Code Merging using Transformations and Member Identity

André R. Teles
adrts@iscte-iul.pt
Instituto Universitário de Lisboa (ISCTE-IUL)
Portugal

André L. Santos
andre.santos@iscte-iul.pt
Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR-IUL
Portugal

Abstract

Conventionally, merging code files is performed using generic line-based merging algorithms (e.g., diff3) that are unaware of the syntax and semantics of the programming language, outputting conflicts that could be avoided. Structured and semistructured merging techniques are capable of reducing conflicts, but they still suffer from *false positives* (conflicts that could be avoided) and *false negatives* (conflicts that go undetected). We propose a merging technique that combines semistructured and transformation-based strategies, where conflict detection is aware of semantic aspects of the programming language. We extract transformations of two branches and apply a merging process that analyzes incompatible transformations, avoiding false positives and false negatives that occur in existing approaches. We developed Jaid, a prototype merging tool for Java based on the assumption that structural code elements evolve with attached UUIDs (representing identity). We performed an early experiment with 63 merge scenarios from two open-source projects to test the technique and assess its feasibility.

CCS Concepts: • **Software and its engineering → Software configuration management and version control systems**.

Keywords: software merging, version control systems, transformations, conflicts, identity

ACM Reference Format:
André R. Teles and André L. Santos. 2023. Code Merging using Transformations and Member Identity. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '23), October 25–27, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3622758.3622891

Onward! '23, October 25–27, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0388-1/23/10.
https://doi.org/10.1145/3622758.3622891

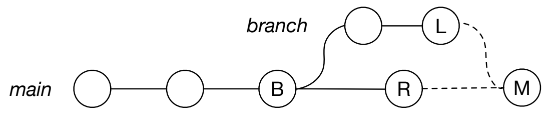


Figure 1. Three-way merge in Git-flow processes. left (L) and right (R) versions with a base ancestor (B) originate a merged version (M).

1 Introduction

A version control system (VCS) allows teams to manage and parallelize tasks in order to deliver a software artifact as quickly as possible. Part of their functionality is the ability to merge branches with other branches. A three-way merge is based on three different versions of a software artifact (see Figure 1). Throughout this paper, we refer to those as *left*, *right*, and *base*. Both left and right have the base as their nearest common ancestor. When left and right are merged, we obtain a new version that combines their changes.

Branches may be merged based on different algorithms, which may produce different results. If one or more developers change the same piece of code and the merging algorithm cannot automatically merge both changes into a new version, a *merge conflict* arises. Manual intervention is required to resolve the conflict to successfully merge the two branches. However, resolving a conflict typically requires developer communication and a true understanding of the related code, making it a time-consuming task during the development cycle, delaying it, and reducing team productivity [9].

Most VCSs have built-in merge tools that are based on algorithms that compute the differences through a line-based comparison [16]. In other words, the unit of comparison between files is a line, and the merging process is purely textual. This happens because the representation of files as plain text is the most generic way of representing files in many programming languages. This merge strategy is referred to in the literature as *unstructured merging*.

Despite the genericity offered by unstructured merging, this strategy has obvious limitations, since it ignores the syntax and semantics of each programming language. Depending on the merge strategy, some conflicts may occur where there is, in fact, no conflict — these are *false positive* cases. Taking into account the drawbacks of unstructured merging, new merge tools have been proposed that take into account the syntax and structure of the programming language of the source code [2, 4, 10, 11, 26, 33]. These aim at improving the accuracy of merging processes, by obtaining fewer non-existent conflicts (*false positives*) and recognizing more conflicts that are not detected and cause compilation errors (*false negatives*).

Despite previous approaches being capable of reducing conflicts, there is room for improvement with respect to false negatives and false positives. One of the main challenges relates to matching source code elements of different branches, given that textual programming languages have no stable identifiers [32]. The difficulty resides in the fact that source code elements have no *identity*, and hence, a form of realizing *origin tracking* [31] is necessary. During compilation, references to code members are essentially resolved by name, which is simply a value that is looked up in a symbol table. If an element is renamed (i.e., type, field, method), all the existing dependent calls become invalid. Issues pertaining to renaming in three-merging are particularly sensitive to this problem, because a branch may evolve to add new references to a member that has been renamed in parallel by the other branch [27].

In this paper, we propose an approach that combines aspects of previous research on semi-structured [2] and operation-based merging [20]. We perform version differencing and union using models [1] of the code artifacts (Abstract Syntax Trees) in a language-specific manner, where code members are assumed to be augmented with a form of representing their *identity.* This characteristic requires encoding *ids* in source code files, and hence, it implies a slightly different form of maintaining source code, but nevertheless, still compatible with current practices. In particular, we propose to represent the identity of referenceable elements in the code by "attaching" comments that hold universally unique identifiers (UUID) to their headings. This brings advantages that are inherent to software development using *projectional editors* [28], where code artifacts are stored in tool-specific formats that embody those identifiers (e.g., JetBrains MPS[1]), while not needing to adopt such tools.

We developed *Jaid*, a merge tool for Java that extracts transformations from two branches given a base version, and further analyzes them to check for conflicts, taking into account semantic aspects of the language to avoid both false positives and false negatives. Namely, we enable broken identifier references (due to renames) to be fixed when applying a three-way merge, and hence, avoid this type of false negative. Merging processes using models, as in projectional editors, are capable of addressing this issue, whereas previous structured merging techniques are not [11, 26].

We performed an early experiment with Jaid involving 63 merge scenarios extracted from two open-source projects. An analysis of false negatives on merges without conflicts revealed that 79% could successfully build, while the remaining cases did not build due to the lack of Jaid's Java coverage on some constructs. On the other hand, we found no false positives when analyzing merge scenarios where conflicts were found. The execution times are the order of a few seconds for the whole project merges. Although these are clearly slower than previous tools [11, 26], we argue that the capability of avoiding false positives and false negatives, which otherwise would be fixed by hand, is likely to compensate for the performance trade-off.

In this paper, we first motivate the need for more precise merging processes in Section 2 in light of the state-of-the-art merge tools. Section 3 describes the proposed merging process. Section 4 discusses details regarding the implementation of Jaid. Section 5 presents a first evaluation of our merging approach in real scenarios extracted from Github projects, and Section 6 discusses the tradeoffs of our approach and threats to validity. Section 7 presents related work, and Section 8 presents our conclusions.

2 Motivating Example

This section presents an example to motivate our approach in contrast to industrial practice and state-of-the-art approaches. Figure 2 presents a three-merge scenario involving a single class *Point*, where a base version (b) has evolved to a left (a) and right version (c) which will be merged. The left changes consist of adding the modifier `final` to both fields in order to have immutable objects, adding a constructor, renaming the getter methods to adhere to the usual convention, and adding a method to check if the point is the origin. The right changes consist of adding the modifier `private` to both fields and adding the same method to check if the point is the origin (using the method identifiers of the base version).

Merging the left and right versions into a valid new version of the class is possible, as the set of changes are not incompatible. Figure 2 (d) presents the ideal merge result, where both field modifications are integrated, method renames are performed, and references to it are updated accordingly. However, automated methods are not yet able to obtain such a merge. The goal of our approach is to have a merge process that is capable of outputting such a result.

We ran the merge process using the three versions of the class presented in Figure 2 (a,b,c) with git-merge[2], a widely-used industrial form of merging, jFSTMerge [11][3], IntelliMerge [26][4], which are state-of-the-art approaches to this problem, and MPS, an industrial language workbench

[1]https://www.jetbrains.com/mps/

[2]https://git-scm.com/docs/git-merge

[3]https://github.com/guilhermejccavalcanti/jFSTMerge

[4]https://github.com/Symbolk/IntelliMerge

```
class Point {
  final int x;
  final int y;

  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  int getX() {
    return x;
  }

  int getY() {
    return x;
  }

  boolean isOrigin() {
    return getX() == 0
        && getY() == 0;
  }
}
```

(a) Left branch version.

```
class Point {
  int x;
  int y;

  int x() {
    return x;
  }

  int y() {
    return y;
  }
}
```

(b) Base version.

```
class Point {
  private int x;
  private int y;

  int x() {
    return x;
  }

  int y() {
    return y;
  }

  boolean isOrigin() {
    return x() == 0
        && y() == 0;
  }
}
```

(c) Right branch version.

```
class Point {
  private final int x;
  private final int y;

  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  int getX() {
    return x;
  }

  int getY() {
    return x;
  }

  boolean isOrigin() {
    return getX() == 0
        && getY() == 0;
  }
}
```

(d) Ideal merged version.

Figure 2. Illustrative three-way merge scenario involving a single class.

```
class Point {
<<<<<<< LEFT
  final int x;
  final int y;

  Point(int x,int y){
      this.x = x;
      this.y = y;
=======
  private int x;
  private int y;
 >>>>>>> RIGHT

 int getX() {
  return x;
 }

 int getY() {
  return x;
 }

 boolean isOrigin(){
  return getX()==0
      && getY()==0;
 }
<<<<<<< LEFT
}
=======

 boolean isOrigin(){
  return x()==0
    && y()==0;
  }
}
>>>>>>> RIGHT
```

(a) Git-merge (diff3).

```
class Point {
<<<<<<< LEFT
  final int x;
=======
  private int x;
>>>>>>> RIGHT

<<<<<<< LEFT
  final int y;
=======
  private int y;
>>>>>>> RIGHT

  Point(int x,int y){
      this.x = x;
      this.y = y;
  }

  int getX() {
    return x;
  }

  int getY() {
    return x;
  }

  boolean isOrigin(){
<<<<<<< LEFT
    return getX()==0
        && getY()==0;
=======
    return x()==0
        && y()==0;
>>>>>>> RIGHT
  }
}
```

(b) jFSTMerge.

```
class Point {
<<<<<<< LEFT
  final int x;
=======
  private int x;
>>>>>>> RIGHT

<<<<<<< LEFT
  final int y;
=======
  private int y;
>>>>>>> RIGHT

  Point(int x,int y){
      this.x = x;
      this.y = y;
  }

  int getX() {
    return x;
  }

  int getY() {
    return x;
  }

  boolean isOrigin(){
    return x()==0
        && y()==0;
  }
}
```

(c) IntelliMerge.

```
class Point {
  private final int x;
  private final int y;

  Point(int x,int y){
      this.x = x;
      this.y = y;
  }

  int getX() {
    return x;
  }

  int getY() {
    return x;
  }

<<<<<<< LEFT
  boolean isOrigin(){
    return getX()==0
        && getY()==0;
  }
=======
  boolean isOrigin(){
    return x()==0
        && y()==0;
  }
>>>>>>> RIGHT
}
```

(d) JetBrains MPS.

Figure 3. Merge results with industry and state-of-the-art approaches.

providing projectional editing. We used the standard git-merge tool that ships with Git and the latest releases available from the research project repositories.

The results are presented in Figure 3. The git-merge result has a conflict due to the unawareness of structure in line-based merging (*false positive*) and also detects a conflict in the *isOrigin* method (*false positive*), since the two versions (left and right) clash in the same text region. The merge results of jFSTMerge and IntelliMerge originate a conflict in the fields, given that these approaches do not address semantic-aware combinations of modifiers. This conflict is more fine-grained than the one of git-merge, but nevertheless, it could be avoided. Their results also suffer from the same problem in the *isOrigin* method, but in a different way. Although jFSTMerge's merge result shows more structure awareness compared to git-merge's result by marking the conflict only in the body of the *isOrigin* method, both approaches present a non-existent conflict (*false positive*). On the other hand, IntelliMerge does not raise the conflict but outputs a non-compilable result (*false negative*) since the method references have not been renamed according to the methods. A large-scale study has revealed that many build conflicts are due to missing declarations removed or renamed by one version but referenced by another [27].

Finally, we tested the merging scenario in MPS, which carries out the merge using the model representation of the code (in Figure 3 we depict the conflict using the typical source code marks). Despite its structural nature, a conflict is detected between the two versions (left and right) of the *isOrigin* method (*false positive*). Even though they are semantically equivalent, MPS, due to the apparent lack of semantic awareness, did not recognize that they could be merged into a single method, in particular, because the id of both *isOrigin* methods is different, as they were created in different revisions and are not derived from a common one (base). As another case, not illustrated in the scenario, if one branch would add the `final` modifier and another branch the `static` modifier, we would face a false positive, because they both belong to the same modifiers container (given the way the Java language was modeled in MPS). However, it is worth noting that MPS can resolve broken references due to renames (as in the given scenario).

3 Approach

3.1 Representing Member Identity

We propose to use UUIDs as a form of providing identity to source code elements, a technique that is widely used, for instance in the XMI standard (XML Metadata Interchange [23]) to encode references between elements. A UUID is a 36-character alphanumeric string that is assumed to be unique due to a large number of different combinations of hexadecimal digits. The probability of having two identical UUIDs is so remote that it can be neglected.

```
//03ece596-e4dd-11ed-b5ea-0242ac120002
class Point {

  //51447490-5ae5-468c-adb0-27ec4a39bff2
  private final int x;

  //f9b7a663-bd12-4ce5-a52e-db6e8b0ad3e0
  private final int y;

  //b94372dd-d103-4e07-997f-cfd93408ca9e
  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  //7a67ca25-8b0c-4bd6-97b1-141d56988c2d
  int getX() {
    return x;
  }

  int getY() {
    return y;
  }
}
```

Figure 4. Java source code with member identity encoded as UUIDs in comments.

UUIDs are attached to structural elements of the code that can be referenced from statements and expressions. Hence, these elements can be classes, interfaces, methods, constructors, or fields. Such an identity has to be encoded as part of the source code files so that it is persisted on a VCS. In order to have a source code format that includes UUIDs and is still valid source code in the host language, we opt for storing the UUIDs as source code comments.

Figure 4 illustrates a Java class with UUID comments attached. Notice that every structural element has a comment holding a UUID. The highlighted code would be an addition created locally that is not yet stored in the VCS. An obvious downside of having UUIDs in the code is the extraneous noise they introduce, as well as their fragility, as the identity can be broken accidentally (deletes, copy-paste, etc). Addressing these usability issues requires a slightly specialized code editing environment, which we discuss later in Section 6.

If two elements from two different branches have the same UUID in their comments, they are considered to be the same, even if one or all of their properties have changed. By storing these identifiers in the comments throughout the development cycle of a software artifact, the identity of each element is maintained over time. Since new additions to the source code have no UUID associated, their identity is non-existent. When a branch adds new elements to the source, these are preprocessed to inject UUIDs automatically when committing to a branch. When the local changes of a branch become available in the remote repository, updates will have access to the newly added elements with their identity attached. In this way, all structural elements stored

Table 1. Examples of transformations and their parameters.

Transformation	**Parameters**
adding a field	UUID of the owner class; field type; field name; field modifiers; [initializer expression]
removing a method	UUID of the method
modify method signature	UUID of the method; method identifier; list of parameters
moving a static method	UUID of the method; UUID of the destination type
modifying Javadoc	UUID of the member; documentation content

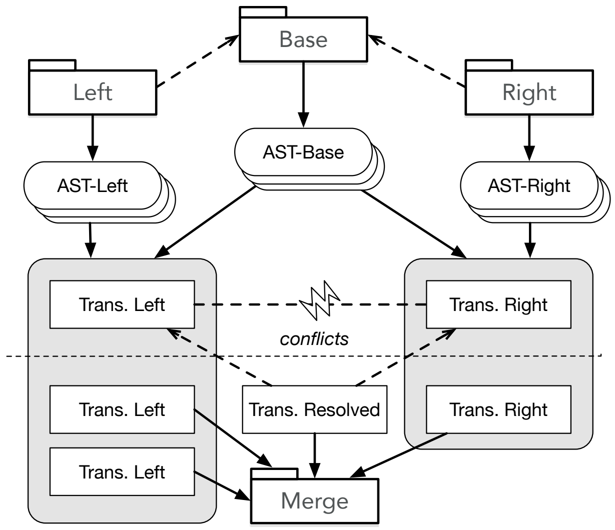


Figure 5. Merge process overview.

in the VCS have an identity attached, which will be available to newly created branches.

3.2 Transformations

The fundamental part of an operation-based approach are operations [20], also called *transformations*. Any atomic operation which can be applied to a single structural element of the source code can be represented as a transformation. Any property of a structural element that is likely to be modified can produce a transformation, such as changing the body of a method, renaming a class, changing the modifiers of an element, and so forth.

We consider that a transformation is a unit of modification that a version (branch) has performed over a base version. A branch transformation fits into one out of four broad types: *addition*, *removal*, *modification*, or *move*. We also considered modifications to the documentation of classes as transformations of their own. We represent each transformation as a *command* (Command pattern [14]) that can be applied to the model of the base version (AST). Hence, each transformation object holds all the necessary information to carry out the modifications. Our current implementation comprises 35 transformation types. Table 1 presents examples of transformations and their parameters.

3.3 Merging Process

In a three-way merge scenario, the process of extracting transformations is performed between the versions of each branch and their common ancestor (base version). Figure 5 presents an overview of our merge process, with the assumption that structural elements of source code have an attached identity (as explained in Section 3.1). Merge is performed using the whole source code of the project versions in the three-way merge (left, base, right). Each version is parsed into abstract syntax trees (AST), and element identifiers are indexed by UUID in order to allow efficient resolution (Section 4.2). The AST pairs (left, base) and (right, base) are analyzed to extract a set of transformations at the level of granularity of the indexed elements. For example, an addition of a method, or renaming of a class (Section 3.3.1). In the following stage, conflicting transformations between the two sets and the shared set are detected (Section 3.3.2). These have to be resolved with human intervention, whereas non-conflicting transformations are suitable to be applied directly in the merged version of the project (Section 3.3.3).

Algorithm 1 describes the overall merge process. For each branch, we determine a set of transformations (left and right). The shared transformations, that is, those that are equivalent in both branches (same type and same parameters), are factored out into a separate set. Using the three sets of transformations we compute the set of conflicts. The decomposition of the process is further detailed in the next sections.

When two transformations in the two versions are the same, we refer to them as a single *shared* transformation. In some cases, redundancy is not a problem, but nevertheless pointless, such as renaming the same method twice with the same new name. However, there are some situations where applying these two transformations could cause the merge process to malfunction. If we consider a pair of transformations involving the addition of the same structural element, this situation is particularly problematic, because after inserting the element once, the second transformation will insert it again, causing an identifier conflict. For this reason, all redundant transformations are factored at this phase, so that only one of them is applied.

3.3.1 Extracting Transformations. Algorithm 2 describes the process of extracting transformations from the set of files of a version (branch) when compared to a base version. The

Algorithm 1 Procedure to merge two versions (left and right), considering a base version. Merging is performed if there are no conflicts, otherwise a non-empty set of conflicts is returned. The parameter $conflictTypes$ is a set of conflict detectors for transformation pairs.

```
procedure MergeVersions($conflictTypes$, $baseFiles$, $leftFiles$, $rightFiles$)
    $left \leftarrow$ VersionTransformations($baseFiles$, $leftFiles$)
    $right \leftarrow$ VersionTransformations($baseFiles$, $rightFiles$)
    $shared \leftarrow \{(a, b) \in (left \times right) : a = b\}$
    $left \leftarrow left - shared$
    $right \leftarrow right - shared$
    $conflicts \leftarrow$ ComputeConflicts($conflictTypes$, $left$, $right$, $shared$)
    if $conflicts = \emptyset$ then
        ApplyTransformations($baseFiles$, $left$, $right$, $shared$)
    end if
    return $conflicts$
end procedure
```

Algorithm 2 Function to obtain the set of transformations of a version (branch) in relation to a base version.

```
function VersionTransformations($baseFiles$, $versionFiles$)
    $transformations \leftarrow \emptyset$
    for all $f \in (versionFiles - baseFiles)$ do
        $transformations \leftarrow transformations \cup \{AddFile(f)\}$
    end for
    for all $f \in (baseFiles - versionFiles)$ do
        $transformations \leftarrow transformations \cup \{RemoveFile(f)\}$
    end for
    for all $(a, b) \in \{(a, b) \in (baseFiles \times versionFiles) : a.uuid = b.uuid\}$ do
        AddNodeTransformations($transformations$, $a$, $b$)
    end for
    AddMoveTransformations($transformations$)
    return $transformations$
end function
```

first step is to understand which files are not in both versions and generate their respective insertion/removal transformations. The match is accomplished by comparing the identities of the files, whose UUIDs are stored in the first line of each file.

The next step consists of finding the pairs of corresponding files between the two versions and checking their differences (Algorithm 3). For each language-specific structure from which transformations may be extracted (assumed to be identified by UUIDs), a handler must be implemented to check for possible changes in the properties of that structure. Each supported modification will result in a transformation, whose type is specific to the kind of modification (e.g., RenameClass, ChangeBody). Similar to the detection of file insertions and removals, the extraction of additions and removals of children is based on the following idea: if in the branch version and not in the base, it is considered an addition of a node; if in the base and not in the branch, it is considered a removal of a node.

When member bodies are altered we handle this as a coarse-grained modification transformation involving all its statements. However, we perform a special comparison strategy that instead of verifying the equality of AST nodes with element references by value (token in the source), compares those using the identity of the referenced element. The name present in a reference is ignored and what is used for comparison is the UUID of the element to which they refer. This comparison strategy implies that if one performs a rename refactor, for instance on a method, all the member bodies that include dependent expressions will be considered unchanged. In fact, only a symbol has changed, and we represent and further apply this sort of change as a cohesive transformation unit, instead of a scattered set of modifications that include all the bodies holding dependent expressions. This allows us to consider that in the scenario of Figure 2 both versions of the *isOrigin* method are equivalent.

The final step in extracting the transformations is to check for any *move transformations* between files (Algorithm 4). Since all the addition and removal transformations of all the

Algorithm 3 Procedures to collect transformations performed on an AST Node, abbreviated with respect to node types. The parameters b (base node) and v (version node) are assumed to have the same type.

```
procedure AddNodeTransformations($transformations, b, v$)
    if $typeOf(b) = File$ then
        AddMemberTransformations($transformations, b, v$)
    else if $typeOf(b) = Class$ then
        if $b.name \neq v.name$ then
            $transformations \leftarrow transformations \cup \{RenameClass(b.uuid, v.name)\}$
        else if $b.modifiers \neq v.modifiers$ then
            $transformations \leftarrow transformations \cup \{ChangeModifiers(b.uuid, v.modifiers)\}$
        end if
        ...
        AddMemberTransformations($transformations, b, v$)
    else if $typeOf(b) = Method$ then
        if $b.name \neq v.name$ then
            $transformations \leftarrow transformations \cup \{RenameMethod(b.uuid, v.name)\}$
        else if $b.body \neq v.body$ then
            $transformations \leftarrow transformations \cup \{ChangeBody(b.uuid, v.body)\}$
        end if
        ...
    end if
    ...
end procedure

procedure AddMemberTransformations($transformations, b, v$)
    $baseChildren \leftarrow b.childNodes$
    $versionChildren \leftarrow v.childNodes$
    for all $n \in (versionChildren - baseChildren)$ do
        $transformations \leftarrow transformations \cup \{AddNode(n)\}$
    end for
    for all $n \in (baseChildren - versionChildren)$ do
        $transformations \leftarrow transformations \cup \{RemoveNode(n)\}$
    end for
    for all $(a, b) \in \{(a, b) \in (baseChildren \times versionChildren) : a.uuid = b.uuid\}$ do
        AddNodeTransformations($transformations, a, b$)
    end for
end procedure
```

Algorithm 4 Procedure to add move transformations by converting matching addition-removal pairs.

```
procedure AddMoveTransformations($transformations$)
    $additions \leftarrow \{t \in transformations : typeOf(t) = AddNode\}$
    $removals \leftarrow \{t \in transformations : typeOf(t) = RemoveNode\}$
    for all $(a, r) \in \{(a, r) \in (additions \times removals) : a.uuid = r.uuid \land a.node.isStatic\}$ do
        $transformations \leftarrow transformations \cup \{MoveNode(a, r)\}$
        $transformations \leftarrow transformations - \{a, r\}$
        AddNodeTransformations($transformations, r.node, a.node$)
    end for
end procedure
```

files in both versions have been extracted, if there is one file with an addition and another with a removal transformation where both refer to the same UUID, we consider that the involved element has been moved between files. The same strategy is used to check if static methods, fields, or enum constants have been moved between types. After isolating all the insertion and removal transformations, the addition-removal pairs that refer to the same UUID will result in adding a move transformation to the set of transformations, while the original addition and removal transformations are removed. The last step is to extract the transformations between the two versions (base and branch) of the moved node, since these two versions have not been compared before due to the moved node having different parents when it is removed from one member and inserted into another.

3.3.2 Conflict Detection.

Once the two sets of transformations and the shared set have been obtained, the next phase is to find pairs of conflicting transformations. Our notion of *conflict* is slightly different than what is generally considered in code merging. We identify a conflict when two transformations cannot both be performed without leading to a syntactic or semantic error in the source code. For example, when two renames of the same class use different values. In these sorts of cases, it is not possible to have an automated process that decides which of the two transformations to select for merging. Hence, our aim is that a conflict is a situation that *necessarily* needs human intervention in order to be resolved. We aim at a merge process that is as deterministic as possible, while not performing merges that will require post-merge manual fixes.

To achieve this kind of conflict reporting, we defined a set of conflict types that defines which pairs of transformation types may lead to a conflict. Appendix A contains tables describing a non-exhaustive summary of potentially conflictual transformation types. If there is no conflict type between two transformation types, there is no situation where the application of both transformations could cause a conflict. A conflict type is an object that has information about:

- the two types of transformations that lead to a conflict;
- a method to check if it applies to two unordered transformations;
- a message generator to explain the conflict cause.

For example, for a field *Add-Rename* transformation pair, a conflict type shall be declared with types *Add* and *Rename*, along with a handler that evaluates whether the new field name and the new rename value are the same. If they are different, there is no conflict, otherwise, a conflict is found. A conflict is formed by the two conflicting transformations, as well as a message justifying why they are incompatible. Our conflicts are accompanied by a human-readable explanation and precise references to the involved code elements. As an example conflict message, "two renames of the same method: *a* (left) and *b* (right)".

Algorithm 5 describes the process of finding conflicting pairs of transformations. Each pair of the Cartesian product of the two sets of transformations and the shared set is checked for a possible conflict, which when positive, is added to the set of conflicts. A pair of transformations is straightly rejected as a possible conflict if their types are not identified as conflictual (see Appendix A).

Regarding modifiers, we follow a semantic-aware strategy to verify conflicts. First, for each set of modifiers of an element, two subsets are created: the access modifiers (`public`, `private`, `protected`) and the remaining modifiers (`final`, `static`, etc.). There are two reasons why two lists of modifiers may conflict. One is when the two subsets of access modifiers combined have more than one element. For example, consider that one subset has a `public` modifier and another has a `private` modifier. If we sum both subsets, we will get a new one with two elements (`public` and `private`), so the situation is considered conflicting. If the two subsets have the same modifier, since sets do not allow duplicates, only one will be stored in the resulting set of the sum of them and both modifiers' lists are not conflicting. The other reason is if at least one of the subsets of non-accessible modifiers has an `abstract` modifier and the other has one of the remaining modifiers that can be used (`static`, `final`). In other words, if there is an `abstract` modifier, there can be no other non-access modifier. If neither of these two reasons is true, then the two lists of modifiers are not conflicting and can be merged automatically. This process is able to solve the false positive problem illustrated in Figure 3.

3.3.3 Applying Transformations.

If there are no conflicting transformations, all the transformations of the three sets (left, right and shared) can safely be merged into a copy of the base version in order to obtain a newly merged version. In the presence of conflicts, we propose that developers should opt for not performing the merge, but rather go through the conflicts given by our process and fix the issues in one or both branches until no conflicts are obtained (hence, the description of Algorithm 1).

Algorithm 6 describes the process of applying transformations. The order in which the transformations are applied is important, based on the state of the version when they were all extracted. When local move transformations are extracted, the removed and inserted members are ignored, and then the index to which the element should be moved is calculated based on a list of members without the removed and newly inserted members. For this reason, to correctly apply a local move transformation, we must first remove all the members that should be removed, so that the member list is equal to the member list when the local move transformation was extracted (without the removed and newly inserted members). Note that an inter-type move transformation, e.g. an element moved from one file to another, can be decomposed into two transformations: the removal transformation of the element from the origin node and the insertion transformation of that element into the destination node. After all local move transformations have been applied, the insertion transformations can be applied properly. The order in which

Algorithm 5 Function to compute the conflicts with the transformations of two versions (left and right), taking into account their shared transformations. The parameter $conflictTypes$ is a set of conflict detectors for transformation pairs.

```
function ComputeConflicts($conflictTypes, left, right, shared$)
    $conflicts \leftarrow \emptyset$
    $transformationPairs \leftarrow (left \times right) \cup (left \times shared) \cup (right \times shared)$
    for all $(a, b) \in transformationPairs$ do
        for all $c \in conflictTypes$ do
            if $c.isApplicable(a, b) \wedge c.existsConflict(a, b)$ then
                $conflicts \leftarrow conflicts \cup \{Conflict(c, a, b)\}$
            end if
        end for
    end for
    return conflicts
end function
```

Algorithm 6 Procedure to apply merge transformations.

```
procedure ApplyTransformations($baseFiles, left, right, shared$)
    $transformations \leftarrow left \cup right \cup shared$
    $mergedFiles \leftarrow baseFiles$
    for all $t \in \{t \in transformations : typeOf(t) = AddFile\}$ do
        $t.apply(mergedFiles)$
    end for
    $globalMoves \leftarrow \{t \in transformations : typeOf(t) = MoveNode\}$
    for all $gm \in globalMoves$ do
        $gm.removeTransformation.apply(mergedFiles)$
    end for
    for all $t \in \{t \in transformations : typeOf(t) = RemoveNode \vee RemoveFile\}$ do
        $t.apply(mergedFiles)$
    end for
    for all $lm \in \{t \in transformations : typeOf(t) = LocalMoveNode\}$ do
        $lm.apply(mergedFiles)$
    end for
    for all $gm \in globalMoves$ do
        $gm.additionTransformation.apply(mergedFiles)$
    end for
    for all $t \in \{t \in transformations : typeOf(t) = AddNode\}$ do
        $t.apply(mergedFiles)$
    end for
    for all $t \in \{t \in transformations : typeOf(t) \neq AddFile \vee RemoveNode \vee RemoveFile \vee MoveNode \vee LocalMoveNode \vee AddNode\}$ do
        $t.apply(mergedFiles)$
    end for
    $write(mergedFiles)$
end procedure
```

the transformations from the final list of transformations are applied is as follows:

1. Apply all file additions;
2. Filter out all inter-types move transformations and apply only their corresponding removal transformation;
3. Apply all other removal transformations (which remove files, methods, fields, etc);
4. Apply all local move transformations in the order defined when they were all extracted;
5. Filter all inter-type move transformations and apply only their insertion transformations;
6. Apply all other insertion transformations (which add methods, fields, etc);
7. Apply all other transformations in any order.

Note that each time a new file or node is inserted, its references are indexed (see Section 4.2) in order that all other transformations are applied correctly. When the process of applying all these transformations to the common ancestor is complete, a merged version is created with the contributions from the two branches.

3.3.4 Translation of Identifiers by Reference. Most of the transformations are straightforward to apply, with the exception of method body modifications are handled with a non-trivial mechanism that makes the merging process more robust. If a member is referenced in new code (e.g., a method call) and that element is renamed in the other branch, the code will have missing references when merged (e.g., the false negative described in Figure 3). We address this problem so that those "outdated" references are translated into the correct ones.

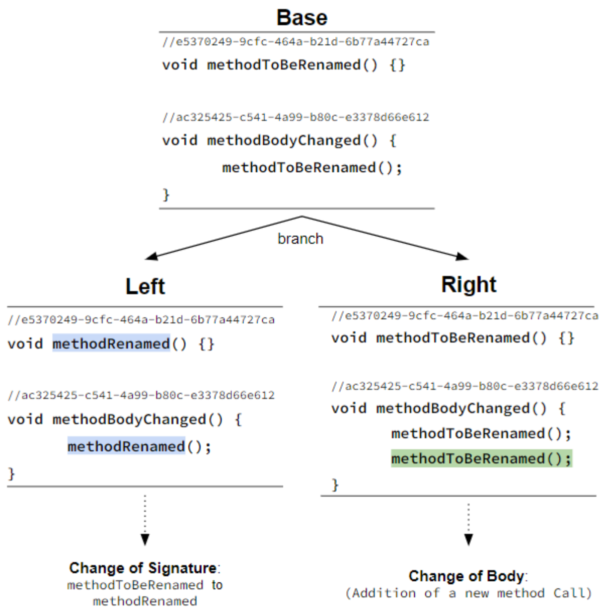
Figure 6 shows a three-way merge scenario to illustrate this case in which two branches are derived from a base branch: the left branch, in which the method named *methodToBeRenamed* is renamed to *methodRenamed* along with all its calls, resulting in a Signature transformation; and the right branch, in which a new method call to *methodToBeRenamed* is added to the method named *methodBodyChanged*, resulting in a Body modify transformation. Notice that the latter call is made using the method name that is going to be renamed by the left branch. The two transformations are not considered to form a conflict, but the order in which they are applied produces different outputs. Appendix B illustrates that performing right followed by left leads to the desired output, whereas the opposite ordering will lead to broken references in the body of *methodBodyChanged*.

The statements of a method body hold identifiers that refer to other elements (types, methods, fields). In our approach, we maintain an identity for all the referenceable elements through the UUIDs. Instead of simply copying the new method body into the merged version when applying the transformation, we translate all the contained identifier references to match those of the current version rather than those of the version where changes were introduced. We illustrate this mechanism in Figure 7.

As discussed ahead in Section 7, the order in which transformations are applied is a limitation of operation-based merging. Appendix C illustrates how applying our merging process involving the two transformations of Figure 6 with different ordering leads to the same result.

4 Implementation

As a proof of concept of the proposed approach, we developed *Jaid*, a merging tool for Java projects. The extraction of transformations and the set of conflict types are language-dependent. We currently do not support the whole Java syntax, as our implementation efforts have focused on the essential constructs to be able to have a working proof of concept.


Base
branch
Left
Right
Change of Signature:
methodToBeRenamed to methodRenamed
Change of Body:
(Addition of a new method Call)

```
//e5370249-9cfc-464a-b21d-6b77a44727ca
void methodToBeRenamed() {}

//ac325425-c541-4a99-b80c-e3378d66e612
void methodBodyChanged() {
        methodToBeRenamed();
}
```

```
//e5370249-9cfc-464a-b21d-6b77a44727ca
void methodRenamed() {}

//ac325425-c541-4a99-b80c-e3378d66e612
void methodBodyChanged() {
        methodRenamed();
}
```

```
//e5370249-9cfc-464a-b21d-6b77a44727ca
void methodToBeRenamed() {}

//ac325425-c541-4a99-b80c-e3378d66e612
void methodBodyChanged() {
        methodToBeRenamed();
        methodToBeRenamed();
}
```

Figure 6. Three-way merge scenario.

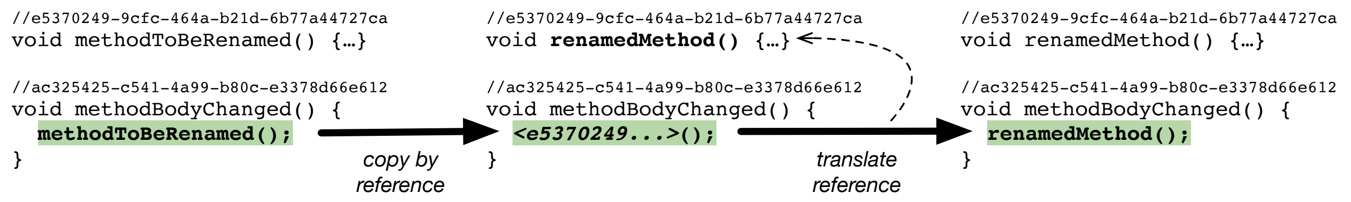
For instance, annotations, lambdas, and generic types are constructs that are not currently supported, despite that we do not foresee that they present any particular implementation challenge. Jaid is developed in Kotlin and it currently has about 6K lines of code.

4.1 Abstract Syntax Trees for Java

A critical library that was fundamental for developing Jaid was JavaParser[5], an open-source parser for the Java programming language that also provides tools for analyzing, transforming, and generating new code, through AST manipulations. JavaParser is a widely used library that has even been used in other studies (e.g., [26]). Overall, JavaParser is the backbone of most Jaid processes, as it loads the code structure into memory and these parsed nodes are the units that crosscut the entire merging process.

JavaParser handles code comments so that they are nodes in the AST. Comments that immediately precede type and member declarations are represented as child nodes of those. We use JavaParser to deal with all operations related to UUIDs in member comments, as their representation in the AST facilitates the process of matching members to their comments. The process of appending a UUID to a member's comment depends on the type of comment associated with a member. If the member does not have a comment, a new line comment (`//...`) is appended with a newly generated UUID. If the member already has a line comment, the comment is converted to a block comment (`/*...*/`), which stores the previous content of the line comment as well as the UUID.

[5]https://javaparser.org



```
//e5370249-9cfc-464a-b21d-6b77a44727ca
void methodToBeRenamed() {…}

//ac325425-c541-4a99-b80c-e3378d66e612
void methodBodyChanged() {
  methodToBeRenamed();
}
```

```
//e5370249-9cfc-464a-b21d-6b77a44727ca
void renamedMethod() {…}

//ac325425-c541-4a99-b80c-e3378d66e612
void methodBodyChanged() {
  <e5370249...>();
}
```

```
//e5370249-9cfc-464a-b21d-6b77a44727ca
void renamedMethod() {…}

//ac325425-c541-4a99-b80c-e3378d66e612
void methodBodyChanged() {
  renamedMethod();
}
```

Figure 7. Translating identifiers by reference in body modify transformations.

In case of a block comment or a Javadoc comment, the new UUID is appended in a new line at the end of its content.

Jaid also uses JavaParser for extracting transformations. When comparing nodes to understand whether they have changed, the built-in AST node comparison is extended so that identifiers, present in method bodies or field initializers, are compared by reference.

Finally, JavaParser's node properties are accessed in the handlers of conflict detection (Section 3.3.2), and transformations are applied through AST manipulations to obtain the final merged version (Sections 3.3.3 and 3.3.4).

4.2 Parsing and Indexing

After the projects have been parsed, all elements that have UUIDs are indexed by mapping a UUID to its corresponding element. The indexed elements are files, types (classes, interfaces, and enums), constructors, enum constants, methods, and fields. Indexing members allows for more efficient searches and comparisons of elements of the same type. There is also an index that maps a method to all its calls, an index of field and enum constant references, and another with all the usages for classes, interfaces, and enums. These indexes are particularly important during the merge process phase when transformations need to be applied to a particular element, and it is necessary to know which calls and expressions refer to that element in order to apply the transformation. More concretely, a renaming of an element relies on this mechanism in order to reach all the references to that particular element. There are also the reverse indexes of those mentioned above, where each reference is mapped to the element to which it refers. These indexes are used when translating identifiers by reference.

This stage is the most costly of the process (see Section 5), because, as opposed to other merging techniques (e.g., [11, 26]), we load and index the entire version of the project. This cost comes with the advantage of allowing us to perform renaming transformations across the project, as well as moving elements. Such transformations will reduce the number of false negatives related to missing references (as in the example of Figure 3).

5 Experiment

To evaluate the feasibility of our approach in practice we carried out an experiment involving code from open-source code repositories. However, to our knowledge, there are no real merge scenarios with UUIDs attached available, nor any other form of identity on its elements. Therefore, we decided to alter existing merge scenarios by injecting UUID comments, and hence, obtain scenarios where we could test our merge process. We achieved this by using tree-matching tools to pair elements from different versions and further attach the same UUID to both elements.

5.1 Collecting Merge Scenarios

We extracted two sets of real merge scenarios from the supplementary material of the paper by Cavalcanti et al. [11]. A shortcoming of the provided merge scenarios was the fact that only the files that were textually modified were available in the material. Our approach requires not only the changed files but also the unchanged files to resolve the references. Thus, we obtained the commit ids that were also available in the authors' package in order to fetch the complete versions from the projects' Git repositories. We selected two out of the four projects available in the small sample contained in the package, Bukkit[6] and jsoup[7].

Once all the versions containing the whole project code had been fetched from Git, the next phase was to set up these projects by artificially "fabricating" the identity of their elements, as if it would have been maintained over time using our approach. The first step was to create a set of 3-tuples, each holding one file for each version (base, left, right), where all three files are the same file in three different versions. This correspondence is provided by the path within the project. If a file has been removed from one version to another, the file tuple represents the missing file as null.

Having obtained the set of file tuples, the next step was to find all the elements that are mapped between the base/left and base/right pairs of files from a single file tuple. This is done using a tree matcher, already used in other studies [8, 26] as a tool to calculate differences in ASTs, but in our case to give identity to their elements. We used GumTree [12] for this purpose, one of the state-of-the-art tools, with a tree generator based on JavaParser's ASTs.

After finding all the mappings resulting from the GumTree tool, correspondence was made between a GumTree node

[6]https://github.com/Bukkit/Bukkit/
[7]https://github.com/jhy/jsoup

and a JavaParser node based on their positions in the source, and a UUID comment was injected into these elements. After setting up all the versions, an experiment was carried out involving a total of 63 merge scenarios, 19 from Bukkit and 44 from jsoup.

5.2 Results

Jaid was used to detect conflicts and to check for potential false positives and false negatives. To achieve the objective a manual analysis was performed. The merge scenarios were divided into two groups: the group of merge scenarios without conflicts and the group with conflicts. Table 2 presents the results per project of the number of merge scenarios and how many of them are conflict-free or not. Note that some merge scenarios were excluded because at least one of their transformation sets (left or right) is empty due to changes over members not being covered by Jaid (see section 4).

5.2.1 False Negatives. The main purpose of analyzing the group without conflicts was to find potential false negatives. After finding that there were no conflicts, both branches were merged into the base and the output of each file was written to a separate path. After copying the merge output files into a folder with a pre-configured project according to the version's project (Bukkit or jsoup), we investigated if these files would build successfully, and if not, what was the reason.

We found 9 (21%) scenarios in which errors were related to missing Java's coverage of Jaid. The most frequent case (7 out of 9 scenarios) were methods defined inside class bodies of enum constants that were changed in the branches, as well as their calls. As these methods are also called by other methods covered by Jaid, the bodies were changed, but the references were broken, not least because the methods not covered by Jaid did not even have a UUID attached. The remaining two scenarios failed to build successfully due to the absence of transformations regarding converting a class to an interface and exception classes changes in the throws keyword used along with the method signature. The remaining 34 (79%) merge scenarios were built successfully.

5.2.2 False Positives. The main goal of the conflict group analysis was to find potential false positives. After finding the versions with conflicting transformations, a manual evaluation was performed to understand the type of conflicts and the transformations that caused them, in order to investigate whether the generated conflicts were in fact real conflicts or not (*false positives*). A total of 40 conflicts were found, but only 6 different types of conflict, a small fragment of the conflictual pairs. Table 3 presents the types of conflict found. All of these conflicts were correctly detected since they all actually reference two conflicting transformations. Therefore, no false positives were found.

5.2.3 Execution Time. We also evaluated the performance of the Jaid merge processes with the objective of finding out if the process would require long execution times that would make the approach inviable. The merge executions were performed on a laptop with Intel Core i7, 14 cores @ 3 GHz, and 16 GB RAM on Windows 10 (64-bit).

We calculated the execution times of the entire merge process, as well as the parsing/indexing and applying transformations phases separately. Based on the average of the measured times, parsing/indexing is the most time-consuming task of the entire merging process, consuming approximately 50% and 70% of the time for Bukkit and jsoup, respectively. The remaining time is consumed in extracting transformations (45% for Bukkit and 28% for jsoup), detecting conflicts, and applying transformations phases.

Our implementation, being a prototype, has many points where it can be optimized. Besides, the whole merging process is done sequentially, while parsing and especially indexing could take advantage of parallelization. The overall execution times of the merging process took on average 2.9 and 13.7 seconds for Bukkit and jsoup projects, respectively, which we consider acceptable execution times to make the proposed approach a viable option in practice.

Additionally, since the parsing/indexing phase is the most time-consuming task of the whole process, we explored whether the number of lines of code and the execution time of the whole process is directionally proportional to each other. Figure 8 presents the relationship between lines of code and the merge process execution time for each version, and the results seem to point to a linear relationship.

6 Discussion

6.1 Drawbacks

The proposed approach, being language-specific, requires a different implementation for each programming language, despite the overall process being similar. As with testing toolkits, profilers, linters, etc, we argue that language-specific merging tools could pay off, given that they would provide significant improvements in conflict detection and in the overall usability of the merging process.

Another disadvantage of our approach is to require compilable code without broken references. If some references to members cannot be resolved, this implies that Jaid has incomplete information for the transformations, and consequently, that may affect their application and accurate conflict detection. Nevertheless, it is not common practice to commit project versions with compilation errors, and hence, we consider that this is not a severe limitation.

Another aspect that could compromise the approach is if UUID comments are accidentally broken by using operations such as delete or cut-and-paste, which would lead to dangling or incomplete UUID comments. This situation could be particularly common for members with only the UUID in a

Table 2. Summary of results for all merge scenarios.

	Scenarios	Excluded	Conflict-free	With Conflicts
Bukkit	19	0 (0%)	13 (68%)	6 (32%)
jsoup	44	6 (14%)	30 (68%)	8 (18%)
Total	**63**	**6 (10%)**	**43 (68%)**	**14 (22%)**

Table 3. Summary of results for all merge scenarios. (*) Refers to any transformation that involves a modification to the child elements of the removed one.

Conflicts	Conflict type	Explanation
1	SetJavadoc–SetJavadoc	Both Javadoc changes are different.
6	Imports–Imports	The same file's imports are being changed to two different import lists.
17	RemoveFile–*	The removed file has changes in it.
5	Body–Body	Both body changes are different.
10	RemoveCallable–*	The removed callable has changes in it.
1	AddCallable–Signature	The new added callable has the same signature as the changed one.

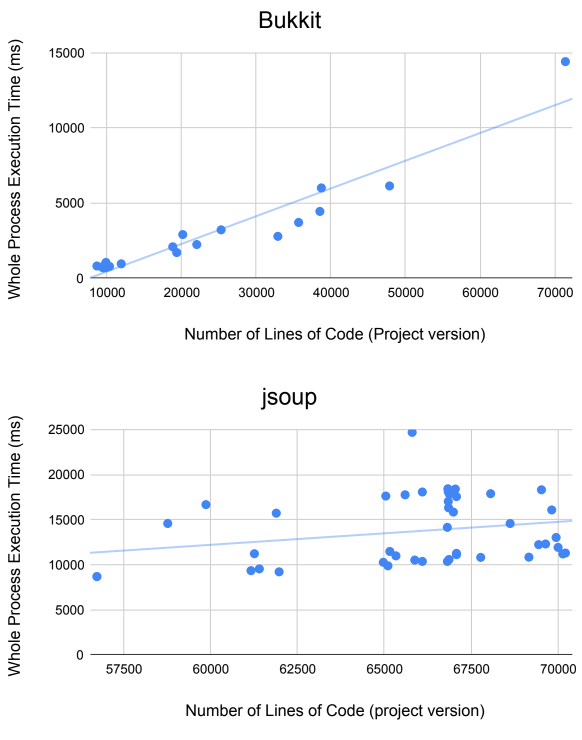


Figure 8. Merge times in relation to project size.

single line comment, which, by being hidden, could move the entire member declaration elsewhere, leaving the comment behind. In the case of members with block or Javadoc comments, we believe this would be less likely to occur, as the selection of the entire member declaration would start from the beginning of the comment until the end of the member declaration.

Having UUIDs in the member comments is harmless and negligible with respect to the storage and machine-processing of source code. However, from a developer's perspective, it is obvious that the UUIDs embody additional visual clutter that may hinder the usability of the code editor (recall Figure 4). These extraneous elements add no value to usual development settings, so we speculate that it may simply annoy most developers. Therefore, we believe that applying our approach in practice would be smoother if code editors are slightly adapted. One solution to the clutter problem could be based on a well-known feature of modern popular IDEs (e.g., Eclipse, IntelliJ, VS Code) — collapsing of lines of code. A simple plugin for those IDEs would hide UUID lines, and the appearance of the code editor would not differ much from the conventional one.

Another option would be to use a *projectional editor* that handles UUID comments in a specialized way. Projectional editors are closely related to the notion of *structured editors*, an old idea that never gained wide popularity but still with active research (e.g., [6, 15, 17, 25]). As opposed to conventional code editors, projectional editors typically use a different representation for storage (i.e. file content) and editing, implying that what is visible in the editor is not necessarily a direct representation of the file content (e.g., Domain Workbench [28], MPS). We aim at a similar outcome, but we do it in a non-disruptive way by not requiring a different storage format for source files. In this way, UUID comments would be completely hidden from the editor, as they would not be even part of the projection. Furthermore, when types and members are created, UUIDs comments could be injected directly by the editor, as opposed to having that performed at the commit phase (as discussed in Section 3.1).

6.2 Benefits

The downside of having a language-specific approach also has its advantages, given that such a specificity allows more precise conflict detection due to distinct features of programming languages. Consider the following examples: Java and C++ support method overloading whereas Javascript and Python do not; the modifiers and their valid combinations are different among languages, as well as namespace schemes. Having generic transformations that work well for numerous, diverse languages, would be a very complex endeavor.

The experiment conducted suggests that, with an exhaustive list of transformations and a complete library of conflict types, this approach, assuming the UUIDs are preserved properly, could evolve towards nearly eliminating false positives and false negatives, resulting in an accurate merge (or no merge, if there are conflicts). By accurate merge, we do not mean a fully automated merge, but rather an error-free merge that correctly detects all existing conflicts, along with the insertion of all features introduced by the branch, and the display of all conflicts that could not be resolved automatically, since there will always be conflicts that require human intervention.

We envision a user-friendly GUI facility, which we did not implement so far, where one could select among alternative transformations to solve at least a part of the conflicts. For example, when facing a clashing rename conflict, one would decide which one to use without having to edit the code. In turn, the chosen transformation would become conflict-free, whereas the discarded one would be removed from its set. We are confident that some decisions could be made by means of light interaction with tool assistance, but others certainly would require manual intervention. Our merge process, by working with typed transformations and conflict objects formed using those, may facilitate having a good conflict resolution usability, because the modifications involved are categorized and well-defined.

Finally, a pragmatic advantage of our approach is its compatibility with existing toolchains. It allows one to obtain the benefits of projectional editing (having UUIDs) in code merging without using a projectional editor (as discussed in Section 1).

6.3 Threats to Validity

The fact of running an experiment with only two projects, even though 63 scenarios were evaluated, not only is a relatively small sample of the Java projects, but also the diversity of merge scenarios is questionable, since all merge scenarios of the same project may share a similar development style. Therefore, the sets of transformations extracted from versions of those projects are not diversified. Running experiments with more projects and diversified scenarios is necessary to support our claimed benefits.

The lack of Java's coverage does not allow us to confidently claim that there is no Java grammar element that, if covered, would not cause some sort of malfunction in the overall merge process. Also, checking for conflicts between only two transformations may not be sufficient to detect some more advanced and complex conflicts, since a conflict may not exist between two transformations, but with the addition of a third one, a conflict may arise.

7 Related Work

7.1 Unstructured Merging

Unstructured merging is a purely textual merging technique [8]. Line-based algorithms are the most common strategy for this type of merging [16, 21], meaning that these tools compare the files on a line-by-line basis and detect conflicts based on chunks, the lines that are different between versions. Most of the available unstructured merge tools are based on the definition of the *diff3* algorithm [16]. Examples of unstructured merge tools are the Concurrent Version System (CVS) [7] and the *rcsmerge* tool in the Revision Control System (RCS) [30]. The genericity offered by tools based on this technique, since all software artifacts can be represented as plain text or in binary files, and the small computational times required are the major advantages, are most likely the reason why it is still the state of the practice. Unstructured merging is not only imprecise, as it does not take into consideration the grammar of the programming language, which sometimes results in unexpected conflicts (*false positives*), but also untrustworthy because it does not report situations that are in fact conflictual (*false negatives*) [11].

7.2 Structured Merging

Structured merge tools, because they take the grammar of the source code into consideration, are more precise [2] because they detect conflicts that are related to syntax errors, the so-called syntax conflicts, and ignore conflicts detected by unstructured merge tools that make no difference to the overall structure of the source code. On the other hand, since the merge tools require knowledge of the particular programming language, the genericity advantage over text-based merge tools is lost. Also, as noted by Apel et al. [2], even after auto-tuning an implemented structured merge tool, JDime, both syntactic merge techniques with and without auto-tuning have significantly lower performance than unstructured merge tools. OperV [22] and JDime [2] are examples of tree-based merging techniques. In addition, Asenov et al. [5] proposed a novel approach to versioning trees that provides more accurate diffs through a novel algorithm for a three-way merging of trees based on tree differencing algorithms such as GumTree [12] or ChangeDistiller [13] and node IDs, which are stored in a custom storage format. These approaches compute differences and detect renames based on the relative position of nodes within trees. As long as the nodes remain in the same position and with the same parent, they are correctly matched. Therefore, their process of extracting differences between versions is an approximation, whereas in our case, assuming that UUIDs are correctly preserved, the match between two members with the same UUID is always exact.

7.3 Semi-Structured Merging

Semi-structured merge tools, as the name suggests, are not fully structured, but are a combination of unstructured and structured techniques. Semi-structured merge tools use tree-based representations of the source code and differencing and matching algorithms to compute the similarity between trees and the differences between them. FSTMerge [3] and JDime [2] are examples of semi-structured merging tools. Both use a tree-based representation up to the method level and their full-text bodies in the leaves. Thus, structured merging techniques are used on the tree nodes up to the method level, and unstructured merging techniques are used in the method bodies. FSTMerge's approach differs from ours in the way it does tree differencing, which is done by superimposing trees, and in the way it handles body differencing.

Large-scale experiments have demonstrated that semistructured merge significantly reduces conflicts when compared to unstructured merge, with a performance that is not prohibitively slow [11, 26]. The FSTMerge approach was improved in jFSTMerge [11], reducing false positives and false negatives. IntelliMerge [26] has further improved precision with a graph-based approach to match code elements of branches to be merged and detect refactoring operations, while also improving merge execution time. IntelliMerge's approach differs from ours not only in being graph-based, but also in that it relies on matching vertices for graph differencing and uses the GumTree algorithm for body differencing. Note that none of these approaches has an infallible differencing step, as explained above in Section 7.2, nor they are operation-based. A key difference of our approach is that it puts more emphasis on language-specific constructs, allowing a more precise conflict detection.

7.4 Model Differencing

Our approach is performing model differencing [1] when extracting the transformations. Previous approaches and tools have addressed this problem using methods to differentiate models that conform to a given metamodel. DSMDiff [19] describes differencing algorithms for metamodels of the Generic Modeling Environment (GME [18]). The Eclipse Modeling Framework (EMF [29]), an implementation of the Meta-Object Facility OMG's standard [24], embodies a metamodeling language (Ecore) that can be used to model arbitrary domains, including the structure of a program in a given programming language. In this context, the EMF Compare project[8] provides facilities to compare models that conform to an Ecore metamodel. MPS also offers differencing capabilities for models described in the tool-specific metalanguage. Despite the several existing methods for generic model differencing, in our approach, we opted for working with Java-specific models (JavaParser's AST) in other to perform fine-grained conflict detection (recall Section 3.3.2). We could have modeled Java code, for instance, using EMF, and in turn benefit from its model comparison facilities. However, that would imply redoing much of what is already well-implemented in JavaParser, most notably, the model definition, the parsing of Java code to obtain model instances, the API to manipulate those, and the resolution facilities for types and references that are specific to Java.

7.5 Operation-Based Merging

Change-based merging tools [21] capture changes as they occur. Since the changes are captured, there is no need for the differencing process, which consists of computing the differences between two versions of the same software artifact. Operation-based merging [20] is a variant of change-based merging because it represents the changes as operations (or transformations) that, when applied to a particular state of the software artifact, become its subsequent state. Some advantages and problems of operation-based merging are discussed in [20], as well as the proposal and implementation of algorithms capable of segregating the conflicting transformations into sets. OperV [22] also uses an operation-based approach, where changes to the system are represented by editing transformations on the tree used to store the structural information of the project.

As pointed out by [20], one of the limitations of operation-based merging is that the order in which certain operations are applied affects the result, since two different orders can produce two different outputs. Appendix B illustrates this issue with the scenario of Figure 6.

8 Conclusions

Over the years, one of the major challenges in software merging has been the inability to correctly match the same members between versions. Thus, members with fully defined identities are a form of avoiding ambiguous and possibly inaccurate matches that indirectly affect the merge result. By providing a more controlled evolution of software artifacts through the sequential application of transformations that should not introduce syntactic errors if implemented correctly, false negative conflicts related to semantic compilation errors can be reduced. Also, more precise merge commits can be achieved by reducing false positive conflicts, with a detailed and exhaustive knowledge base about the programming languages' syntax incompatibilities. Our contribution was a first step towards this goal.

[8] https://projects.eclipse.org/projects/modeling.emfcompare

We have shown how more cases of false negatives and false positives can be successfully addressed, most notably, with respect to changes involving renamings. The experiment results provided some evidence that the approach is feasible in practice. Despite the constraint of maintaining UUIDs as code comments, a slight deviation from standard practice, our approach is still largely compatible with software development practices. Despite that code editing environments would have to be lightly adapted, we could benefit from some advantages of projectional editors but remain close to regular development settings.

In future work, we plan to improve the coverage of Java’s constructs in Jaid, and run a large-scale experiment to evaluate the approach with more depth. Up to this point, we were focused on achieving a proof of concept, and no efforts have been made regarding optimization — we believe there is room for improvement here, too. As discussed in Section 6.2, a GUI tool to select among alternative transformations would also be a useful addition to our work.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback on earlier versions of this paper. This work was partially supported by Fundação para a Ciência e a Tecnologia, I.P. (FCT) [ISTAR Projects: UIDB/04466/2020 and UIDP/04466/2020].

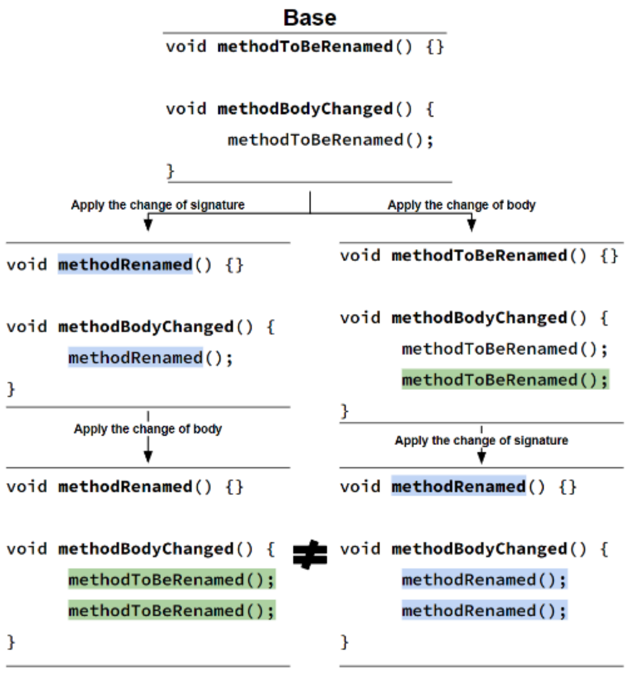
A Conflict Type Pairs

Type	Package	Imports	Add	Remove	Rename	Modifiers	Implements	Extends	Move
Move			•	•	•				•
Extends				•				•	
Implements				•			•		
Modifiers				•		•			
Rename			•	•	•				
Remove									
Add			•						
Imports		•							
Package	•								

Method	Add	Remove	Signature	Body	Modifiers	Return Type	Move
Add	•		•				•
Remove			•	•	•	•	•
Signature			•				•
Body				•			
Modifiers					•		
ReturnType						•	
Move							•

Field	Add	Remove	Rename	Type	Modifiers	Initializer	Move
Move	•	•	•				•
Initializer		•		•		•	
Modifiers		•			•		
Type		•		•			
Rename	•	•	•				
Remove							
Add	•						

B Transformation Ordering in Operation-Based Merge



```
void methodToBeRenamed() {}

void methodBodyChanged() {
    methodToBeRenamed();
}
```

```
void methodRenamed() {}

void methodBodyChanged() {
    methodRenamed();
}
```

```
void methodToBeRenamed() {}

void methodBodyChanged() {
    methodToBeRenamed();
    methodToBeRenamed();
}
```

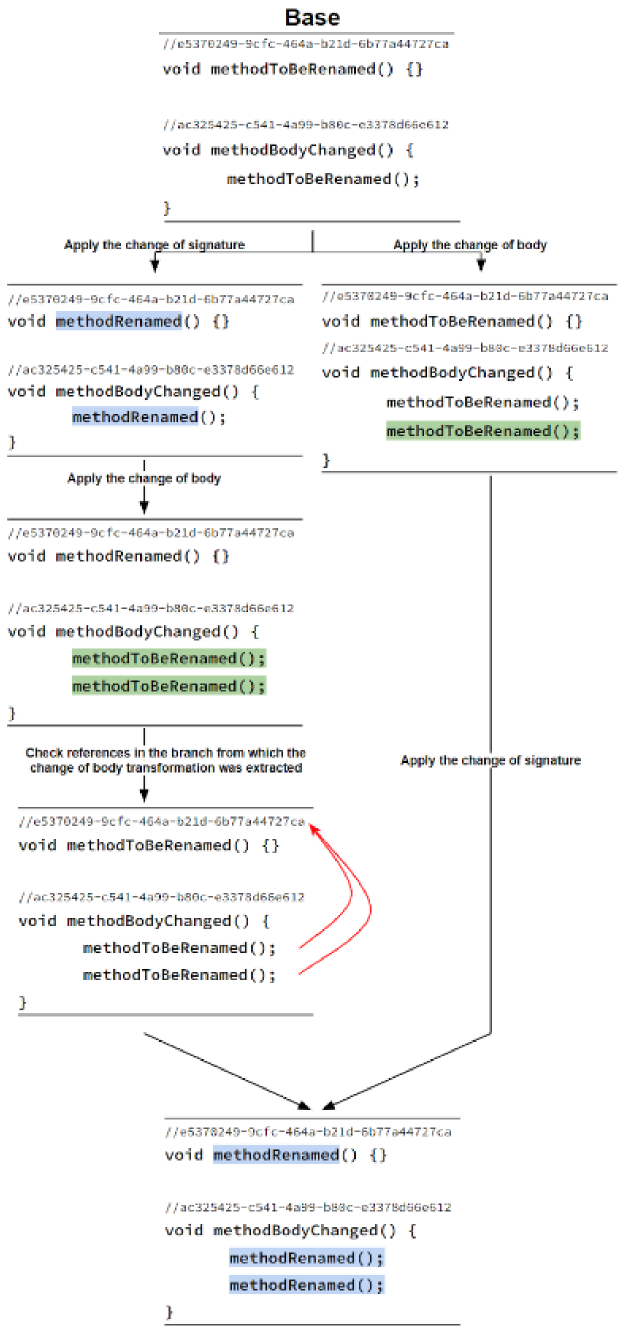
```
void methodRenamed() {}

void methodBodyChanged() {
    methodToBeRenamed();
    methodToBeRenamed();
}
```

```
void methodRenamed() {}

void methodBodyChanged() {
    methodRenamed();
    methodRenamed();
}
```

C Translate Identifiers by Reference


Base
Apply the change of signature
Apply the change of body
Apply the change of body
Check references in the branch from which the change of body transformation was extracted
Apply the change of signature

```
//e5370249-9cfc-464a-b21d-6b77a44727ca
void methodToBeRenamed() {}

//ac325425-c541-4a99-b80c-e3378d66e612
void methodBodyChanged() {
        methodToBeRenamed();
}
```

```
//e5370249-9cfc-464a-b21d-6b77a44727ca
void methodRenamed() {}

//ac325425-c541-4a99-b80c-e3378d66e612
void methodBodyChanged() {
        methodRenamed();
}
```

```
//e5370249-9cfc-464a-b21d-6b77a44727ca
void methodToBeRenamed() {}
//ac325425-c541-4a99-b80c-e3378d66e612
void methodBodyChanged() {
        methodToBeRenamed();
        methodToBeRenamed();
}
```

```
//e5370249-9cfc-464a-b21d-6b77a44727ca
void methodRenamed() {}

//ac325425-c541-4a99-b80c-e3378d66e612
void methodBodyChanged() {
        methodToBeRenamed();
        methodToBeRenamed();
}
```

```
//e5370249-9cfc-464a-b21d-6b77a44727ca
void methodToBeRenamed() {}

//ac325425-c541-4a99-b80c-e3378d66e612
void methodBodyChanged() {
        methodToBeRenamed();
        methodToBeRenamed();
}
```

```
//e5370249-9cfc-464a-b21d-6b77a44727ca
void methodRenamed() {}

//ac325425-c541-4a99-b80c-e3378d66e612
void methodBodyChanged() {
        methodRenamed();
        methodRenamed();
}
```

References

[1] Marcus Alanen and Ivan Porres. 2003. Difference and Union of Models. In *«UML» 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2863)*, Perdita Stevens, Jon Whittle, and Grady Booch (Eds.). Springer, 2–17. https://doi.org/10.1007/978-3-540-45221-8_2

[2] Sven Apel, Olaf Leßenich, and Christian Lengauer. 2012. Structured merge with auto-tuning: balancing precision and performance. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 120–129. https://doi.org/10.1145/2351676.2351694

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

[4] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2007. JDiff: A Differencing Technique and Tool for Object-Oriented Programs. *Automated Software Eng.* 14, 1 (mar 2007), 3–36. https://doi.org/10.1007/s10515-006-0002-0

[5] Dimitar Asenov, Balz Guenat, Peter Müller, and Martin Otth. 2017. Precise Version Control of Trees with Line-Based Version Control Systems. In *Fundamental Approaches to Software Engineering*, Marieke Huisman and Julia Rubin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 152–169.

[6] Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. 2023. Structured Editing for All: Deriving Usable Structured Editors from Grammars. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI ’23)*. Association for Computing Machinery, New York, NY, USA, Article 595, 16 pages. https://doi.org/10.1145/3544548.3580785

[7] Brian Berliner. 1990. CVS II: Parallelizing Software Development. In *Proc. The Advanced Computing Systems Professional and Technical Association (USENIX) Conf.* 22–26.

[8] Caius Brindescu. 2018. How Do Developers Resolve Merge Conflicts? An Investigation into the Processes, Tools, and Improvements. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 952–955. https://doi.org/10.1145/3236024.3275430

[9] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. 2020. An Empirical Investigation into Merge Conflicts and Their Effect on Software Quality. *Empirical Softw. Engg.* 25, 1 (jan 2020), 562–590. https://doi.org/10.1007/s10664-019-09735-4

[10] Jim Buffenbarger. 1995. Syntactic software merging. In *Software Configuration Management*, Jacky Estublier (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 153–172.

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

[12] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE ’14)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[13] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. 2007. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743. https://doi.org/10.1109/TSE.2007.70731

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc.

[15] David B. Garlan and Philip L. Miller. 1984. GNOME: An Introductory Programming Environment Based on a Family of Structure Editors. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*. Association for Computing Machinery, New York, NY, USA, 65–72. https://doi.org/10.1145/800020.808250

[16] J. W. Hunt and M. D. Mcilroy. 1975. An algorithm for differential file comparison. *Computer Science* (1975). http://www.cs.dartmouth.edu/%7Edoug/diff.pdf

[17] Amy J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada) *(CHI '06)*. Association for Computing Machinery, New York, NY, USA, 387–396. https://doi.org/10.1145/1124772.1124831

[18] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. 2001. Composing Domain-Specific Design Environments. *Computer* 34, 11 (nov 2001), 44–51.

[19] Yuehua Lin, Jeff Gray, and Frédéric Jouault. 2007. DSMDiff: a differentiation tool for domain-specific models. *European Journal of Information Systems* 16, 4 (2007), 349–361. https://doi.org/10.1057/palgrave.ejis.3000685

[20] Ernst Lippe and Norbert van Oosterom. 1992. Operation-Based Merging. *SIGSOFT Softw. Eng. Notes* 17, 5 (nov 1992), 78–87. https://doi.org/10.1145/142882.143753

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

[22] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, and Tien N. Nguyen. 2010. Operation-Based, Fine-Grained Version Control Model for Tree-Based Representation. In *Fundamental Approaches to Software Engineering*, David S. Rosenblum and Gabriele Taentzer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 74–90.

[23] Object Management Group 1999. *OMG XML Metadata Interchange (XMI) Specification Version 1.1.* Object Management Group, Framingham, Massachusetts.

[24] OMG. 2013. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1. http://www.omg.org/spec/MOF/2.4.1

[25] André L. Santos. 2020. Javardise: A Structured Code Editor for Programming Pedagogy in Java. In *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) *(Programming '20)*. Association for Computing Machinery, New York, NY, USA, 120–125. https://doi.org/10.1145/3397537.3397561

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

[27] Léuson Silva, Paulo Borba, and Arthur Pires. 2022. Build conflicts in the wild. *Journal of Software-Evolution and Process* (2022), (also appeared in ICSME'2022 Journal First track).

[28] Charles Simonyi, Magnus Christerson, and Shane Clifford. 2006. Intentional Software. *SIGPLAN Not.* 41, 10 (oct 2006), 451–464. https://doi.org/10.1145/1167515.1167511

[29] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework* (2 ed.). Addison-Wesley, Boston, MA.

[30] Walter F. Tichy. 1985. Rcs — a system for version control. *Software: Practice and Experience* 15, 7 (1985), 637–654. https://doi.org/10.1002/spe.4380150703 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380150703

[31] A. van Deursen, P. Klint, and F. Tip. 1993. Origin Tracking. *J. Symb. Comput.* 15, 5–6 (may 1993), 523–545. https://doi.org/10.1016/S0747-7171(06)80004-0

[32] Riemer van Rozen and Tijs van der Storm. 2015. Origin Tracking + Text Differencing = Textual Model Differencing. In *Theory and Practice of Model Transformations*, Dimitris Kolovos and Manuel Wimmer (Eds.). Springer International Publishing, Cham, 18–33.

[33] Bernhard Westfechtel. 1991. Structure-Oriented Merging of Revisions of Software Documents. In *Proceedings of the 3rd International Workshop on Software Configuration Management* (Trondheim, Norway) *(SCM '91)*. Association for Computing Machinery, New York, NY, USA, 68–79. https://doi.org/10.1145/111062.111071

Received 2023-04-28; accepted 2023-08-11