Renaming and Shifted Code in Structured Merging: Looking Ahead for Precision and Performance

Olaf Leßenich,* Sven Apel,* Christian Kästner,† Georg Seibt,* Janet Siegmund*
*University of Passau, Germany
†Carnegie Mellon University, USA

***Abstract*—Diffing and merging of source-code artifacts is an essential task when integrating changes in software versions. While state-of-the-art line-based merge tools (e.g., GIT MERGE) are fast and independent of the programming language used, they have only a low precision. Recently, it has been shown that the precision of merging can be substantially improved by using a language-aware, structured approach that works on abstract syntax trees. But, precise structured merging is $\mathcal{NP}$ hard, especially, when considering the notoriously difficult scenarios of *renamings* and *shifted code*. To address these scenarios without compromising scalability, we propose a syntax-aware, heuristic optimization for structured merging that employs a *lookahead mechanism* during tree matching. The key idea is that renamings and shifted code are not arbitrarily distributed, but their occurrence follows patterns, which we address with a syntax-specific lookahead. Our experiments with 48 real-world open-source projects ($4{,}878$ merge scenarios with over 400 million lines of code) demonstrate that we can significantly improve matching precision in 28 percent of cases while maintaining performance.**

I. Introduction

Due to the success of distributed version control systems, merging software artifacts has become a common task for software developers. However, the state-of-the-art technique used to perform merging in practice is basically still the same as thirty years ago [1]: unstructured, line-based merging of text files (e.g., using GIT MERGE). The success of line-based merging is based on the fact that it is easy to use, applicable to all kinds of text files, and cheap to compute. However, line-based merging also has several shortcomings: Since it completely ignores the (syntactic) structure of the artifacts that it merges, it is not able to either resolve or properly present conflicts to the developer conducting the merge.

It has been shown that merging at a structural level can improve the quality of the merged artifacts significantly [2, 3, 4, 5, 6, 7]. Instead of using text lines as a basic unit, a structural merge tool operates on the abstract syntax trees (ASTs) of the input artifacts. As in line-based merging, the key to merging artifacts is to identify matches and differences between these artifacts. The shortcoming of structured merging compared to line-based merging is that comparing ASTs is computationally expensive in general.

In previous work [5], we have proposed an approach to achieve a balance between precision and execution time of merging, using a simple, but effective mechanism: As long as no conflicts occur, a line-based merge strategy is used. When a conflict is encountered, one switches to structured merge and remerges the respective part, to increase precision. As a further performance optimization, when computing the matching between two ASTs, a top-down approach is used that considers matches only between corresponding tree levels. This optimization significantly reduces the runtime complexity from one that was only feasible for the smallest examples to one that is practical for real-world software projects [5].

While comparing ASTs only top-down and level-wise does significantly improve performance and scales to real-world projects, it lowers precision, in particular, when program elements are renamed (e.g., methods) and when code is shifted (e.g., shifting a code block into an exception handler). As renamings and shifted code are not uncommon changes in practice, this limitation is of practical importance. To catch renamings and shifted code, one could perform the matching only structurally (ignoring identifiers) and search also for embedded subtrees (not only level-wise), both of which are known to be $\mathcal{NP}$ hard and out of reach for a practical solution.

To gain precision without compromising performance, we propose a *syntax-aware, heuristic optimization* for structured merging that employs a *lookahead mechanism* during tree matching. The key idea is that renamings and shifted code are not arbitrarily used in software development, but their occurrence follows patterns. For example, only few syntactic program elements have names that are relevant for matching, and code is typically not shifted arbitrarily across the AST. While it is reasonable to test whether a block of statements has been shifted into an *if* or *try* block, it is highly unlikely (and even invalid) to shift a method declaration into a subexpression.

The main idea of our approach is employ a *syntax-aware lookahead* mechanism that diverges from level-wise tree matching depending on the kinds of syntax elements to be matched. For example, if level-wise matching is not able to match a block of statements on both sides, it searches for exception handlers and steps one level down (i.e., looks ahead) to find a match within the block. While this is more expensive than level-wise matching, it is still feasible, as we will demonstrate. Interestingly, both renamings and shifted code can be treated equally with this lookahead mechanism.

To demonstrate the practicality of our approach, we implemented the lookahead mechanism in our open-source structured merge tool JDIME[1], and we conducted a series of experiments on $4{,}878$ merge scenarios of 48 open-source projects, with over

[1]http://fosd.net/JDime/

400 million lines of code (aggregated over the merge scenarios). Overall, we found that we can significantly improve matching precision in 28 percent of the cases, without compromising performance (i.e., it scales to real-world projects of substantial size).

In summary, we make the following contributions:

- We present a heuristic matching technique based on a syntax-aware lookahead mechanism to precisely match ASTs in the presence of renamings and shifted code.
- We provide a practical implementation of our approach on top of our open-source tool JDIME.
- We evaluate our approach on a substantial set of real-world merge scenarios (48 projects, 4,878 merge scenarios, 400 million lines of code); we found that our approach significantly improves matching precision in 28 percent of cases while maintaining performance.

The implementation and a replication package are available on a supplementary Web site.[2]

II. Problem Statement

A. Structured Merge

Structured merging is an advanced merging strategy that operates at the level of ASTs. While a regular line-based merge tool is limited to the granularity of lines, a structured tool is aware of what kind of source code elements it is handling, and, therefore, is able to present better solutions in terms of quality, especially when it comes to conflict handling. A typical example where structured merging is superior to line-based merging is reordering of methods. The matching result produced by a line-based tool is limited by its capability of comparing just lines, which in most cases fails to match the reordered methods. A structured merge tool knows what methods are and that, for example, in JAVA, the order of their definition does not affect program semantics, hence it is able to match method declarations independently of their order in the source code.

The improved matching precision of structured merging comes at a cost, though: Beside technical overhead, such as parsing and pretty printing, matching of trees is per se more expensive as matching of sequences of text lines. In general, there are different algorithms for tree matching and merging with different degrees of precision and computational complexity. To maximize precision, one would need to handle cases where subtrees are shifted and nodes are renamed, which corresponds to solving the Tree Amalgamation Problem [8] and the Maximum Common Embedded Subtree Problem [9], both of which are known to be $\mathcal{NP}$ hard.

To reduce computational complexity to a practical degree, a top-down matching approach, which compares corresponding tree levels only (corresponds to the Maximum Common Subtree Problem [10]), is feasible (see Section VIII). In Figure 1, we illustrate an example: In the base version, *Method A* has three statements, *Stmt 1*, *Stmt 2*, and *Stmt 3*. While version *A* does not change *Method A*, version *B* shifts the statements

[2]https://www.infosun.fim.uni-passau.de/se/papers/lookahead/

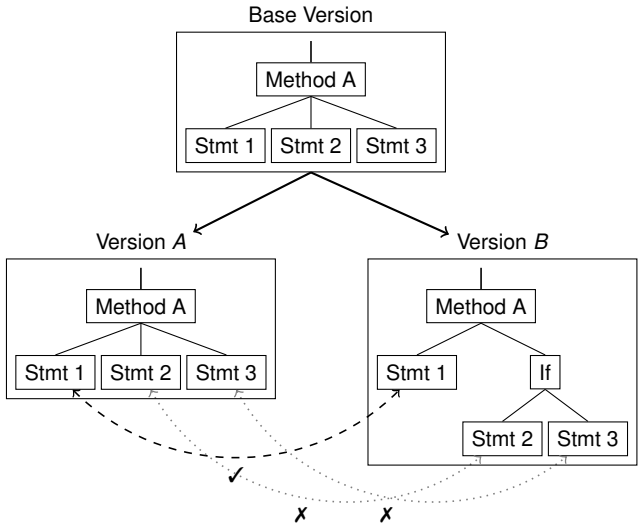

Figure 1. Top-down, level-wise AST matching (*Stmt 1* can be matched (✓); *Stmt 2* and *Stmt 3* can not be matched (✗) as they are not at the same level)

Stmt 2 and *Stmt 3* into an *If* block. As a result, *Stmt 1* can be matched across the versions *A* and *B*, but both *Stmt 2* and *Stmt 3* cannot, as they are at different levels in the two versions. Furthermore, the matching procedure terminates recursion early when an element could not be matched. These optimizations lead to quadratic (for ordered elements) [10] or cubic (for unordered elements) [11] runtimes, which is still practicable for merging real-world projects and has proved to provide practically relevant results [5]. There are, however, situations where structured merging is less precise than a line-based approach. In previous work [5], we found that these less precise situations are a direct result of the two optimizations (level-wise matching, early return) and often related to two common change scenarios, which are also confirmed by our empirical data (see Section IV): *renaming* and *shifted code*.

B. Renaming

Almost any programming language contains syntax elements that are uniquely identified by their names. This fact is exploited during structured merging, but poses problems when program elements are renamed. As an example of a renaming, we present a scenario of merging two versions of a JAVA stack implementation in Figure 2. The methods *size* and *length* have the same implementation in Version *A* and *B*, and differ only in their names. The bottom-left listing shows the result of a structured merge with JDIME. JDIME stops matching when the method declarations themselves cannot be matched, instead of looking further for similarities inside the method bodies. As a result, JDIME does not recognize the renaming and, instead, inserts both versions while merging—clearly, not the solution that is wanted in this scenario. Instead, we would expect a conflict that prompts the developer to make a decision, as illustrated in the bottom-right listing. This way, the developer can select the correct version.

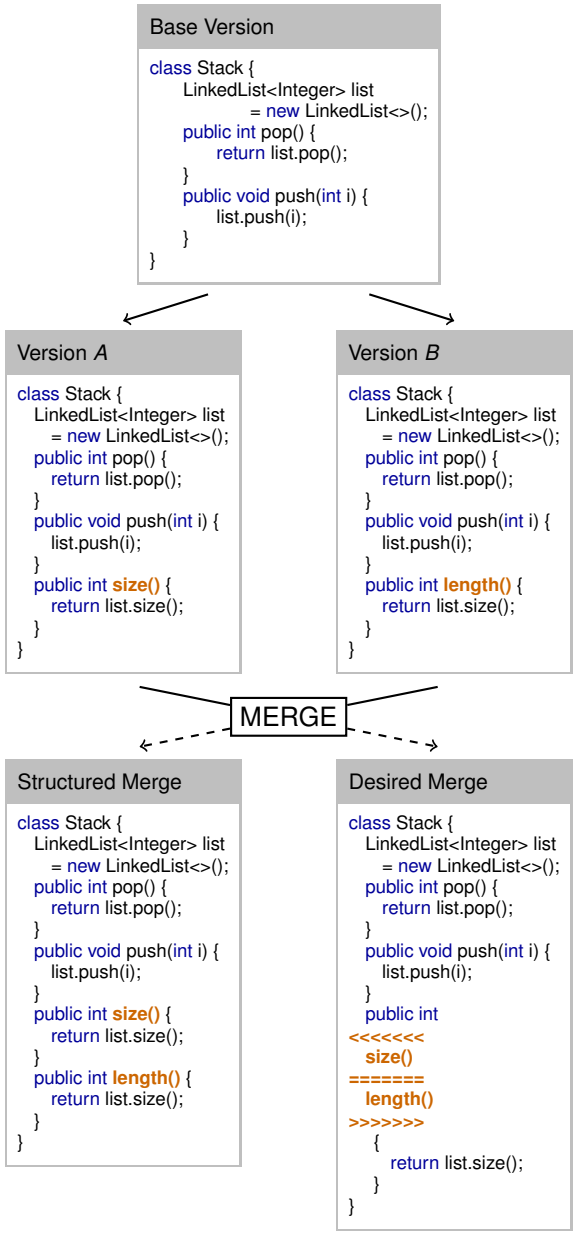

Figure 2. Method renaming example: *Stack.java*

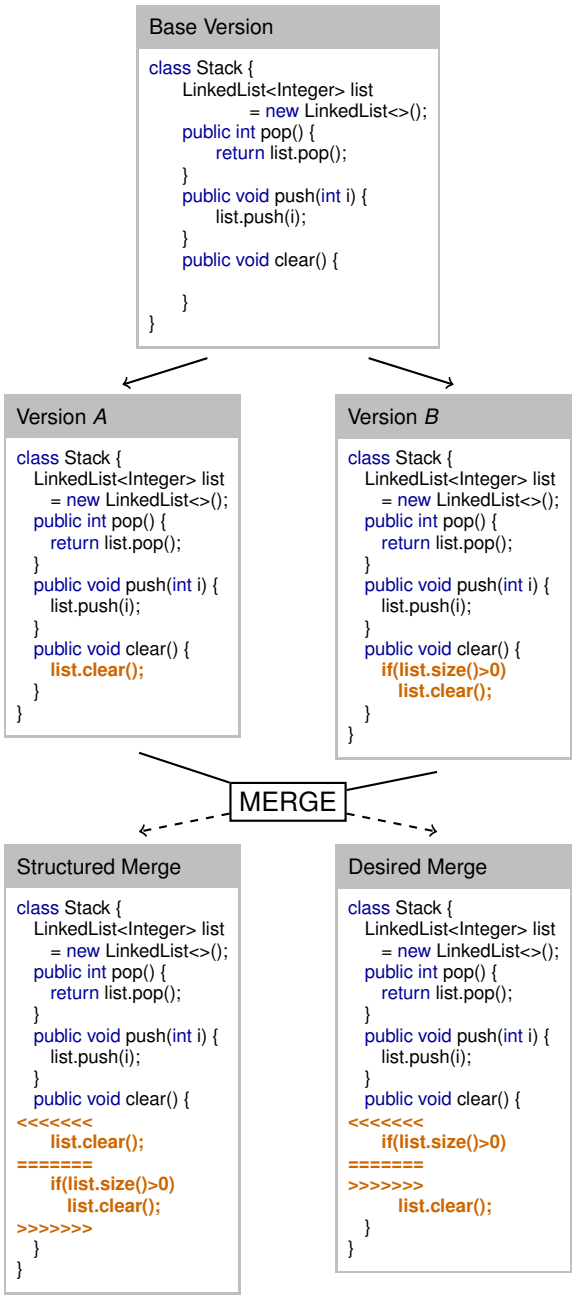

Figure 3. Shifted code example: *Stack.java*

C. Shifted Code

Program elements without unique names are structurally matched based on their code (i.e., equality of AST nodes). If a program element is moved in one version, but not the other, they cannot be matched easily; the matching algorithm needs to search for corresponding pairs. As an example, we show in Figure 3 a new piece of code that was introduced by both versions, but wrapped by an *if* statement only in one version. In version *A*, the surrounding *if* is missing; in version *B*, it is present. Performing a structured merge with JDIME results in an unnecessary large conflict (bottom-left listing), because the tool does not match the statement *list.clear* in the two versions. This is due to the optimization that matching is limited to corresponding tree levels only. The *if* statement inserts an additional level in the tree, which causes JDIME to miss the match between the list calls. The desired output includes a smaller conflict (i.e., fewer lines involved in the conflict), as illustrated in the bottom-right listing.

D. Discussion

To produce the desired merge result for both examples, we must improve the matching algorithm of the structured merge approach. One option is to resort to established algorithms from graph theory. In particular, one could perform the matching

only structurally (ignoring identifiers, which corresponds to the Tree Amalgamation Problem [8]) and search also for embedded subtrees (not only level-wise, which corresponds to the Maximum Common Embedded Subtree Problem [9]) However, due to the sheer size of ASTs of real-world programs, this is not feasible in practice (e.g., for unordered trees, the Maximum Common Embedded Subtree Problem is $\mathcal{NP}$ hard.[3])

Our key insight is that identifying arbitrarily renamed program elements and shifted code is not necessary in practice. The reason is that renaming and shifted code are not arbitrarily used in software development, but their occurrence follows patterns. For example, only few syntactic program elements have names that are relevant for matching, and even less have substantial nested code that would lead to imprecise conflicts as in our example of Figure 2. Thus, we concentrate on renamed methods in our experiments (see Section IV). Furthermore, code is typically not shifted arbitrarily across the AST. While it is reasonable to test whether a block of statements has been shifted into an *if* or *try* block, it is highly unlikely (and even invalid) to shift a method declaration into a subexpression or field initialization. Instead, we aim for a syntax-aware heuristic that, for certain (language-specific) elements in the AST, allows the matching algorithm to descend a few levels in the subtree and *look ahead* for a better match. Specifically, we propose such a heuristic, implemented on top of JDIME, using a configurable lookahead mechanism that produces significantly better matches while still achieving a performance comparable to state-of-the-art structured merging.

III. A Syntax-Aware Lookahead Mechanism

In Algorithm 1, we show the existing top-down matching approach used in JDIME, which matches only corresponding tree levels and stops descending when two nodes do not match. The algorithm starts matching at the root nodes of both ASTs and, if the nodes match, inspects their children to determine whether ordered or unordered matching is required (Line 7). Ordered matches between nodes are computed with an adaptation of the Longest Common Subsequence Algorithm [10], unordered matches are computed using the Hungarian method [11]. Both subroutines for ordered and unordered matchings include recursive calls to the match function shown in Algorithm 1. For more details on the ordered and unordered matching in JDIME, we refer the reader elsewhere [5].

One of the limitations that we discussed previously arises if two nodes cannot be matched (even if one could be matched with a child of the other; Line 3). As discussed in Section II, we need to change this behavior in certain situations to achieve better overall matching results.

If a program element cannot be matched (by name or structure), we can still look ahead (i.e., further descend in the ASTs) and search for potential matches. So, we have to alter the algorithm to continue the descent when specific AST nodes (e.g., two method declarations) could not be matched.

[3]More precisely, the problem is $\mathcal{APX}$ hard [9]; $\mathcal{APX}$ is the set of $\mathcal{NP}$ optimization problems that can be approximated with polynomial-time algorithms with an approximation ratio bounded by a constant.

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

```
1:  function MATCH(Node L, Node R)
2:    if L ≠ R then
3:      return 0                          ▷ Nodes do not match, early return
4:    end if
5:    l ← children of L
6:    r ← children of R
7:    if ISORDERED(l) ∨ ISORDERED(r) then
8:      return ORDEREDMATCHING(L, R)      ▷ Considering order
9:    else
10:     return UNORDEREDMATCHING(L, R)    ▷ Ignoring order
11:   end if
12: end function
```

A. Looking Ahead

To prevent a combinatorial explosion of potential matchings, and therefore, exponential runtime complexity, we add a parameter, called "lookahead distance", to our matching algorithm that tracks the number of levels it has descended in each tree without matching a node. As the influence of this parameter on runtime is quite high, we have to explore the structure of our ASTs to infer a reasonable configuration. For most syntax elements, a maximum value of 3 or 4 should be sufficient to match the relevant code blocks in most situations, this way, identifying renamed program elements and shifted code.

While it is technically possible to use a generic lookahead distance (i.e., independent of the type of program element to match), it is highly inefficient in practice, as it leads to numerous matching calls that yield no better result. For example, it is highly unlikely to identify a matching of method declarations by descending into sub-statement level. Instead, we exploit structural knowledge of typical changes: Because the subtree structure of, say, method declarations and *if* statements differs, this parameter has to be set specifically for each AST node type that is subject of our lookahead procedure to find an optimal balance between matching precision and algorithm performance.

In Figure 5, we illustrate the lookahead procedure for a renaming scenario. In version *A*, *Method A* has been renamed to *Method B*, thus, it cannot be matched with its counterpart in version *B*. Looking ahead three levels on both sides, they can be matched, as their bodies are identical (both containing *Stmt 1* and *Stmt 2*). Note that this example illustrates a further issue: Real-world parse trees and ASTs typically contain various auxiliary nodes (*Body* and *StmtList*, in our case). This needs to be taken into account when choosing the distance of the lookahead.

In Figure 4, we illustrate the lookahead procedure for an example with shifted code. In the base version, *Method A* contains the statements *Stmt 1* and *Stmt 2*.[4] In version *A*, *Stmt 2* is shifted into an *if* block (one level down); in version *B*, *Stmt 1* is shifted into an *try* block, which itself is wrapped by an *if*

[4]For simplicity, we do not show auxiliary nodes, such as *Body* and *StmtList*.

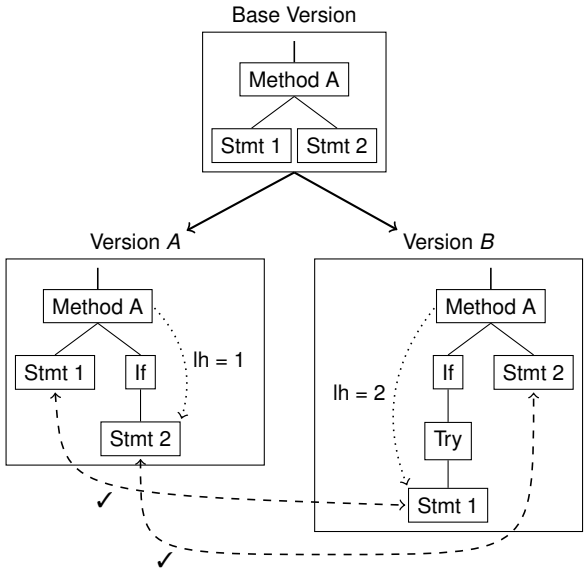

Figure 4. Looking ahead to match shifted statements

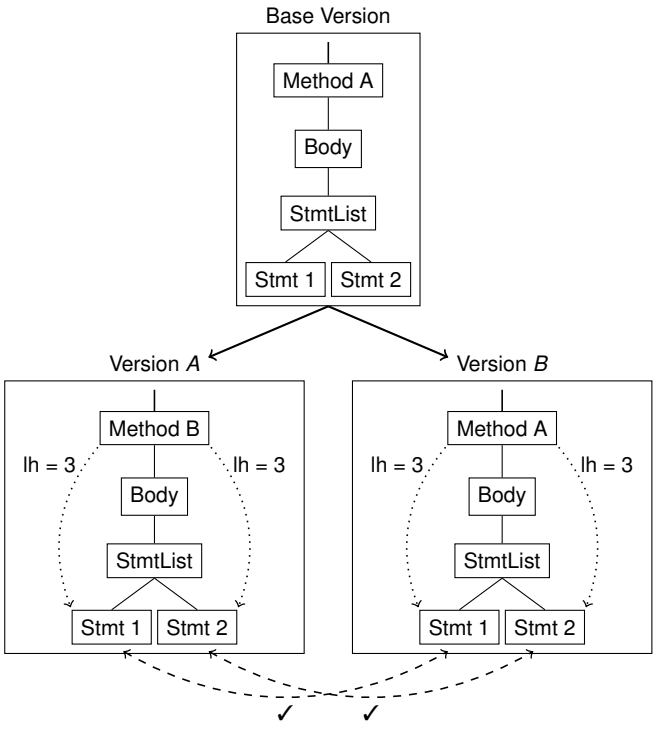

Figure 5. Looking ahead to match a renamed method

block (two levels down). While for matching *Stmt 1*, we need to look ahead two levels, for *Stmt 2*, we need to look ahead only one level.

B. Algorithm

Algorithm 2 shows the whole procedure of AST matching with lookahead. If two nodes do not match and the algorithm is allowed to look ahead (Lines 8 and 9)—instead of an early return (Line 6)—we still compute the matches between the child nodes. First, the algorithm descends the left AST (Line 8) and then tries the right AST (Line 9). The distance of looking ahead depends on the type of the nodes involved (Lines 3 and 4). The lookahead mechanism itself, shown in Algorithm 3, is a straightforward recursive descent in the AST, controlled by the given lookahead distance (Lines 5 and 9). To guarantee a certain matching quality, the resulting matches are used only if the similarity between the subtrees is, at least, the value of a given threshold (Algorithm 2, Lines 24–28). Note that the threshold is not always 1 (i.e., perfect match) as in the case of a method renaming, at least the names of the methods differ. So, the threshold depends on the type of the node to take that aspect into account.

In particular, we are interested in cases where the nodes themselves do not match, but their entire subtrees (or the most relevant parts, excluding some auxiliary nodes). In these cases, we optimize the lookahead by incorporating the *identical subtree optimization* proposed by Dotzler et. al [7] (not shown in Algorithm 2). In a nutshell, prior to matching, we create a hash for each node, using its type, label (if available), and children hashes as input for the hash function. Looking back at our method renaming example, we can determine whether their subtrees are completely identical by comparing the hash values. This optimization significantly reduced the number of matching operations for the renaming example, because the identity of the method bodies can be detected in $\mathcal{O}(1)$ by looking at the hash, instead of $\mathcal{O}(n^2)$ using an ordered matching algorithm.

Algorithm 2 AST MATCHING WITH LOOKAHEAD

```
1:  function MATCH(Node L, Node R, double threshold)
2:      if L ≠ R then
3:          l_dist ← LOOKAHEADDIST(L)        ▷ depends on the node's type
4:          r_dist ← LOOKAHEADDIST(R)        ▷ depends on the node's type
5:          if l_dist = 0 ∧ r_dist = 0 then
6:              return 0                     ▷ Nodes do not match, early return
7:          else                             ▷ Perform lookahead
8:              L ← LOOKAHEAD(L, R, l_dist)  ▷ Descending left
9:              R ← LOOKAHEAD(R, L, r_dist)  ▷ Descending right
10:             if L = NULL ∨ R = NULL then
11:                 return 0                 ▷ No matching nodes deeper in subtree
12:             else
13:                 return MATCH(L, R)       ▷ Recursive call
14:             end if
15:         end if
16:     end if
17:     l ← children of L
18:     r ← children of R
19:     if ISORDERED(l) ∨ ISORDERED(r) then
20:         matching ← ORDEREDMATCHING(L, R)
21:     else
22:         matching ← UNORDEREDMATCHING(L, R)
23:     end if
24:     if matching ≥ threshold then         ▷ Maximum depth reached?
25:         return matching
26:     else
27:         return 0
28:     end if
29: end function
```

Algorithm 3 LOOKAHEAD

```
1: function LOOKAHEAD(Node N, Node M, int dist)
2:   if N = M then
3:     return N                                  ▷ Match found
4:   end if
5:   if dist < 0 then                            ▷ Maximum depth reached?
6:     return NULL
7:   end if
8:   for c in children of N do
9:     n ← LOOKAHEAD(c, M, dist − 1)             ▷ Recursive call
10:    if n ≠ NULL then                          ▷ Further children?
11:      return n
12:    end if
13:  end for
14:  return NULL
15: end function
```

IV. EVALUATION

To evaluate our approach with respect to relevance, precision, and performance, we conducted an empirical study on 4,878 merge scenarios from 48 projects. In this section, we describe the setup and the results of our experiments. We provide a replication package on the supplementary Web site.

A. Research Questions

To learn about the precision and performance of matching with lookahead, we address six research questions. In particular, we concentrate on renamed methods and statements shifted into *if*, *try*, and *catch* blocks; as for the latter, we limit our attention to shifts across one level. To understand whether these types of changes are indeed relevant in practice (i.e., whether they occur frequently), we determine how often they occur in real-world merge scenarios, as determined by our lookahead mechanism.

RQ1.1 *How often do renamings occur in real-world merge scenarios?*

RQ1.2 *How often does shifted code occur in real-world merge scenarios?*

Apart from the frequency of occurrences, we also evaluate whether we can handle these situations better with the lookahead mechanism. Therefore, we measure precision in terms of the percentage of matched nodes, both with and without lookahead enabled. We expect a significantly higher precision with the lookahead mechanism.

RQ2.1 *What is the precision of matching in the presence of renaming with and without lookahead?*

RQ2.2 *What is the precision of matching in the presence of shifted code with and without lookahead?*

In practice, the time a merging algorithm needs to perform its task is relevant to the user. To evaluate whether the lookahead mechanism is applicable in practice, we determine what the cost of the additional matching operations (induced by looking ahead) is. To this end, we measure the runtime of the algorithm with and without lookahead.

RQ3.1 *What is the runtime cost of matching in the presence of renaming with and without lookahead?*

RQ3.2 *What is the runtime cost of matching in the presence of shifted code with and without lookahead?*

B. Subject Systems and Merge Scenarios

We selected 48 popular GIT projects from various application domains from the collaboration platform GITHUB, including popular projects, such as RXJAVA, ELASTICSEARCH, GUAVA, APACHE STORM, and APACHE WEEX. Our main criteria for selecting subject projects were that they are written (mainly) in JAVA and have a substantial number of merge scenarios in their history. We identified the merge scenarios by iterating over all commits in the version control history and by extracting those with more than one parent. Then, we performed a two-way diff between the competing versions of the merge. Of course, for an actual merge, a three-way comparison would be done, using the additional information from the common ancestor to decide which changes to select. However, as our goal is to evaluate the matching precision of the diff algorithms, it is sufficient to use two-way comparisons. A list of all subject systems along with relevant information is available in Table I.

C. Procedure

In our experiments, we compute a structured diff for each merge scenario with and without our lookahead mechanism. Then, we compare the achieved matchings as well as the time needed to perform the diff. To this end, we let the GIT client use our extension of JDIME as an external diff tool. Based on the output of JDIME, we measure whether we matched fewer or more AST nodes, overall and aggregated per type of node. Furthermore, we manually inspected about 200 scenarios to estimate the quality of matchings produced by our tool (see Section VII). This step is necessary to avoid that our approach simply matches more AST elements, while producing "wrong" results that would be harmful as input for the actual merge.

Our analysis framework is written in PYTHON; it is available at the supplementary Web site. We executed all experiments on Intel Xeon E7-4870 machines with 128 GB of memory.

V. RESULTS

A. RQ1.1 and RQ1.2 (Frequency)

In our experiments, we found that renaming of methods occurs in 25.44 % of the considered merge scenarios. On average, these renamings were spread over 2.06 files. Code shifted into *if*, *try*, and *catch* blocks occurred in 18.02 % of all merge scenarios, in most cases located in just one file.

Figure 6 and Figure 7 show the distributions of numbers of files affected by renaming and shifted code. While, in both cases, most occurrences concern only a single or few files, there are outliers, which point to major refactorings.

B. RQ2.1 and RQ2.2 (Precision)

By using our lookahead mechanism, we matched 141 additional AST nodes per affected scenario, on average (123 nodes by detecting renamings and 18 nodes by detecting shifted code), increasing our matching precision. Overall, matching

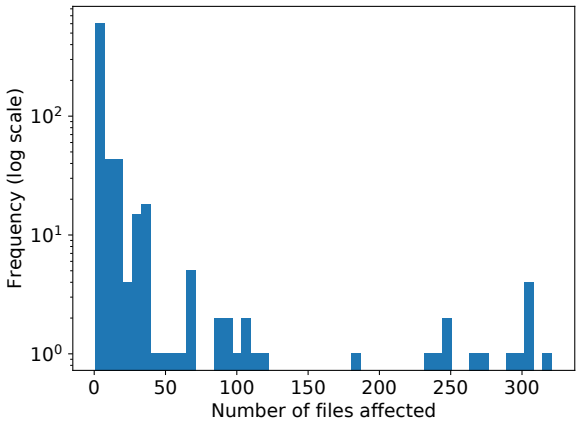

Figure 6. Files per scenario that contain renamings

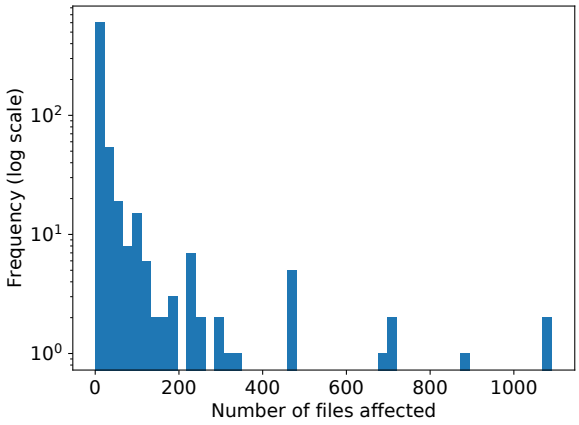

Figure 7. Files per scenario that contain shifted code

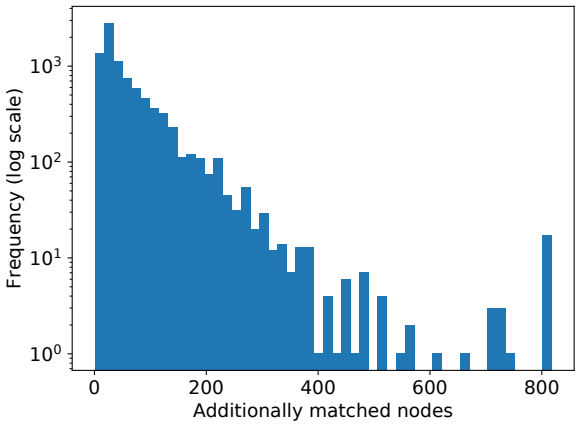

Figure 8. Number of additionally matched AST nodes by renaming detection

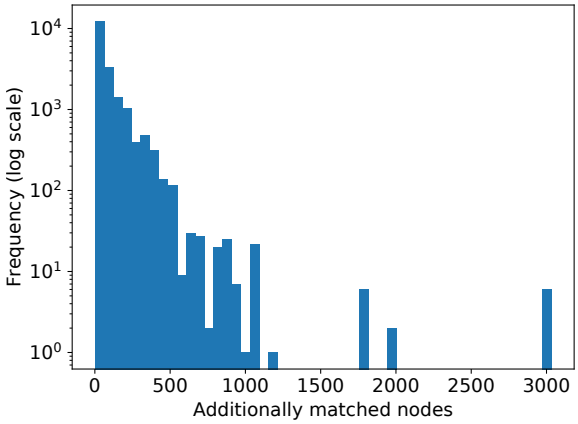

Figure 9. Number of additionally matched AST nodes by shifted code detection

precision could be improved in 28 % of the evaluated merge scenarios.

Figure 8 and Figure 9 show the distributions of numbers of nodes additionally matched by looking ahead for renamed methods and shifted code. In many cases, the number is 150 and below, but there are cases with up to several hundreds and thousands of nodes. As the number of nodes is a proxy of the amount of code, these results show that code of substantial size could be matched, which would be missed without lookahead.

C. $RQ_{3.1}$ and $RQ_{3.2}$ (Costs)

JDIME without lookahead had a total runtime of 31 min, 383 ms per scenario, on average. By enabling our lookahead mechanism, we spent a total runtime of 41 min, 505 ms per scenario, on average. Figure 10 and Figure 11 show the runtime per scenario with and without lookahead.[5] Overall, matching with lookahead is—as expected—slower than without, but still reasonably close. On average, looking ahead is less than half a second slower per scenario.

[5]Note that matching with lookahead is slightly faster in some cases in the figures, which is just an artifact of measurement inaccuracy, though.

VI. Threats to Validity

A. Internal Validity

For our experiments, we selected only merge scenarios actually committed to the GIT repositories, this way, ignoring scenarios where a merge was possibly too hard and the developers gave up instead of integrating the changes. We deliberately made this decision to increase confidence in the practical relevance of our merge scenarios, especially, as there is no reliable way to identify and extract merge scenarios that failed. Likewise, there is also no reliable way of identifying rebases, so these are also excluded.

In our experiments, we considered only on a few instances of renaming and shifted code (method renaming, shifting blocks of statements into *if*, *try*, and *catch* blocks). The mechanism we propose here is much more general, though, so our results mark just a lower bound of its applicability. Still, we were able to improve matching precision in a practical relevant number of situations.

We adjusted the lookahead distance parameter based on the structure of JDIME ASTs that we had at our disposal. Different AST implementations may require (slightly) different parameter settings, but a deviation from our high-level results are not to be expected. We set the threshold for matching

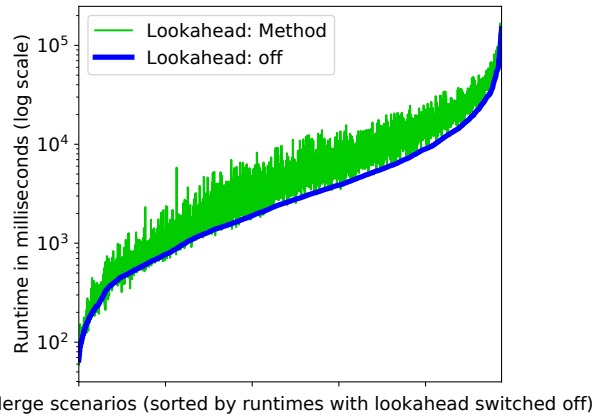

Figure 10. Runtimes in milliseconds with a lookahead for renamed methods switched on and off

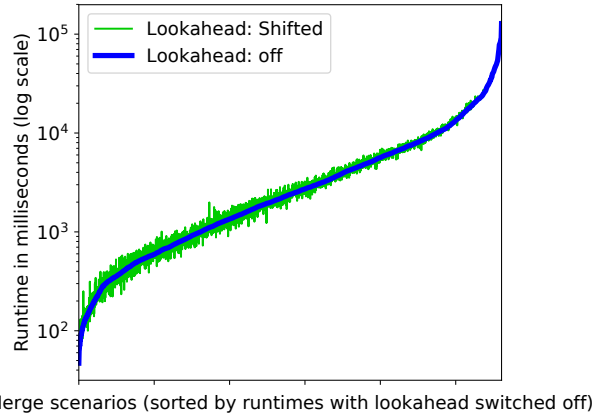

Figure 11. Runtimes in milliseconds with a lookahead for shifted code switched on and off

similarity such that only "perfect matches" are obtained (e.g., equal method bodies and blocks of shifted code). Relaxing this requirement would produce more desired matches, but possibly also undesired matches.

Finally, in our research questions, we concentrated on the matching routine, which is only one part of the overall merging procedure. Performing the actual merge based on the improved matching would add a further dimension of possible design decision and optimizations, which is well beyond the scope of this paper.

B. External Validity

One has to be careful with generalizing our findings beyond our corpus. We selected a substantial and diverse corpus of projects, covering different development practices and application domains, but other projects may have different characteristics. Likewise, our implementation and experiments are limited to JAVA. While our findings do not necessarily translate to any language, our results likely also apply to languages with similar syntactic structure, especially, with regard to method or function declarations and nesting of code blocks (e.g., C++, C#, and PYTHON).

VII. DISCUSSION

Summarizing the results of our experiments, looking ahead pays off! First of all, renaming methods and shifting code are changes that happen in practice (in 25.44 % of the merge scenarios, methods are renamed; in 18.02 % of the merge scenarios, code is shifted). This finding illustrates the relevance of targeting these changes in our lookahead mechanism. Furthermore, pursuing a syntax-aware lookahead mechanism improves precision considerably. Overall, matching precision could be improved in 28 % of the considered merge scenarios. Finally, the improvement in precision does not affect the practicality of the approach in terms of performance. On average, enabling the lookahead mechanism is less than half a second slower per merge scenario (up to 23 s).

Matching is an automatic procedure and must be automatic to be practical. To gain confidence in the correctness or "quality" of the computed matchings, we manually inspected about 200 samples of code that were matched differently by our lookahead, compared to without using the lookahead. For the manual inspection, we compared both pretty-printed diff outputs from JDIME with the output computed by a line-based diff tool, to judge whether our matching approach produced a correct result. During this inspection, we did not find any indication that our approach produces false positives. The reason for this is very likely the threshold parameter that is used by the algorithm to decide whether or not two trees are similar: In our evaluation, this value was set to 90 %, which leads to matches of mostly identical code fragments. When reduced, say to, 50 %, we assume that false positives will be produced along with correct matches, but this is outside the scope of our study.

The other tunable parameters are the language-specific lookahead distances: They need to be high enough to find matching code, but at the same time, introduce a performance penalty, due to the additional matching operations. Good values can be deduced from the structure of the AST: To find shifted code, a distance of 1 or 2 is sufficient. For renaming, a value of 4 or 5 is required to find better matches.

To learn more about the influence of the syntax-specific lookahead distances, we experimented with a generic, unbounded lookahead variant that always tries to find matching code in subtrees, irrespective of the type of node. As expected, this variant was much slower, with average runtimes of up to 200 % compared to the unoptimized version of JDIME.

Most of the shifted code that we looked at during our manual inspection, appeared like bug fixes of some sort: statements wrapped by a *try* block for better exception handling, or code wrapped by an *if* statement to execute it conditionally. We also observed the opposite situations: *try* blocks or *if* statements being removed. Typically, only one file was affected by shifting code per merge scenario, whereas renaming seems to be more common in practice.

VIII. RELATED WORK

After the seminal work of Westfechtel [2] and Buffenbarger [3], several of approaches of structured merging for

mainstream programming languages such as JAVA have been proposed [4, 6, 5, 12].

JDIME is a structured merge tool that merges JAVA ASTs top-down and level-wise [5]. It distinguishes between ordered and unordered nodes and treats them differently during matching, which improves precision. Leßenich et al. [5] proposed an auto-tuning approach on top of JDIME, which essentially switches between structured and unstructured merge based on the presence of conflicts (if there are conflicts, a more precise, but also more costly, structured merge is used; otherwise, only a line-based merge). Without lookahead, JDIME is not able to match renamed program elements and shifted code.

Apel et al. [13] addressed problem of computational complexity of structured merging by treating individual artifacts only to some extent in terms of their syntactic structure (e.g., until the method level) and the fine-grained parts as plain text, which is effectively a *semistructured merging* approach. Much like JDIME, renamed program elements and shifted code cannot be matched.

Asenov et al. [12] encode the tree structure of ASTs in plain text, such that a line-based merging tool can be used. The matching algorithm relies on node identifiers, though, which must be supplied from an external source (e.g., from an editor), which simplifies the matching problem. However, in many practical settings, such as using version control systems, this information is not available.

GUMTREE uses a two-phase strategy for AST matching [6]. In the first phase, it searches top-down for nodes whose names match and whose subtrees are isomorphic. In the second phase, it revisits unmatched nodes and searches for pairs which have a significant number of matching descendants. This way, renamed program elements and shifted code can be detected. In contrast to our lookahead mechanism, their approach is not syntax-specific, thus, searches for potential matches irrespective of syntactic categories of the program elements involved. As discussed in Section VII, this incurs an overhead for searching matches that are very unlikely or even invalid.

Dotzler et al. [7] discuss five optimizations that may be used as pre- or postprocessing steps for tree matching algorithms, with the aim of shortening the resulting edit scripts. In particular, they search for identical subtrees as a preprocessing step and target specific cases such as unmapped or moved leaf nodes. Much like GUMTREE, they do not optimize for matching program elements of different syntactic categories.

JDIFF goes beyond the AST and matches at the level of control flow [4], which changes the problem from tree matching to graph matching. Graph matching is in general $\mathcal{NP}$ hard and not feasible for the scenarios that we addressed in our experiments.

Malpohl et al. [14] developed a language-independent detector of renamed program elements that operates on parse trees. Besides tree matching, it considers also the static program semantics in the form of def–use pairs, which is more precise but harms performance (unfortunately, there is no evaluation on substantial programs available). The detector is not syntax-aware in the sense that it incorporates the types of nodes to find better matchings, as we do in our lookahead mechanism. So, it is not surprising that, in a case study, they had to calculate similarity measures for all possible pairs of identifiers to arrive at over 3000 identifier pairs, about 50 of which were renamings.

Beside mainstream programming languages, some approaches target model artifacts [15, 16, 17]. They are mostly based on graphs, which allow precise merging but harm performance. So, it is unlikely that they scale to problem sizes in the order of our subject projects.

Finally, tracing which change operations give rise to the versions to be merged can help in the detection and resolution of conflicts [18, 19, 20, 21, 22]. However, such an *operation-based* approach is infeasible when traces are not available or difficult to obtain, which is common in practice. Dig et al. [23, 24] proposed techniques to detect renamings in such scenarios. Other approaches require that the artifacts to be merged come with a formal semantics [25, 26], which is also rarely the case in practice (e.g., for mainstream programming languages). Another line of work attempts to detect refactorings by identifying semantic changes in a diff [27, 28, 29]. Finally, approaches that rely on model finders for semantic merge have substantial limitations with regard to performance [30].

IX. CONCLUSION

Diffing and merging software artifacts are central tasks in software development. While the state of the art still relies on an unstructured, lined-based approach to merging, recent developments demonstrate the merits and prospects of structured merging approaches.

Precise structured merging is computationally expensive, though, so practical approaches introduced a number of optimizations (top-down, level-wise matching and early return if nodes do not match). While these optimizations improve performance to a degree that makes structured merging practical, the imprecision induced touches two very relevant change scenarios: renaming and shifted code.

To improve precision without compromising performance, we have developed a syntax-aware, heuristic optimization of structured merging by devising a lookahead mechanism that, based on the type of the program element to match, descends the matching traversal efficiently. This way, we can handle renaming and shifted code uniformly.

In a series of experiments, on 48 real-world open-source projects ($4{,}878$ merge scenarios with over 400 million lines of code), we demonstrate that a syntax-aware lookahead mechanism can significantly improve matching precision in 28 percent without compromising performance. A qualitative analysis confirms the relevance and quality of the matchings produced.

X. ACKNOWLEDGEMENTS

This work has been supported by the German Research Foundation (AP 206/4, AP 206/5, and AP 206/6).

Table I
SUBJECT PROJECTS AND EXPERIMENT DATA (LOC: LINES OF CODE)

Project	# Scenarios	First Commit	# LOC	# Renamings	# Pieces of Shifted Code
ACTIONBARSHERLOCK	90	2011-03-21	41,993	13	19
ANDROID-CLEANARCHITECTURE	11	2014-09-05	5,207	0	0
ANDROID-OBSERVABLESCROLLVIEW	13	2015-01-29	17,043	0	3
ANDROID-PULLTOREFRESH	50	2011-12-10	6,102	11	11
ANDROID-UNIVERSAL-IMAGE-LOADER	78	2011-12-10	13,857	7	21
ANDROIDSWIPELAYOUT	34	2014-08-25	3,383	0	2
ANDROIDUTILCODE	20	2016-07-31	21,359	10	0
EVENTBUS	26	2012-07-31	7,709	8	6
HOMEMIRROR	14	2015-09-09	2,462	0	2
HYSTRIX	6	2012-04-09	78,541	0	2,105
MPANDROIDCHART	276	2014-04-25	42,570	68	84
MATERIAL-ANIMATIONS	7	2015-03-17	1,240	0	0
MATERIALDESIGNLIBRARY	21	2014-10-28	3,771	4	28
MATERIALDRAWER	188	2014-03-15	13,964	94	86
PHOTOVIEW	50	2012-10-08	2,303	2	4
POCKETHUB	114	2011-10-17	36,594	474	2,267
RXANDROID	70	2014-02-05	1,666	28	42
RXJAVA	438	2012-04-09	360,054	1,477	109
SLIDINGMENU	41	2012-07-01	4,932	8	16
VIEWPAGERINDICATOR	21	2011-09-26	4,059	15	6
ANDROID-ULTRA-PULL-TO-REFRESH	17	2014-12-09	6,879	0	1
ANDROID-UNIVERSALMUSICPLAYER	22	2015-03-10	9,222	0	8
ANDROID-ASYNC-HTTP	167	2011-03-15	12,609	7	9
ANDROIDANNOTATIONS	113	2011-10-13	68,297	140	338
BUTTERKNIFE	145	2013-03-06	12,058	228	232
DUBBO	49	2012-06-19	163,252	10	10
ELASTICSEARCH	10	2011-04-21	939,086	1	2
FASTJSON	37	2011-11-09	172,913	7	355
FRESCO	2	2015-03-26	109,561	32	65
GLIDE	77	2013-07-20	58,141	284	308
GUAVA	3	2010-07-07	732,697	52	102
IOSCHED	243	2014-04-02	70,691	247	31
JAVA-DESIGN-PATTERNS	81	2014-08-16	62,014	3,473	2,698
KOTLIN	40	2010-12-10	702,579	30	286
LEAKCANARY	109	2015-05-08	5,736	21	106
LOTTIE-ANDROID	5	2016-10-07	9,241	6	0
MATERIAL-DIALOGS	138	2014-11-08	9,013	8	36
NETTY	4	2010-11-23	365,984	0	0
OKHTTP	750	2012-07-23	74,251	955	754
PICASSO	337	2013-02-18	12,676	48	56
PLAID	32	2015-09-07	24,438	2	10
REALM-JAVA	12	2012-04-26	101,040	31	60
RETROFIT	482	2010-10-13	23,987	444	256
SPRING-BOOT	129	2013-05-27	353,034	353	607
STORM	231	2011-09-21	67,422	50	455
TINKER	13	2016-09-24	42,508	6	2
WEEX	21	2016-04-13	185,533	5	5,210
ZXING	41	2014-01-24	66,148	181	571

References

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

[2] B. Westfechtel, "Structure-Oriented Merging of Revisions of Software Documents," in *Proc. SCM*. ACM, 1991, pp. 68–79.

[3] J. Buffenbarger, "Syntactic Software Merging," in *Selected Papers from SCM-4 and SCM-5*, vol. LNCS 1005. Springer, 1995, pp. 153–172.

[4] T. Apiwattanapong, A. Orso, and M. Harrold, "JDiff: A Differencing Technique and Tool for Object-Oriented Programs," *ASE J.*, vol. 14, no. 1, pp. 3–36, 2007.

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

[6] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and Accurate Source Code Differencing," in *Proc. ASE*. ACM, 2014, pp. 313–324.

[7] G. Dotzler and M. Philippsen, "Move-optimized Source Code Tree Differencing," in *Proc. ASE*. ACM, 2016, pp. 660–671.

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

[9] 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.

[10] W. Yang, "Identifying Syntactic Differences Between Two Programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991.

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

[12] D. Asenov, B. Guenat, P. Müller, and M. Otth, "Precise Version Control of Trees with Line-Based Version Control Systems," in *Proc. FASE*, vol. LNCS 10202. Springer, 2017, pp. 152–169.

[13] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured Merge: Rethinking Merge in Revision Control Systems," in *Proc. ESEC/FSE*. ACM, 2011, pp. 190–200.

[14] G. Malpohl, J. J. Hunt, and W. F. Tichy, "Renaming Detection," *ASE J.*, vol. 10, no. 2, pp. 183–202, 2003.

[15] A. Mehra, J. Grundy, and J. Hosking, "A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design," in *Proc. ASE*. ACM, 2005, pp. 204–213.

[16] D. Kolovos, R. Paige, and F. Polack, "Merging Models with the Epsilon Merging Language (EML)," in *Proc. MODELS*, vol. LNCS 4199. Springer, 2006, pp. 215–229.

[17] C. Treude, S. Berlik, S. Wenzel, and U. Kelter, "Difference Computation of Large Models," in *Proc. ESEC/FSE*. ACM, 2007, pp. 295–304.

[18] E. Lippe and N. van Oosterom, "Operation-Based Merging," in *Proc. SDE*. ACM, 1992, pp. 78–87.

[19] D. Dig, K. Manzoor, R. Johnson, and T. Nguyen, "Refactoring-Aware Configuration Management for Object-Oriented Programs," in *Proc. ICSE*. IEEE, 2007, pp. 427–436.

[20] M. Koegel, J. Helming, and S. Seyboth, "Operation-Based Conflict Detection and Resolution," in *Proc. CVSM*. IEEE, 2009, pp. 43–48.

[21] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer, "Conflict Detection for Model Versioning Based on Graph Modifications," in *Proc. ICGT*, vol. LNCS 6372. Springer, 2010, pp. 171–186.

[22] H. Gall, B. Fluri, and M. Pinzger, "Change Analysis with Evolizer and ChangeDistiller," *IEEE Software*, vol. 26, no. 1, pp. 26–33, 2009.

[23] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated Detection of Refactorings in Evolving Components," in *Proc. ECOOP*, vol. LNCS 4067. Springer, 2006, pp. 404–428.

[24] D. Dig, K. Manzoor, R. Johnson, and T. Nguyen, "Effective Software Merging in the Presence of Object-Oriented Refactorings," *IEEE TSE*, vol. 34, no. 3, pp. 321–335, 2008.

[25] V. Berzins, "Software Merge: Semantics of Combining Changes to Programs," *ACM TOPLAS*, vol. 16, no. 6, pp. 1875–1903, 1994.

[26] D. Jackson and D. Ladd, "Semantic Diff: A Tool for Summarizing the Effects of Modifications," in *Proc. ICSM*. IEEE, 1994, pp. 243–252.

[27] B. Fluri, M. Würsch, M. Pinzger, and H. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE TSE*, vol. 33, no. 11, pp. 725–743, 2007.

[28] P. Weißgerber and S. Diehl, "Identifying Refactorings from Source-Code Changes," in *Proc. ASE*. IEEE, 2006, pp. 231–240.

[29] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based Reconstruction of Complex Refactorings," in *Proc. ICSM*. IEEE, 2010, pp. 1–10.

[30] S. Maoz, J. Ringert, and B. Rumpe, "CDDiff: Semantic Differencing for Class Diagrams," in *Proc. ECOOP*, vol. LNCS 6813. Springer, 2011, pp. 230–254.