$; and a left-deletion-change conflict when $b \neq r$.

The above is realized as Alg. 3: First, traverse the elements in B and collect any shifting (line 4) or left-deletion-change conflict (line 7, right-deletion-change conflict is symmetric) in T. Meanwhile, mark every relevant left/right element as "visited" (lines 6 and 9). Then, all elements yet not marked must be left/right-insertions: thus insert them into T (line 10).

3.5 Ordered Merging

Merging ordered lists is more complex than merging unordered lists in that the elements of the target list must be in an order preserving the original occurrence order of associated elements in the merge scenario; and it is necessary to decide whether such an order uniquely exists—if not, to fit the merge algorithm into a conservative setting, conflicts shall be reported as well. For example, the conflicting scenario depicted in Fig. 3 is due to the ambiguity that whether `Stmt2` should precede `Stmt3` (note that both should be included in T as insertions).

Order-preserving Before presenting the merge algorithm, we first need a formal interpretation of "preserving the original occurrence order". Since the occurrence order is a partial order relation, it is natural to regard the three ordered lists in the merge scenario (B, L, R) as three ordered sets $\langle B, \lhd_B \rangle$, $\langle L, \lhd_L \rangle$ and $\langle R, \lhd_R \rangle$.

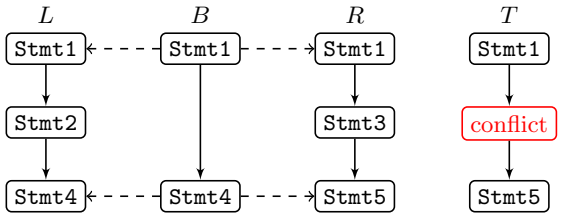

Fig. 3: A conflicting merge scenario of three statement-lists (B, L, R) with the target T. A dashed edge between two statements indicates they are matched.

The occurrence order relation is denoted by $\lhd_X$ (for $X \in \{B, L, R\}$), formally defined as $X[i] \lhd_X X[j] \iff i < j$.

Let S be the set of elements that should appear in the target list T, computed by `Unordered(`B`, `L`, `R`)`. We encode the relationship as partial functions $\pi_B : B \rightharpoonup T$, $\pi_L : L \rightharpoonup T$ and $\pi_R : R \rightharpoonup T$, associating an element in the input merge scenario with the corresponding element in the target list. For example, if $t^* \in S$ is a left-insertion, then t^* is a copy of some $l^* \in L$, thus we let $\pi_L(l^*) = t^*$.

Definition 2 (Order-preserving). *We say an ordered list T is an* order-preserving *w.r.t. (B, L, R) if T is a permutation of $S =$ `Unordered(`B`, `L`, `R`)` such that π_B, π_L, and π_R are monotone. A partial function $f : X \rightharpoonup Y$ is said* monotone *if for every $x_1, x_2 \in X$ such that $f(x_1)$ and $f(x_2)$ are both defined, $x_1 \lhd_X x_2$ entails $f(x_1) \lhd_Y f(x_2)$.*

In the above, the monotonicity condition precisely encodes our requirement of "preserving the original occurrence order".

Algorithm The main goal of the algorithm (Alg. 4) is to solve an order-preserving list and to decide the uniqueness of such lists. We compute an order-preserving list via constraint-solving—the constraints encode the monotonicity condition for the target list T by Def. 2. Technically each constraint has the form $e_1 \lhd e_2$, meaning "e_1 precedes e_2 in T". We propose an algorithm `GenConstraints` to produce them by traversing the elements of B, L, and R in their occurrence order, following the same structure of Alg. 3, e.g., we generate the constraint "$\pi_B(b_1) \lhd \pi_B(b_2)$" for $b_1 \lhd_B b_2$.

Let Φ be the set of computed constraints (line 2). We represent the constraints as a directed graph (line 3) $G_\Phi = \langle V, E \rangle$, where: (1) the set of vertices are the elements of `Unordered(`B`, `L`, `R`)`, i.e., $V = S$, and (2) for each constraint $(e_1 \lhd e_2) \in \Phi$, let $(e_1, e_2) \in E$ be an edge of G_Φ. It is well-known from graph theory that: there is a one-one correspondence between a topological sort of G_Φ and a satisfying solution of Φ, which further implies that: there is a one-one correspondence between an order-preserving list and a topological sort of G_Φ.

We compute an order-preserving list using classic topology sort algorithms such as Kahn's algorithm [13]. Following Kahn's algorithm, we facilitate a loop (lines 6 – 14) to compute a topological sort and save it to a list Y. The auxiliary

Algorithm 4: Ordered merge

```
1  Function Ordered(B: List, L: List, R: List):
2      Φ ← GenConstraints(B, L, R);
3      Represent Φ as a directed graph G_Φ = ⟨V, E⟩;
4      Y ← [];
5      Z ← {u | u ∈ V, indeg(u) = 0};
6      while Z ≠ ∅ do
7          if |Z| ≥ 2 then return "conflict";
8          remove the sole vertex u from Z;
9          Y += u;
10         foreach (u, v) ∈ E do
11             E ← E \ {(u, v)};
12             if indeg(v) = 0 then
13                 Z ← Z ∪ {v};
14         V ← V \ {u};
15     if V ≠ ∅ then return "cyclic";
16     return Y;
```

set Z (line 5) maintains all vertices with zero in-degree (i.e., without incoming edge). To further enable the uniqueness checking, we make the following extension: Each time the loop is entered, we check if Z has multiple elements. If so, choosing either vertex of Z gives a topological sort—in other words, the topological sort is not unique—so we report a conflict (line 7, highlighted). Otherwise, we follow the original Kahn's algorithm in lines 8 – 14. The loop repeats until Z is emptied. Suppose V is nonempty even when the loop exits, G_Φ must be cyclic and has no topological sort at all (based on the property of Kahn's algorithm), where a conflict exhibits as well.

The correctness of our ordered merge algorithm is stated as the following theorem:

Theorem 1. *Alg. 4 returns an order-preserving list (without conflict) if and only if the order-preserving list w.r.t. input merge scenario uniquely exists.*

The ordered merge algorithm is linear because both Kahn's algorithm and the generation of constraints are linear. Moreover, the other algorithms mentioned before are also linear, thus:

Theorem 2. *The time complexity of the entire structured merge algorithm is linear (to the size of the input merge scenario).*

4 Implementation

We implemented the proposed approach as a structured merge framework, Mastery, written in Java. This framework consists of four modules:

1. a parser that translates input source files into ASTs;
2. a tree matcher that generates mappings between different program versions, using an adapted GumTree [7] algorithm;
3. a tree merger that computes the target AST, following the algorithms presented in § 3;
4. a pretty printer that outputs the formatted code from the merged AST.

Mastery currently supports merging Java programs. We use JavaParser [7] to build ASTs from source code and pretty print source code from ASTs.

5 Evaluation

We extract 40,533 merge scenarios from 78 Java open-source projects hosted on GitHub, and then conduct a series of experimental evaluations to answer the following research questions:

RQ1: How often does shifted code occur in real-world merge scenarios?
RQ2: What is the merge accuracy of Mastery when compared to state-of-the-art merge tools?
RQ3: How many merge conflicts are reported by these tools?
RQ4: What is their performance from the perspective of runtime?

5.1 Experimental Setup

To select realistic and representative merge scenarios as our evaluation dataset, we seek the top-100 most popular open-source Java projects hosted on GitHub[8]; exclude any non-software-project (e.g., tutorials). On the remaining 78 projects, we extract merge scenarios via an analysis of their commit histories:

1. On all *merged commits*, we extract its two parents and their *base commit* from the Git history. The three source files with the same name extracted from the three commits each form a merge scenario (*base*, *left*, *right*), where *base* is from the base commit and the two variants are from the parent commits. The file (with the same name) in the merged commit is marked as the *expected* version, i.e., the *ground truth* of the merged result.
2. If any two of *base*, *left*, and *right* are equivalent, the target version of this merge scenario is obvious. To better examine the differences between the merge tools, we elide such trivial merge scenarios and instead only collect the merge scenarios where the three versions are pairwise distinct, judged by the `git diff` command.
3. Some source files cannot be correctly parsed, e.g., they include unresolved conflicts. We have to elide them too because structured approaches assume the input files to have valid syntax, checked by JavaParser.

[7] https://javaparser.org/
[8] According to the following list, until July 12, 2021: https://github.com/EvanLi/Github-Ranking/blob/master/Top100/Java.md.

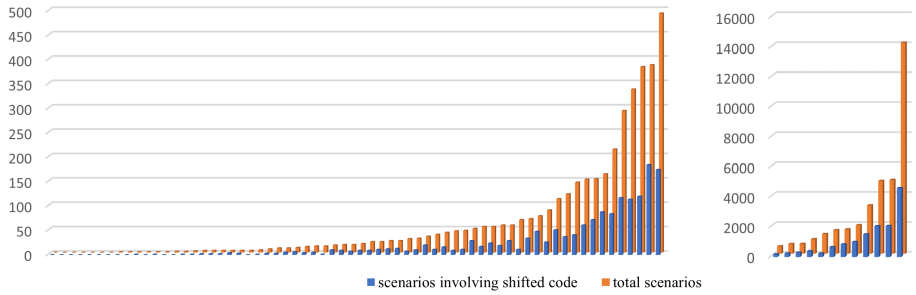

Fig. 4: Distribution of shifted code in each project. The projects are sorted by their number of merge scenarios in ascending order. We split them into two subfigures according to if the number of merge scenarios is less than 500 (left) or not (right).

4. In total, we collect 40,533 merge scenarios across 78 Java projects.

We compare Mastery with four state-of-the-art merge tools:

- JDime [16], a state-of-the-art structured merge tool,
- jFSTMerge [5], a well-known tree-based semistructured merge tool,
- IntelliMerge [20], a refactoring-aware graph-based semistructured merge tool,
- GitMerge, the default merging algorithm in Git.

All experiments were conducted on a workstation with AMD EPYC 7H12 64-Core CPU and 1TB memory, running Ubuntu 20.04.3 LTS.

5.2 Frequency of Shifted Code (RQ1)

To calculate the frequency of shifted code, we use the state-of-the-art AST differencing tool, GumTree [7], to compute the matching sets among *base*, *left*, and *right*. Note that we don't need any merge tool for this evaluation. We detect shifted mappings from these matching sets according to Def. 1. Fig. 4 presents how many merge scenarios in each studied project *involve* shifted code, meaning at least one shifted mapping is detected. Among the 40,533 merge scenarios, we find 15,620 merge scenarios involve shifted code—the frequency is 38.54%. In those merge scenarios, we detect 90,982 shifted mappings—on average 2.24 shifted mappings per merge scenario.

5.3 Taxonomy of Results

To understand the behavioral performance of the merge tools, we classify a merged result (for each merge scenario of each tool) into one of the following four categories:

Table 2: Distribution of the merged results.

Tool	Expected		Unexpected		Conflicting		Failed	
	Number	Accuracy	Number	Percentage	Number	Percentage	Number	Percentage
Mastery	**33342**	**82.26%**	3504	8.64%	3686	9.09%	1	0.00%
JDime	32789	80.89%	2446	6.03%	4610	11.37%	688	1.70%
jFSTMerge	30063	74.17%	3837	9.47%	6627	16.35%	6	0.01%
IntelliMerge	9774	24.11%	24555	60.58%	**3442**	**8.49%**	2762	6.81%
GitMerge	30643	75.60%	**791**	**1.95%**	9099	22.45%	**0**	**0.00%**

- *expected*: the merged file and the expected version (the ground truth) are *syntactically equivalent*, i.e., their ASTs are isomorphic to each other (allowing permutations of elements in an unordered list node);
- *unexpected*: the merged file is conflict-free and is nonequivalent to the expected version;
- *conflicting*: there is at least one conflict block in the merged file;
- *failed*: either the tool crashes or the execution exceeds the time limit 300 s.

Table 2 lists the merge results of the five tools. JDime and IntelliMerge failed on a considerable number of scenarios, mainly caused by their implementation bugs. IntelliMerge tends to produce more unexpected results because it adopts a radical merging algorithm that may violate the three-way merge principles when a deletion happens. For GitMerge, only 1.95% results are unexpected. To understand this phenomenon, we have to notice that all projects use GitMerge as default. If GitMerge does not report any conflict, the merged codes will usually become the *ground truth* in our evaluation, without being reviewed by the developers. Thus, the expected version is a kind of *biased* ground truth in favor of GitMerge. This finding is consistent with a previous work [20].

5.4 Merge Accuracy (RQ2)

The merge *accuracy* is calculated as the percentage of expected results. As shown in Table 2, Mastery achieves the highest accuracy of 82.26% among all tools. Comparing to JDime, Mastery gains 1.37% higher accuracy. Among the 1,398 scenarios where Mastery's results are expected whereas JDime's are not, we find 48.78% involves shifted code—10.25% higher than the overall frequency.

The scenarios where GitMerge produces unexpected or conflicting results are of special interest to us—in these merge scenarios, the expected versions (i.e., the merged versions in Git histories) must have been reviewed by the developers. If we consider only these 9,890 scenarios, the accuracy of the five tools except GitMerge are:

Mastery	JDime	jFSTMerge	IntelliMerge
32.17%	31.94%	17.26%	6.98%

Mastery still achieves the highest accuracy.

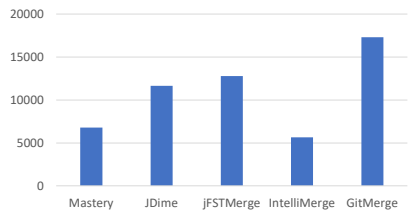

Fig. 5: Numbers of conflict blocks.

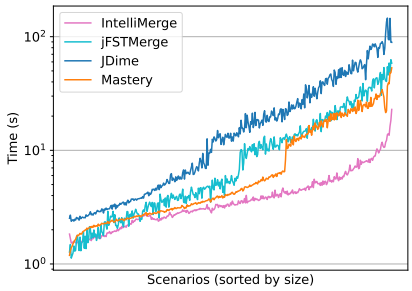

Fig. 6: Time cost of merging.

5.5 Reported Conflicts (RQ3)

In addition to the numbers of conflicting merge scenarios listed in Table 2, we also count the numbers of conflict blocks (or conflict hunks) as depicted in Fig. 5. Especially, IntelliMerge's radical strategy ignores three-way merge principles, making it achieve the lowest in both metrics. The other four tools all follow the three-way merge principles. Among them, Mastery reports the fewest 6,791 conflict blocks and the fewest 3,686 conflicting scenarios. Among the 1,650 scenarios where JDime's results are conflicting whereas Mastery's are not, we find 51.82% involves shifted code—higher than the overall frequency.

5.6 Runtime Performance (RQ4)

As unstructured GitMerge is inherent particularly efficient, we only compare the runtime performance among semistructured and structured tools. Ignoring the failed runs, Fig. 6 shows the runtime on merge scenarios sorted by the size (i.e., total file size in unit of byte) of merge scenarios in ascending order. Since the runtime of each tool has considerable ups and downs even on the merge scenarios of a similar size, for clearer illustrating, we plot each point as the average of adjacent 100 merge scenarios. The average times for the four tools are:

Mastery	JDime	jFSTMerge	IntelliMerge
10.33 s	24.06 s	13.21 s	4.34 s

Mastery is about 2.4 x as fast as JDime, and about 1.3 x as fast as jFSTMerge, which shows Mastery, as a structured merging tool, has competitive efficiency to semistructured merging tools.

5.7 Discussions

Threats to Validity We find the following threats to our ground truth:

1. The expected version may not exactly be the merged version but the one postponed by a few commits, a.k.a. supplementary commits. Our dataset

extraction process does not consider such supplementary commits. As a future direction, it is interesting to investigate how to obtain better ground truth taking the supplementary commits into account (such as [12,11]).
2. Because GitMerge is the default merging tool that developers use, if GitMerge reports no conflicts, its merging results will usually become the ground truth without any careful review, even if they are indeed wrong.
3. By empirical inspection, we found developers introduce additional changes in some merging scenarios, rather than only collaborating on the changes from *left* and *right*.

The ideal ground truth is to only merge changes in *left* and *right* correctly without introducing additional changes, and conflict blocks get reported if the changes are indeed semantically contradicted. Unfortunately, manual efforts seem inevitable approaching this ideality.

Limitations Among the 438 merge scenarios where Mastery produces unexpected results while JDime produces expected results, we manually studied 10 random samples. We found that: In 4 merge scenarios, the expected versions introduce additional changes by developers or break three-way merge principles in other ways. Mastery produces the desired merge results w.r.t. three-way merge principles. The other 6 scenarios failed due to our limited support for *two-way merging*, where a merge scenario consists of only the two variants but not the base version. JDime realizes some heuristic two-way merging strategies, which can handle these merge scenarios better. These strategies can be realized in Mastery in the future.

6 Related Work

Structured Merge Westfechtel [21] and Buffenbarger [3] pioneered in proposing merge algorithms that exploit structures of programs.

JDime [16] is a state-of-the-art tool for merging Java programs at AST level. In their AST representation, ordered and unordered lists are distinguished, and they propose distinct algorithms for merging them. We further distinguish ordered list nodes from constructor nodes (§ 2), as a list node can have an arbitrary number of children while a constructor node cannot. Their algorithm is in a top-down and level-wise manner, and is unable to merge shifted code.

Later, two extensions of JDime are proposed. One is an auto-tuning technique that switches between structured and unstructured merge algorithms for better efficiency [15]; the other is a syntax-aware looking ahead mechanism for identifying shifted code and renaming in the AST matcher [14]. To be scalable, the lookahead mechanism has restrictions on the types of nodes when lookahead is enabled (an if- or try-statement), and the maximum search distance of lookahead (3 or 4). Note that in their work, the lookahead mechanism is not applied to merging. Unlike them, our merge algorithm efficiently handles shifted code in a general sense (i.e., without the above restrictions).

Asenov et al. [2] propose an algorithm for matching and merging trees using their textual encoding, which enables the usage of standard line-based version control systems. To yield precise matching, external information, for example, unique identifiers across revisions, is required. Unfortunately, they are directly unavailable. Furthermore, they have to perform expensive tree matching algorithms.

Semistructured Merge Apel et al. [1] invented *semistructured merge*—a novel way of combining unstructured and structured approaches—that aims to balance the generality of unstructured merge and the precision of structured merge. Since semistructured approaches represent only part of the programs (typically high-level structures) as ASTs and keep the rest (low-level structures, e.g., method bodies) as plain text, they are not as precise as fully-structured approaches. An empirical study [4] on over 40,000 merge scenarios reveals that semistructured merge reports more false positives than structured merge.

Shen et al. [20] propose a graph-based refactoring-aware semistructured merging algorithm for Java programs, which is implemented as a tool IntelliMerge. The major difference between refactoring and shifted code is that refactoring must preserve semantics while shifted code usually does not.

Conflict Resolution Mens [17] thinks the resolution of conflicts caused by inconsistent changes made by variants is a major problem in version control. Since the resolutions of those conflicts are ambiguous, developers have the responsibility to resolve them manually. To alleviate manual efforts, Zhu and He [23] propose a synthesis-based technique that can automatically suggest candidate resolutions. In a real-time collaborative environment, it is also possible to simply prevent any presence of conflicts using locks [10,19,18,8].

7 Conclusion & Future Directions

We present Mastery, a three-way structured merge framework based on the methodology of combining the top-down and bottom-up visits of ASTs. This framework benefits from both the efficiency of handling trivial merge scenarios via a top-down pass and the effectiveness of handling non-trivial merge scenarios via a bottom-up pass, which makes it possible to handle shifted code elegantly. In the future, we plan to support more programming languages in our framework and further improve the tree matching and merging algorithms based on our evaluation findings.

Acknowledgement

This work was supported in part by the National Natural Science Foundation of China (No. 62072267 and No. 62021002) and the National Key Research and Development Program of China (No. 2018YFB1308601).

References

1. Apel, S., Liebig, J., Brandl, B., Lengauer, C., Kästner, C.: Semistructured merge: Rethinking merge in revision control systems. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 190–200. ESEC/FSE '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2025113.2025141, http://doi.acm.org/10.1145/2025113.2025141
2. Asenov, D., Guenat, B., Müller, P., Otth, M.: Precise version control of trees with line-based version control systems. In: Huisman, M., Rubin, J. (eds.) Fundamental Approaches to Software Engineering. pp. 152–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)
3. Buffenbarger, J.: Syntactic software merging. In: Estublier, J. (ed.) Software Configuration Management. pp. 153–172. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
4. Cavalcanti, G., Borba, P., Seibt, G., Apel, S.: The impact of structure on software merging: Semistructured versus structured merge. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1002–1013 (Nov 2019). https://doi.org/10.1109/ASE.2019.00097
5. Cavalcanti, G., Borba, P., Accioly, P.: Evaluating and improving semistructured merge. Proc. ACM Program. Lang. **1**(OOPSLA), 59:1–59:27 (Oct 2017). https://doi.org/10.1145/3133883, http://doi.acm.org/10.1145/3133883
6. Dotzler, G., Philippsen, M.: Move-optimized source code tree differencing. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 660–671. ASE 2016, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2970276.2970315, http://doi.acm.org/10.1145/2970276.2970315
7. Falleri, J.R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 313–324. ASE '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2642937.2642982, http://doi.acm.org/10.1145/2642937.2642982
8. Fan, H., Sun, C.: Dependency-based automatic locking for semantic conflict prevention in real-time collaborative programming. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing. p. 737–742. SAC '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2245276.2245417, https://doi.org/10.1145/2245276.2245417
9. Fluri, B., Wuersch, M., PInzger, M., Gall, H.: Change distilling:tree differencing for fine-grained source code change extraction. IEEE Transactions on Software Engineering **33**(11), 725–743 (Nov 2007). https://doi.org/10.1109/TSE.2007.70731
10. Ho, C.W., Raha, S., Gehringer, E., Williams, L.: Sangam: A distributed pair programming plug-in for eclipse. In: Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology EXchange. p. 73–77. eclipse '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1066129.1066144, https://doi.org/10.1145/1066129.1066144
11. Ji, T., Chen, L., Yi, X., Mao, X.: Understanding merge conflicts and resolutions in git rebases. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE). pp. 70–80 (2020). https://doi.org/10.1109/ISSRE5003.2020.00016
12. Ji, T., Pan, J., Chen, L., Mao, X.: Identifying supplementary bug-fix commits. In: 2018 IEEE 42nd Annual Computer Software and Applications Conference

(COMPSAC). vol. 01, pp. 184–193 (2018). https://doi.org/10.1109/COMPSAC.2018.00031
13. Kahn, A.B.: Topological sorting of large networks. Commun. ACM **5**(11), 558–562 (Nov 1962). https://doi.org/10.1145/368996.369025, https://doi.org/10.1145/368996.369025
14. Leßenich, O., Apel, S., Kästner, C., Seibt, G., Siegmund, J.: Renaming and shifted code in structured merging: Looking ahead for precision and performance. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 543–553 (Oct 2017). https://doi.org/10.1109/ASE.2017.8115665
15. Leßenich, O., Apel, S., Lengauer, C.: Balancing precision and performance in structured merge. Automated Software Engineering **22**(3), 367–397 (Sep 2015). https://doi.org/10.1007/s10515-014-0151-5, https://doi.org/10.1007/s10515-014-0151-5
16. Leßenich, O., Lengauer, C.: Adjustable Syntactic Merge of Java Programs. Master's thesis, Department of Informatics and Mathematics, University of Passau (2012)
17. Mens, T.: A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering **28**(5), 449–462 (May 2002). https://doi.org/10.1109/TSE.2002.1000449
18. Reeves, M., Zhu, J.: Moomba – a collaborative environment for supporting distributed extreme programming in global software development. In: Eckstein, J., Baumeister, H. (eds.) Extreme Programming and Agile Processes in Software Engineering. pp. 38–50. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
19. Salinger, S., Oezbek, C., Beecher, K., Schenk, J.: Saros: An eclipse plug-in for distributed party programming. In: Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering. p. 48–55. CHASE '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1833310.1833319, https://doi.org/10.1145/1833310.1833319
20. Shen, B., Zhang, W., Zhao, H., Liang, G., Jin, Z., Wang, Q.: Intellimerge: A refactoring-aware software merging technique. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). https://doi.org/10.1145/3360596, https://doi.org/10.1145/3360596
21. Westfechtel, B.: Structure-oriented merging of revisions of software documents. In: Proceedings of the 3rd International Workshop on Software Configuration Management. pp. 68–79. SCM '91, ACM, New York, NY, USA (1991). https://doi.org/10.1145/111062.111071, http://doi.acm.org/10.1145/111062.111071
22. Zhang, K., Jiang, T.: Some max snp-hard results concerning unordered labeled trees. Inf. Process. Lett. **49**(5), 249–254 (Mar 1994). https://doi.org/10.1016/0020-0190(94)90062-0, http://dx.doi.org/10.1016/0020-0190(94)90062-0
23. Zhu, F., He, F.: Conflict resolution for structured merge via version space algebra. Proc. ACM Program. Lang. **2**(OOPSLA) (Oct 2018). https://doi.org/10.1145/3276536, https://doi.org/10.1145/3276536
24. Zhu, F., He, F., Yu, Q.: Enhancing precision of structured merge by proper tree matching. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 286–287. IEEE (2019)